UNIVERSITÉ DE NAMUR
Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

DOCTOR OF SCIENCES

On automatic, constraint-based test-case generation for Mercury and its application to imperative languages

Degrave, François

Award date:
2013

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2024

University of Namur
Faculty of Computer Science

March 2013

On automatic, constraint-based test-case generation for Mercury and its application to imperative languages

François Degrave

Jury :
Prof. Puri Arenas (Complutense University of Madrid, Spain)
Prof. Vincent Englebert (University of Namur)
Prof. Baudouin Le Charlier (Université Catholique de Louvain)
Prof. Tom Schrijvers (Ghent University)
Prof. Wim Vanhoof (University of Namur)

UNIVERSITÉ DE NAMUR
FACULTÉ D'INFORMATIQUE

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science

Thesis director: Prof. Wim Vanhoof

Abstract

Creating a piece of software behaving the way the user expects it to behave is a central problem in computer science. Once a system is implemented, it needs to be evaluated in order to verify that it accurately and completely fulfils the initial expectations. Arguably the most commonly applied strategy to achieve this is *testing*. In testing terminology, we call a *test case* the combination of a single input for a software component with the expected result of its execution using that input, whereas a *test suite* refers to a collection of individual test cases. Testing refers to the activity of running a software component with respect to a well-chosen test suite and comparing, for each test case, the output that is produced with the expected result in order to find errors.

The difficulty in software testing is due to the complexity of the systems; this complexity has never stopped growing over the years, which renders the need for constant improvement of testing techniques crucial. In particular, a test suite must be constructed in such a way that it will allow the testers to discover as many errors as possible in the program. While it is provably impossible to design test suites in such a way that the program is exercised in *all* the possibles ways it can possibly be, computer scientists have defined different *adequacy criteria* which, if satisfied by a given test suite, indicate that the successful execution of this test suite will *sufficiently* increase the confidence of the testers in the correctness of a program.

The hard part of the testing process is *constructing* a test suite which satisfies the chosen adequacy criteria. This activity can be very time-consuming when performed manually; moreover, the resulting test suites are very large and complex, to such an extent that they can themselves contain errors. There exists therefore a strong interest in automating this process.

In this work, we present a testing framework for the logic programming language Mercury able to *generate* and *execute* test suites that satisfy a given set of adequacy criteria. The technique we define is based on symbolic execution and constraints, and is able to deal with complex (possibly user-defined) data types. We also show how we can adapt this method in order to generate test suites satisfying sets of adequacy criteria in the context of imperative programming languages using heap-allocated pointer-based data structures.

Résumé

La création de logiciels fonctionnant de la manière attendue par l'utilisateur est un problème central de l'informatique. Lorsqu'un système a été implémenté, il doit être évalué afin de vérifier s'il satisfait complètement et précisément aux attentes initiales. La stratégie la plus répandue pour effectuer cette évaluation est sans doute le *test*. Dans la terminologie du testing, un *cas de test* est la combinaison d'une donnée d'entrée pour un composant logiciel avec le résultat attendu de l'exécution de ce composant en utilisant cette donnée d'entrée ; d'autre part, une *suite de test* désigne un ensemble de cas de test. Le test lui-même consiste donc en l'exécution du composant logiciel avec une suite de test bien choisie et en la comparaison, pour chaque cas de test, du résultat produit en regard du résultat attendu, dans le but de détecter des erreurs. La difficulté du test de logiciels est dûe à la complexité des systèmes ; cette complexité n'a cessé de grandir avec les années, rendant crucial le besoin d'une amélioration continue des techniques de test. Plus particulièrement, la construction d'une suite de test doit être réalisée d'une façon telle qu'elle permette aux testeurs de découvrir un maximum d'erreurs présentes dans le programme. Si l'impossibilité de concevoir des suites de test éprouvant un programme de *toutes* les manières possible a été formellement établie, les informaticiens ont toutefois défini différents *critères d'adéquation* qui, s'ils sont satisfaits par une suite de test donnée, indiquent que l'exécution réussie de cette suite de test pourra *suffisamment* accroître la confiance des testeurs en l'exactitude d'un programme. La phase la plus difficile du processus de test est la *construction* d'une suite de test satisfaisant des critères d'adéquation sélectionnés. Cette activité peut être extrêmement consommatrice en temps lorsqu'elle est effectuée manuellement ; de plus, les suites de test qui en résultent sont généralement très longues et complexes, à tel point qu'elles peuvent elles-mêmes contenir des erreurs. Il existe dès lors un fort intérêt dans l'automatisation de cette procédure.

Dans cet ouvrage, nous présentons une plateforme de test pour le langage de programmation Mercury capable de *générer* et *exécuter* des suites de test satisfaisant un ensemble de critères d'adéquation. Notre technique basée sur l'exécution symbolique et les contraintes est capable de traiter des types de données complexes (éventuellement définis par l'utilisateur). Nous adaptatons ensuite notre méthode afin de générer des suites de test satisfaisant des ensembles de critères d'adéquation dans le contexte de langages de programmation impératifs utilisant des structures de données basées sur les pointeurs.

Contents

1 Introduction **1**

1.1 Test cases and test suites 3
1.2 Black-box and Glass-box testing 4
1.2.1 Black-box testing 4
1.2.2 Glass-box testing 5
1.3 Adequacy criteria 6
1.3.1 Control-flow-based adequacy criteria 7
1.3.2 Data-flow adequacy criteria 11
1.4 Fault-based evaluation of a test suite effectiveness 12
1.5 Test data generation: state-of-the-art 14
1.5.1 Test data generation using symbolic execution 14
1.5.2 Test data generation using numerical analysis 21
1.5.3 Test data generation using the specifications of the program 23
1.5.4 Other test data generation techniques 24
1.5.5 Structure of the work 25

2 Technical background **27**

2.1 The Mercury language 27
2.1.1 Overview of logic programming 28
2.1.2 Mercury's Type System 31
2.1.3 Mercury's Mode and Determinism System 33
2.1.4 Mercury superhomogeneous form 35
2.1.5 A semantics for Mercury 38

3 A test automation framework for Mercury **43**

3.1 Test automation framework 43
3.2 Unit testing tool for Mercury 44
3.2.1 Determinism 45
3.2.2 Implementation details 47
3.2.3 Handling exceptions 48
3.3 Evaluation 49

4 A control flow graph for Mercury 51
4.1 Constructing the graph . . . 52
4.2 Deriving execution sequences . . . 55
4.2.1 Formal definition of symbolic execution . . . 59
4.2.2 Correspondence between execution sequences and semantics traces . . . 63
4.3 Adapting control-flow-based adequacy criteria to Mercury . . . 64
4.4 Using the control flow graph in coverage measurement . . . 67
4.4.1 Implementation . . . 68
4.4.2 Switches vs. disjunctions . . . 69
4.4.3 Computing the coverage rate with respect to coverage criteria . . . 73
4.4.4 Graphical visualization of the atom coverage and arc coverage criteria . . . 74
4.4.5 Evaluation . . . 74

5 Test data generation for Mercury 77
5.1 A brief introduction to Constraint Programming . . . 77
5.1.1 CSP, propagation and search . . . 78
5.1.2 Constraint Programming and Constraints Solving . . . 80
5.2 Segment conditions and sequence conditions . . . 82
5.3 Properties of the analysis . . . 84
5.4 Constraint Solving . . . 86
5.5 Implementation and Evaluation . . . 87
5.6 Automatic completion of test suites . . . 90

6 TDG for a pointer-based imperative language 93
6.1 Existing work on TDG for imperative languages using complex data structures . . . 93
6.2 The ImpL language . . . 98
6.3 Generating test inputs . . . 99
6.3.1 Overview . . . 99
6.3.2 Constraint Generation . . . 102
6.3.3 Properties . . . 106
6.3.4 Constraint Propagation . . . 107
6.3.5 Search . . . 108
6.3.6 Generalized Data Structures . . . 109
6.4 Applications . . . 110
6.5 Proof of Completeness Theorem . . . 113
6.6 Proof of Soundness Theorem . . . 115

7 Mercury normal form 119
7.1 Introduction and motivation . . . 119
7.2 Mercury Core Syntax . . . 121
7.3 Transformation to Normal Form . . . 123
7.4 Detecting duplicated functionality and experimental results . . . 131

8 Conclusion **135**

Appendix **141**

Bibliography **145**

Chapter 1

Introduction

Creating a piece of software behaving the way the user expects it to behave is a central problem in computer science. For that purpose, the development process is structured and controlled using frameworks called *software development methodologies*. Those frameworks have been widely used since the 1960's and a wide variety of them have evolved over the years (Yourdon 1977; DeMarco 1996; Mohagheghi 2008). Nowadays, the most well-known such methodologies are the waterfall, spiral and prototyping methodologies, along with techniques that are fully based on formal methods (Fujita and Zualkernan 2008).
So-called formal methods are techniques enabling software construction with a formal verification based on mathematical logic. The creation of software based on a fully formal method consists in two steps; the first step is the specification of the system using a mathematical notation, or at least a formal language of which the semantics is well defined (Gravell 1990). This specification is seen as the initial design of the system, and is used as a reference during the implementation process. Once the system is implemented, it is used to (formally) check if the final system accurately and completely fulfills the initial expectations. The second step is the implementation itself; this implementation is considered to be a *refinement* of the specification into a working system. This refinement can itself be performed in several steps, each step being related to the previous one by a mathematical relation, each of which has to be *proven*. Many of those proofs can be automated using dedicated development tools (such as *Atelier B* for the B Method (Abrial 1996)). The formal methods are the only methods that don't use testing in order to detect unexpected behaviours of a system under construction. However, because they are time-consuming and expensive, those formal methods are in general used only for mission-critical software, which represent a very small part of all information systems produced.
Besides this small part, most other information systems are produced using approaches that can be seen as including, in some way, three main phases. First, the computer scientists analyse, identify and specify the requirements and usually use models to represent many different aspects of the system to create: its functionalities, its architecture, its internal processes interleaving, etc. This

analysis/modelisation phase allows the computer scientists not only to understand and formalize the requirements, make the application well-structured and therefore easier to maintain and upgrade, but also to avoid a wide range of logical errors afterwards (Bailey and Whiddett 1996). Then during the implementation phase, programmers apply standardized methodologies, respect good practices and refer to the specifications and models elaborated in the previous phase; it is also a wide-spread practice for programmers to frequently check if their code under development runs the way they expect it to by performing tests. Finally, the phase after the implementation is the verification and validation of the system (Royce 1987); this phase can possibly involve formal verification methods (such as model checking (Clarke, Long, and McMillan 1989)), however *testing* is the most widely used technique for this purpose (Harrold 2008). Testing can therefore be seen as the cornerstone of the evaluation of a system; unfortunately it is not always as efficient as expected. In practice, despite the multiple efforts undertaken during the development, estimates range from 50% to 75% the portion of a software development budget spent in corrective maintenance (Glass 1997). More than half of the maintenance cost is actually due to the correction of errors that were already present at delivery time. The other half is due to the addition of new functionalities, which can themselves introduce new errors in the code. That is why it is very important to improve the efficiency of testing, in order to reduce as much as possible the number of errors in a system before delivery time. A study conducted by NIST (National Institute of Standards and Technology, USA) in 2002 reports that software bugs cost the U.S. economy $ 59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed (Tassey 2002).

When it comes to developing a software system, the first thing computer scientists do (possibly together with the client ordering that system) is to define *what* the system should do and *how* it should do it; this first step is the definition of the *requirements*. The *what* part is referred to as the *functional* requirements; they specify the functions that the system must be able to perform. On the other hand, *non-functional* requirements refer to *how* the system should perform its tasks, the qualities it should have. Those qualities are usually divided in two categories: qualities regarding the execution, which are observable when running the system – such as security and usability –, and qualities regarding the evolution which are not observable at runtime and which depend on the inner structure of the system – such as maintainability, extensibility or scalability (Berztiss 1994).

Since requirements can be classified in two categories, there exist two kinds of testing processes as well, depending on what aspect of the system is tested: *functional testing* is the process of testing the system in order to determine if it satisfies all the functional requirements, i.e. if it is able to perform all the required functions correctly. *Non-functional testing* focuses on verifying if the system has the required qualities. In this work, we concentrate on functional testing and we will use the term “testing” as a synonym of “functional testing”.

1.1 Test cases and test suites

The word "testing" refers to the activity of using a system with the intent of finding errors (Myers 1979). This activity has been performed since long before the arising of computerized systems, in order to check if physical processes – machines – were running correctly. However, when a physical process can usually fail in a fixed and reasonably small set of ways, software can in general fail in an infinite number of ways, the reasons of which are sometimes far from obvious. Moreover, most of the errors in software originate from logical mistakes made by those who specified or implemented the code, not from manufacturing defects. This is why testing has been substantially adapted and has considerably evolved since it is used in the domain of software development(Beizer 1990).
The difficulty in software testing is due to the complexity of the systems; this complexity has never stopped growing over the years, which renders the need of constant improvement of testing techniques crucial.
In practice, a test resembles a scientific experiment. It examines an hypothesis represented by a triplet: the input data, the object to test (a program or a part of a program) and an expected output. Performing the test consists in executing the object (program or function) using the input data contained in the test, then compare the output produced with the expected output. If they are the same, the test *succeeds*, otherwise it *fails*. Similarly to a scientific experiment, a test is considered as valid only if it can be repeated, i.e. the result of the result of the test is the same each time it is performed if the triplet (input, object, expected output) is not modified. We now define two major notions used in testing terminology: the *test case* and the *test suite*.

Definition 1.1 The combination of an input data together with the output as expected to be produced by the program (fragment) under concern when executed using that input data, is referred to as a *test case*. ◇

Definition 1.2 A test case is said to be *executed* when the program (fragment) under concern is executed using the input data contained in that test case, and the actual result of this execution is compared with the expected result contained in the test case. If the actual and expected results are identical, the test case *succeeds*; it *fails* otherwise. ◇

Definition 1.3 A *test suite* refers to a collection of well-chosen test cases, executed in order to fully test a program (fragment). ◇

The notion of "fully testing" a program (fragment) is not defined here; it can refer to the fact of "testing the program with all possible input data", but this is in general impossible since the set of input data for a program is in general infinite. This is why a test suite is generally created in order to test a (part of a) program "as fully as required" by so-called *adequacy criteria* when executed. These notions are explained in further detail later in the current chapter.

Definition 1.4 A *test suite* is said to be *executed* when each test case contained in it is executed. The *result* of this execution is a report identifying the test cases that failed during the process. If this report is empty, the test suite *succeeds*. ◇

We can now define the notion of *testing* to which we will refer in the remaining.

Definition 1.5 *Testing a program (fragment)* is the fact of executing a test suite specifically created for that (part of) program. ◇

Note that, in the present work, we focus on testing techniques that aim at detecting defects in programs source code. However, testing can be used for other purposes; for example, *stress testing* is a technique used to check if a system meets its performance objectives. It usually consists in simulating the behaviour of real users of the system in a controlled environment, and mimicking a range of workload conditions including those observed at real systems. A number of measurements are performed during the test, such as response time, memory/CPU consumption, etc. and used to support sizing and capacity planning for example (Krishnamurthy, Rolia, and Majumdar 2006).

1.2 Black-box and Glass-box testing

There exist many ways of categorizing testing techniques. One of them classifies the techniques in two categories, depending on whether the test suites are created using knowledge about the internal structure of the source code or not. In the latter case, the system is seen as a black-box; this is why the tester performs, in that case, so-called *black-box testing.* End-users (i.e. participating to the public testing phase of a program) usually perform such black-box testing – either because they don't have access to the source code, or because the purpose wouldn't be worth the effort to understand that code. Other black-box testing techniques are used by testers during software development; in that case, testers *choose* for different reasons to design the test suites without taking the code structure into account. On the other hand, if the tester takes the code structure into account when designing the test suites, he performs so-called *glass-box testing* (or white-box testing). This kind of testing is widely used during software development, usually in combination with black-box testing techniques. In what follows we will discuss both techniques in a somewhat more detailed way.

1.2.1 Black-box testing

As its name indicates, black-box testing treats the system – or a component of the system – as a "black-box"; the tests to perform are designed without using knowledge of its internal structure (Beizer 1995). Using such a methodology, the tester determines what are the possible inputs for the application and what output should respectively result from the execution using each input. For example, if the tested object is a search engine, the tester enters text that he (or she) wants to search for in the text area, presses the "Search" button and checks if the results returned seem to be what he expected. In such case, he doesn't know about the specific process that is employed to obtain those results. If the tested object is a single component of a system, such as a procedure of

a program, the tester is not always able to determine if the output he gets is actually the expected output. He usually needs to refer to the *specification* of this component in order to check if it behaves the way it should. Such *specification-based testing* – focused on examining whether all the claims being made in the specifications are verified in the product – constitute an important part of black-box testing as used in the software industry (Hutcheson 2003).
Among the other numerous techniques belonging to this methodology there is of course "beta-testing" – the fact of releasing the product to people outside the company in order to confront it to real-world use and thus discover unknown errors (Fine 2002) – or *scenario testing*. In scenario testing, a test is based on a credible story about how the program is used; this story involves a complex use of the program, a complex environment or a complex set of data. The scenarios are usually created with respect to the requirements analysis performed before the system was implemented (Ambler 1995).

Black-box testing has many advantages; among them, we can obviously cite the *ease of use*. Indeed, because testers do not have to concern themselves with the inner structure and mechanisms of the system, it is easy for anybody to simply work through the application in order to test it. This kind of testing also enables a *quick test case development*, since the testers avoid spending time on identifying the internal execution paths involved in a specific process (Beizer 1995).
However, one of the greatest drawbacks of black-box testing is precisely that it is not based on the structure of the code – unlike glass-box testing – and as a result it can happen that (1) a tester writes many test cases to check something that could have been tested by a single test case, and/or (2) some parts of the back-end may not be tested at all. Moreover, due to the lack of a link with the source code, there is no way to detect the location of the bug in case running the test resulted in a failed test case (Chen, Tse, Chan, and Chen 1998). In order to overcome these limitations the testers can, when necessary, apply techniques of *glass-box testing*.

1.2.2 Glass-box testing

The glass-box testing methodology, also known as white-box testing, clear-box testing or structural testing, aims to develop tests for an application with full knowledge of its inner working. It allows the tester to create the test cases according to the program structure, ensuring that the execution of all the tests will result in the detection of a high rate of the errors possibly present in the system (Beizer 1990). Indeed, using this approach allows the tester to design test cases that (1) exercise independent execution paths within a module of the system; (2) exercise logical expressions for both their true and false values; (3) execute loops – which have been shown to be the most common cause of faults in programs – with different number of iterations; and (4) exercise the data structures used (and possibly defined) within the system to ensure their validity (Pressman 2001). On the other hand applying such techniques is obviously more

time-consuming (and thus more expensive) than black-box testing techniques; this is why techniques from both categories are usually used when developing a system. Some methods also use *both* the specifications and the source code to develop the test suites; such methods are often called *grey box* testing methods (Omar and Ibrahim 2010).

One can also classify testing according to the kind of software component it applies to; if the object under test is a function or a procedure, in other words the *smallest testable piece of a software*, it is called *unit testing*. When the interactions between those units are tested, it is called *integration testing*, and finally if the entire system is tested it is called *system testing* (Runeson 2006). An advantage of white-box testing is that it can be applied to any of those levels of the system (Beizer 1990).

1.3 Adequacy criteria

Dijkstra claimed in his *Notes on Structured Programming*, that "testing can be used to show the presence of bugs, but never to show their absence" (Dijkstra 1972). Of course, he was right; a test suite is a limited representation of possible inputs for a program. Testing a program in order to prove its correctness would require the test suite to contain test cases for the *entire set* of possible input values for the program. This is generally impossible since this set can be infinite. Therefore, the successful execution of a test suite can only increase the confidence of the testers in the correctness of a program. If this test suite contains *well-chosen* test cases, this confidence can be high enough to be reassured that the program will behave correctly in a large majority of use cases, and that the program can be released for a real-world use. Therefore, the central question when designing test suites is: how can we choose the test cases well? According to which *criterion*? That is, the criterion that defines what constitutes an adequate test suite. Since the '70s, this question has been a major research topic in software engineering (Goodenough and Gerhart 1975). A substantial number of such test criteria – usually called *adequacy criteria* – have been created and investigated. For black-box testing techniques, the only thing the tester can rely on for designing good test suites is the *specification* of the tested object, since there is no access to the source code. The so-called *specification-based* criteria specify the test cases required according to the features identified in the specifications of the tested object, so that a test suite is considered adequate if it fully exercises all those features. In glass-box testing, testers have access to the code and can therefore define *program-based* criteria for the test suites. Those criteria specify the test cases required according to whether the program has been thoroughly exercised (of course, adequacy criteria for glass-box testing can also be defined using a combination of the knowledge of the source code with the specifications). In this work we concentrate on this second class of adequacy criteria for test suites, that is the adequacy criteria for glass-box testing. Many such criteria exist, as well as various ways to classify them; in what follows we

choose to distinguish 2 categories. First, *control-flow-based adequacy criteria*, which are the most well-known and used such criteria; they are based on a flow-graph model of a program structure. Secondly we look at *data-flow-based adequacy criteria* which take into account the data flow information added to the flow graph of the program. Finally, we present techniques for *fault-based evaluation* of test suites, aiming to measure the quality of a test suite according to its ability to detect faults[1].

1.3.1 Control-flow-based adequacy criteria

Before we define the different control-flow-based adequacy criteria, we first give an introduction to the notion of control flow and control flow graph of a program.

Control flow

The control flow of a program refers to the order in which the individual statements, instructions, or function calls of this program are executed or evaluated (Dahl, Dijkstra, and Hoare 1972). Control flow analysis of programs has long been used in compilers in order to produce optimized code (Allen 1970); it is a indeed a useful tool for implementing program analyses and optimizations such as dead-code elimination, branch prediction, loop transformations, etc. (Muchnick 1997; Allen and Kennedy 2002). A *basic block* is a linear sequence of program instructions having a single entry point (the first instruction executed) and a single exit point (the last instruction executed). A control flow graph is a directed graph in which the nodes represent basic blocks, together with two additional nodes: the "begin" node (the entry block through which control enters into the flow graph) and the "end" node (the exit block through which all control flow leaves) which have no inward, respectively outward edges. There exist an edge between two nodes n_1 and n_2 (representing the basic blocks B_1 and B_2 respectively) if the execution of B_1 can be followed by the execution of B_2 ; each edge $\langle n_1, n_2 \rangle$ is associated with a predicate representing the condition under which the control is transferred from B_1 to B_2. Those conditions are introduced by the conditional control structures of the program, such as an *if-then-else* or a *(while) loop*. Every node in a control flow graph has to be on a path from the begin node to the end node. In an edge $\langle a, b \rangle$, b is said to be a *immediate successor* of a, and a is said to be a *immediate predecessor* of b. A *path* in a directed graph is a directed subgraph expressed as a sequence of nodes $\langle n_1, n_2, ..., n_m \rangle$ where n_{i+1} is an immediate successor of n_i(Berge 1958). For convenience, we will say that a statement *is in* a path if this statement belongs to one of the basic blocks figuring as a node traversed by the path. Similarly, we will say that an edge, denoted by a couple of nodes (n_1, n_2), is in a path if $\langle n_1, n_2 \rangle$ is a subsequence of the sequence of nodes that represents the path.

[1]The IEEE standard 610.12-1990 defines a **fault** as a collection of program source code statements that causes failure whereas an **error** is defined as a mistake made by a programmer during the implementation of a software system (Electrical and (ieee) 1990)

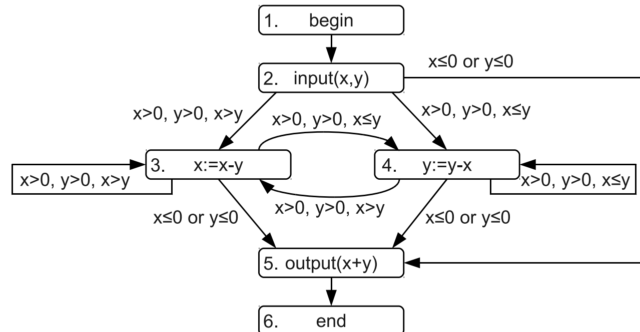


Figure 1.1: Control Flow Graph for the program of Example 1.1

Example 1.1 This example is an extract from (Zhu, Hall, and May 1997). The following (imperative) program computes the greatest common divisor of two natural number by applying Euclid's algorithm.

```
begin
  input(x,y)
  while (x>0 and y>0) do
      if (x>y)
        then x:=x-y
        else y:=y-x
      endif
  endwhile;
  output(x+y);
end
```

In the above, we have 4 basic blocks (plus the begin and end blocks), all of which contain one of the 4 primitive statements figuring in the program. Those are `input(x,y)`, `x:=x-y`, `y:=y-x` and `output(x+y)`. In order to transfer the control from the first one to the second one, the condition of the *while* loop has to be satisfied (i.e. `x>0` and `y>0`) and the condition of the *if-then-else* has to succeed (i.e. `x>y`). The conditions for transfer between other states can easily be deducted similarly. The control flow graph for this program is depicted in Figure 1.1. Note that the control flow graph constitutes a finite representation of all possible program executions. A path through this graph represents one of the possible executions of the corresponding program. For convenience, we will often represent a basic block by its identifier (i.e. a unique natural number associated to each node in the control flow graph) and a path as a sequence of node identifiers. For example, in Figure 1.1, an execution using input values x and y that are both less or equal to zero would be represented by the path $\langle 1, 2, 5, 6 \rangle$.

Path coverage criterion

Any execution of a program can be represented by a path through its control flow graph, beginning with the "begin" node and concluding with the "end" node. The other way round, any path through a program's control flow graph beginning with the "begin" node and finishing with the "end" node represents an execution of this program. In what follows, we will refer to such a path through the control flow graph of a program representing an actual execution of this program as an *execution path* and we will say that the execution *follows* a path in the control flow graph if the sequence of basic blocks in the path equals the sequence of statements executed. The intuitive notion of "fully testing" a program corresponds to designing and executing a test suite such that the program is executed in "all the possible ways". This idea corresponds to the path coverage criterion (Brown and Lipow 1975). First we define the notion of *coverage* as follows:

Definition 1.6 An execution path, respectively a program point is said to be *covered* by a test suite if when this test suite is executed at least one execution follows this path, respectively a path containing that program point. A set of execution paths, respectively program points is covered by a test suite if all the execution paths, respectively program points of this set are covered by this test suite. $\diamond$

The path coverage criterion mentioned above can now be de fined as follows:

Definition 1.7 A test suite satisfies the *path coverage criterion* iff the set P of execution paths representing the execution of each test case comprised in this test suite contains all the execution paths of the program. $\diamond$

If the use of this criterion seems to result in the creation of efficient test suites, it is however inapplicable in practice since there can be an infinite number of paths through a program with loops. We can however define modified versions of the path coverage criterion that can be used in all cases; the resulting criteria will then be satisfied if a well-defined *finite subset* of all execution paths of the program is covered by the test suite. We present here a few examples of such modified versions of the path-coverage criterion.
First, the *length-n path coverage criterion* requires all the execution paths of which the length is less or equal to n to be covered by the test suite (Gourlay 1983). Beside limiting the length of the paths, another simple way of ensuring to consider a finite subset of execution paths is to limit the number of times a loop or a recursive call is executed. There exist a whole class of criteria based on this limitation of the path coverage criterion (called loop count criteria, synthesised (Bently and Miller 1993)), the most well-known of which being the *loop count-K criterion* (Howden 1977). According to the latter, a test suite satisfies the criterion if, given a natural number K, for each loop l in the control flow graph, and for each $0 \leq i \leq K, i \in \mathbb{N}$ such that l can possibly be executed i times, there exists at least one test case in the test suite the execution of which causes l to be

executed i times. Another variant of the path coverage criterion is similar to this one: the *block count-K criterion* which is satisfied if, given a natural number K, all the execution paths traversing each block of the control flow graph less than K times are executed in the test suite (Albert, Gómez-Zamalloa, and Puebla 2009).

Statement coverage criterion

The statement coverage adequacy criterion requires that all the statements in the program are exercised during the test process (Hetzel and Hetzel 1991). It can be defined as follows:

Definition 1.8 A test suite satisfies the *statement coverage criterion* iff for any statement s of the program there exists a path p in the set of execution paths covered by this test suite such that s is in p. $\diamond$

Even though the program is finite, full statement coverage cannot be achieved in all cases because of so-called "unreachable code". A code fragment of a program is unreachable if, in the control flow graph of the program, there is no path from the "begin" node to the node representing the basic block containing this code fragment, such that the conjunction of the predicates labelling the edges along the path is satisfiable (Debray, Evans, Muth, and De Sutter 2000). If a statement is a part of an unreachable code fragment, it can never be executed, and therefore no test suite can be constructed such that all the statements are exercised. We can then modify the criterion such that only *reachable* statements are covered by the test suite; however, determining if a statement is unreachable is known as being an undecidable problem (Weyuker 1982; Kan 2002).

Branch coverage criterion

The branch coverage criterion aims to define whether all control transfers in the program (branches) are tested by a test suite. Since those control transfers correspond to the edges in the control flow graph, this criterion can be defined as follows:

Definition 1.9 A test suite satisfies the *branch coverage criterion* iff for any edge e in the control flow graph of the program there exists a path p in the set of execution paths covered by this test suite such that e is in p. $\diamond$

This criterion is obviously weaker than the path coverage criterion defined earlier; indeed, even if all branches are exercised, that doesn't mean that all the possible combinations of those branches are exercised. It is however stronger than the statement coverage criterion since if all edges of the control flow graph are covered, all nodes are necessarily covered.
The branch coverage criterion is also called decision coverage criterion, because it implies that for every decision in the program both outcomes are covered, i.e. there is at least one test case such that the decision is evaluated to true and at least one test case such that the predicate is evaluated to false.

Condition coverage criterion

A condition is the predicate associated to an edge in the control flow graph of a program – representing the condition from an `if-then-else` or a loop; it consists of a boolean expression typically containing several atomic predicates combined with the logical operators *not*, *and* and *or*. In case of the branch (or decision) coverage criterion it suffices that this boolean expression as a whole is evaluated both to true and false during the execution of a test suite; condition coverage criterion is stronger in the sense that it requires all the atomic predicates to evaluate both to `true` and `false` (Gupta and Jalote 2008; Myers 1979). It can be defined as follows:

Definition 1.10 A test suite satisfies the *condition coverage criterion* iff for any predicate p associated to an edge of the control flow graph containing the atomic predicates $(p_1, p_2, ..., p_n)$, $\forall p_i,\ (1 \leq i \leq n)$ there exists paths π_1, π_2 in the set of execution paths covered by this test suite such that p_i evaluates to $true$ in π_1 and to $false$ in π_2. $\diamond$

Multiple condition coverage criterion

Multiple condition coverage criterion is even stronger than condition coverage; indeed, it requires *all the combinations* of truth value of the atomic predicates to be tested (Zhu, Hall, and May 1997). For example, if the condition $a \wedge b$ appears in the program, a test suite satisfies the condition coverage criterion for this condition if (a, b) evaluates to $(true, true)$ once and $(false, false)$ once (for example) whereas multiple condition coverage requires it to evaluate to $(true, true)$ once, $(false, false)$ once, $(true, false)$ once and $(false, true)$ once.

Definition 1.11 A test suite satisfies the *multiple condition coverage criterion* iff for any predicate p associated to an edge of the control flow graph containing the atomic predicates $(p_1, p_2, ..., p_n)$ in the program, for each combination $(b_1, b_2, ..., b_n)$ of truth values there exists at least one path π in the set of execution paths covered by this test suite such that $(p_1, p_2, ..., p_n)$ evaluates to $(b_1, b_2, ..., b_n)$ in π. $\diamond$

Function coverage, call coverage

Among the (many) other variants of control-flow based adequacy criteria, let us cite the very simple function coverage and call coverage criteria, which require respectively all the functions to be invoked and all the function calls to be executed (Woodward, Hedley, and Hennell 1980).

1.3.2 Data-flow adequacy criteria

Data-flow adequacy criteria examine the life-cycle of data variables. Use of such adequacy criteria leads to test suites concentrating on detecting improper use of data due to coding errors. In order to detect improper use of data, all the

occurrences of the variables are examined during definition (where a value is bound to the variable), predicate use (where the variable is used to determine the truth value of a predicate), computational use (where the variable is used to compute the values of other variables or as an output value) and termination (where the variable is killed).
In order to discover bugs in data usage, test suites are created in such a way that they cover paths that trace each variable definition to each of its uses and every use is traced back to its definition. Various criteria are employed for the creation of such test suites (Rapps and Weyuker 1982; Parrish and Zweben 1995), among which the *all definitions* criterion, the *all uses* criterion and the *all definition-use paths (or chains)* criterion.
A test suite satisfies the *all definitions* criterion – often called all-def criterion in the literature – if for each variable definition in the code, there is at least one test case that will cause the program to follow an execution path containing this definition and traversing at least one use of the variable.
The *all uses* criterion is stronger since it requires, for each variable definition *and* for each use of this variable, that the test suite contains at least one test case that will cause, when executed, the program to follow a path containing the definition and this particular use of the variable. It was introduced for the first time in (Herman 1976). Of course, there exists in general many paths containing both the definition and a particular use of a variable. The *all uses* criterion requires only one of these path to be executed. This requirement could be strengthened in order to get *all* those paths to be executed; unfortunately, there could exist an infinite number of such paths, due to cycles. To avoid that problem, it was proposed to restrict the paths to cycle-free paths (Frankl and Weyuker 1988; Clarke, Podgurski, Richardson, and Zeil 1989). The resulting criterion is called *all definition-use paths* (all DU paths) criterion.

1.4 Fault-based evaluation of a test suite effectiveness

The goal of fault-based test suites evaluation is to measure the ability of a test suite to detect faults that are present in a program. In order to perform this measurement, faults are willingly introduced in the source code. There exist different classes of fault-based adequacy criteria; however we focus on the technique of *mutation testing*, which is the most well-known and the most used one. We refer to (Zhu, Hall, and May 1997) for explanations and details about the other options. Mutation testing originates from another technique called *error seeding* which consists in randomly introducing errors in a program's source code (Meek and Siu 1989). After the testing phase, the ratio between detected and undetected artificial errors is examined. This ratio is supposed to provide an estimation of the ratio of actual errors that the test suite used is able to detect. Of course, this estimation can be trustworthy only if the errors artificially introduced are as difficult to detect as actual errors. In practice,

this is not the case; artificial errors are usually much easier to detect than actual ones (Offutt 1989). This measurement is therefore far from accurate. *Mutation testing* has been defined by DeMillo (DeMillo, Lipton, and Sayward 1978) to overcome these limitations. To perform mutation testing one proceeds as follows: first, a number of "alternative" programs, called *mutants*, are created. Those mutants are obtained by slightly modifying the original program code. The modifications introduced are themselves called *mutations* and are based on *mutation operators* that ideally mimic typical programming errors made by programmers. Each mutant is then tested using the test suite to evaluate, as well as the original program. For each mutant, the following situations can arise: either at least one test case produces a solution that is different for the mutant than for the original program – in that case, one says that the mutant has been "killed" – either all the test cases produced the same result for both the original program and the mutant – in that case one says that the mutant "lives". A mutant can have been left alive for two reasons:

1. The test cases are inadequate. The test suite was not designed well enough to detect the error in the mutant; if a large number of mutants are alive after mutation testing, there is no more reason to be confident in the original program's correctness than in the correctness of the living mutants.

2. The mutant is semantically equivalent to the original program. This should happen only in a small percentage of cases when the introduction of mutants is performed correctly.

The *mutation adequacy score* (MAS) can then be computed. It corresponds to the ration between the number of mutants killed (K) and the number of mutants left alive (A), ignoring the mutants semantically equivalent to the original program (E):

$$MAS = \frac{K}{A - E}$$

More test cases can of course be added in order to kill non-equivalent mutants.

Mutation testing can be considered efficient if we assume the two following statements are true: (1) the programmers are competent. They write programs which are *close to* be correct (DeMillo, Guindi, McCracken, Offutt, and King 1988). That means that if the program is not correct, it differs from a correct program by at most a few small errors. It implies that the mutants to be considered only have to be a slighly modified version of the original program. (2) There exists a so-called "coupling effect"; indeed, mutation testing tests only for simple errors, but if simple errors and complex ones are coupled then test data that kills simple (nonequivalent) mutants will be likely to kill complex mutants as well. Theoretical studies have shown the actual existence of such a coupling effect (Offutt 1989).

1.5 Test data generation: state-of-the-art

The hard part of the testing process is *constructing* a test suite which satisfies the chosen adequacy criteria. This activity can be very time-consuming (and thus very expensive) when performed manually; moreover, the resulting test suites are very large and complex, to such an extent that they can themselves contain errors and can therefore require one to test and correct them (Li and Wu 2004).

A large amount of work exists in the field of automatic test case generation, most of it focusing on imperative programming languages. Interest for this research field began in the 70's; in 1975 and 1976, several papers were published proposing different promising approaches, *symbolic execution* being then (and still) the most used one.

1.5.1 Test data generation using symbolic execution

A programming language is always associated to a semantics describing the objects that program variables may represent, the way the statements provided by the language manipulate those data objects and the control flow of a program written in that language. One can also define a so-called "symbolic execution" semantics, which describes the semantics for the programming language in which (some of) the data objects are replaced by *symbols*, i.e. logical variables representing those data objects through an execution (King 1976). Symbolic execution is a generalization of the normal execution of the program – the latter being a special case in which no value is replaced by a corresponding symbol. In symbolic execution semantics, the usual definitions of the basic operators are modified in order to accept symbolic inputs and produce symbolic formulas as output. The symbolic execution of a program (in which *every* input is replaced by a symbol) following a chosen execution path will result in the production of symbolic formulas – constraints – over input variables; input values satisfying those formulas would cause the program to follow the chosen execution path. For this reason, we call such a symbolic formula associated to an execution path a *path condition*. Let us reconsider Example 1.1 to illustrate these concepts. The algorithm has two input variables. We replace their values by symbolic values, i.e. logical variables that we note in upper case, in order to distinguish them from program's variables.

Example 1.2

```
begin
  x:=X;
  y:=Y;
  while (x>0 and y>0) do
      if (x>y)
        then x:=x-y
        else x:=y-x
      endif
  endwhile;
  output(x+y);
end
```

We perform the symbolic execution of the execution path that traverses the while-loop body twice, and goes through the "then" case the first time and the "else" case the second time. In the control flow graph of Figure 1.1, that corresponds to the path $p = \langle 1, 2, 3, 4, 5, 6 \rangle$. The path condition resulting of this symbolic execution is the following:

$$X > 0 \ \wedge \ Y > 0 \ \wedge \ X > Y \ \wedge \ (X - Y) > 0 \ \wedge \ (X - Y) \leq Y \ \wedge \ (2Y - X) \leq 0$$

One possible solution of this path condition would be $X = 4$, $Y = 2$. Running the program from Example 1.1 with these input values would indeed cause the execution to follow the path p.

By repeating this technique for each path in a set of well-chosen execution paths, one can derive a test suite satisfying different adequacy criteria. All possible symbolic execution paths of a program can be represented under the form of a (possibly infinite) tree. This symbolic execution tree is built recursively from the root – the first statement of the program – to the leaves. Two actions allow one to construct the symbolic execution tree (Lindquist and Jenkins 1988):

- First, when an assignment is encountered, a new node is added to the tree; this node contains a formula describing the new value of the assigned variable, obtained by substituting the variables in the right-hand expression by their current symbolic values. This new node is the root of a sub-tree representing the execution of the rest of the program.

- Second, when an decision is encountered, a new node is created and each branch originating from this node is associated to a decision predicate.

Once the symbolic execution tree is created, each path through it corresponds to an execution path through the program. The conjunction of all the conditions (formulas) encountered along the path is the path condition describing the constraints on the input variables' values that would cause the path to be executed. The symbolic execution tree for Example 1.1 with both input variables replaced by symbols is depicted in Figure 1.2; symbols are in upper case and program's variables in lower case. The path through the symbolic execution tree that corresponds to symbolic execution described in Example 1.2 is drawn in bold lines.

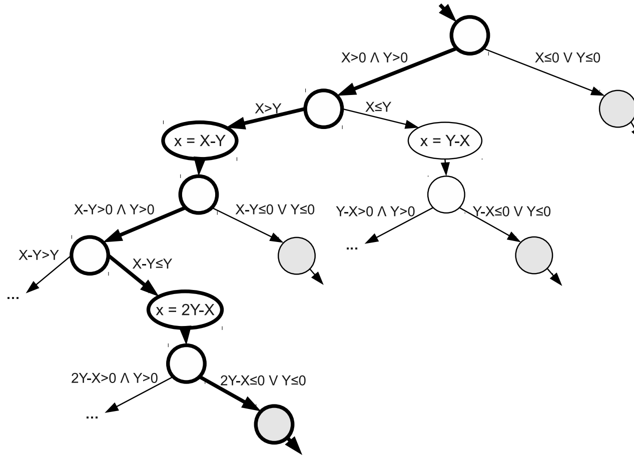


Figure 1.2: Symbolic execution tree for Example 1.1

In (Miller and Melton 1975), Miller and Melton use the symbolic execution technique in order to generate numerical (integer) test data for programs written in Fortran. They consider a single given execution path, derive the associated (numerical) constraints on the input variables and solve them – the solving must be performed manually – to compute the input values that would cause the program to follow the path under consideration. The authors don't have interest in trying to generate sets of data satisfying adequacy criteria, since the latter only began to be developed and theorised at about the same period (Goodenough and Gerhart 1975). Moreover, the technique doesn't handle procedure calls. In (Howden 1975) a very similar method is described, in which the constraints must be solved manually and procedure calls are not handled. However, a technique to select a set of execution paths is defined; it decomposes the set of the execution paths of the program in which the loops can be executed at most once[2] into classes. One path from each class is then selected. Clarke (Clarke 1976) proposed a somewhat more evolved technique for test data generation, which does not only take into account loops and procedure calls, but also solves automatically the constraints obtained from a given execution path thanks to Altran, a former Fortran extension providing rational algebra, and able to solve inequality constraints (Hall 1971). The author also evokes – but does not provide its solution – the problem introduced by the use of arrays when performing symbolic execution; indeed, the fact that the values of variables used as subscripts are not known during the symbolic execution can introduce ambiguous situations. For example, in the following code, if i has the same value as j, the test will fail, otherwise it will succeed.

[2]Note that if this whole subset of paths in which the loops are executed at most once is covered by a test suite, then this test suite satisfies the loop count-1 criterion, as described in Section 1.3

```
    input(i,j)
    a(j) := 2
    a(i) := 0
    a(j) := a(j)+1
if (a(j) == 3)...
```

In (Boyer, Elspas, and Levitt 1975), the authors develop a test case generation technique for (a numerical subset of) LISP in which the solution to this particular problem is to impose additional constraints (hypotheses) to the test path in their system so that both cases ($i = j$ and $i \neq j$) are examined. Ramamoorthy et al. (Ramamoorthy, Ho, and Chen 1976) proposed another approach to solve the problem by duplicating arrays used in such problematic ways. This problem is similar to the problems introduced by the references in programs with pointer-based data structures or by the aliasing in object-oriented programming language; work that has addressed those problems is presented later.

Voges et al. integrated a variant of the algorithm of (Miller and Melton 1975) in a testing tool for Fortran called SADAT (Voges, Gmeiner, and Mayrhauser 1980); few technical details are provided in the paper, but it appears that the test suites generated by the tool satisfy the branch coverage criterion. In (Lindquist and Jenkins 1988) a test case generator for a subset of Ada using symbolic execution is described. It has the ability to deal with procedure calls and assignment statements, but is limited to integer data. It concentrates on generating test suites satisfying a variant of the decision coverage criterion. In symbolic execution, there exist two different approaches to deal with procedure calls. The first one is to continue execution as if the procedure was inlined in the main routine. This method has the important drawback that it causes the procedure to be retested with every call. The approach of (Lindquist and Facemire 1985) is rather bottom-up; test data for the called procedure is first generated, then this data is re-used in the symbolic execution tree of the caller in order to generate its own test data. Therefore, if N different test cases were generated for the called procedure, the branch originating from a node representing a call to this procedure is duplicated N times in the symbolic execution tree.
In (Offutt 1991), an automated test data generator called Godzilla is described. It consists of an integrated collection of at the time existing techniques. The test data generation is limited to numerical (integer) data, and is based on the mutation analysis criterion. First a set of mutants is generated, then a collection of input values is generated in such a way that it causes the mutant program to have a program state that differs from the state of the original program after the execution of the mutated statement. In order to achieve this, the first obvious condition is that the execution of the program using that input *reaches* the mutated statement. Another condition is that once this mutated statement is executed, the mutant program behaves erroneously. Reachability conditions are described by constraint systems called "path expressions" by the author; each statement in the program is associated to a path expression that describes each execution path through the program reaching that statement. Such "path

expression" associated to a statement S represents in fact (a subset of) the formulas obtained by performing (a subset of) all the possible symbolic executions of the program from the beginning up to this statement S. The condition that the test case must cause an erroneous state is described by a constraint that is specific to the type of coding error modelled by each mutation and requires that the computation performed by the mutated statement creates an incorrect intermediate program state. It is interesting that Godzilla is able to generate test suites satisfying statement coverage directly with the path expressions. Test suites satisfying branch coverage can also be generated by cleverly using mutations, i.e. introducing mutations of the program that can only be killed if each condition evaluates first to true, then false. Godzilla is composed of over 15,000 lines of C code and has been integrated in the Mothra Software Test Environment Project (DeMillo and Offutt 1991), a mutation-testing-based testing system for Fortran-77. Experimental results about Godzilla are provided in (DeMillo and Offutt 1993).
Gotlieb et al. introduce in (Gotlieb, Botella, and Rueher 1998) a method to generate test data which cannot be strictly called symbolic execution, but is very similar to it. Given a selected program point (statement or branch), test data that will cause the program to execute that program point when used as input is generated using a two-step method:

1. The program/procedure under concern is statically transformed into a constraint system thanks to the use of the so-called "Single Static Assignment" (SSA) form (Cytron, Ferrante, Rosen, Wegman, and Zadeck 1988) and control dependencies. The result of this step is a set of constraints which is formed of:
    - the constraints generated for the whole procedure;
    - the constraints that are specific to the selected program point
2. The constraint system is solved to check whether at least one feasible path which covers the selected point exists. Finally, test data corresponding to one of these paths is generated.

In that method, the authors take advantage of the constraint programming paradigm that has emerged during the nineties (Hentenryck and Saraswat 1997; Jaffar and Maher 1994) and that allows them to overcome the difficulties encountered earlier in the field; in particular, it allows them to generate test data containing arrays and records, thanks to the use of lists – note that atomic values are still limited to integer values, though (latter, the method has been improved to handle floating-point computations thanks to a dedicated constraint solver using projection functions (Botella, Gotlieb, and Michel 2006)). Besides, their work really concentrates on the constraint solving process and pays little attention to the generation of test suites satisfying adequacy criteria. However, given that the method is based on the coverage of a given program point, it is obvious that one could use it to achieve branch, block or statement coverage criteria. Though it is not clear whether the method would create "minimal" test suites

for those criteria. The authors have developed a prototype implementation called InKa for a subset of the C language (the `array-if-while` language over integers), using the CLP(FD) library of Sicstus Prolog. In (Gotlieb, Botella, and Rueher 2000), they provide experimental results about this prototype, and compare its efficiency (for the generation of test suites satisfying the block coverage criterion) to random test data generation and to Testgen. The latter is a test data generator based on a *dynamic* approach, that is an approach using actual executions of a program under test and numerical optimisation methods. They reach the conclusion that InKa is the only method able to generate test suites within a reasonable time with one hundred percent block coverage.
Sy and Deville focus on the problem of generating test data for programs manipulating boolean, integers and *floating point numbers* (Sy and Deville 2001; Sy and Deville 2003). The latter is particularly challenging, because it introduces a subtle issue. Indeed, there exist constraint solvers able to handle *real* variables (e.g. (Hong 1992; Benhamou and Older 1997)); solving a path condition using such solvers will produce a (small) interval containing its mathematical solution. However, this mathematical solution may involve real values which are not floating point numbers and therefore the resulting test input is not guaranteed to traverse the specified path. In order to solve this problem the authors concentrate on the constraint solving process and particularly on the *consistency technique* used.
In order to explain the notion of consistency technique, we need to introduce the notion of Constraint Satisfaction Problem first. A Constraint Satisfaction Problem (CSP) is a triple (Z, D, C) where Z is a finite set of variables, D is a function which maps every variable in Z to a set of objects of arbitrary type (the *domain* of the variables) and C is a finite (possibly empty) set of constraints on an arbitrary subset of variables in Z. A solution of a CSP is a simultaneous assignment of values (from the respective domains of each variable provided by D) to the whole set of variables Z in such a way that the constraints in C are satisfied (Tsang 1993). A CSP can be solved using a generate-and-test method, i.e. systematically generate each possible value assignment and then test it to check if it satisfies all the constraints. A more efficient method uses backtracking; it incrementally attempts to extend a partial solution towards a complete solution, by repeatedly choosing a value for another variable (Dechter and Frost 1998). However, both of these "search-based" methods have as disadvantage the fact that they detect inconsistencies late in the process. That is why there exist *reduction techniques*, also called *consistency techniques* which aim at reducing the search space by removing from a CSP the assignments of values to variables that cannot be part of a solution (that is domain reduction). In order to speed up the search of a solution, one needs to find a good trade off between the time needed for the problem reduction and the efficiency of the consistency technique used. The most well-known consistency techniques are called node consistency, arc consistency and k-consistency – for further details see (Tsang 1993). In order to solve (continuous) constraints on real variables one uses *interval methods*; the idea is to associate with each variable a domain which is an interval. Consistency techniques especially designed for this purpose are then used to reduce the

intervals without removing solutions for the constraints. A well known example of such a consistency technique is box consistency (Benhamou, McAllester, and Hentenryck 1994; Hentenryck 1997).
In (Sy and Deville 2001) the authors define a new consistency algorithm called *eBox consistency* which generalizes box consistency in order to deal with both integer and float variables at once. The `FindSolutions` algorithm is then defined to search for an (arbitrary) solution satisfying the set of constraints among the variables' domains. CLP using eBox consistency and the `FindSolutions` search algorithm is used to solve a given path condition and therefore create a test input causing this path to be traversed. A method is also proposed that selects a set of execution paths in such a way that all (reachable) statements/branches are executed, in order to create test suites satisfying statements/branches coverage criteria. This method is based on the definition of a Control Dependence Graph. In such a graph a node a is linked to a node b if any execution path reaching b also traverses a.

All the work we have presented up to here does not address some important issues, one of them being the generation of test cases containing complex data structures, possibly involving *aliasing* issues, dynamically allocated and pointer-based data structures. These issues are further detailed in Chapter 6, dedicated to test case generation with pointer-based data structures, using symbolic execution (and constraint logic programming).
As we can notice, symbolic execution is the base of a large number of test data generation techniques; however, a significant scalability challenge for these techniques is the fact that symbolic execution produces an exponential number of execution paths through the code. In order to solve this problem, Cadar and Engler propose a method to prune redundant paths (Cadar and Engler 2008) – two paths are considered redundant if they result in the same program state – by tracking the memory locations read and written by the code, in order to determine if the remainder of a given execution path is able to explore behaviours not encountered yet. This technique is called *read-write set* (RWset) analysis. In the evaluation of their prototype, the authors show that the technique is able to avoid generating from 30% up to nearly 90% of the test data that would be generated by a (naïve) symbolic execution-based test data generation technique in order to satisfy a given coverage criterion.
Another technique that aims to improve the scalability of symbolic execution-based test data generation techniques is called *concolic testing* and has been introduced in (Majumdar and Sen 2007). The idea is to combine random testing and symbolic execution based test data generation in order to partly overcome the limitations of each technique: random testing is fast and scalable but fails at satisfying a given adequacy criterion, while symbolic execution-based testing is just the other way round. Concolic testing is based on concolic execution; the word "concolic" is a compression of "concrete" and "symbolic". In concolic testing, random (concrete) input values are first generated (pointers are assigned to the NULL value), then the algorithm does the following: it performs a concolic

execution of the code with the generated input, i.e. it concretely executes the program and collects at the same time the symbolic path condition along the concrete execution path. At the end of this execution, the path condition is negated and solved with a constraint solver able to generate a new test input, with which the process can be repeated until the algorithm has explored all execution paths up to a given length.

1.5.2 Test data generation using numerical analysis

Test data generation using numerical optimisation techniques was first introduced in (Miller and Spooner 1976). Their work is focused on numerical programs. It requires a part of the input values (the integer values) to be manually computed (e.g. the dimensions of the data in a matrix program or the number of iterations in an iterative method). Once this has been done, an execution path takes the form of a computation containing only assignments of floating-point values and path conditions of the form $c_i(\vec{x}) = 0,\ c_i(\vec{x}) > 0,\ c_i(\vec{x}) \geq 0$, where each c_i is a real-valued function defined in terms of program's input values, represented by the vector $\vec{x}$. Suppose we have k different constraints, sorted in such a way that $c_i(\vec{x}) \geq 0$ for $1 \leq i \leq m$ and $c_i(\vec{x}) = 0$ for $m < i \leq k$. Then, a continuous real-valued function f/m is chosen such that f is (strictly) negative if one of its arguments is (strictly) negative, and (strictly) positive if all its arguments are (strictly) positive. The problem is then to find $\vec{x}$ satisfying $f(c_1(\vec{x}), ..., c_m(\vec{x}))$ and $c_i(\vec{x}) \geq 0$ for $1 \leq i \leq m$. This problem can be solved by choosing a (random) initial vector of input values $\vec{x}_0$ and using an iterative constrained maximization method, described in (Gill and Murray 1974).

The idea introduced by Miller and Spooner has been followed and widely improved by Korel in (Korel 1990; Korel 1992). The method he describes is qualified as "dynamic" since it requires the program to be actually executed in order to produce the test data. The idea is to select an execution path $P = \langle n_1, ..., n_m \rangle$ – where $n_1, ..., n_m$ are nodes of the control flow graph – to be followed through the code. In the context of numerical programs, all the branch predicates along that path are of the form E_1 op E_2, where E_1 and E_2 are arithmetic expressions and op is one of the operators $<, \leq, >, \geq, =, \neq$ and the author assumes that predicates do not contain boolean operators. Those predicates can be transformed into an equivalent predicate of the form F rel 0, where F is called a "branch function" and is either $E_2 - E_1$ or $E_1 - E_2$ and rel is one of $<, \leq$, either F is $abs(E_1 - E_2)$ and rel is one of $=, \neq$. F is real-valued function over program's input values $\vec{x}$ and is positive if the branch predicate is false, negative if it is true. Like in (Miller and Spooner 1976), a first vector of values $\vec{x}_0$ is selected on which the program is executed, traversing an execution path P_1. If $P_1 = P$, $\vec{x}_0$ is the solution; if not, le $S = \langle n_1, ..., n_k \rangle$ be the longest common prefix of P and P_1. It means the wrong branch was chosen on node n_k, and the branch (n_k, n_{k+1}) was not followed. Let us assume that F_k is the branch function corresponding to that last branch; the first sub-goal of the process is now to find $\vec{x}_1$ such that $F_k(\vec{x}_1)$ is negative (or zero) and P_1 is still traversed

by the program using $\vec{x}_1$ as input. This problem is similar to the constrained minimization problem, identical to the maximization problem described in (Gill and Murray 1974) and used in (Miller and Spooner 1976). This process can then be repeated with $\vec{x}_1$ as initial input vector until P is entirely traversed, or one of the sub-goals cannot be solved, in which case this procedure fails. The author provides advanced search procedures for constrained minimization problems, the details of which are out of scope of our current work.
It is however interesting to note that a method to generate dynamic data structures (records and pointers) is proposed. Every record is simply treated as a separate variable; in order to achieve this, a list of dynamic records is created and manipulated during the execution of the program. Each record is associated to a unique name, and each field can then be accessed with record_ name.field_ name. The approach to deal with pointers is based on backtracking. The goal of finding an input data structure to traverse a selected execution path is achieved, as earlier, by solving sub-goals. The method starts with an arbitrary input data structure with the fields initialized with arbitrarily chosen values. If the wrong branch is chosen at some node during the execution, there are two possibilities: either the branch was chosen because of a wrong arithmetic value, in which case the method described earlier is applied. Or the branch was chosen because of a wrong *shape* of the data structure. In that case, the method determines those input pointer variables that influence the choice, and tries to systematically assign other values for those variables. If no new assignment can cause the execution path to follow the selected branch, then a new solution is sought for the previous sub-goal – the search procedure backtracks in order to assign new values to the input pointers that prevent the selected branch to be followed. This method is repeated until the solution of the main goal is found or no solution of the sub-goals can be computed. A criticism of the efficiency of this method can be found in (Visvanathan and Gupta 2002), based on the fact it tries to generate the shape and the arithmetic values in a single process. It means that if backtracking is performed because an incorrect choice about the shape was made at some point, the values in the data structure that were generated subsequently to this incorrect choice become useless. In addition, the backtracking can be extensive in the presence of pointer aliasing.
This latter method has been refined to create the *chaining approach* described in (Ferguson and Korel 1995; Ferguson and Korel 1996; Korel 1996). This method is based on the idea that selecting a given execution path and then try to generate the input value that would cause the program to follow that exact path can be a weakness of the method. Indeed, it is not possible to know in advance if the selected path is feasible or not; according to the authors, it happens really often that an infeasible path is selected and then significant computational effort is wasted in analysing it. Instead of choosing a path to be executed, the authors propose to choose a *goal* to be executed – i.e. a node or a branch in the program – irrespectively of the path taken to reach it. Then a sequence of "essential" nodes $\langle n_l, n_2, ..., n_m \rangle$ to be executed prior to execution of the goal g is identified thanks to the control flow graph and a data dependency analysis. Similarly to the method explained above, a first execution using an

arbitrary program input $\vec{x}_0$ is performed; for each executed branch (p, q), a search process decides whether the execution should continue along this branch or if another branch should be taken – because the current branch is not likely to lead the execution to the goal g. In the latter case, the execution is suspended and a new program input $\vec{x}_1$ is computed to change the flow at this branch. If the search process fails to determine such a new input, a new sequence of "essential" nodes in the program is identified by using data dependence concepts and requiring that these nodes are executed before reaching branch (p, q). Note that the search process used in the chaining approach needs to determine path conditions for each sub-path through the essential nodes, and uses therefore symbolic execution in the process.
The work presented by Gupta in (Gupta, Mathur, and Soffa 1998; Gupta, Mathur, and Soffa 2000) is very similar to Korel's work. It is based on a numerical analysis technique called *relaxation technique* for iteratively refining a randomly chosen input –relaxation techniques are usually used to improve upon an approximate solution to an equation representing the roots of a function (Scheid 1968). Similarly to (Korel 1990; Korel 1992), an execution path to follow is first selected. If the program does not follow the selected path when executed with a randomly chosen input, the method attempts to modify the current input in such a way that *all* the branch predicates on the path evaluate to the desired outcome when the program is executed with the new input. According to the authors, if all the branch conditions on the path are linear functions on the input values then the method is able to derive the desired input values in a single iteration or guarantees that the path is infeasible. However, if at least one path condition is a non-linear function on the input then several iterations could be necessary. This ability to generate the input values for a given path in a single iteration makes that approach much more scalable than the one proposed in (Korel 1990; Korel 1992), which considers one branch predicate at a time, in particular when branch conditions are linear functions on the input. This technique has been used in other work, notably (Shan, Wang, and Qi 2001).

1.5.3 Test data generation using the specifications of the program

In this section we briefly present the test case generation techniques based on the specifications of the program in a broad sense; a specification can either be a textual specification (possibly written using a specification language), or a (graphical) model of the structure such as UML diagrams, etc. ATLAS (Jessop, Kane, Roy, and Scanlon 1976) is a system that generates test cases based on a directed graph model of the software under test, describing the sequential behaviour of the software system and its inner components. This graph is a kind of simplified control-flow graph of the system, in which the nodes are the components and the arcs are labelled with assertions describing the input/output behaviour of the transitions; the graph can possibly be *constrained*, which means that certain paths through the graph don't correspond to a real execution

of the program – paths corresponding to a real execution are called *admissible paths*. The system searches for all admissible paths, and collects the assertions along each of them in order to derive test data.
In (Gargantini and Heitmeyer 1999) a method for constructing test suites based on the SCR (Software Cost Reduction) requirements is presented. The goal of the SCR requirements specification is to describe both the behaviour of the system (which is usually deterministic) and the system environment (which is non-deterministic) (Hager 1989). The system is modelled as a state machine, which begins execution in an initial state and then changes state and possibly produces output events when an input event occurs. These state transitions are represented by functions mapping an input event and a state to a new state; the method collects a finite set of transitions sequences from the initial state to a final state a derives a predicate for each of them, based on the transition functions. The method then derives test suites by using a model checker.
The Korat test generator (Boyapati, Khurshid, and Marinov 2002) translates method specifications into Java predicates; any specification language can be used as long as it can be translated into Java predicates – however the prototype presented supports only JML (Java Modelling Language) as specification language. Based on the method precondition, Korat generates all the non-isomorphic possible input values up to a given (small) size. Then the method is executed using each input, and each corresponding output is checked against the method postcondition.
There exists an amount of work about test case generation based on UML diagrams (Linzhang, Jiesong, Xiaofeng, Jun, Xuandong, and Guoliang 2004; Samuel, Mall, and Kanth 2007; Kim, Kang, Baik, and Ko 2007). Note however that in this work, "test case" refers to a "test scenario", a sequence of interactions between the user and the system rather than a combination of concrete input values and expected output. These techniques are therefore difficult to compare with the work we have presented until now.

1.5.4 Other test data generation techniques

The easiest way of generating test cases is obviously to generate them randomly, i.e. choosing input data randomly based on some input distribution; it is also often seen as the worst way of doing it from the efficiency or a coverage point of view. However it can sometimes be a cost-effective testing technique for some classes of programs (Duran and Ntafos 1981; Bird and Munoz 1983). It is notably used with success in some real-life tools such as QuickCheck (Claessen and Hughes 2000); this tool aims at automatically test Haskell programs. The authors of the tool also defined a specification language for Haskell. Quickcheck can therefore generate a random input, execute the function with that input and check the result with respect to the specification without any user intervention. Note however that writing a (correct) formal specification is not always easy to do, particularly for large programs. Besides, there exists work based on random testing that tries to overcome random testing limitations. That is for example the case of *antirandom testing* (Malaiya and Malaiya 1996; Yin, Lebne-Dengel,

and Malaiya 1997). The basic idea is that a generated test input should depend on the test inputs generated before. An *antirandom test sequence* is then defined as "a test sequence such that a test t_i is chosen such that it satisfies some criterion with respect to all tests $t_o, t_1, ...t_{i-1}$ generated before". The choice of the criterion is a research subject itself: in (Wu, Jandhyala, Malaiya, and Jayasumana 2008) each (numerical) test is chosen such that its total *distance* from all previous tests is maximal. Assuming that a test input is a vector, two notions of distance are taken into account: the Hamming distance, which is the number of bits in which two binary vector differ – this distance is not defined for continuous values – and the usual Cartesian distance.
Another variant of random testing is described in (Gotlieb and Petit 2006) and mixes random testing with symbolic execution. In this paper, the goal is to generate random test data based on a uniform distribution for a subset of execution paths. This approach is called Path-oriented Random Testing and uses symbolic execution to derive the path conditions for each path of the selected set. The uniform random test data generator is then built in such a way that it minimizes the number of randomly generated test data satisfying no path condition. To achieve that, the approach combines constraint propagation with random test data generation.
Search-based test data generation is one approach that has attracted recent interest (Alshraideh, Bottaci, and Mahafzah 2010). This approach is based on the definition of an evaluation or cost function that is able to discriminate between candidate test cases with respect to achieving a given test goal. The cost function is implemented by appropriate instrumentation of the program under test. The candidate test is then executed on the instrumented program. This provides an evaluation of the candidate test in terms of the distance between the computation achieved by the candidate test and the computation required to achieve the test goal. Providing the cost function is able to discriminate reliably between candidate tests that are close or far from covering the test goal and the goal is feasible, a search process is able to converge to a solution, i.e., a test case that satisfies the coverage goal. For some programs, however, an informative cost function is difficult to define. The operations performed by these programs are such that the cost function returns a constant value for a very wide range of inputs. A typical example of this problem arises in the instrumentation of branch predicates that depend on the value of a Boolean-valued (flag) variable although the problem is not limited to programs that contain flag variables.

1.5.5 Structure of the work

The current work is structured a follows: Chapter 2 introduces the reader to a number of notions necessary for the comprehension of the next chapters, such as different aspects of logic programming and some notable particularities of the Mercury programming language as well as its semantics. In Chapter 3, we present a test automation framework for Mercury, i.e. a tool able to automatically execute each test case of a test suite and produce a report about which test cases failed and why. This chapter adapts notions used in the con-

text of testing techniques for imperative programming languages and presents the methods used in the implementation of our tool to handle the particular features of Mercury. The work presented in this chapter has been published in (Biener, Degrave, and Vanhoof 2010).
In Chapters 4 and 5, we present work that has been published in (Degrave and Vanhoof 2007a), (Degrave 2008) and (Degrave, Schrijvers, and Vanhoof 2008). In Chapter 4, we first define a control flow graph for a Mercury program, and show how one can use it to symbolically execute this program. We also use it to adapt test coverage criteria existing for imperative programming languages to the context of Mercury, and finally we show how we enhanced our test framework for Mercury with a complementary module that computes the coverage rate with respect to some of the latter coverage criteria. We define in Chapter 5 how to represent the symbolic execution of a Mercury program program under the form of sets of constraints containing both numeric and symbolic data and explain how we used the CHR language to define a constraints solver able to deal with such data.
We explore in Chapter 6 a similar approach to the one presented in the preceding chapters in order to generate test suites for a pointer-based imperative language; this research was published in (Degrave, Schrijvers, and Vanhoof 2009).
Finally, we deviate slightly from our main topic in Chapter 7 as we present researches published in (Degrave and Vanhoof 2007b) and (Vanhoof and Degrave 2008), the goals of which are to study the conditions under which two (fragments of) logic programs can be considered equivalent, and detect program fragments that are susceptible for refactoring, aiming in particular to the removal of duplicated code or to the generalisation of two related predicates into a new (higher-order) one.

Chapter 2

Technical background

2.1 The Mercury language

Mercury is a programming language that was designed and implemented in 1993 in Australia by researchers of the university of Melbourne. It is based on the purely declarative programming paradigm and was conceived in order to create large and reliable software. If it can be categorized as a logic programming language – since it uses the traditional execution model for this family of languages – it also allows the user to define functions, as functional programming languages do. The advantages brought by pure declarative programming languages compared to the imperative languages, summarized in (Somogyi, Henderson, and Conway 1995) for example, are well-known: they provide a higher level of expressivity (the programmer *declares* the properties of the system – *what* the system should do – rather than describing the operations to perform – *how* the system should work), they have much more useful formal semantics than imperative languages (which makes the development of automatic analyses and transformations much simpler and more effective), their semantics is independent of any order of evaluation (which makes it much easier for a compiler to parallelise the code) and can potentially be used with declarative debuggers that make debugging easier (Lloyd 1987a; Maclarty 2005).
The goal of the developers of Mercury was to create a *successor* for Prolog; indeed, the latter uses impure features that destroy the possibility to exploit all the advantages cited above. The qualities that the developers wanted this successor to have are the following (Somogyi, Henderson, and Conway 1995):

- **Support for the creation of reliable software**. The language should provide mechanisms to prevent some classes of bugs at compile time.
- **Support for the creation of efficient programs**. Programs written in that language should be at least as fast as if they were written in an alternative language.
- **Support for programming in teams**. This feature requires the lan-

guage to support modularity and information hiding in order to allow the programmers to effectively isolate themselves from the effects of the changes made by the other programmers.

- **Support for program maintenance**. Programs written in the language need to be easily readable and understandable.

- **Support for accessing external databases**.

In order to satisfy these requirements, the researchers from Melbourne decided Mercury had to be a purely declarative language with a strong type-, mode- and determinism declarations system. Those declarations are an excellent indication on how and with what kind of data the predicate should be used, and also allow the compiler to perform analyses that detect certain classes of bugs at compile time. Besides, they provide the basis for an efficient execution mechanism of the language (Somogyi, Henderson, and Conway 1994; Conway, Henderson, and Somogyi 1995; Somogyi, Henderson, and Conway 1996). Mercury supports for higher order programming and is equipped with a modern module system that enables to hide some data definitions and to encapsulate both data and code, and provides as such support for programming-in-the-large activities. We will now present an overview of the general characteristics of the logic programming paradigm, and then take a closer look at the different Mercury declarations and other particularities of the language.

2.1.1 Overview of logic programming

In this section, we present the basics of logic programming; we refer to (Lloyd 1987b; Apt 1990) for further details. Logic programming languages contain *variables*, *function symbols* (functors) and *predicate symbols*. The sets of variables, function symbols and predicate symbols are denoted in this work by $\mathcal{V}$, Σ and Π respectively. Function and predicate symbols have an associated arity, that is a natural number indicating the number of arguments for this function or predicate symbol. A function symbol with no argument is often called a "constant". In this work we use the wide-spread convention of denoting variables by uppercase letters whereas function symbols and predicate symbols are denoted by lowercase letters. Sometimes, we will denote the arity of a function or predicate symbol f using the notation f/n where $n \in \mathbb{N}$. A *term* is defined as a construction using elements from $\mathcal{V}$ and Σ and is either a variable (from $\mathcal{V}$) or a function symbol $f/n \in \Sigma$ applied to a sequence of n terms. We denote the set of all such terms by $\mathcal{T}(\mathcal{V}, \Sigma)$. We name terms using again lowercase letters, and we define $\mathcal{V}(t)$ as the set of variables occurring in t. An *atom* is a predicate symbol $p/n \in \Pi$ applied to a sequence of n terms, and a *literal* is either an atom or the negation of an atom. The latter is denoted by an atom preceded by $\neg$. A *clause* is an implication of a head from a body

$$H \leftarrow B_1, \ldots, B_n, n \geq 0$$

where H (the head) is an atom and $B_1, \ldots, B_n$ (the body) is a (possibly empty) sequence of literals. If the body is empty, the clause is called a *fact*. A *program* is constituted by a set of clauses. A *query* is a clause, the head of which is empty. A query is thus of the form

$$\leftarrow B_1, \ldots, B_n, n \geq 1.$$

In what follows, we will use "expression" to denote any object that is a term, an atom, a literal, a clause or a query. Expressions that do not contain any variables are said to be *ground*. If the body of a clause or a query contains only positive literals – that is literals that are not negated atoms –, it is called a definite clause. If all the clauses of a program are definite, the program is called a definite program.
A *substitution* σ is a finite mapping from variables to terms, represented as a finite set of pairs $(v, t) \in \mathcal{V} \times \mathcal{T}(\mathcal{V}, \Sigma)$. Each pair is noted v/t, that is $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ such that:

1. $\forall i, j : i = j \Rightarrow X_i = X_j$ and
2. $\forall i : X_i \neq t_i$

The first condition states that all the variables in the first member of the pairs must be distinct, and the second condition states that a variable cannot be mapped to itself. The *domain* of a substitution $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ is the set of variables defined as

$$dom(\sigma) = \{X_1, \ldots, X_n\}$$

whereas its *codomain* is the set of variables defined as

$$codom(\sigma) = \bigcup_{k=1}^{n} \mathcal{V}(t_k)$$

A *ground substitution* is a substitution the codomain of which is empty, i.e. a substitution that maps variables to ground terms. If E is an expression and σ a substitution, than $E\sigma$ denotes the result of applying σ to E and is defined as the expression obtained from E by simultaneously replacing the variables from the domain of σ that occur in E by their corresponding term in σ. We call $E\sigma$ an instance of E. If F is an instance of the expression E, then E is said to be *more general* than F, denoted $F \leq E$. If F is an instance of the expression E, and E is an instance of the expression F, then E and F are called *variants*, denoted by $F \approx E$. If $F \leq E$ and $E \not\approx F$, we say that E is *strictly more general* than F, denoted with $F < E$. From two substitutions $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ and $\theta = \{Y_1/s_1, \ldots, Y_m/s_m\}$, the *composition* of these substitutions, denoted $\sigma\theta$, is defined to be the substitution:

$$\begin{gathered}\{X_i/t_i\theta \mid 1 \leq i \leq n \wedge t_i\theta \neq X_i\} \\ \cup \\ \{Y_i/s_i \mid 1 \leq i \leq m \wedge Y_i \notin \{X_1, \ldots, X_n\}\}\end{gathered}$$

If τ and ρ are substitutions, and there exists a substitution σ such that $\tau = \rho\sigma$, τ is said to be *more precise* than ρ and ρ is said to be *more general* than τ, denoted $\tau \leq \rho$. A *unifier* for two expressions E and F is a substitution σ verifying the property $E\sigma = F\sigma$. Among all the unifiers for E and F, we call the *most general unifier* (denoted $\text{mgu}(\{E, F\})$) any substitution σ such that

1. $E\sigma = F\sigma$ and
2. $\forall \rho$ such that $E\rho = F\rho$, we have $\exists \rho' : \rho = \sigma\rho'$

The most general unifiers of a set S are unique modulo variable renaming; hence we often refer to *the* most general unifier of a set S of expressions. We will sometimes refer to the most general unifier simply by "mgu" as is common practice.
We now present the usual execution model for logic programs under the form of a procedural semantics; for clarity and concision reasons, we restrict our attention to definite logic programs. The procedural semantics model is very well-known and is the most commonly used in logic programming; it is called the *SLD-resolution* (SLD stands for *Selective Linear Definite*). We now define the basic notions related to SLD-resolution; these definitions can be found in (Lloyd 1987b; Apt 1990).

Definition 2.1 Let Q be the query $\leftarrow A_1, \ldots, A_k, \ldots, A_n$ and C be the clause $A \leftarrow B_1, \ldots, B_q$. Then Q' is *derived* from Q and C using the most general unifier θ if the following conditions hold:

1. A_k is an atom, called the *selected* atom in Q
2. θ is a most general unifier of A_k and A
3. Q' is the query $\leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_q, A_{k+1}, \ldots, A_n)\theta$.

$\diamond$

Definition 2.2 Let P be a definite program and Q_0 a definite query. An SLD-derivation of $P \cup \{Q_0\}$ consists of a possibly infinite sequence $Q_0, Q_1, Q_2 \ldots$ of queries, a sequence of renamed apart variants of program clauses $C_1, C_2, \ldots$ of P and a sequence $\theta_1, \theta_2, \ldots$ of most general unifiers such that each Q_{i+1} is derived from Q_i and C_{i+1} using θ_{i+1}. $\diamond$

An SLD-derivation can be finite or infinite. If a finite SLD-derivation ends in the empty query, this SLD-derivation is called a *successful* derivation, or an *SLD-refutation*. If a finite SLD-derivation ends in a query of which the selected atom does not unify with any of the heads of any clause in the program, this SLD-derivation is said to *fail*. The execution mechanism of logic programs consists in constructing SLD-derivations for a query and a program. When a finite SLD-derivation succeeds, one is interested in what is actually "computed" by the derivation. This is formally defined by the concept of a computed answer (substitution).

Definition 2.3 Let P be a definite program and Q_0 a definite query. A *computed answer (substitution)* θ for $P \cup \{Q_0\}$ is the substitution obtained by restricting the composition $\theta_1 \dots \theta_n$ – being the sequence of most general unifiers used in an SLD-refutation of $P \cup \{Q_0\}$ – to the variables of Q_0. $\diamond$

Of particular interest is the fact that one has to select a particular atom in the query in order to continue the execution and, if the selected atom unifies with more than one clause in the program, to select one of these clauses. Because of this selection process, based on a selection rule, SLD resolution implicitly defines a search tree of alternative computations. Such a tree is called a *SLD-tree*.

Definition 2.4 Let P be a definite program and Q a definite goal. An SLD-tree for $P \cup \{Q\}$ is a tree in which each node of the tree is a possibly empty definite query, the root node is the query Q and for each node $\leftarrow A_1, \dots, A_k, \dots, A_n$ ($n \geq 1$) we have the following: if A_k is the selected atom, then for each variant of a clause $A \leftarrow B_1, \dots, B_m$ in P such that A_k and A are unifiable with most general unifier θ, the node has a child of the form

$$\leftarrow (A_1, \dots, A_{k-1}, B_1, \dots, B_m, A_{k+1}, \dots, A_n)\theta.$$

$\diamond$

Each branch in an SLD-tree is a SLD-derivation; such a branch can possibly be an infinite derivation. If a tree contains one or more such infinite derivation, the tree is called *infinite*, otherwise it is *finite*. A leaf node (i.e. a node which has no children) is called a *success node* if the query it is associated to is empty. It is a *failure node* if the associated query is non-empty and its selected atom unifies with the head of no clause in the program. A branch ending with a failure node is a *failing branch*. If all branches of a finite SLD-tree are failing, the tree is called a *finitely failing SLD-tree*.

Example 2.1 Let us examine the following definite program P (from (Lloyd 1987b)):

$$\begin{array}{l} p(X, X) \leftarrow \\ p(X, Y) \leftarrow q(X, Z), p(Z, Y) \\ q(a, b) \leftarrow \end{array}$$

Using the selection rule that always selects the leftmost atom, the resulting SLD-tree for $P \cup \{\leftarrow p(X, b)\}$ is depicted in Fig. 2.1. Branches are annotated with the necessary substitutions to allow the reconstruction of computed answer substitutions.

2.1.2 Mercury's Type System

Mercury's type system is based on a polymorphic many-sorted logic, and corresponds to the Mycroft-O'Keefe type system (Mycroft and O'Keefe 1984), which has the same basis as the type system of Haskell (Leivant 1983). The idea is

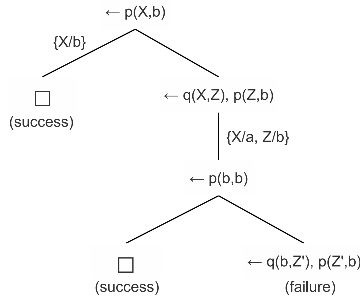


Figure 2.1: An SLD-tree.

to base type definitions on discriminated unions to support parametric polymorphism. Therefore, a type is defined by giving the set of function symbols to which variables of this type can be bound, and the type of each argument for those functors (Somogyi, Henderson, and Conway 1996). For example, the definition of the well-known $list(T)$ type is the following:

Example 2.2

```
:- type list(T) ---> [] ; [T|list(T)].
```

According to this example, if T is a type representing a given set of terms, values of type `list(T)` are either the empty list `[]` or a term $[t_1|t_2]$ where t_1 is of type T and t_2 of type $list(T)$. The type T is called a *type variable*. Formally, a user-defined type is constituted of terms constructed from two sets: a set of type constructors denoted $\Sigma_{\mathcal{T}}$ and a set of type variables of a language $\mathcal{L}$, denoted $V_{\mathcal{T}}$. The set of types associated to $\mathcal{L}$ is defined using the terms from the set represented by $\mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$; that is the set of terms that can be constructed from $\Sigma_{\mathcal{T}}$ and $V_{\mathcal{T}}$. A *type definition* is constituted of *type rules*, one for each type constructor. The previous example (Example 2.2) shows the type rule for the `list/1` type constructor. Formally, a type rule is defined as follows:

Definition 2.5 The *type rule* associated to a type constructor $h/n \in \Sigma_{\mathcal{T}}$ has the form

$$h(\overline{T}) \rightarrow f_1(\bar{t}_1) \; ; \; \ldots \; ; \; f_k(\bar{t}_k).$$

where $\overline{T}$ is a sequence of n type variables from $V_{\mathcal{T}}$ and for $1 \leq i \leq k$, $f_i/m \in \Sigma$ with $\bar{t}_i$ a sequence of m types from $\mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ and all of the type variables occurring in the right hand side occur in the left hand side as well. The function symbols $\{f_1, \ldots, f_k\}$ are said to be associated with the type constructor h. A finite set of type rules is called a type definition. $\diamond$

In addition to these so-called *algebraic types*, Mercury defines a number of primitive types that are builtin in the system. Among these are the *numeric types* `int` (integers) and `float` (floating point numbers). In order to deal with polymorphic types – type containing type variables – the Mercury compiler uses *type*

substitutions, which are substitutions from type variables to other types (builtin, monomorphic – type containing no variable – or even polymorphic types). The result of applying such a substitution to a type is the creation of a new type, an *instance* of the original polymorphic type.

Example 2.3 If `list/1` and `int/0` are type constructors from $\Sigma_{\mathcal{T}}$, `list(T)` is a polymorphic type. If we apply the type substitution that maps the type variable `T` to the type `int` onto this type, we get the monomorphic type `list(int)`.
It is also possible to apply the type substitution that maps the type variable `T` to the type `list(T)` onto this type, we get the new polymorphic type `list(list(T))`.

A type t is said to be *atomic* if it is not defined in terms of other types (t is builtin or the type rule defining t's type constructor uses only function symbols of which the arity is 0).
The Mercury type declaration system requires the programmer to declare the type for each argument of each predicate of the program.

Example 2.4 Consider a predicate `append/3` declared as follows:

```
:- pred append(list(T), list(T), list(T)).
```

According to this declaration, each argument of the predicate is of type `list(T)`.

From this information the compiler infers the type of every local variable and verifies that the program is well-typed (or *type-correct*) (Mycroft and O'Keefe 1984; Pfenning 1992). For example, all the arguments of every predicate call in a type-correct program are instances of the types declared for the predicate. For example, the arguments used when the `append/3` predicate (defined above) is called must be instances of the `list(T)` type, such as `list(int)`, `list(list(int))`, `list(list(T))` or simply `list(T)` itself.

2.1.3 Mercury's Mode and Determinism System

The Mercury *mode system* describes how the instantiation of a variable changes over the execution of a goal. Each predicate is associated to one or more mode(s); each such mode maps for each argument an initial instantiatedness (describing the state of the argument at the time the predicate is called) onto a final instantiatedness (describing the state of the argument at the time the predicate exits). It is the responsability of the caller to achieve the initial instantiation state, and the responsibility of the predicate to achieve the final state. According to (Somogyi, Henderson, and Conway 1995) the vast majority of predicates in real-world programs (can) have modes defined using the two most basic notions of instantiatedness: the *input* mode (denoted by `in`), stating that the corresponding term has a ground instantiatedness before and after a call) and the *output* mode (denoted by `in`), stating that the argument is a free variable at the time of the call that will be instantiated to a ground term when the predicate exits). However, Mercury allows one to define new instantiation

states, and thus new modes. For now we assume that the modes are restricted to `in` and `out` modes.

Example 2.5 One of the most used modes for the `append/3` predicate is declared as follows:

```
:- mode append(in,in,out).
```

This mode corresponds to a usage of the predicate `append/3` such that when it is called, its first two arguments are ground whereas the third one is a free variable that will be bound to a ground term when the predicate succeeds.

Mercury allows the programmer to declare more than one mode for a predicate; each mode can have its own implementation and represents a particular usage of the predicate. Based on the mode declaration, the compiler infers the (declared) instantiation of the variables in the different subgoals of a predicate. The compiler therefore checks if each predicate is *well-moded*. Intuitively, this means that the goals in the predicate's body can be reordered in such a way that values are produced (mapped from free to ground) before they are consumed (mapped from ground to ground) when the predicate is executed by a left-to-right selection rule (Overton, Somogyi, and Stuckey 2002a). In order to guarantee well-modedness, the compiler duplicates each predicate as many times as there are mode declarations for that predicate. The body of each resulting copy of the predicate is reordered in such a way that it becomes well-moded with respect to the corresponding mode declaration. Each reordered predicate is called a *procedure* in Mercury terminology.
Finally, a *determinism declaration* is associated with each procedure – that is, with each mode of a predicate. This determinism declaration indicates whether the corresponding procedure may produce one ore more answers and whether it can fail or not. The different existing determinisms and corresponding meanings are the following:

- A *deterministic* procedure (noted `det` in Mercury) succeeds each time it is called and produces exactly one solution.
- A *semi-deterministic* procedure (noted `semidet` in Mercury) can fail or succeed with exactly one solution when it is called.
- A *multisolution* procedure (noted `multi` in Mercury) succeeds each time it is called and produces at least one (but possibly more) solutions.
- A *nondeterministic* procedure (noted `nondet` in Mercury) can fail or produce at least one (but possibly more) solutions when it is called.
- A *failure* procedure (noted `failure` in Mercury) cannot succeed but may fail when it is called.
- An *erroneous* procedure (noted `erroneous` in Mercury) cannot succeed nor fail when it is called (it either loops forever or aborts execution).

The two last determinisms are rarely used in Mercury and are of little interest; they won't be considered further.

Example 2.6 Let us consider the `append/3` predicate again. We provide two mode declarations for this predicate, and each mode corresponds to a determinism:

```
:- pred append (list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
append([], Y, Y).
append([E|Es], Y ,[E|Zs]):- append(Es, Y, Zs).
```

This predicate basically implements a ternary relation in which one argument is the result of concatenating both other arguments. For `append(in, in, out)` (presented in Example 2.5) a call to this procedure is deterministic, meaning that it will succeed exactly once. In the `append(out, out, in)` mode, the third argument is input (ground) and the first two are output (free variables) in which case a call to this procedure may generate multiple solutions. Note that no call to `append/3` in either of these modes can fail.

Example 2.7 We provide as additional example the well-known `member/3` predicate, with two mode declarations:

```
:- pred member(T, list(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.
member(X, [X|_]).
member(X, [Y|T]) :- not (X=Y), member(X, T).
```

This predicate implements a binary relation in which one argument is a list and the other one is an element of this list. In `member(in, in)` both arguments are input and a call to this procedure will either succeed once or fail, in `member(out, in)` only the second argument is input, in which case a call to this procedure can fail, or generate one or more solutions.

2.1.4 Mercury superhomogeneous form

In this work, we restrict attention to first-order Mercury programs on which no module structure is imposed. We will also consider that the Mercury program is in *superhomogeneous form*. The translation of a Mercury program into superhomogeneous form comprises different analysis and transformation steps, including the translation of n-ary function into $n + 1$-ary predicates in which the return value is transformed into an additional argument. Indeed, Mercury uses functions as syntactic sugar for predicates with a single output argument (Somogyi, Henderson, Conway, Bromage, Dowd, Jeffery, Ross, Schachte, and Taylor 1996). Also, each multi-moded predicate is transformed into different procedures, one for each mode. Each resulting procedure is well-typed and

well-moded – all programs that are not well-moded or well-typed are rejected by the compiler (Somogyi, Henderson, and Conway 1996).
Formally, we define the syntax of Mercury programs in superhomogeneous form as follows. In what follows, we use symbol Π to refer, in Mercury context, to the set of *procedure* symbols (rather than predicate symbols) underlying the language associated to the program. As such, we consider two procedures that are derived from the same predicate as having different procedure symbols.

Definition 2.6

$$
\begin{array}{llll}
\textit{Proc} & & ::= & p(X_1,\ldots,X_k)\ \texttt{:-}\ C. \\
\textit{Conj} & C & ::= & G \mid G\texttt{,}C \\
\textit{Disj} & D & ::= & C\texttt{;}C' \mid D\texttt{;}C \\
\textit{Goal} & G & ::= & A \mid D \mid \texttt{not}(C) \\
\textit{Atom} & A & ::= & X\texttt{==}Y \mid X \Rightarrow f(Y_1,...,Y_n) \mid X \Leftarrow f(Y_1,...,Y_n) \\
 & & \mid & Z\texttt{:=}X \mid p(X_1,...,X_n)
\end{array}
$$

where $p/n \in \Pi$, $X_1,\ldots,X_k \in \mathcal{V}$, $f/n \in \Sigma$ and $Y_1,\ldots,Y_n \in \mathcal{V}$. $\diamond$

The definition of a procedure p in superhomogeneous form consists of a single clause in which the arguments in the head of the clause (denoted $\mathcal{A}rgs(p)$) and in procedure calls in the body are all distinct variables. Explicit unifications are generated for these variables in the body, and complex unifications are broken down into simple ones. The body of a procedure is a conjunction of goals (possibly containing a single goal). A goal is either an atom, a disjunction or a negated conjunction of goals (*not*). An atom is a unification or a procedure call. Moreover, using mode information each unification is classified as either:

- A *test* between two atomic (ground) values (X==Y). In that case, both X and Y are input to the unification.
- An *assignment* between two variables $X := Y$. The variable Y is input, whereas X is output.
- A *deconstruction* denoted $X \Rightarrow f(Y_1,...,Y_n)$ in which X is input to the unification and $Y_1,\ldots,Y_n$ are output variables.
- A *construction* of the form $X \Leftarrow f(\overline{Y})$ in which X is output of the unification and $Y_1,\ldots,Y_n$ are input variables

Example 2.8 Consider the definition of the `append/3` predicate, both in normal syntax and in superhomogeneous form for the mode `append(in,in,out)` as depicted in Fig. 2.2. The `append(in,in,out)` procedure in superhomogeneous form consists of a single clause, the arguments of which are all distinct, and the body of which is a disjunction. The first disjunct is a transformation of the first clause of the original predicate, `append([],Y,Y)`, whereas the second disjunct is a transformation of the second clause of the original predicate.

append/3	append/3 in superhomogeneous form

`append([],Y,Y).`
`append([E|Es],Y,[E|R]):-`
`    append(Xs,Y,R).`
	`:- mode append(in,in,out).`
	`append(X,Y,Z):-`
	`  (X⇒ [], Z:=Y ;`
	`  (X ⇒ [E|Es],append(Es,Y,W), Z ⇐ [E|W]).`

Figure 2.2: The `append/3` predicate and `append(in,in,out)` in superhomogeneous form.

member/2	member/2 in superhomogeneous form

`member(X, [X|_]).`
`member(X, [Y|T]) :-`
`    not (X=Y), member(X, T).`
	`:- mode member(in,in).`
	`member(X,Y):-`
	`  Y ⇒[E|Es],(X==E ; member(X,Es)).`
	`:- mode member(out,in).`
	`member(X,Y):-`
	`  Y ⇒[E|Es],(X:=E ; member(X,Es)).`

Figure 2.3: The `member/2` predicate and `member(in,in)` and `member(out,in)` in superhomogeneous form.

Now consider the definition of the `member/2` predicate, both in normal syntax and in superhomogeneous form for the mode `member(in,in)` and `member(out,in)` as depicted in Fig. 2.2.

Note that the only difference between the two procedures for `member` is the use of test, respectively an assignment in the first disjunct of the body.

According to Definition 2.6, conjunctions and disjunctions are considered binary constructs. This differs from their representation inside the Melbourne compiler (Somogyi et al.), where conjunctions and disjunctions are represented in flattened form. Our syntactic definition however facilitates the conceptual handling of these constructs during analysis.

If-then-else

The Mercury language offers a *if-then-else* construction. It does not explicitly appear in the Mercury syntax used in this work; however, such a construction can easily be transformed into an equivalent code fragment using constructions from this syntax. Indeed, $(Cond \rightarrow Then; Else)$ is logically equivalent to $(Cond, Then; \texttt{not}Cond, Else)$ (Somogyi, Henderson, and Conway 1996). From an operational point of view, such a transformation does not change the order of the solutions, and a derivation tree of an execution of the original code using a given input substitution is nearly identical as an execution of the transformed code using the same input substitution.

2.1.5 A semantics for Mercury

In this section, we formally define a denotational semantics for the subset of the Mercury language presented in Section 2.1.4. We consider a program as being a set of procedures translated to *superhomogeneous form*. We assume thus that every mode of a predicate has been translated into a different procedure and that, in addition, every procedure is well-typed and well-moded. To the best of our knowledge, there is a single published formal semantics for the Mercury language in (Baldan, Le Charlier, Leclre, and Pollet 1999); however, the one we present here is based on the informal description of the Mercury execution algorithm from (Somogyi, Henderson, and Conway 1996). This semantics is intended to be used as a reference model for the symbolic execution of Mercury programs described in the remaining chapters. This symbolic execution will be used in order to generate test cases using path-based – and therefore operational – adequacy criteria, and thus the semantics needs to capture the exact operational behaviour of a Mercury execution, i.e. provide not only the solutions computed by the program (in the right order) but also a representation of the *exact execution path* followed during that computation. In order to easily distinguish the different program points traversed during an execution, we define a *labelled syntax* for procedures in superhomogeneous form. That is, we associate a distinct *label* to a number of program points of interest. In formulas and examples, these labels are written in subscripts and attached to the left and/or right side of a goal.

Definition 2.7 Let Π denote the set of procedure symbols, Σ the set of function symbols and $\mathcal{V}$ and $\mathcal{L}$ respectively the set of variables and labels in a given program P. The syntax of a procedure in labelled superhomogenous form is defined as follows:

$$
\begin{array}{llcl}
LProc & & ::= & p(X_1,\ldots,X_k)\ \texttt{:-}\ C. \\
LConj & C & ::= & {}_lG_{l'} \mid {}_lG\texttt{,}C \\
LDisj & D & ::= & C\texttt{;}C' \mid D\texttt{;}C \\
LGoal & G & ::= & A \mid D \mid \texttt{not}(C) \\
Atom & A & ::= & X\texttt{==}Y \mid X \Rightarrow f(Y_1,...,Y_n) \mid X \Leftarrow f(Y_1,...,Y_n) \\
 & & \mid & Z\texttt{:=}X \mid p(X_1,...,X_n)
\end{array}
$$

where X,Y,Z and $X_j,Y_i(0 \leq j \leq k,\ 0 \leq i \leq n) \in \mathcal{V}, p/k \in \Pi, f \in \Sigma, l, l' \in \mathcal{L}$. A program in labelled superhomogenous form is a set of procedures in labelled superhomogenous form, in which all labels are assumed to be distinct. $\diamond$

Note that according to the definition above, a label is placed between two successive conjuncts, as well as at the beginning and at the end of a conjunction and a disjunction.

Example 2.9 The `append(in,in,out)`, `member(in,in)` and `member(out,in)` procedures – of which the superhomogeneous forms are depicted in Example 2.8

– in labelled superhomogeneous form look as follows.

$$\begin{array}{l}\texttt{append}(\texttt{X} :: \texttt{in}, \texttt{Y} :: \texttt{in}, \texttt{Z} :: \texttt{out}) \texttt{ :- } \\ \quad {}_{1_1}({}_{1_2}\texttt{X} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}],_{1_3} \texttt{append}(\texttt{E}_\texttt{s}, \texttt{Y}, \texttt{W}),_{1_4} \texttt{Z} \Leftarrow [\texttt{E}|\texttt{W}]_{1_5} \ ; \ {}_{1_6}\texttt{Z} = \texttt{Y}_{1_7})_{1_8}.\end{array}$$

$$\begin{array}{l}\texttt{member}(\texttt{X} :: \texttt{in}, \texttt{Y} :: \texttt{in}) \texttt{ :- } \\ \quad {}_{1_1}\texttt{Y} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}],_{1_2} ({}_{1_3}\texttt{X} == \texttt{E}_{1_4} \ ; \ {}_{1_5}\texttt{member}(\texttt{X}, \texttt{E}_\texttt{s})_{1_6})_{1_7}.\end{array}$$

$$\begin{array}{l}\texttt{member}(\texttt{X} :: \texttt{out}, \texttt{Y} :: \texttt{in}) \texttt{ :- } \\ \quad {}_{1_1}\texttt{Y} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}],_{1_2} ({}_{1_3}\texttt{X} := \texttt{E}_{1_4} \ ; \ {}_{1_5}\texttt{member}(\texttt{X}, \texttt{E}_\texttt{s})_{1_6})_{1_7}.\end{array}$$

We can now define the semantics of Mercury under the form of a semantic function with two arguments: a goal and a substitution. The substitution is assumed to map the goal's input variables to ground terms, and thus we will sometimes refer to it as the *input substitution*.
The signature of the semantic function is the following:

$$\mathcal{S} : \mathit{Goal} \times \mathit{Subst} \mapsto (\mathcal{L}^*, \mathit{Subst})^*$$

We use $\mathcal{S}[\![G]\!]$ to denote the meaning of a goal G and we define it as a function from an input substitution to a sequence of pairs, each of those pairs being composed of a sequence of labels and a substitution. Within each such pair, the sequence of labels are the labels encountered from the beginning of the execution up to a success (and thus the creation of a solution) or a failure of this part of the execution. In other words, the sequence of labels is a representation of one branch of the SLD-derivation tree for $G \cup \{\mathit{Subst}\}$. We call such a sequence of labels a *trace*. The order in which they are placed in the sequence of pairs reflects the actual order in which Mercury will produce the different solutions thanks to backtracking. Note that two consecutive segments contain redundant information; indeed, backtracking resumes the execution at the last choicepoint encountered, and not at the entry point of the program. Therefore, the common prefix of two consecutive segments represents, in the second one, a part of execution path that is not actually followed during a real execution. We use the operator $\cdot$ to denote the concatenation of two sequences; the notation $\bullet_{i=1}^{n} s_i$ stands for $s_1 \cdot \ldots \cdot s_n$. The solution computed (or the failure) at the end of each trace is represented by the substitution in the second member of the pair. A special substitution noted `Fail` denotes a failure. The composition of `Fail` with any other substitution always leads to the creation of the substitution `Fail`. We call a *succeeding goal* a goal the semantics of which contains at least one pair containing a substitution that is not `Fail`. Conversely, a *failing goal* has a semantics in which all the pairs of the sequence contain the `Fail` substitution.
By representing multiple solutions of a goal by a sequence and computing a representation of the execution trace, the semantic function $\mathcal{S}$ can be seen as representing the actual behaviour of the execution of a Mercury goal, taking into account the order in which the successive solutions are produced, and therefore

the order in which the backtrackings are performed. This order is indeed necessary in the context of testing and automatic test data generation, since the symbolic execution of the program – obviously based on the semantics – used to generate test data should represent an actual execution of that program as faithfully as possible.
Mercury executes programs using a left-to-right computation rule after having reordered the goals with respect to the mode analysis (Somogyi, Henderson, and Conway 1996), as described in Section 2.1.4. For a goal G and input substitution θ, $\mathcal{S}[\![G]\!]\theta$ is a sequence of pairs (sequence of labels,substitutions) where each such substitution is of the form $\theta\sigma$, that is, an update of the original input substitution θ.
The definition of $\mathcal{S}$ is depicted in Fig. 2.4. A test unification succeeds if the input substitution maps both variable to the same ground term; it fails otherwise, but never creates any bindings. A deconstruction succeeds if the input substitution θ maps the left-hand variable to a term having the same outermost functor f/n as the right-hand term. A new input substitution is created from the former one in which each of the variables in $\overline{Y}$ are bound to the corresponding term in $\bar{t}$. Note that we use $\{\overline{Y}/\bar{t}\}$ as a syntactic sugar to denote $\{Y_1/t_1, \ldots, Y_n/t_n\}$ if $\bar{t} = \langle t_1, \ldots, t_n \rangle$. A construction simply adds a new binding to the input substitution, as does the assignment; neither of them can fail. In all the four last cases, the corresponding trace is empty, since no label has been encountered during the execution of the atom. The semantics of a procedure call $p(\overline{X})$ is given by the semantics of the body of this procedure definition, i.e. the semantics of B if the procedure is defined as $p(\overline{F}) \leftarrow B$, in which the variables have been renamed as follows:

- the variables that are the formal arguments of p are renamed respectively to the actual arguments of the procedure call. This renaming is denoted by σ, defined as $\sigma = \{\overline{F}/\overline{X}\}$
- all the variables used in B that are not in the procedure formal arguments are renamed using *fresh* variables. This renaming is denoted ρ and its domain is $dom(\rho) = \mathcal{V}(B) \setminus \mathcal{V}(\overline{F})$

Using those two renamings, we can therefore define the semantics of a procedure call $p(\overline{X})$ as the semantics of $B\rho\sigma$.
In order to define the meaning of a conjunction we first define the meaning of a goal preceded and succeeded by a label. Since the preceding label is encountered before entering the goal, the first trace of the semantics of the goal it precedes begins with that label. The succeeding label is encountered only when the execution of the goal finishes, i.e. when the goal succeeds. This is why this label is added at the end of each trace corresponding to a success in the semantics of the goal. Similarly, in a conjunction, the execution of the first conjunct continues with the execution of the next conjunct only when the first conjunct succeeds. That is why each trace of the first conjunct leading to the creation of a substitution θ_i is concatenated with the first trace of the semantics of the rest of the conjunction using θ_i as input substitution. Note that if $\theta_i = \texttt{Fail}$,

$$
\begin{array}{lcl}
\mathcal{S}[\![_]\!]\texttt{Fail} & = & \langle(\langle\rangle,\texttt{Fail})\rangle \\
\mathcal{S}[\![X\texttt{==}Y]\!]\theta & = & \begin{cases} \langle(\langle\rangle,\theta)\rangle & \text{if } \theta(X)=\theta(Y) \\ \langle(\langle\rangle,\texttt{Fail})\rangle & \text{otherwise} \end{cases} \\
\mathcal{S}[\![X \Rightarrow f(\overline{Y})]\!]\theta & = & \begin{cases} \langle(\langle\rangle,\theta\{\overline{Y}/\overline{t}\})\rangle & \text{if } \theta(X)=f(\bar{t}) \\ \langle(\langle\rangle,\texttt{Fail})\rangle & \text{otherwise} \end{cases} \\
\mathcal{S}[\![X \Leftarrow f(\overline{Y})]\!]\theta & = & \langle(\langle\rangle,\theta\{X/f(t_1,\ldots,t_n)\})\rangle \text{ where } \forall i: t_i=\theta(Y_i) \\
\mathcal{S}[\![X\texttt{:=}Y]\!]\theta & = & \langle(\langle\rangle,\theta\{X/t\})\rangle \text{ where } \theta(Y)=t \\
\mathcal{S}[\![p(\overline{X})]\!]\theta & = & \mathcal{S}[\![B\rho\sigma]\!]\theta \\
 & & \quad \text{where } p(\overline{F}) \leftarrow B \in \mathit{Proc} \\
 & & \qquad\qquad \sigma=\{\overline{F}/\overline{X}\} \\
 & & \qquad\qquad \rho \text{ is a fresh renaming} \\
 & & \qquad\qquad \mathit{dom}(\rho)=\mathcal{V}(B)\setminus\mathcal{V}(\overline{F}) \\
\mathcal{S}[\![(_l G)]\!]\theta & = & \bullet_{i=1}^{n}\langle(\langle l\rangle\cdot t_i,\theta_i)\rangle \\
 & & \text{where } \mathcal{S}[\![G]\!]\theta=\langle(t_1,\theta_1),\ldots,(t_n,\theta_n)\rangle \\
\mathcal{S}[\![(_l G_{l'})]\!]\theta & = & \bullet_{i=1}^{n}\langle(t'_i,\theta_i)\rangle \\
 & & \text{where } \mathcal{S}[\![(_l G)]\!]\theta=\langle(t_1,\theta_1),\ldots,(t_n,\theta_n)\rangle \\
 & & \qquad\qquad t'_i=\begin{cases} t_i & \text{if } \theta_i=\texttt{Fail} \\ t_i\cdot\langle l'\rangle & \text{otherwise} \end{cases} \\
\mathcal{S}[\![(_l G\ ,\ C)]\!]\theta & = & \bullet_{i=1}^{n}\langle(t_{Gi}\cdot t_{Ci1},\theta_{Ci1}),\ldots,(t_{Gi}\cdot t_{Cim},\theta_{Cim})\rangle \\
 & & \text{where } \mathcal{S}[\![(_l G)]\!]\theta=\langle(t_{G1},\theta_{G1}),\ldots,(t_{Gn},\theta_{Gn})\rangle \\
 & & \qquad\qquad \mathcal{S}[\![C]\!]\theta_{Gi}=\langle(t_{Ci1},\theta_{Ci1}),\ldots,(t_{Cim},\theta_{Cim})\rangle \\
\mathcal{S}[\![(C\ ;\ C')]\!]\theta & = & \mathcal{S}[\![C]\!]\theta\cdot\mathcal{S}[\![C']\!]\theta \\
\mathcal{S}[\![(D\ ;\ C)]\!]\theta & = & \mathcal{S}[\![D]\!]\theta\cdot\mathcal{S}[\![C]\!]\theta \\
\mathcal{S}[\![\mathit{not}(C)]\!]\theta & = & \begin{cases} \langle(t_1,\texttt{Fail}),...,(t_{n-1},\texttt{Fail}),(t_n,\emptyset)\rangle & \text{if } \theta_i=\texttt{Fail}, \\ & \forall 1\le i\le n \\ \langle(t_1,\texttt{Fail}),...,(t_k,\texttt{Fail})\rangle & \text{if } \theta_k\neq\texttt{Fail}, \\ & k\le n\ \wedge \\ & \theta_1,\ldots,\theta_{k-1}=\texttt{Fail} \end{cases} \\
 & & \text{where } \mathcal{S}[\![C]\!]\theta=\langle(t_1,\theta_1),...,(t_n,\theta_n)\rangle
\end{array}
$$

Figure 2.4: Definition of $\mathcal{S}$.

then the semantics of the rest of the conjunction using θ_i as input substitution is $\langle(\langle\rangle, \mathtt{Fail})\rangle$, according to the rule $\mathcal{S}[\![_]\!]\mathtt{Fail} = \langle(\langle\rangle, \mathtt{Fail})\rangle$. The meaning of a disjunction is the concatenation of the meaning of the first disjunct with the meaning of the second disjunct. Consequently, a disjunction only fails when both the disjuncts fail (denoted by a sequence in which all the pairs contain a substitution Fail). If the negated conjunction in a not-goal fails – that means that all the traces in its semantics lead to failures ($\theta_i = \mathtt{Fail}, \ \forall 1 \leq i \leq n$) – the not-goal succeeds. This success is denoted by the replacement of the failure substitution Fail of the last trace in the negated conjunction semantics by the empty substitution $\emptyset$. Indeed, a negated goal in Mercury is not allowed to bind variables that are used outside the negated goal. If the negated conjunction succeeds – at least one of the traces in its semantics leads to a solution $\theta_i \neq \mathtt{Fail}$ – then the result of the not-goal is a failure that occurs when the negated conjunction succeeds for the first time. The semantics of the not-goal is thus the prefix of the semantics of the negated conjunction containing k couples – where k is the least index of the couples representing a succeeding execution – in which the substitution of kth couple is replaced by Fail in order to denote the failure of the not-goal.

Example 2.10 Let us reconsider the member(out,in) written in labelled syntax as shown in Example 2.9.

$$\mathtt{member(X :: out, Y :: in)} :- \quad {}_{\mathtt{l_1}}\mathtt{Y} \Rightarrow [\mathtt{E}|\mathtt{E_s}],_{\mathtt{l_2}} ({}_{\mathtt{l_3}}\mathtt{X} := \mathtt{E}_{\mathtt{l_4}} \ ; \ {}_{\mathtt{l_5}}\mathtt{member(X, E_s)}_{\mathtt{l_6}})_{\mathtt{l_7}}.$$

The semantics of a call member(V,W) using as input substitution $\theta = \{W/[0,1]\}$ is the following:

$$\begin{aligned} \mathcal{S}[\![\mathtt{member(V,W)}]\!]\theta = \quad & \langle(\langle l_1, l_2, l_3, l_4, l_7\rangle, \{V/0, W/[0,1]\}), \\ & (\langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_6, l_7\rangle, \{V/1, W/[0,1]\}), \\ & (\langle l_1, l_2, l_5, l_1, l_2, l_5, l_1\rangle, \mathtt{Fail})\rangle \end{aligned}$$

Chapter 3

A test automation framework for Mercury

3.1 Test automation framework

As stated in Definition 1.5, once a test suite has been created the testing process consists in executing, for each test case, the (part of the) program under concern using the input values contained in that test case and comparing the actual result of the execution with the expected result (also recorded in the test case). Performing these actions for each test case of a large test suite – possibly containing several *thousands* of test cases – is a very repetitive and time-consuming task to be done manually. This is why there exist tools that enable the automation of that process; such tools are called *test automation frameworks*. They are able to perform automatically the testing process for a (part of a) program using a test suite that was created previously. There exist a large number of such tools, for a large variety of programming languages – mainly imperative and object-oriented languages (Hunt and Thomas 2003; Davis, Chirillo, Gouveia, Saracevic, Bocarsley, Quesada, Thomas, and Lint 2009; McMahon 2009). A very well-known example of such a tool is JUnit for Java (Hunt and Thomas 2003; Massol and Husted 2003). One of the main advantages of such tools is that they enable easy repetitive testing once a test suite has been created; this is particularly useful for performing so-called *regression* testing (Leung 1992). Regression testing consists in testing a program after modifications have been introduced using the same test suite as the one used to test the program before it was modified. The goal is to check if no errors were introduced during the implementation of the modifications. Indeed, if a test case fails during the testing process, and if this test case succeeded during the test of the previous version of the program, it means that new errors ("regressions") were introduced and should be corrected.
If most of the work focuses on imperative programming, there exist however test frameworks for declarative programming languages, such as Prolog Unit Tests

(Wielemaker 2006) – an integrated test framework for SWI-Prolog –, the basic `test_util` library for ECLiPSe Prolog (Schimpf), and HUnit (Herington 2002) for Haskell. Note that the way we deal with I/O operations is quite simple compared to (Wielemaker 2006); the latter tool provides for example a feature called "cleanup", able to revert the side effects induced by the execution of the code fragment. This tool also provides other advanced features, such as the possibility to add a pre-condition to a test, such that if this condition fails the test is skipped.
The target of the work presented in the current chapter is to develop a test automation framework for unit testing the Mercury language, which had no such tool available yet. This framework represents a very useful and convenient base to build on in order to add automatic test data generation capabilities.
Our framework has been conceived by building on the same principles as the previously mentioned tools, though it was not possible port any of those tools directly to Mercury because of the particularities of the language. For example, the strict type- and mode-checking mechanisms make it difficult to adopt most of the methods used in Prolog, even if we can of course re-use some of the ideas in the design phase.

In what follows we first present an implementation of our unit testing framework and some interesting characteristics of this implementation (Section 3.2), then we show and discuss the results of a limited evaluation of the prototype (Section 3.3).

3.2 Unit testing tool for Mercury

The goal of this work is to create a framework for Mercury that lets the user define test cases through a simple language. From that point on, this framework can propose different tools; the first one, presented in this section, automatically performs the whole testing process.
This testing process is completely independent from the tested code: one can write test cases without having any knowledge of the source code, therefore the tool is usable for black-box as well as white-box testing.
A schematic diagram of the testing process will shown in Figure 3.1. The first step is to transform a given test suite into Mercury source code, the effect of which – when run – is to execute and evaluate all the test cases, as explained in Section 3.2.2. In order to be successfully compiled, this generated source code must be put together with the source code of the different modules it depends on (i.e. the modules containing the tested procedures and the procedures called therein). The effect of running the resulting compiled code is the production of a test report. An optional input of the tool is a renaming information file; its usage is explained in Section 4.4.
In Definition 1.1, we defined a test case as the combination of some test together with the output as expected to be produced by the (part of the) program under test when executed using that input data. In our tool, we generalize

this definition by defining a test case as the combination of a *code fragment* together with one or more *assertions on the expected results* of the execution of this fragment. A “classical” test case (i.e. a combination of input data with the corresponding expected output) can be represented in this formalism by using, as code fragment, a single call to the (part of the) program under test with the input data provided in the test case as argument and, as only assertion, a proposition on the expected result including an equality test on the possible output arguments. The reasons we defined such a generalisation for the notion of test case is 1) to provide a way to initialize the program environment before executing the test and 2) to provide a mechanism to specify of the expected behaviour of a procedure/program which is different than just a single set of expected output values; indeed, in a logic programming context, one could want to assert about the possible success/failure of the execution, the number of output values produced, properties about those output values such as their order, etc. In our implementation, a test case is therefore represented as a triple, denoted test(t, c, a), where t is the name of the test case (a Mercury string), c is a Mercury code fragment – a conjunction – represented as a list of atoms, and a is a list of assertions. An assertion can be either a condition on the variables used in c – a condition is represented by any *semidet* goal, including conjunctions, disjunctions, procedure calls, etc. –, and/or a specification of the expected behaviour of the execution.
Let us examine a simple example of the syntax of a test case:

Example 3.1

```
test(t1, [reverse([1,2],L)], [true(L=[2,1])]).
```

`t1` is the name that will be used to refer to the test case in the report generated by the tool. The code fragment to test contains only one goal (a call to the list reverse predicate), while the only assertion is a condition verifying whether the only value computed for `L` is indeed the result of reversing the list `[1,2]`. This example is a representation in our formalism of a “classical” test case.

3.2.1 Determinism

If only the features mentioned above are used, then execution of the test code is limited to the first solution, even if the predicate under test has possibly multiple solutions. In the latter case, all the solutions but the first one are dropped. Nevertheless, more extensive examination of `multi` and `nondet` predicates is also possible. In general, the following conditions can be used in the assertions part:

success Successful if the code fragment succeeded.

failure Successful if the code fragment failed.

true(G) Successful if the goal Gθ – with θ being the first answer returned by the tested code fragment in the test case – succeeds.

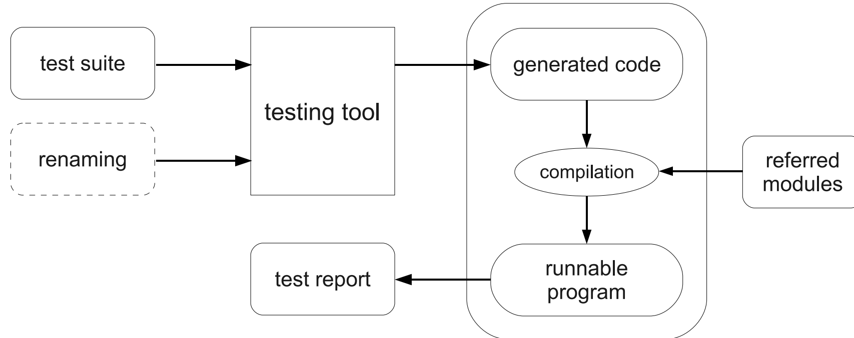


Figure 3.1: Testing framework

some_true(G) Successful if there exists θ among the answers returned by the tested code fragment in the test case which causes the goal Gθ to succeed.

all_true(G) Successful if for each θ among the answers returned by the tested code fragment in the test case, the goal Gθ succeeds.

true(N, G) Successful if the goal Gθ – with θ being the Nth answer returned by the tested code fragment in the test case – succeeds.

solutions_cardinality(N) Successful if the execution of the tested code fragment in the test case produces N solutions. N can be either a variable or a constant.

Together with these assertions, one can use the following statements that will affect the execution of the code fragment:

type(V, T) Allows the user to define type information of a variable; this is useful in some particular situations, as explained in Section 3.2.2.

limit(N) Limits the execution to N solutions. This can be useful when testing predicates with a large number of solutions.

Example 3.2 illustrates the usage of some of these conditions.

Example 3.2

```
test(t2, [member(X,[1,3,4,2])],
     [limit(2),some_true(X>1)]).
test(t3, [member(X,[1,2,3,4])],
     [solutions_cardinality(N),true(N>3),
     all_true(X<5)]).
```

The semantics of the assertions in `t2` is "there is a solution among the first two in which the value bound to X is bigger than 1", while the meaning of the assertions in `t3` is "there are at least 3 solutions, and all of the solutions are less than 5".

In reality the framework has two different execution modes of the test tool: "multi" and "IO". Within the "IO" execution mode, usage of input/output operations is allowed in the tested code fragment; however, testing multi-solution predicates is not possible in this mode. Within the "multi" mode, it is the other way round; testing multi-solution predicates is allowed but one can not perform any input/output operation. The execution mode of the tool can be chosen by a command line option. The reason why these two different modes exist is due to a limitation of the Mercury language, that enables input/output operations in deterministic (or cc-multi) predicates only – for further details see (Henderson, Conway, Somogyi, Jeffery, Schachte, Taylor, Speirs, Dowd, Becket, and Brown 1996).

3.2.2 Implementation details

The code generated for executing each test case is relatively straightforward. For example, Figure 3.2 depicts the generated code for the test case `t1` defined in Example 3.1. Since the expected behaviour is specified as success, if the `reverse/2` predicate fails, the result of the test case will be "failed because of failure (instead of success)".

```
...
testcase(t1, Result) :-
  (if
    reverse([1,2], L)
  then
    (if
      L = [2,1]
    then
      Result = succeeded
    else
      Result = condition_failed
    )
  else
    Result = failed(failure)
  ).
```

Figure 3.2: Generated code (det)

Testing multi-solution predicates needs some considerations. Our implementation uses Mercury's `solutions` library for handling predicates that can succeed more than once. However, this library has an important restriction, namely that the given predicate can have only one output argument. An easy solution to this problem is to wrap all the output variables into a compound term, then unwrap the variables after execution of the code and then perform the checks of the assertions part. Unfortunately, for the generation of this compound type declaration, the type of each output variable should be known. The need of type

analysis could strongly limit the usability of the tool since all the sources of used modules should be known in that case. This is usually not feasible, especially in case of built-in modules. The workaround we developed is to use the type analysis facility of the compiler itself. It is possible with the `univ` library, which allows to wrap any Mercury type into a universal type. For the unwrap operation, the compiler must know the type of the wrapped object. Usually, it can be inferred from the assertions, but if not, we have to give the type manually, as a help to the compiler. Example 3.3 shows the usage of this feature.

Example 3.3

```
test(t4, [append(L1,L2,[1,2,3])], [type(L2,list(int)),
     some_true((L1=[1,2],length(L2,1)))]).
test(t5, [append(L1,L2,[1,2,3])],
     [some_true((L1=[1,2],L2=[3]))]).
```

The tested code fragment is the same in both test cases: the `(out,out,in)` mode of `append/3`. In `t4`, the compiler can infer the type of `L1`, but the type of `L2` must be given explicitly. In the other test case, the compiler doesn't need any complementary information. Notice that if there is no more than one common variable between the two code parts, then no wrapping is used, and thus no type information needs to be provided.

Generation of code for the `some_true/1`, `all_true/1` and `true/2` conditions is based on the same principle. Unwrap instructions of output variables are appended before the given condition if necessary, then this code fragment is called in an appropriate way. For example in the case of `true/2`, after selecting the required solution from the list, the constructed predicate is called simply using the `call/2` predicate. The optional solution number limitation is implemented with the help of `do_while` predicate in the `solutions` library.
Figure 3.3 shows the generated code for the test case `t4`, where we can see the declaration for the generated type. The two lines just after the call to `append/3` wrap the output variables into a single compound term. The reverse operation is performed by the two lines just before the assertions. The combination of the latter together with the assertions themselves constitute the body of a meta-predicate. This meta-predicate must succeed for at least one solution for the test case to be considered as successful.

3.2.3 Handling exceptions

In this thesis, we focus on a subset of the Mercury language, containing only declarative functionalities. However, Mercury is conceived as an industrial language, an includes therefore many features similar to the ones available in the most well-known imperative and object-oriented programming languages. For example, Mercury includes a mechanism allowing the programmer to throw *exceptions*. Our framework is able to deal with these exceptions. By default, every exception thrown within either the tested code or some of the assertions is caught by the framework. The expected result can also be declared as being

```
...
:- type t4_type ---> t4_t(univ, list(int)).
testcase(t4, Result) :-
  solutions( ((pred (IF1 :: out)) is nondet :-
      append(L1, L2, [1,2,3]),
      type_to_univ(L1, L1_U),
      IF1 = t4_t(L1_U, L2)
    ), Vs),
  (if
    some_true( ((pred (IF2 :: in)) is nondet :-
        IF2 = t4_t(L1_U, L2),
        det_univ_to_type(L1_U, L1),
        L1 = [1,2], length(L2,1)
      ), Vs)
  then
    Result = succeeded
  else
    Result = condition_failed
  ).
```

Figure 3.3: Generated code (nondet)

an exception. However, the framework is not able to distinguish between these exception according to their origin; an exception thrown by an assertion is handled in the same way as if it had been thrown by the tested code. Currently it is impossible to make assertions about the exception itself, the only thing that can be declared in the assertion part is that the desired result is an exception. Nevertheless, it can happen that exceptions need to be left uncaught, especially when the user wants to know the exact source of an exception, usually to know where to find a given bug. If the exception is caught, the result will only be "the test case threw an exception", but the real cause remains hidden. To help to identify these problems, the exception handling mechanism can be entirely switched off by a command line switch, so in that "debug" mode, the details of the problem becomes observable.

3.3 Evaluation

Table 3.1 shows the results of a small evaluation of our tool. Three different properties were examined: for a given testsuite, we measure the size of the generated code, the time needed for its generation by our tool, and the execution overhead of the generated code compared to the execution time of a script that executes the testsuite in an ad-hoc way.
As one expects, the size and generation time of the code depends on the number and complexity of the given set of test cases. Although the execution time also depends on the complexity of the test cases, the evaluation shows a rather

Goals	Determinism	Test cases	Generated code size (lines)	Code generation (ms)	Execution (ns)	
					gross	net
member(in,in)	semidet	6	169	12	40	2
member(out,in)	nondet	4	189	12	40	11
bubblesort(in,out)	det	24	475	16	60	50
transpose(in,out)	det	11	288	12	40	8

Table 3.1: Performance of the testing tool

constant overhead for the execution of the testcode generated by our framework.

Chapter 4

A control flow graph for Mercury

The control flow graph of a program – see Section 1.3 for an overview of the notions of control flow and control flow graph – is a widely used structure in many software development tools such as compilers and debuggers. Their main interest lies in the fact that they provide an explicit representation of a program's control flow structure which makes them well-suited as a building block for implementing program analyses and optimizations such as dead-code elimination, branch prediction, loop transformations, etc. (Muchnick 1997; Allen and Kennedy 2002). Moreover, the fact that they can easily be visualised makes that they are frequently used in debugging and (semi) automatic test-case generation (e.g. (Visser, Păsăreanu, and Khurshid 2004)).

Notwithstanding these applications, the construction and use of control flow graphs for logic programs have received little attention. Some notable exceptions include (Lindgren 1995; Cameron, de la Banda, Marriott, and Moulder 2003; Brayshaw and Eisenstadt 1991). This should not be surprising, given that in logic programming languages control information is far less explicit in programs and hence more difficult to catch in a static structure.

In this work, we define how one can build and use a control flow graph for the logic programming language Mercury. The fact that Mercury is a *moded* language makes it easier to extract control flow information from a program than it would be the case for an unmoded language such as Prolog. Nevertheless, the resulting structure is a non-trivial extension of its counterpart for imperative programs, since it needs to allow for reasoning about success and failure of goals, backtracking, and multiple answers.

The construction of the control-flow graph is based on the labelled syntax of Mercury programs defined in Definition 2.7. The labels are intended to identify the nodes of the program's control flow graph.

$$
\begin{array}{lcl}
\mathcal{A}\ {}_lG_{l'}\ l_s\ l_f & = & (\mathcal{A}\ {}_lG\ l'\ l_f) \cup \{(l', l_s)\} \\
\mathcal{A}\ {}_lG, C\ l_s\ l_f & = & (\mathcal{A}\ {}_lG\ first_C\ l_f) \cup (\mathcal{A}\ C\ l_s\ l_f) \\
\mathcal{A}\ {}_lX == Y\ l_s\ l_f & = & \{(l, l_s), (l, l_f)\} \\
\mathcal{A}\ {}_lX \Rightarrow f(\overline{Y})\ l_s\ l_f & = & \{(l, l_s), (l, l_f)\} \\
\mathcal{A}\ {}_lX := Y\ l_s\ l_f & = & \{(l, l_s)\} \\
\mathcal{A}\ {}_lX \Leftarrow f(\overline{Y})\ l_s\ l_f & = & \{(l, l_s)\} \\
\mathcal{A}\ {}_l(C; C')\ l_s\ l_f & = & (\mathcal{A}\ C\ l_s\ l_F) \cup (\mathcal{A}\ C'\ l_s\ l_f) \cup \{(l, first_C), (l, first_{C'})\} \\
\mathcal{A}\ {}_l(D; C)\ l_s\ l_f & = & (\mathcal{A}\ {}_lD\ l_s\ l_F) \cup (\mathcal{A}\ C\ l_s\ l_f) \cup \{(l, first_C)\} \\
\mathcal{A}\ {}_l not(C)\ l_s\ l_f & = & \mathcal{I}o\ ((\mathcal{A}\ C\ l_f\ l_s) \cup \{(l, first_C)\})\ l\ l_s\ l_f \\
\mathcal{A}\ {}_lp(\overline{X})\ l_s\ l_f & = & (\mathcal{A}\ B\ l_s\ l_f) \cup \\
 & & \{(l, first_B), (l, ls)^{rs}, (l, lf)^{rf}\}
\end{array}
$$

Figure 4.1: Definition of function $\mathcal{A}$

Most of the work about the control flow graph for Mercury presented in this chapter has been published in (Degrave and Vanhoof 2007a) and (Degrave, Schrijvers, and Vanhoof 2008).

4.1 Constructing the graph

Before defining the control flow graph for a Mercury program, let us introduce two special labels l_S (the *success label*) and l_F (the *failure label*), representing respectively success and failure of a (partial) derivation. Note that since l_F represents failure of a (partial) derivation, it usually causes the execution to backtrack when reached.

Definition 4.1 The *control flow graph* for a Mercury program P written in the labelled syntax is denoted $\mathcal{G}(P)$ and defined as a couple (N, A) where N is the set of nodes of the graph, constituted by the labels appearing in P together with the success label l_S and the failure label l_F, and A is the set of arcs of the graph, containing three different kinds of arcs: *regular arcs*, *return-after-success arcs* and *return-after-failure arcs.* It is defined as

$$A = \bigcup_{p(\overline{X}) \leftarrow B \in P} \mathcal{A}\ B\ l_S\ l_F$$

where $\mathcal{A}$ is a function the signature of which is

$$\mathcal{A}\ : Goal \times \mathcal{L} \times \mathcal{L} \mapsto \wp(\mathcal{L} \times \mathcal{L})$$

and the definition of which is provided in Fig. 4.1. $\diamond$

In the definition of $\mathcal{A}$ depicted in Fig. 4.1, we denote by $first_C$ the first label appearing in a labelled conjunction C, i.e. the label preceding the first conjunct of the conjunction C.

The two labels l_s and l_f used as second and third arguments of the function $\mathcal{A}$ are the nodes representing respectively the program point on which the execution should be resumed upon success and failure of the (labelled) goal passed as first argument. In the remaining we call these program points the *local success point*, respectively the *local failure point*. Note that they are not (necessarily) the success label l_S and the failure label l_F; however, the only case in which the local failure point l_f differs from the failure label l_F is when the goal under concern is the last conjunct of the negated conjunction in a $not(C)$ goal. In this case, failure of the conjunct does not imply failure of the derivation but the success of the negated conjunction (and thus the continuation of the execution). The arcs of a goal surrounded by two labels ${}_lG_{l'}$ are the arcs of ${}_lG$ with l' used as local success point, together with the regular arc (l', l_s) representing the transition from l' to the local success point l_s. The arcs of a conjunction beginning with a goal ${}_lG, C$ are the arcs of ${}_lG$ using the first label of the rest of the conjunction C as local success point, together with the arcs of C. For a test unification or a deconstruction preceded by a label l, the arcs are (l, l_s) linking the preceding label to the local success point (denoting the possible success of the test, respectively deconstruction) and (l, l_f) linking the preceding label to the local failure point (denoting the possible failure of the test, respectively deconstruction). Since neither an assignment nor a construction can fail, there is a single arc for an assignment, respectively construction preceded by a label l, that links l to the local success point l_s. The arcs of a disjunction preceded by a label ${}_l(C; C')$ are the arcs linking l to the first label of each disjunct C and C', together with the arcs of the first disjunct C and the arcs of the second disjunct C' using the failure label as local failure point, and the arcs of the second disjunct C' using the local failure point. The reason the failure label l_F is used as local failure point for the first disjunct is because the failure of one of the disjuncts is not sufficient to cause the whole disjunction to fail. The execution *can* be resumed at the local failure point only after the last disjunct has failed, and after each preceding disjunct has failed causing backtrackings to be performed. This is why we compute the arcs of the last disjunct of C' with a local failure point which is the same as the one of the whole disjunction, whereas the local failure points of the other disjuncts are the failure label l_F, that causes the execution to backtrack when reached. We assume that regular arcs are annotated by a natural number called its *priority*. Each arc initiating a disjunct is annotated by the *position* of the disjunct in the disjunction when counted from right to left. Other arcs are annotated by zero. The function $\mathcal{A}$ can easily be modified in order to add these annotations explicitly; however given this addition would be straightforward, we prefer not adding those modifications here in order to improve readability.

The arcs of a not-goal ${}_lnot(C)$ are the arcs of the negated conjunction C computed with inverted local success point and local failure point, together with the arcs making the transition between l and the first label of C. Indeed, the success, respectively failure of C would lead to the failure, respectively success of $not(C)$. Moreover, we have to introduce a special mechanism to deal with the particularities of the negation. Indeed, the derivation of a negated goal is

performed, in logic programming, in isolation from the main derivation (Apt and van Emden 1982). The consequence is that if the execution of the negated goal has exited once (either upon a failure or a success), not backtrack can occur to a program point inside this negated goal. Therefore, we use a function $\mathcal{I}o$ that transforms the graph of the the negation by assigning a fresh negative even number n as a priority to the entry arc of the graph (i.e. the arc originating from l) and $n-1$ to the exit arcs (i.e. the arc ending at l_s and l_f, the local success respectively failure labels). This allows us to clearly identify the entry and the corresponding exit of a not-goal in the graph of a procedure. Finally, the arcs of a procedure call preceded by a label ${}_lp(\overline{X})$ are the arcs of the conjunction B where B is the body of the procedure, i.e. $p(\overline{F}) \leftarrow B \in Proc$, together with the transition arcs from l to the first label of B and two special arcs: a *return-after-success* and a *return-after-failure* arcs. A *return-after-success* or *return-after-failure* arc, denoted $(l, l')^{rs}$ respectively $(l, l')^{rf}$ denotes the fact that the execution after procedure call preceded by l should be resumed at l' upon success, respectively failure, of the call.

Definition 4.2 We call *choicepoint* a node of a control flow graph from which leave several arcs bearing different priorities. $\diamond$

Example 4.1 Figure 4.2 depicts two control flow graphs. The left one corresponds to a program defining the `member(in,in)` procedure, the right one to a program defining the `member(out,in)` procedure, as defined in Example 2.9.

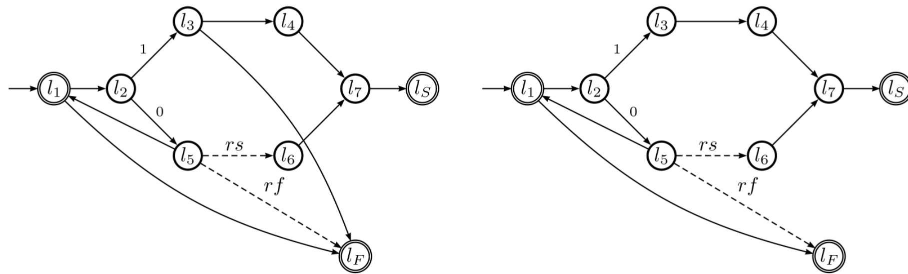


Figure 4.2: `member(in,in)` and `member(out,in)`

In both graphs, the arc (l_1, l_2) represents success of the atom $Y \Rightarrow [E|E_s]$ whereas the arc (l_1, l_F) represents failure of the atom. In the first case, the execution continues at l_2, in the latter it fails. The node l_2 represents a choicepoint since two different arcs – (l_2, l_3) and (l_2, l_5) – originate from this node. These two arcs are associated to priorities denoting the order of appearance, in the source code, of the disjunct they lead to. The return-after-success arc $(l_5, l_6)^{rs}$ and the return-after-failure arc $(l_5, l_F)^{rf}$ denote respectively the fact that the execution should resume on l_6, respectively l_F, upon success, respectively failure of the call preceded by l_5 in the labelled source code. The only

difference between both graphs is the presence of the arc (l_3, l_F) in the graph for `member(in,in)`; it represents the fact that the atom at l_3 (the test X`==`E) can fail whereas the assignment X`:=`E in `member(out,in)` cannot. In order to avoid overloading the figures, we depict priorities only when relevant, i.e. when they annotate an arc representing the entry into a disjunct.

4.2 Deriving execution sequences

A program's control flow graph allows to reason about *all* possible executions of a given procedure. We first define the notion of a *complete execution segment* that represents a straightforward derivation from a call to a success (and thus the production of an answer) or a failure, in which an arbitrary disjunct is chosen at each encountered choicepoint. The definition is in two parts:

Definition 4.3 An *execution segment* for a labelled procedure $p(\overline{X}) \leftarrow B$ is a finite sequence of labels $\langle l_1, \ldots, l_n \rangle$ where l_1 is $first_B$, and l_n is either the success label l_S or the failure label l_F, where for each pair of consecutive labels (l_i, l_{i+1}) the following conditions hold:

1. If $l_i \neq l_S$ and $l_i \neq l_F$ then l_i is connected to l_{i+1} in the program's control flow graph with a regular arc.
2. If $l_i = l_S$ then there exists l_c $(c < i)$ such that l_c and l_{i+1} are connected in the graph with a *return-after-success* arc, and the sequence $\langle l_{c+1}, ..., l_i \rangle$ is itself an execution segment;
3. If $l_i = l_F$ then there exists l_c $(c < i)$ such that l_c and l_{i+1} are connected in the graph with a *return-after-failure* arc, the sequence $\langle l_{c+1}, ..., l_i \rangle$ is itself an execution segment *and* each pair of consecutive labels (l_j, l_{j+1}) with $c + 1 \leq j \leq i$ is connected in the graph with a regular arc of which the priority equals zero.

◇

Definition 4.3 basically states that an execution segment is a path trough the graph in which a label l_{i+1} follows a label l_i if both labels are connected by a regular arc (condition (1)). If, however, l_i represents the exit from a procedure call – either by success (condition (2)) or failure (condition (3)) – then the next label should be a valid resume point. Moreover, conditions 2 and 3 impose that each return has a corresponding call, and guarantee that the sequence of labels representing the execution through the callee is a valid execution segment as well. Condition 3 also denotes that the return after the *failure* of a call can be performed only if the corresponding call definitely failed, i.e. it is impossible to perform backtracking to a choicepoint created after the call that would make the latter succeed. In order to be useful, an execution segment must be *complete*, intuitively meaning that there should be no calls without a corresponding return, unless the derivation ends in failure and contains unexplored alternatives for

backtracking. A complete execution segment can be seen as representing a branch, from the root to a leaf, of one (or more) of the possible execution trees of the procedure under concern.

Definition 4.4 An execution segment β for a procedure p is *complete* if the following conditions hold:

1. If β ends with l_S, then no proper suffix of β is an execution segment for any procedure of the program;

2. If β ends with l_F then if β has a proper suffix β' which is an execution segment for a procedure of the program, then β' contains at least one pair of consecutive labels (l_j, l_{j+1}) connected in p's control flow graph by a regular arc annotated by a priority $n \geq 1$.

$\diamond$

In the above definition, condition 1 guarantees that, in a derivation leading to a success, every call has a corresponding return while condition 2 imposes that, in a derivation leading to a failure, if there exists a call with no corresponding return, it must be possible to backtrack inside this call. Note that Definitions 4.3 and 4.4 only allow for *finite* (complete) execution segments.

Example 4.2 Let us consider `member(in,in)`, defined in Example 2.9 and the corresponding graph, defined on the left side of Figure 4.1. The sequence of labels $\beta = \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle$ represents a complete execution segment in the control flow graph depicted on the left in Figure 4.2. It corresponds to the execution of a call in which the deconstruction of the list succeeds, the first disjunct is chosen at the choicepoint l_2, and the equality test between the first element and the call's first argument also succeeds, leading to the success of the predicate. In other words, it represents a call `member(X,Y)` in which the element X appears at the first position in the list Y. Likewise, the sequence of labels $\beta' = \langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S \rangle$ represents an execution in which the value of the first argument of the call to *member* occurs at the second position of its second argument. Indeed, the deconstruction of the list succeeds, then at choicepoint l_2 the second disjunct is chosen. This second disjunct causes a recursive call to be performed, in which the deconstruction (of the tail of the original list) succeeds, the first disjunct is chosen at l_2 and and reaches the success label. This success causes the execution to be resumed upon label l_6, and finally ends on a success.

Example 4.3 Let us now consider the nondeterministic `member(out,in)` procedure, also defined in Example 2.9. The sequence of labels $\beta = \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle$ represents a complete execution segment in the control flow graph depicted on the right in Figure 4.2. The execution segment β represents the execution leading to the first solution of a call `member(X,Y)` in which the list Y is not empty.

Of particular interest are the choices committed to at each choicepoint encountered along a given complete execution segment. In the remaining we represent these choices by a sequence of integers, which are the priorities of the arcs chosen at each choicepoint.

Definition 4.5 Let $\beta = \langle l_1, ..., l_n \rangle$ be a complete execution segment. The *sequence of choices* associated to β, noted $\mathcal{SC}(\beta)$, is defined as follows:
$\mathcal{SC}(\langle l_1 \rangle) = \langle\rangle$
$\mathcal{SC}(\langle l_1, ..., l_n \rangle) = \mathcal{P}rior(l_1, l_2) \cdot \mathcal{SC}(\langle l_2, ..., l_n \rangle)$
where $\cdot$ denotes sequence concatenation and $\mathcal{P}rior(l_i, l_{i+1}) = \langle nb \rangle$ if l_i is a choicepoint and l_i and l_{i+1} are connected in the graph with a regular arc annotated by a number *nb*, or $\langle\rangle$ if l_i is not a choicepoint. $\diamond$

Example 4.4 Let us consider again the complete execution segment $\beta = \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle$ for `member(in,in)`, defined in Example 4.2. The sequence of choices associated to this segment is $\mathcal{SC}(\beta) = \langle 1 \rangle$. On the other hand, for the second complete execution segment $\beta' = \langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S \rangle$ defined in Example 4.2, we have $\mathcal{SC}(\beta') = \langle 0, 1 \rangle$.

A complete execution segment for a procedure p represents a single derivation for a call to p with respect to some (unknown) input values in which for each encountered choicepoint an arbitrary choice is made. In order to model a real execution of the procedure, several such derivations need in general to be combined, in the right order. The order between two complete execution segments is determined by the sequence of choices that have been made. The sequence of choices being a sequence over natural numbers, we first define the following operation:

Definition 4.6 Let $\langle i_1, ..., i_m \rangle$ denote a sequence over $\mathbb{N}$, we define

$$decr(\langle\rangle) = \langle\rangle$$

$$decr(\langle i_1, ..., i_m \rangle) = \begin{cases} \langle i_1, ..., (i_m - 1) \rangle & \text{if } i_m > 0 \\ decr(\langle i_1, ..., (i_{m-1}) \rangle) & \text{if } i_m < 0 \text{ and } i_m \text{ is even} \\ decr(\langle i_1, ..., (i_{k-1}) \rangle) & \text{if } i_m < 0 \text{ and } i_m \text{ is odd} \\ decr(\langle i_1, ..., i_{m-1} \rangle) & \text{otherwise} \end{cases}$$

where $i_k < 0,\ i_k = i_m + 1$ and $\neg\exists\, j:\ k < j < m$ such that $i_j = i_m + 1$. $\diamond$

For a sequence of choices $\beta = \langle i_1, \ldots, i_n \rangle$, $decr(\beta)$ represents a new sequence of choices that is obtained from β by deleting the rightmost zeros and decrementing the rightmost non-zero choice by one, and ignoring the choices comprised between an even negative priority n and the next priority $n-1$. Operationally, for a complete execution segment β, $decr(\beta)$ represents the stack of remaining choicepoints after performing a backtrack operation together with the entry and exit points of a negated goals; the choices performed between the entry and the exit points of a given negated goals are to be ignored, for the reasons explained earlier (see Section 4.1).
We can now define an *execution sequence* for a procedure, representing a complete derivation tree for this procedure.

Definition 4.7 An *execution sequence* for a procedure p is defined as a sequence of complete execution segments $\langle \beta_1, ..., \beta_n \rangle$ for p having the following properties:

1. For all $0 < i < n$, $decr(\mathcal{SC}(\beta_i))$ is a proper prefix of $\mathcal{SC}(\beta_{i+1})$;
2. For all $0 < i < n$, there does not exist a segment β ($\beta \neq \beta_i \wedge \beta \neq \beta_{i+1}$) such that $decr(\mathcal{SC}(\beta_i))$ is a proper prefix of $\mathcal{SC}(\beta)$ and $decr(\mathcal{SC}(\beta))$ is a proper prefix of $\mathcal{SC}(\beta_{i+1})$;
3. There does not exist a complete execution segment β' for the procedure such that $decr(\mathcal{SC}(\beta'))$ is a proper prefix of $\mathcal{SC}(\beta_1)$.

$\diamond$

An execution sequence $\mathcal{T} = \langle \beta_1, \ldots, \beta_n \rangle$ an be seen as representing a derivation *tree* for a call to the predicate under consideration with respect to some (unknown) input values. Indeed, the first segment β_1 represents the first branch, i.e. the derivation in which for each encountered choicepoint the first alternative is chosen (the one having the highest priority in the graph). Likewise, an intermediate segment β_{i+1} ($i \geq 1$), represents the same derivation as β_i except that at the last choicepoint having an unexplored alternative, the next alternative is chosen. Note that the derivation tree represented by $\langle \beta_1, \ldots, \beta_n \rangle$ is not necessarily complete. Indeed, the last segment β_n might contain choicepoints having unexplored alternatives. However, by construction, there doesn't exist a complete execution segment representing an unexplored alternative between two consecutive segments β_i and β_{i+1}.
While the definition allows to consider infinite execution sequences, an execution sequence cannot contain an infinite segment, nor can it contain a segment representing a derivation in which one of the choicepoints has a previous alternative that would have led to an infinite derivation. It follows that an execution sequence represents a finite part of a real execution of the Mercury procedure under consideration (always with respect to a set of particular but unknown input values). The attentive reader will notice that if $\mathcal{SC}(\beta_n)$ is a sequence composed of all zeros, then the execution sequence $\langle \beta_1, \ldots, \beta_n \rangle$ represents a complete execution in which all answers for the call have been computed.

Example 4.5 Reconsider the nondeterministic procedure `member(out,in)` and the following complete execution segments:

$$\begin{aligned} \beta_1 = \quad & \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle, \\ \beta_2 = \quad & \langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S \rangle, \\ \beta_3 = \quad & \langle l_1, l_2, l_5, l_1, l_2, l_5, l_1, l_F, l_F, l_F \rangle \end{aligned}$$

Labels refer to the graph depicted at the right of Figure 4.2. The reader can easily verify that $\mathcal{SC}(\beta_1) = \langle 1 \rangle$, $\mathcal{SC}(\beta_2) = \langle 0, 1 \rangle$, and $\mathcal{SC}(\beta_3) = \langle 0, 0, 1 \rangle$. Obviously, $decr(\mathcal{SC}(\beta_1)) = \langle 0 \rangle$ is a prefix of $\mathcal{SC}(\beta_2)$ and $decr(\mathcal{SC}(\beta_2)) = \langle 0, 0 \rangle$ is a prefix of $\mathcal{SC}(\beta_3)$ (Condition 1 of Definition 4.7). Moreover, there does not exist a complete execution segment β such that $decr(\beta)$ is a prefix of β_1 (Condition 3 of Definition 4.7), and there are no "intermediate" complete execution segments

that could be placed between β_1 and β_2 or between β_2 and β_3 (Condition 2 of Definition 4.7). Hence $\langle\beta_1, \beta_2, \beta_3\rangle$ is an execution sequence for `member(out,in)`. This execution sequence represents a tree, that we depict in Figure 4.5

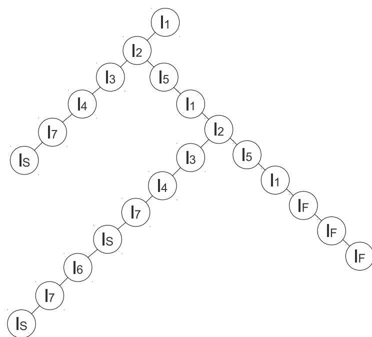


Figure 4.3: Tree representation of the execution sequence defined in Example 4.5

The execution sequence from Example 4.5 corresponds to the execution of a call $member(X, Y)$ in which a first solution is produced by assigning the first element of the list Y to X and returning from the call (expressed by the first segment of the sequence, ending in l_S). A second solution is produced by backtracking, choosing the disjunct corresponding to l_5, performing a recursive call, assigning the second element of the list to X, and performing the return (the second segment, also ending in l_S). The execution continues by backtracking and continuing at l_5 and performing a recursive call in which the deconstruction of the list argument fails. In other words, the execution sequence e corresponds to a call to member in which the second argument is instantiated to a list containing exactly two elements.

4.2.1 Formal definition of symbolic execution

An execution sequence for a goal represents the execution of that goal using determined but unknown input data. Since we would like to derive a *set* of input data for that goal, we first need to derive a *set* of execution sequences. Since an execution sequence represents a (symbolic) execution tree, a set of execution sequences represents a *forest*. For a given goal G we denote the result of deriving corresponding (symbolic) execution trees as $\Phi(G)$. This result can be computed using the different steps presented in this chapter. A more formal definition of this symbolic execution is given in Definition 4.8.
In order to ease the readability in what remains, we define the set of execution segments $\xi = \mathcal{L}^*$ and the set of execution sequences $\Upsilon = (\mathcal{L}^*)^*$

Definition 4.8 The symbolic execution is represented by the Φ function, the signature of which is:

$$\Phi : Goal \mapsto \wp(\Upsilon)$$

$$\begin{array}{lcl} \Phi(X \Leftarrow f(\overline{Y})) & = & \{\mathcal{T}\} \\ \text{where } \mathcal{T} & = & \langle\langle l_S \rangle\rangle \end{array}$$

$$\begin{array}{lcl} \Phi(X \Rightarrow f(\overline{Y})) & = & \{\mathcal{T}_1, \mathcal{T}_2\} \\ \text{where } \mathcal{T}_1 & = & \langle\langle l_S \rangle\rangle \\ \mathcal{T}_2 & = & \langle\langle l_F \rangle\rangle \end{array}$$

$$\begin{array}{lcl} \Phi(X\texttt{==}Y) & = & \{\mathcal{T}_1, \mathcal{T}_2\} \\ \text{where } \mathcal{T}_1 & = & \langle\langle l_S \rangle\rangle \\ \mathcal{T}_2 & = & \langle\langle l_F \rangle\rangle \end{array}$$

$$\begin{array}{lcl} \Phi(X\texttt{:=}Y) & = & \{\mathcal{T}\} \\ \text{where } \mathcal{T} & = & \langle\langle l_S \rangle\rangle \end{array}$$

Figure 4.4: Base cases of the Φ function

From a goal, Φ computes *all possible execution sequences* for that goal, as determined by different instantiations of the goal input arguments. We define Φ by induction on the structure of the goal to execute symbolically. The base cases are depicted in Figure 4.4, the inductive cases are depicted in Figure 4.7. $\diamond$

The unifications are the base cases of the definition, depicted in Figure 4.4. For a unification that can fail (== and $\Rightarrow$) the result is a forest with two trees. Both of them have a single branch; the first one represents the success of the unification, whereas the second one represents its failure. The result for a unification that can only succeed (:= and $\Leftarrow$) is forest with a single tree having a single branch. Before defining Φ for the recursive cases, we define different functions and operators on forests and trees. The function *ext*, depicted in Figure 4.5 has the following signature:

$$ext : (\xi, \wp(\Upsilon)) \mapsto \wp(\Upsilon)$$

From a complete execution segment and an execution forest (a set of complete execution sequences), *ext* creates a new execution forest by extending the single execution segment representing a single execution of a goal with the forest representing the semantics of another goal. If the execution segment ends with a success, *ext* creates this new execution forest by concatenating this segment with each execution sequence (execution tree) in the forest. If the execution segment ends with a failure, the result is a forest with a single execution tree, containing this execution segment as single branch. The formal definition of this operator is provided in Figure 4.5. The concatenation of an execution segment ending with the success label $\langle l_1, \ldots, l_{q-1}, l_q, l_S \rangle$ with an execution sequence $\langle \beta_1, \ldots, \beta_p \rangle$ is defined as an execution sequence $\langle \beta'_1, \ldots, \beta'_i, \ldots, \beta'_p \rangle$, where β'_i $(1 \leq i \leq p)$ is the concatenation of $\langle l_1, \ldots, l_{q-1}, l_q \rangle$ with β_i (the $\bullet$ operation in Figure 4.5).

The result of the $\boxplus$ operator (the definition of which is depicted in Figure 4.6) is the creation, from two execution forests, of a new execution forest composed of *all the execution sequences* that can be created by:

1. Choosing an execution sequence $\mathcal{T}_1$ from the first forest;
2. Each segment of $\mathcal{T}_1$ ending with l_S is concatenated with one of the execution sequences of the second forest (using the concatenation of an execution segment and an execution sequence defined previously).

$$\begin{array}{rcl} ext(\beta_j, \{\mathcal{T}'_1, \ldots, \mathcal{T}'_m\}) & = & \left\{ \begin{array}{ll} \{\beta_j \bullet \mathcal{T}'_1, \ldots, \beta_j \bullet \mathcal{T}'_m\} & \text{if } \beta_j \text{ ends with } l_S \\ \{\langle \beta_j \rangle\} & \text{otherwise} \end{array} \right. \\ \langle l_1, \ldots, l_{q-1}, l_q, l_S \rangle \bullet \langle \beta_1, \ldots, \beta_p \rangle & = & \cdot_{i=1}^{p} \langle \langle l_1, \ldots, l_{q-1}, l_q \rangle \cdot \beta_i \rangle \end{array}$$

Figure 4.5: Definition of the *ext* operator

$$\{\mathcal{T}_1, \ldots, \mathcal{T}_n\} \boxplus \{\mathcal{T}'_1, \ldots, \mathcal{T}'_m\} = \bigcup_{i=1}^{n} F_i \text{ where } F_i = \bigotimes_{\beta \in \mathcal{T}_i} ext(\beta, \{\mathcal{T}'_1, \ldots, \mathcal{T}'_m\})$$

Figure 4.6: Definition of the $\boxplus$ operator

Basically, $\boxplus$ extends each "branch" of each tree in an execution forest (representing one of the possible execution paths of a goal) with the forest representing the semantics of another goal.

Definition 4.9 We define $\otimes$ as the "cartesian concatenation", the signature of which is:

$$\otimes : \wp(T^*), \wp(T^*) \mapsto \wp(T^*)$$

T being any type and formally defined as:

$$A \otimes B = \{a \cdot b \mid (a, b) \in (A \times B)\}$$

◇

The cartesian concatenation applies to two sets of sequences, and its result is the result of a cartesian product in which the members of each tuples are concatenated in a single sequence.
The inductive cases of the Φ function are depicted in Figure 4.7. The symbolic execution of a labelled conjunction ${}_lG$, C results in the creation of the forest using the $\boxplus$ operator, with the forest resulting from the symbolic execution of ${}_lG$ as first argument, and the forest resulting from the symbolic execution of C as second argument. Thanks to this operator, we combine all the execution sequences for ${}_lG$ with all the execution sequences for C. We create therefore all the possible execution sequences through ${}_lG$, C. The symbolic execution of a disjunction (C ; C'), respectively (D ; C), is the forest of all the execution sequences that can be created by concatenating an execution sequence from $\Phi(C)$, respectively $\Phi(D)$, with an execution sequence from $\Phi(C')$, respectively $\Phi(C)$. The symbolic execution of a *not* goal `not`(C) can be obtained by transforming each execution sequence of $\Phi(C)$ as follows:

1. If at least one of the segments of the execution sequence ends with l_S, then each segment ending with l_S is tranformed by replacing this last label by the failure label l_F;

$$\begin{aligned}
\Phi({}_lG) &= \langle \mathcal{T}'_1, \ldots, \mathcal{T}'_m \rangle \\
\text{where } \Phi(G) &= \langle \mathcal{T}_1, \ldots, \mathcal{T}_m \rangle \\
\mathcal{T}'_i &= \langle \langle l \rangle \cdot \beta^i_1, \ldots, \langle l \rangle \cdot \beta^i_n \rangle \\
\mathcal{T}_i &= \langle \beta^i_1, \ldots, \beta^i_n \rangle
\end{aligned} \tag{4.1}$$

$$\begin{aligned}
\Phi({}_lG_{l'}) &= \langle \mathcal{T}'_1, \ldots, \mathcal{T}'_m \rangle \\
\text{where } \Phi({}_lG) &= \langle \mathcal{T}_1, \ldots, \mathcal{T}_m \rangle \\
\mathcal{T}'_i &= \langle \beta'^i_1, \ldots, \beta'^i_n \rangle \\
\mathcal{T}_i &= \langle \beta^i_1, \ldots, \beta^i_n \rangle \\
\beta'^i_j &= \begin{cases} \langle l, l_1, \ldots, l_p, l', l_S \rangle & \text{if } \beta^i_j = \langle l, l_1, \ldots, l_p, l_S \rangle \\ \langle l, l_1, \ldots, l_p, l_F \rangle & \text{if } \beta^i_j = \langle l, l_1, \ldots, l_p, l_F \rangle \end{cases}
\end{aligned} \tag{4.2}$$

$$\Phi({}_lG\ ,\ C) = \Phi({}_lG) \boxplus \Phi(C) \tag{4.3}$$

$$\Phi(C\ ;\ C') = \Phi(C) \otimes \Phi(C') \tag{4.4}$$

$$\Phi(D\ ;\ C) = \Phi(D) \otimes \Phi(C) \tag{4.5}$$

$$\begin{aligned}
\Phi(p(\overline{X})) &= \Phi(B) \\
&\quad \text{where } p(\overline{F}) \leftarrow B \in \mathit{Proc}
\end{aligned} \tag{4.6}$$

$$\begin{aligned}
\Phi(\texttt{not}(C)) &= \langle \mathcal{N}(\mathcal{T}_1), \ldots, \mathcal{N}(\mathcal{T}_r) \rangle \\
\text{where } \Phi(C) &= \langle \mathcal{T}_1, \ldots, \mathcal{T}_r \rangle \\
\mathcal{N}(\langle \beta_1, \ldots, \beta_n \rangle) &= \begin{cases} \langle \beta_1, \ldots, \beta_{k-1}, \beta'_k \rangle & \text{if } \exists 1 \le k \le n \text{ such that } \mathit{last}_{\beta_k} = l_S \\ & \text{and } \mathit{last}_{\beta_i} \ne l_S\ \forall 1 \le i \le k \\ \langle \beta_1, \ldots, \beta_{n-1}, \beta''_n \rangle & \text{otherwise} \end{cases} \\
\beta'_k &= \langle l_1, \ldots, l_m, l_F \rangle \\
\text{where } \beta_k &= \langle l_1, \ldots, l_m, l_S \rangle \\
\beta''_n &= \langle l_1, \ldots, l_p, l_S \rangle \\
\text{where } \beta_n &= \langle l_1, \ldots, l_p, l_F \rangle
\end{aligned} \tag{4.7}$$

Figure 4.7: Inductive cases of the Φ function

2. If no segment of the execution sequence ends with l_S, then the last segment is transformed by replacing its last label by l_S.

Example 4.6 Let us consider the following goal:

$$_{l_1}\mathtt{Y} \Rightarrow -(\mathtt{E}_1, \mathtt{E}_2)_{,l_2} (_{l_3}\mathtt{X==E}_1\ _{l_4}\ ;\ _{l_5}\mathtt{X==E}_2\ _{l_6})_{l_7}$$

where `-/2` is a pair constructor. The Φ function applied to this goal is computed as follows, using the base cases and lines 4.1 and 4.2 of the inductive cases defined in Figure 4.7:

$$\begin{array}{lcl} \Phi(_{l_1}Y \Rightarrow -(E_1, E_2)) & = & \{\langle\langle l_1, l_S\rangle\rangle, \langle\langle l_1, l_F\rangle\rangle\} \\ \Phi(_{l_3}X\text{==}E_1\ _{l_4}) & = & \{\langle\langle l_3, l_4, l_S\rangle\rangle, \langle\langle l_3, l_F\rangle\rangle\} \\ \Phi(_{l_5}X\text{==}E_2\ _{l_6}) & = & \{\langle\langle l_5, l_6, l_S\rangle\rangle, \langle\langle l_5, l_F\rangle\rangle\} \end{array}$$

And thus, using the lines 4.2 and 4.4 of the definition of the inductive cases:

$$\begin{array}{l} \Phi(_{l_3}X\text{==}E_1\ _{l_4}\ ;\ _{l_5}X\text{==}E_2\ _{l_6}) = \\ \quad \{\langle\langle l_3, l_4, l_S\rangle, \langle l_5, l_6, l_S\rangle\rangle, \langle\langle l_3, l_4, l_S\rangle, \langle l_5, l_F\rangle\rangle, \\ \quad \langle\langle l_3, l_F\rangle, \langle l_5, l_6, l_S\rangle\rangle, \langle\langle l_3, l_F\rangle, \langle l_5, l_F\rangle\rangle, \} \\ \Phi(_{l_2}(_{l_3}X\text{==}E_1\ _{l_4}\ ;\ _{l_5}X\text{==}E_2\ _{l_6})_{l_7}) = \\ \quad \{\langle\langle l_2, l_3, l_4, l_7, l_S\rangle, \langle l_5, l_6, l_7, l_S\rangle\rangle, \langle\langle l_2, l_3, l_4, l_7, l_S\rangle, \langle l_5, l_F\rangle\rangle, \\ \quad \langle\langle l_2, l_3, l_F\rangle, \langle l_5, l_6, l_7, l_S\rangle\rangle, \langle\langle l_2, l_3, l_F\rangle, \langle l_5, l_F\rangle\rangle, \} \end{array}$$

Finally, using line 4.3:

$$\begin{array}{l} \Phi(_{l_1}Y \Rightarrow -(E_1, E_2)_{,l_2} (_{l_3}X\text{==}E_1\ _{l_4}\ ;\ _{l_5}X\text{==}E_2\ _{l_6})_{l_7}) = \\ \quad \{\langle\langle l_1, l_F\rangle\rangle, \langle\langle l_1, l_2, l_3, l_4, l_7, l_S\rangle, \langle l_5, l_6, l_7, l_S\rangle\rangle, \langle\langle l_1, l_2, l_3, l_4, l_7, l_S\rangle, \langle l_5, l_F\rangle\rangle, \\ \quad \langle\langle l_1, l_2, l_3, l_F\rangle, \langle l_5, l_6, l_7, l_S\rangle\rangle, \langle\langle l_1, l_2, l_3, l_F\rangle, \langle l_5, l_F\rangle\rangle, \} \end{array}$$

4.2.2 Correspondence between execution sequences and semantics traces

There exists a correspondence between an execution sequence $\langle\beta_1, \ldots, \beta_n\rangle$ for a procedure p and the sequence of traces provided by the semantics of p for a determined input substitution. This correspondence can be simply found by removing the success labels l_S and failure labels l_F from all the segments of the execution sequence.

Example 4.7 Reconsider the nondeterministic `member(out,in)` procedure and the execution sequence defined in Example 4.5:

$$\begin{array}{rl} \mathcal{T} = & \langle\ \langle l_1, l_2, l_3, l_4, l_7, l_S\rangle, \\ & \quad \langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S\rangle, \\ & \quad \langle l_1, l_2, l_5, l_1, l_2, l_5, l_1, l_F, l_F, l_F\rangle\ \rangle \end{array}$$

After having removed the success and failure labels the result is:

$$\mathcal{T} = \langle\langle l_1, l_2, l_3, l_4, l_7\rangle, \langle l_5, l_1, l_2, l_3, l_4, l_7, l_6, l_7\rangle, \langle l_5, l_1\rangle\rangle$$

which corresponds to the complete trace from the semantics of $member(V :: out, W :: in)$ with an input substitution $\theta = \{W/[0,1]\}$, as shown in Example 2.10:

$$\begin{array}{rl} \mathcal{S}[\![\texttt{member(V,W)}]\!]\theta = & \langle(\langle l_1, l_2, l_3, l_4, l_7\rangle, \{V/0, W/[0,1]\}), \\ & (\langle l_5, l_1, l_2, l_3, l_4, l_7, l_6, l_7\rangle, \{V/1, W/[0,1]\}), \\ & (\langle l_5, l_1\rangle, \texttt{Fail})\rangle \end{array}$$

4.3 Adapting control-flow-based adequacy criteria to Mercury

In Section 1.3.1, we defined different control-flow-based adequacy criteria. These criteria are based on the "classical" notion of control flow graph – that is, control flow graphs representing the control flow of programs written in an imperative programming language – and cannot be directly applied to the control flow graph for Mercury we defined in this chapter. We can nevertheless *adapt* the usual control-flow-based criteria in order to fit the different specificities of logic programming, and Mercury in particular, that are taken into account in our definition of control flow graph: failure of goals, multiple solutions and backtracking.
First, the path coverage criterion can be used as is, using the notion of "execution sequence" instead of the notion of "execution path" as follows:

Definition 4.10 A test suite Σ for a procedure $p(\overline{X})$ satisfies the *sequence coverage criterion* iff for each execution sequence $\mathcal{T}$ in $\Phi(p(\overline{X}))$ there exists a test case in Σ such that the execution trace of that test case corresponds to $\mathcal{T}$. $\diamond$

In other words, the above definition means that a test suite satisfies the sequence coverage criterion iff the set of execution sequences representing the executions of each test case comprised in this test suite contains all the execution sequences that can be computed for the procedure under concern. Similarly to the path coverage criterion, the sequence coverage criterion is inapplicable in practice; indeed, if Mercury does not provide the possibility to define loops in a program, predicates can however perform *recursive calls* and *backtracking*, which can both potentially be performed an unbounded number of times. A simple variant of this criterion can be re-used in our context; that is the *block count-K criterion* which is satisfied if, given a natural number K, all the execution paths which can be built such that the number of times each block is visited within each computation does not exceed the given K are executed in the test suite (Albert, Gómez-Zamalloa, and Puebla 2009).
We can also create a variant of the sequence coverage criterion that limits the number of nested recursive calls in a given execution. In order to define such a new criterion, we first define the notion of "call depth" associated to a given execution sequence.

The definition of the *call depth* associated to an execution sequence requires that we first define the call depth associated to an execution *segment*, which itself requires that we define the number of procedure calls and the number of returns from procedure calls in (a sub-sequence of) an execution segment.
The number of procedure calls in a (sub-sequence of an) execution segment β_s is defined as follows:

$$Calls(\beta_s) = \#\{l | l \in \beta_s,\ l' \text{ is the first label of a procedure's body goal}\}$$

That is the number of times the first label of a procedure's body goal (being part of the program in labelled syntax) is reached along the (sub-sequence of an) execution segment Ss.
The number of returns from procedure calls in a (sub-sequence of an) execution segment Ss is defined as follows:

$$Returns(\beta_s) = \#\{l_S | l_S \in \beta_s\} + \#\{l_F | l_F \in \beta_s\}$$

That is the number of times the success label l_S or the failure label l_F is reached along the (sub-sequence of an) execution segment β_s.
The *call depth* associated to an execution segment β is defined as follows:

$$Depth(\beta) = max\{Calls(\beta_s) - Returns(\beta_s) | \beta_s \in Prefix(\beta)\}$$

where $Prefix(\beta)$ is the set of all the prefixes of β.
The *call depth* associated to an execution sequence $\langle\beta_1, \ldots, \beta_n\rangle$ is then defined as follows:

$$Depth(\langle\beta_1, \ldots, \beta_n\rangle) = max\{Depth(\beta_i) | 0 \leq i \leq n\}$$

According to these definitions, the call depth associated to an execution segment is the greatest difference between the number of arcs representing procedure calls and the number of arcs representing returns from procedure calls that have been followed, among all the prefixes of the segment. That concretely represents the number of calls that haven't been returned from yet, at any moment of the straightforward execution represented by that segment. The call depth associated to an execution sequence is the greatest call depth associated to any of the execution segments composing it – that represents the largest size of the return stack that would be used at runtime.
Based on these new notions, the *call-depth-K* coverage criterion can now be defined.

Definition 4.11 Let $p(\overline{X})$ be a procedure, K a natural number and $\mathcal{D}_K$ the largest subset of $\Phi(p(\overline{X}))$ such that $\forall \mathcal{T} \in \mathcal{D}_K$, $Depth(\mathcal{T}) \leq K$. A test suite Σ satisfies the *call-depth-K coverage criterion* iff for each execution sequence $\mathcal{T}$ in $\mathcal{D}_K$ there exists a test case in Σ such that the execution trace of that test case corresponds to $\mathcal{T}$. $\diamond$

This definition means that a test suite satisfies the call-depth-K coverage criterion iff the set of execution sequences representing the executions of each test case comprised in this test suite contains all the execution sequences of the program of which the call depth is less than K.
Another criterion could be based on the number of times backtracks occur in a given execution sequence.

Definition 4.12 Let $p(\overline{X})$ be a procedure, K a natural number and $\mathcal{B}_K$ the largest subset of $\Phi(p(\overline{X}))$ such that $\forall \mathcal{T} \in \mathcal{B}_K$, $\#(\mathcal{T}) \leq K$. A test suite Σ satisfies the *backtrack-K coverage criterion* iff for each execution sequence $\mathcal{T}$ in $\mathcal{B}_K$ there exists a test case in Σ such that the execution trace of that test case corresponds to $\mathcal{T}$. $\diamond$

The backtrack-K coverage criterion is therefore satisfied by a test suite iff the set S of execution sequences representing the executions of each test case comprised in this test suite contains all the execution sequences of the program containing K or less execution segments.

The *statement coverage criterion* needs to be adapted to fit the notion of atom; of particular interest is the fact that an atom, unlike a statement in imperative programs, can possibly fail without necessarily causing the whole execution to fail. One could therefore consider an atom to be covered by an execution if this atom is executed at least once independently from the result of that execution.

Definition 4.13 A test suite satisfies the *atom coverage criterion* iff for each label l directly preceding an atom in the program written in labelled syntax, l appears at least once in one of the execution trace of one of the test cases comprised in this test suite. $\diamond$

One could also consider an atom to be covered if there exists a least one execution during which this atom fails, and at least one execution during which this atom succeeds. We can notice that if that condition holds for all the atoms, that simply means that all the regular arcs of the program have been followed at least once during the execution of the test suite; this notion is therefore similar to the notion of *branch coverage*.

Definition 4.14 A test suite satisfies the *arc coverage criterion* iff for each regular arc (l, l') in the control flow graph of the program, there exists at least one test case comprised in this test suite, the execution trace of which contains l' directly followed by l. $\diamond$

Note that, similarly to their counterparts in imperative programming, atom coverage and arc coverage criteria are applicable only if all the labels preceding atoms in the program in labelled syntax, respectively all the regular arcs of the control flow graph, are reachable.
Let us finally introduce the *procedure* coverage criterion that aims at verifying if every procedure is called.

Definition 4.15 A test suite satisfies the *procedure coverage criterion* iff for each label l which is the first label of a labelled procedure body, there exists at least one execution segment in one of the execution sequences representing the executions of all the test cases comprised in this test suite, in which l appears. $\diamond$

4.4 Using the control flow graph in coverage measurement

In Chapter 3, we described a test automation framework for the Mercury language. This framework is able to automatically run test suites that were previously created. However, a test framework can also have additional features, which are not necessarily needed for the testing itself; a module could for example interact with an integrated debugger in order to try to identify the code fragment that caused the failure of a given test case (Ducassé and Emde 1988). It this work, we show how the labelled syntax and the control flow graph defined in this chapter can be used to create a coverage tool, i.e. a tool which is able to produce a measure describing the degree to which the source code of the program has been exercised during the execution a given test suite.

This measure is performed with respect to one or more coverage criteria; the coverage criteria used in this section were defined in Section 1.3 and adapted for Mercury and its control flow graph in Section 4.3.

This tool is a complementary module for the base framework that helps to detect parts of the tested code that are not covered by a given set of test cases, for a given coverage criterion. Logic programming languages have a few peculiarities that must be taken into account when constructing a coverage tool. The most important of these is nondeterminism, the fact that statements can fail and/or have multiple solutions. Most coverage tools for declarative languages transform the original program to an instrumented code and place some kind of execution counters before and after calls. This enables tracing calls to and exits from procedures. The counters are usually stored in a non-declarative way, like in the Haskell Program Coverage tool (Gill and Runciman 2007), which records every increment into a file. This is unavoidable in case of logic programming languages, since after backtracking, all changes made on pure declarative variables would be revoked. The same principle is used in the `coverage` library for ECLiPSe Prolog, the output of which is a simple HTML page, where the counter values are shown between the atoms under concern.

Our tool follows the same principles as these tools, but needs to deal with some particularities of the Mercury language. The most notable one is the mode-reordering mechanism of the Mercury compiler, as it strongly affects control-flow based coverage criteria. Another issue that is worth mentioning is the way in which the Mercury compiler treats switches. In the next section, we expose these different issues and present the solutions we developed.

4.4.1 Implementation

As usual, we assume that the programs are well-moded; this condition is checked during compilation by the Mercury compiler. As explained in chapter 2, the latter also re-orders the goals in such a way that they are executed from left to right and multi-moded predicates are transformed into several different procedures, in a so-called superhomogenous form.
Our implementation instruments the examined code into an instrumented one, compiles it and executes it in order to log execution information; the process of coverage measuring is shown in Figure 4.8. The base idea of the transformation is to add counters in the code, implemented by logging calls that write unique identifiers – the labels encountered during execution – into a log file. The counters are placed with respect to the labelled Mercury syntax defined in Section 2.1.5. A counter is assigned to each label l, denoted by counter(l). Basically this means that counters are inserted into every possible place between goals, as well as at the beginning and at the end of a conjunction and a disjunction.

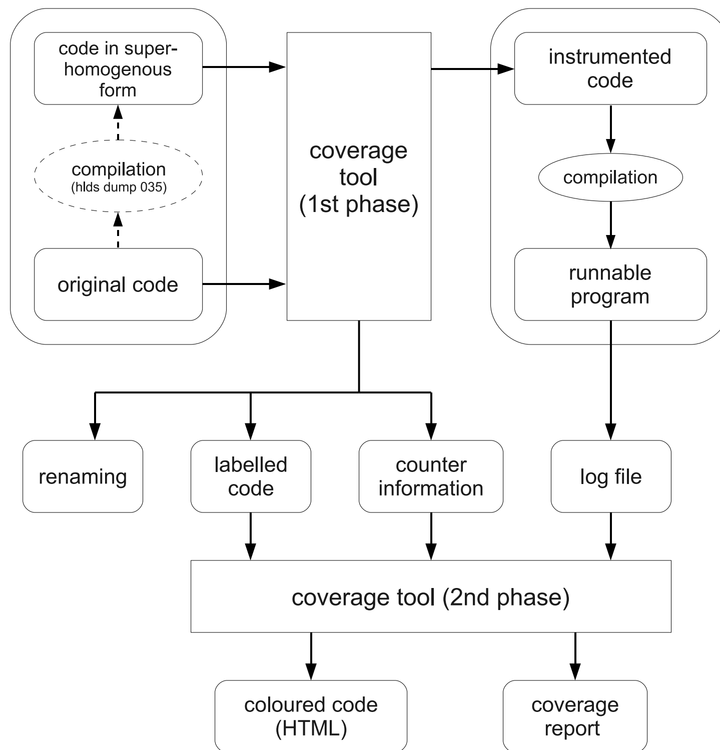


Figure 4.8: Coverage tool

The first step of the transformation process is to transform the code in superhomogeneous form. A part of this process can be achieved by the compiler

(goal reordering, duplication of predicates with multiple modes); however all the multi-moded predicates need to be renamed, in such a way that every procedure is associated to a unique name. Every call to the procedures must therefore be renamed consequently; this can be done using a simplified mode analysis, propagating the instantiations of variables throughout the code. The new name assignments are saved into a file, in order to provide names mapping information to the user at the end of the process.
The second step of the transformation is the addition of logger calls with respect to the labelled Mercury syntax; these calls reify incrementing operations on counters: $\log(l) \equiv \{\text{counter}(l) := \text{counter}(l) + 1\}$.

Example 4.8 Let us examine again the `member(out,in)` procedure. Its labelled syntax, defined in Example 2.9, is the following:

$$\texttt{member}(\texttt{X} :: \texttt{out}, \texttt{Y} :: \texttt{in}) :- \,_{\texttt{l}_1}\texttt{Y} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}],_{\texttt{l}_2} (_{\texttt{l}_3}\texttt{X} := \texttt{E}_{\texttt{l}_4} \;;\; _{\texttt{l}_5}\texttt{member}(\texttt{X}, \texttt{E}_\texttt{s})_{\texttt{l}_6})_{\texttt{l}_7}.$$

The instrumented code derived from that labelled syntax is simply:

$$\begin{aligned} \texttt{member}(\texttt{X} :: \texttt{out}, \texttt{Y} :: \texttt{in}) :- \quad & \texttt{log}(\texttt{l}_1), \texttt{Y} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}], \\ & \texttt{log}(\texttt{l}_2), \\ & (\texttt{log}(\texttt{l}_3), \texttt{X} := \texttt{E}, \texttt{log}(\texttt{l}_4) \;; \\ & \texttt{log}(\texttt{l}_5), \texttt{member}(\texttt{X}, \texttt{E}_\texttt{s}), \texttt{log}(\texttt{l}_6)), \\ & \texttt{log}(\texttt{l}_7). \end{aligned}$$

Once the code has been successfully instrumented, using the coverage tool as a part of the testing framework is easy: in the test suite file, we simply refer to this transformed code instead of the original one.
The direct output of executing instrumented code is, in addition of the usual output of the program, a log file that contains information about reached program points. A log entry means that the execution reached the corresponding point (label) of the program. For each executed test case, a sequence of labels is created, that contains all the labels encountered during an execution. As such, this sequence corresponds to the *complete trace* defined in Chapter2.

Example 4.9 Let us have a look at an execution trace for the `member(out,in)` procedure, defined in Example 2.9 and instrumented with logger calls in Example 4.8. The trace created by the logging calls encountered along the execution described in Example 4.5 (page 58) is the following:

$$\langle \texttt{l}_1, \texttt{l}_2, \texttt{l}_3, \texttt{l}_4, \texttt{l}_7, \texttt{l}_5, \texttt{l}_1, \texttt{l}_2, \texttt{l}_3, \texttt{l}_4, \texttt{l}_7, \texttt{l}_6, \texttt{l}_7, \texttt{l}_5, \texttt{l}_1 \rangle$$

4.4.2 Switches vs. disjunctions

Unfortunately, the addition of logger calls in the source code can render the program not compilable if it contains *switches*. A switch is a special disjunction – with nothing visually distinguishing it from a "regular" disjunction –, in which "each disjunct has near its start a unification that tests the same bound variable

against a different function symbol" (Henderson, Conway, Somogyi, Jeffery, Schachte, Taylor, Speirs, Dowd, Becket, and Brown 1996, Mercury Reference Manual). In the remaining, we call such unifications the *switch conditions* and the variable the *switch variable*. In a single switch, the switch conditions are mutually exclusive; this allows the compiler to consider the switch as being deterministic or semi-deterministic – depending on whether every possible value of the switch variable is covered – whereas regular disjunctions are, in general, non- or multi-deterministic. Switches can be nested into each other and if they test the same variables, they are treated as a single switch.

```
...,
(
  X => f,
  p(Out)
;
  Y := X,
  (
    Y => g,
    I = 42
  ;
    Z := Y,
    Z => h(Arg),
    q(Arg, I)
  ),
  r(I, Out)
),
...
```

```
(
  log(1),
  X => f,
  log(2),
  p(Out),
  log(3)
;
  log(4),
  Y := X,
  log(5),
  (
    log(6),
    Y => g,
    log(7),
    I = 42,
    log(8)
  ;
    log(9),
    Z := Y,
    log(10),
    Z => h(Arg),
    log(11),
    q(Arg, I),
    log(12)
  ),
  log(13),
  r(I, Out),
  log(14)
)
```

Figure 4.9: Naive instrumentation of a switch

The reason switches are considered as particular structures is to allow the compiler to perform a determinism analysis and produce highly optimised code. In order to be recognised as a switch, only unifications can precede switch conditions in the different disjuncts; if not, the compiler is not able to detect the

switch conditions, and therefore considers the disjunction under concern as a regular (*nondet* or *multi*) disjunction. When logger calls are inserted at the beginning of a disjunct, a switch will therefore be considered as a regular disjunction, which can cause the compilation to fail if the enclosing predicate is declared (semi-)deterministic. The example shown at the left side of Figure 4.9 is extracted from the Mercury reference manual (Henderson, Conway, Somogyi, Jeffery, Schachte, Taylor, Speirs, Dowd, Becket, and Brown 1996): it is a switch on X, provided X is an input variable. On the right side is the "naively" instrumented version with the logger calls (in this example, the labels logged are represented by natural numbers); this instrumented code would cause the program to fail at compiling if it is used in a (semi-) deterministic predicate.
The solution we developed is to replace, in a disjunct, logger calls before each unification at the beginning of a disjunct by a single logger call *after* the unifications – just before the first predicate call occurring in the disjunct. This single logger updates all the counters in order to reflect success or failure of all the preceding disjuncts.
However, we need to pay a particular attention to the fact that this work-around should result in the creation of a log file that is the same as it would have been if the counters were placed with respect to the labelled Mercury syntax (assuming that the issue with switches described in the current section did not exist). In particular, if one of the unifications preceding the logger call fails during execution, it is a priori not possible to know which one it was (since no counter was placed between them) and then the coverage information is incomplete.
We take care of those issues by performing a small analysis before entering a switch. The idea is to create a tree representing the switch before executing the switch itself. Knowing the values of the variables at the beginning of the switch, this tree is used to compute the runtime trace through the switch, in order to write the correct log entries in batches after the switch has been executed. The algorithm that determines which counter needs to be updated at which point is presented below (its basic steps are shown on Figure 4.11). The modelling of the switches under the form of a tree is particularly convenient since switches can be nested into each other. Nodes of the tree are the labels of the program written in labelled syntax, while its edges are the unifications positioned between the labels. All the statements after the first predicate call of each disjunct are dropped from the model, so the leaves of the tree are the labels preceding the first predicate call in each disjunct. Complex statements, like conditional structures, etc. are treated as if they were predicate calls, and are thus also dropped from the model. If there are only unifications in a disjunct, then the leaf of the corresponding branch is the last label of the disjunct. We define a *switch path* as the sequence of labels that is the output of a depth-first label traversal of the corresponding model graph. The model of the example of Figure 4.9 is shown in Figure 4.10, while the corresponding switch path is $\langle 1, 2, 4, 5, 6, 7, 8, 9, 10, 11 \rangle$.
When executing a switch, the edges of its model graph are examined by the analysis in order to determine if the corresponding unifications result in success or failure. The examination begins at the first node of the first branch of the graph. If a unification succeeds, then its successor node is marked and

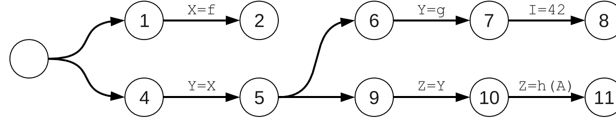


Figure 4.10: Switch execution graph

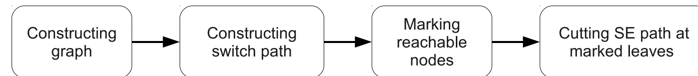


Figure 4.11: Switch transformation

the next edge of the branch is examined, otherwise the examination of that branch is stopped. After this step, the nodes that are not marked correspond to program points that are not reached on the examined program state. The marked nodes are visited using a depth-first search; when a leaf is reached, the sequence of marked nodes encountered on the path up to this leaf is stored and associated to the leaf. This process is repeated starting from the next unvisited marked node until no marked node is left unvisited. Each sequence of marked nodes associated to the leaves is then transformed into a special logger call, the effect of which is to log the sequence of labels corresponding to the sequence of marked nodes. Since the leaves correspond to program points at the end of the switches disjuncts, these special logger calls can be placed at those program points without preventing the compiler to detect the switches conditions.

Example 4.10 Using the example from Figure 4.10 again, let us assume that X is bound to g. In that case, the unifications $Y = X$, $Y = g$, $I = 42$ and $Z = Y$ succeed, while $X = f$ and $Z = h(A)$ fail. The sequence assigned to leaf 8 would be, in that case, $\langle 1, 4, 5, 6, 7, 8 \rangle$. No sequence is assigned to leaf 2 nor leaf 11. As is, we have no trace of what happened within the tree segment from node 9 to node 11.

We now have to decide what to do when the last node (or the few last nodes) of the last segment of the switch path is not marked – as it is the case for nodes 9, 10 and 11 in Example 4.10. Indeed, in that case, the last segment is not associated to any logger call, and we thus keep no track of which atom succeeds or fails within that segment. In order to overcome this, these entries are logged at the same time as the previous batch logging action, or if there is no successful branch, they are logged before the execution of the switch. In Example 4.10, the only batch will therefore be $\langle 1, 4, 5, 6, 7, 8, 9, 10 \rangle$ associated to leaf 8. In the case of a failure of all the unifications, the sequence would be $\langle 1, 4, 9 \rangle$ and would be logged before the execution of the switch.

4.4.3 Computing the coverage rate with respect to coverage criteria

As we can notice by examining Examples 2.10 and 4.9, the *complete trace*[1] of a procedure computed by the semantics function and the trace collected thanks to the logger calls during execution of instrumented code of the procedure are identical, modulo the translation from a label to the corresponding log. That is because the logger calls are placed in the source code with respect to the labelled Mercury syntax used to define the Mercury semantics (see Section 2.1.5) – except for switches structures, but the mechanism we set up has the ability to mimic properly the good placement of the logger calls.

Example 4.11 The complete trace for member(out,in) computed from the semantics given in Examples 2.10 is:

$$\langle l_1, l_2, l_3, l_4, l_7, l_5, l_1, l_2, l_3, l_4, l_7, l_6, l_7, l_5, l_1 \rangle$$

The runtime trace for member(out,in) resulting from the logger calls during its execution and given in Examples 4.9 is identical.

Also, as illustrated in Example 4.7, there exists a correspondence between the execution sequences computed from the control flow graph of a procedure and the traces from the semantics of that procedure. Since runtime traces and the traces from the semantics have also a correspondence, there exists a correspondence between a runtime trace and an execution sequence, that represents that same execution.
Thanks to this correspondence, it is therefore possible to use the coverage tool in order to compute a coverage rate for different coverage criteria, in particular for those defined in Section 4.3. First, the set of execution sequences that should be followed in order to satisfy a coverage criterion is derived from the graph. Each execution sequence can then be transformed into the corresponding trace by substituting each label by the corresponding log, resulting in the creation of a set of traces (we call it the *synthetic* set of traces). Then, the test suite that we wish to evaluate is executed using the program previously instrumented with the different logger calls. The result of this execution is another set of traces (called the *actual* set of trace); the latter is compared to the synthetic set of traces. If a trace appears in the synthetic set and not in the actual set, it means that this trace is not covered by the test suite, and that a test case causing this trace to be logged should be added to the test suite. The other way round, if a trace appears only in the actual set of traces, it means that the test case that caused that trace to be created is unnecessary with respect to the coverage criterion. It can also happen that the same trace appears twice in the actual set; in that case, one of them (if the same trace is in the synthetic set) or both are unnecessary.

[1] Recall that the complete trace of a procedure is the concatenation of all the sub-traces given by the semantics.

4.4.4 Graphical visualization of the atom coverage and arc coverage criteria

The file containing the actual set of traces is hardly readable, particularly if the set of traces or the traces themselves are large. Of course, it is possible to provide a human-readable feed-back about the result of the coverage evaluation: the (number of) traces that were not covered and the (number of) unnecessary traces for example. The control flow graph can prove itself useful in that context, in order to provide a visualization of those different test cases. It is also possible to provide visual feedback by colouring the source code in order to denote which atoms/arcs were followed during the test phase, similarly to what is done by the coverage library of ECLiPSe prolog. Our implementation of this functionality makes use of different sources of information: the program in labelled syntax, the correspondence between the labels and the log entries in the instrumented code, and a file containing meta-information about counters. This meta information consists of the correspondences between pairs of counters and the points of interest of the programs (atoms or more complex structures such as disjunctions). One additional pair of counters is added for each predicate; they are the first and the last counters of the predicate. The log file is generated at runtime, while the others are created at the same time as the instrumentation. The output of our coverage tool is a html file containing the source code to which colours have been added. Those colours are able to denote atom and arc coverage criteria at the same time, using three different colours.

Green If an atom is colored green, all the regular arcs originating from the label that precedes, in the labelled program, an atom coloured green have been followed (at least once) during the execution of the test suite.

Yellow An atom coloured yellow was executed – the label preceding it in the labelled source code was reached at least once – but one of the regular arcs originating from that label in the graph was not followed during the execution of the test suite (note that at most two arcs originate from a label, one denoting the success and one denoting the failure of the atom preceded by this label in the labelled program).

Red An atom coloured red was not executed during the execution of the test suite – the label preceding it was not reached.

The tool also produces a detailed report which enumerates all the pairs of counters, and gives the coverage degree of each atom. Although it is less visual than the html rendering, this report contains more information, since it also gives the coverage degree of complex structures (disjunctions, switches) and predicates (procedure coverage).

4.4.5 Evaluation

Table 4.1 illustrates the performance of the coverage tool. The examined properties are the size of the instrumented code compared to the size of the original

source code, the time needed for instrumentation and the execution overhead caused by the transformation. The table also shows the execution times for both the instrumented and non-instrumented code. The tested procedures are the same as those in Table 3.1 with the additionnal `filter_list`, the latter being a procedure from the code of the test framework itself that does list filtering by a given set of indices. The input parameters are chosen relatively large in order to produce measurable times (lists of a few hundred to few thousand elements).

Goals	Code size (lines)		Instrumentation	Execution (ms)	
	orig.	instr.	(ms)	orig.	instr.
member(in,in)	19	124	32	13	2050
member(out,in)				42	4460
bubblesort(in,out)	38	177	53	11	7130
transpose(in,out)	58	340	96	3.3	1520
filter_list(in,in,out)	37	191	60	18	4760

orig.=original, instr.=intrumented

Table 4.1: Performance of the coverage tool

Since counter update occurs between goals, the size of the instrumented code (in number of lines) is approximately twice the size of the original code. However, when the switch transformation is applied, additional lines are added for every switch test statement, but in any case the size of the instrumented code is limited to a few times the size of the original one. However, as can be seen from Table 4.1, the execution time overhead of the instrumented code can be significant. This can be partially explained by the overhead due to the logging operations, and partially by the fact that the compiler is no longer able to perform a number of optimisations. Nevertheless, it should be noted that the execution overhead is only present when one is measuring test case coverage, and not when a the test suite is executed for testing.

Chapter 5

Test data generation for Mercury

In the previous chapters we have shown – among other things – how we can derive, from a Mercury procedure, the symbolic representations of (a subset of) all the possible executions of that procedure. Such a symbolic representation is called an *execution sequence*, and effectively represents the derivation tree of the program using particular input data (see Section 4.2). An execution sequence consists of a sequence of *execution segments*, each of them representing a branch of the derivation tree, from the root to the leaf. A segment is a sequence of *labels*, whereas the labels are arbitrary unique values introduced in the Mercury source code of the procedure in order to identify the different points of interest of that code (see Section 2.1.5). As explained in Chapter 1, the goal of the present work is to use symbolic execution in order to derive a test suite satisfying a given coverage criterion. As described in Section 1.5.1, the result of the symbolic execution of the program along a chosen execution path is the production of a path condition, i.e. symbolic formulas – constraints – over input variables, such that the input values satisfying those formulas would cause the program to follow the chosen execution path. Similarly, we define in this section the notion of *sequence condition*, that is the notion of path condition adapted to the Mercury execution sequences. The results presented in this chapter were published in (Degrave, Schrijvers, and Vanhoof 2008) and (Degrave 2008).

5.1 A brief introduction to Constraint Programming

Constraint programming is not the main topic of this work. However, the use of this technique is not negligible as it permits the translation of a symbolic execution path – or sequence – into a set(s) of values causing this execution path (sequence) to be followed by the procedure under concern when used as input

data. The most important principles of constraint programming must therefore be understood in order to understand the complete test data generation process that we developed.

5.1.1 CSP, propagation and search

We already evoked in Chapter 1 the notion of Constraint Satisfaction Problem (CSP) which is a triple (Z, D, C) where Z is a finite set of variables, D is a function which maps every variable in Z to a set of objects of arbitrary type (the *domain* of the variables) and C is a finite (possibly empty) set of constraints on an arbitrary subset of variables in Z, and we noted that a solution to a CSP is a simultaneous assignment of values (from the respective domains of each variable provided by D) to the whole set of variables Z in such a way that the constraints in C are satisfied (Tsang 1993).

Propagation

Constraint propagation is a very general concept that is referenced under different names depending on periods and authors. It comprises any reasoning which consists in forbidding assignments or combinations of assignments for variables of a CSP that would prevent a subset of the constraints to be satisfied (Bessiere 2006). This notion has already been evoked in Section 1.5.1 under the name of *domain reduction* as it results in reducing the domain of the variables. For example, if a variable X of the CSP has a domain of $\mathbb{N}$ – that is the set of natural numbers – and there is a constraint $X \leq 3$ in C, the domain of X can be reduced to $\{1, 2, 3\}$ and the other values of $\mathbb{N}$ can be discarded. Such domain reduction techniques based on unary constraints on variables are called *node-consistency techniques.* Of course, set of constraints generally are not limited to unary constraints, and many constraints usually involve two or more variables. A simple example of such a constraint is $X < Y$. If the domains of both X and Y is $\{1, 2, 3\}$, we can observe that X can never be 3 and reduce the domain of X to $\{1, 2\}$ in order to ensure the consistency of the CSP – note that consistency involving pairs of variables is called *arc consistency* and has received much attention in the literature (e.g. (Mackworth 1977; Van Hentenryck 1987; van Dongen 2003; Bessière, Régin, Yap, and Zhang 2005)). If the constraints contain more than two variables, one usually refers to arc consistency under the terms of *hyper arc consistency* or *generalized arc consistency.*

Search

The main algorithmic technique to solve CSPs is *search*(Marriott and Stuckey 1998). A search algorithm can either be complete or incomplete. Complete search algorithms (also called systematic search algorithms) guarantee that a solution will be found if one exists, and can be used to show that no solution exists otherwise. Incomplete search algorithms don't come with the guarantee of finding a solution if one exists, but they are however often effective at finding a

solution. An example of complete search algorithms are *backtracking* algorithms, and an example of incomplete search algorithms are *local search* algorithms (Rossi, van Beek, and Walsh 2006).

Backtracking Backtracking search is realised by a depth-first search traversal of a search tree (Davis, Logemann, and Loveland 1962; Golomb and Baumert 1965). This search tree is generated as the search progresses. The root node of this tree is the empty set, and the children of a node are sets obtained by extending the parent node. The method of extending a node in a search tree is called the *branching strategy.* When the most simple branching strategy is used, a node at level j is a set of j variables assignments for variables $x_1, ..., x_j \in Z$. The outgoing branches of a node n are created by selecting an unassigned variable x_{j+1} from Z and one new outgoing branch is created for each possible value of its domain $D(x_k)$. There are therefore as many children of n as values in the domain $D(x_k)$, and each such child is obtained by extending n with a new assignment $x_k = a$, $a \in D(x_k)$. The search algorithm explores (and builds at the same time) this search tree starting with the empty set; when the current set is extended with a new assignment $x_k = a$ – the assignment $x_k = a$ is said to be *posted* in that case – the algorithm checks if all the constraints are satisfied with the newly extended set of assignments. If not, the next value from the domain $D(x_k)$ is assigned to x_k, and a new branch of the search tree is thus created. Note that another usual branching strategy is to create two branches $x_k = a$ and $x_k \neq a$ for each value $a \in D(x_k)$. Since the first uses of backtracking in computing, many techniques have been created to improve its efficiency. A very important and widely used one is *search and propagation interleaving* (Davis and Putnam 1960; Gaschnig 1974; Mackworth 1977). The principle is to perform constraint propagation at each node in the search tree. This can have very important benefits, such as a significant pruning of the search tree. Indeed, if the propagation results in reducing a variable domain to the empty set, there is no need to continue exploring the current branch since no solution can be found. Likewise, if a single value remains in the domain of a variable, the assignment to this value can be forced and there is no need to branch on this variable in the future. Another effective technique improving the performance of backtracking is the automatic creation a of *implied constraints*, i.e. constraints that, if added to a CSP, don't change the set of solutions for this CSP. Adding such constraints can enable an early detection of deadends in the search tree, and can possibly help pruning the latter if used in combination with search and propagation interleaving (Rossi, van Beek, and Walsh 2006). There are different techniques for adding implied constraints; some of them add the constraints *before* an inconsistency is encountered in the search. Others add the constraints *after* an inconsistency is encountered. In the latter case, the added constraint is a set of assignments that is not consistent with any solution. It is added with the hope that it will help pruning the search tree in the future. Such special type of implied constraints are often referred to as *nogoods* (Stallman and Sussman 1977; Dechter 1986; Schiex and Verfaillie 1993; Ginsberg 1993).

Local search In local search techniques, the nodes of the search graph all represent complete assignments of the variables, unlike the search trees used in backtracking techniques in which only the leaves of the tree represent complete assignments. Each node of such a graph thus represents a set in which each variable of the CSP is assigned to a value of its domain, and is assigned to a *cost value* given by a *cost function* – for example, this function could return the number of constraints violated by the assignment. The search consists in moving from one node to one of its neighbours (provided by a “neighbourhood” function), in search for a node with a lower cost value (Pesant and Gendreau 1996). The cost function chosen will depend on the goal of the search. Indeed, a local search technique can be used to find a solution that satisfies a CSP, in which case the cost function must reflect the satisfaction of the constraints using the assignment represented by the node (such as our previous example of a function returning the number of constraints violated by this assignment). When a local search technique is used to find optimized solutions for a CSP the function must reflect the quality of the solution.

5.1.2 Constraint Programming and Constraints Solving

CLP Constraints are a notion that can be embedded in any programming language. However, some categories of languages are more adapted than others to deal with constraints solving. That is the case of declarative programming languages and particularly logic programming languages (Frühwirth 1998; Rossi, van Beek, and Walsh 2006). Indeed, constraints and logic programming have in common that they do not specify a step or sequence of steps to execute (as it is the case in imperative programming language), but rather the properties of a solution to be found. Constraints can be thus seen as relations (predicates); moreover their conjunction can be seen as a logical “and”, and backtracking is a very common methodology to solve them. They can therefore naturally be embedded in logic programming languages. The resulting languages are called *Constraint Logic Programming* (CLP) languages and have been widely studied (Jaffar and Lassez 1987; Jaffar and Maher 1994; Van Hentenryck 1989; Frühwirth 1992). Syntactically, constraints are added by allowing constraints of chosen types (such as linear equations over real values) to be represented as atoms in the body of a clause. Constraint Logic Programming can be seen as a generalization of logic programming, as *unification* itself can be regarded as a simple form of constraint solving (solving equations over first-order terms) (Van Hentenryck 1989). When executing a program written in a CLP language, two solvers are thus involved: unification and the specific solver for the type of constraints used. The built-in depth-first backtracking search of logic programming is used, interleaved with propagation steps. Choosing a given CLP language means choosing a specific class of constraints to solve; for example, CLP over finite domain (usually noted CLP(FD)) languages use solvers able to deal with variables having possible values taken from a finite domain.

Constraint Handling Rules During the early ages of CLP it was not possible to modify a solver, or to write a new solver in order to deal with a new domain of values. Solvers were written in a low-level language and integrated to the host logic programming engine. In order to tackle this issue, a high-level language extension called *constraint handling rules* was designed in 1991 by Früwirth – usually referred to as CHR (Frühwirth 1992; Frühwirth 1998; Abdennadher, Frühwirth, and Meuss 1999; Frühwirth 2009). CHR allows the programmer to add user-defined constraints into a given host language (Prolog for example). A CHR program consists of guarded rules defining how the constraints must be propagated. There exist essentially two kinds of rules: *simplification* rules define how to replace constraints by simpler ones while preserving logical equivalence. *Propagation* rules define how new constraints can be added, which are logically redundant but can lead to further simplification. In practice, a third kind of rule exists called *simpagation* rules; such a rule is in fact a combination of a simplification and a propagation rule (Frühwirth 1998; Sneyers, Van Weert, Schrijvers, and De Koninck 2010).
Syntactically, a simplification rule is of the form:

$$H_1, \ldots, H_n \ \texttt{<==>}\ G_1, \ldots, G_j \mid B_1, \ldots, B_k\,.$$

A propagation rule is of the form:

$$H_1, \ldots, H_n \ \texttt{==>}\ G_1, \ldots, G_j \mid B_1, \ldots, B_k\,.$$

And a simpagation rule is of the form:

$$H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_n \ \texttt{<==>}\ G_1, \ldots, G_j \mid B_1, \ldots, B_k\,.$$

with $i > 0$, $j \geq 0$, $k \geq 0$, $l > 0$. $H_1, \ldots, H_n$ is a conjunction of CHR constraints called the *head* of the rule. This rule is said to be *n-headed*, and is *multi-headed* if $n > 1$. The sequence $G_1, \ldots, G_j$ is called the *guard* and is a conjunction of constraints written in the host language. The *body* $B_1, \ldots, B_k$ is a sequence of built-in and CHR constraints. Empty sequences can be represented as the constraint `true`; an empty guard `true` can be removed, together with the commit operator $|$.
When applied during the resolution of a CSP (Z, D, C), a simplification rule *replaces* in C the head constraints with the body constraints (which are intended to be simpler than the head constraints) under the condition that the guard is satisfied. That is why a simplification rule uses a double arrow, to indicate that the head and the body are logically equivalent.
A propagation rule *adds* the body constraints to C and still keeps the head constraints, if the guard holds. Since the body is implied by the head, it obviously contains constraints that are logically redundant with the head constraints. However, those new constraints could possibly cause other simplification rules to be applied later on.
A simpagation rule is a combination of a simplification and a propagation rule; the head constraints that appear before the backslash $H_1, \ldots, H_l$ are kept, while the other ones are removed.

5.2 Segment conditions and sequence conditions

Since Mercury programs deal with both symbolic and numeric data, we will now consider two types of constraints: *symbolic constraints* which are either of the form $x = f(y_1, \ldots, y_n)$ or $x \not\Rightarrow f$ (with $x, y_1, \ldots, y_n$ being variables and f a functor) and *numerical constraints* which are of the form $x = y \oplus z$ (with x, y, z being variables and $\oplus$ an arithmetic operator). The constraint $x \not\Rightarrow f$ denotes that the variable x cannot be deconstructed into a term of which the functor is f. Formally, that means $\forall \overline{y}: \ x \neq f(\overline{y})$ with $\overline{y}$ a vector of variables. Furthermore we consider constraints of the form $x = y$ and $x \neq y$ that can be either symbolic or numeric. Note that as a notational convenience, constraint variables are written in lowercase in order to distinguish them from the corresponding program variables.
In the remaining we assume that, in the control flow graph, edges originating from a label associated to an atom are annotated as follows: in case of a predicate call the edge is annotated by the call itself; in case of a unification it is annotated by the corresponding constraint, depending on the kind of atom and whether it succeeds or fails, as follows:

source program	$l, l'(\equiv$ success$)$	l, l'' with $l' \neq l''(\equiv$ failure$)$
$_l X := Y_{l'}$	$x = y$	not applicable
$_l X == Y_{l'}$	$x = y$	$x \neq y$
$_l X <= f(Y_1, \ldots, Y_n)_{l'}$	$x = f(y_1, \ldots, y_n)$	not applicable
$_l X => f(Y_1, \ldots, Y_n)_{l'}$	$x = f(y_1, \ldots, y_n)$	$x \not\Rightarrow f$
$_l X := Y \oplus Z_{l'}$	$x = y \oplus z$	not applicable

In order to collect the constraints associated to an execution segment – that we will refer to as the *segment condition* – the basic idea is to walk the segment and collect the constraints associated to the corresponding edges. However, the constraints associated to each (sub)sequence of labels corresponding to the body of a call need to be appropriately renamed. Therefore, we keep a *sequence* of renamings during the constraint collection phase, initially containing a single renaming (possibly the identity renaming). Upon encountering an edge corresponding to a predicate call, a fresh variable renaming is constructed and added to the sequence. It is removed when the corresponding return edge is encountered. As such, this sequence of renamings can be seen as representing the call stack, containing one renaming for each call in a chain of (recursive) calls.

Definition 5.1 Let E denote the set of edges in a control flow graph and let $\beta = \langle l_1, \ldots, l_n \rangle$ be an execution segment for a procedure p. Given a sequence of renamings $\langle \sigma_1, \ldots, \sigma_k \rangle$, we define the *segment condition* associated to β as $\mathcal{U}(\langle l_1, \ldots, l_n \rangle, \langle \sigma_1, \ldots, \sigma_k \rangle)$ returning the set of constraints C defined as follows :

1. if $(l_1, l_2) \in E$ and $(l_1, l_v)^{rs} \notin E$ then let c be the constraint associated to the edge (l_1, l_2). We define $C = \{\sigma_1(c)\} \cup \mathcal{U}(\langle l_2, \ldots, l_n \rangle, \langle \sigma_1, \ldots, \sigma_k \rangle)$

2. if $(l_1, l_2) \in E$ and $(l_1, l_v)^{rs} \in E$ then let $p(X_1, \ldots, X_m)$ be the call associated to the edge (l_1, l_2). If $head(p) = p(F_1, \ldots, F_m)$ then let γ be a new renaming mapping f_i to x_i (for $1 \leq i \leq m$), and mapping every variable occuring free in $body(p)$ to a fresh variable. Then we define $C = \mathcal{U}(\langle l_2, \ldots, l_n \rangle, \langle \gamma, \sigma_1, \ldots, \sigma_k \rangle)$.

3. if $l_1 = l_S$ or $l_1 = l_F$, then we define $C = \mathcal{U}(\langle l_2, \ldots, l_n \rangle, \langle \sigma_2, \ldots, \sigma_k \rangle)$.

Furthermore, we define $\mathcal{U}(\langle\rangle, \langle\rangle) = \emptyset$. $\diamond$

Note that the three cases in the definition above are mutually exclusive. The first case treats a success or failure edge associated to a unification. It collects the corresponding constraint, renamed using the *current* renaming (which is the first one in the sequence). The second case treats a success arc corresponding to a procedure call, by creating a fresh renaming γ and collecting the constraints on the remaining part of the segment after adding γ to the sequence of renamings. As such, the newly created renaming γ will be used as the current one when collecting the constraints associated to the called predicate's body. The third case, representing a return from a call, collects the remaining constraints after removing the current renaming from the sequence of renamings such that the remaining constraints are collected using the same renamings as those before the corresponding call.

Example 5.1 Let us reconsider the procedure `member(in,in)` and the execution segment $s' = \langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S \rangle$ given in Example 4.4. If we assume that id represents the identity renaming and that, when handling the recursive call at l_5, the constraint variables e and es, corresponding to the local variables of `member`, are renamed into e' and es', we have

$$\mathcal{U}(s', \langle id \rangle) = \{y = [e|es], x \neq e, es = [e'|es'], x = e'\}.$$

As can be seen from Example 5.1, the set of constraints associated to an execution segment s defines the *minimal* instantiation of the procedure's input variables so that the execution is guaranteed to proceed as specified by s. In case of Example 5.1 we have $y = [e, x|es'] \wedge x \neq e$. Indeed, whatever (type correct) further instantiation we choose for the variables x, e and es', as long as the above segment condition is satisfied, the execution of $member(x, y)$ is guaranteed to follow the execution segment s. A test input can thus be computed for a given execution segment by solving the associated set of constraints and further instantiating the free variables by arbitrary values, as long as the instantiation remains type correct.
To collect the constraints associated to an execution sequence, that we call the *sequence condition*, it suffices to collect the constraints associated to each individual execution segment using an appropriate initial renaming in order to avoid nameclashes.

Definition 5.2 Let $\overline{S} = \langle s_1, \ldots, s_n \rangle$ denote an execution sequence for a procedure p. The *sequence condition* associated to $\overline{S}$, denoted $\mathcal{C}\ (\overline{S})$, is defined

as

$$\mathcal{C}\ (\langle s_1, \ldots, s_n \rangle) = \bigcup_{1 \leq i \leq n} \mathcal{U}(s_i, \sigma_i)$$

where each σ_i is a renaming mapping each non-input variable of p to a fresh variable name. $\diamond$

The initial renamings do not change the name of the procedure's input variables. Indeed, since each segment represents a different derivation for the *same* input values, *all* constraints on these values from the different segments must be satisfied.

Example 5.2 Let $\overline{S} = \langle S_1, S_2, S_3 \rangle$ be the execution sequence defined in Example 4.5 for the `member(out,in)` procedure defined in Section 4.2. Assuming that an initial renaming σ_i simply adds the index i to all concerned variables, and assuming that when handling the recursive call variables e and es are renamed into e' and es', one can easily verify that the sequence condition associated to $\overline{S}$ is as follows:

$$\begin{array}{rl} \mathcal{C}\ (\langle S_1, S_2, S_3 \rangle) = & \mathcal{U}(S_1, \sigma_1) \cup \mathcal{U}(S_2, \sigma 2) \cup \mathcal{U}(S_3, \sigma_3) \\ = & \{y = [e_1 | es_1], x_1 = e_1\} \\ & \cup \{y = [e_2 | es_2], es_2 = [e'_2 | es'_2], x_2 = e'_2\} \\ & \cup \{y = [e_3 | es_3], es_3 = [e'_3 | es'_3], es'_3 \not\Rightarrow [\ |\ ]\} \end{array}$$

For a given execution sequence $\overline{S}$, $\mathcal{C}\ (\overline{S})$ defines the minimal instantiation of the procedure's input variables so that the execution is guaranteed to proceed as specified by $\mathcal{S}$. In Example 5.2 above, the set of constraints $\mathcal{C}\ (\langle S_1, S_2, S_3 \rangle)$ implies

$$y = [e_1, e'_2 | es'_3] \wedge es'_3 \not\Rightarrow [\ |\ ] \wedge x_1 = e_1 \wedge x_2 = e'_2$$

and, indeed, whatever type correct instantiation E_1, E'_2 we choose for the variables e_1 and e'_2, we will always have $es'_3 = []$ and the execution of a call $member(_, [E_1, E'_2])$ is guaranteed to proceed along the specified path.
Note that the obtained constraint set defines, for each segment ending in success, the minimal instantiation of the procedure's *output* arguments as well. In Example 5.2, the sequence of output arguments is given by $\langle x_1, x_2 \rangle$. Hence, the computed results could be automatically converted not only into test inputs but into complete test cases. Of course, before such a computed test case can be recorded for further usage, the programmer should verify that the computed output corresponds with the *expected* output.

5.3 Properties of the analysis

Theorem 5.3[Completeness] For all input substitution θ and goal G (in labelled syntax), there exists $\mathcal{T}_i \in \Phi(G)$ such that:

$$\begin{array}{l} \langle t_1, \ldots, t_n \rangle = \mathcal{T}_i \text{ and} \\ last_{\beta_j} = l_F \text{ if } \theta_j = \texttt{Fail} \\ last_{\beta_j} = l_S \text{ otherwise} \end{array}$$

where

$$\mathcal{S}[\![G]\!]\theta = \langle (t_1, \theta_1), \ldots, (t_n, \theta_n) \rangle$$
$$\mathcal{T}_i = \langle \beta_1, \ldots, \beta_n \rangle$$

This theorem states that for all input substitution θ, the complete semantics trace of a goal G (in labelled syntax) computed by $\mathcal{S}[\![G]\!]\theta$ is equal to one execution sequence $\mathcal{T}_i \in \Phi(G)$ (from which the labels l_F and l_S are removed). Moreover, for each individual semantics trace, if the substitution associated to this trace is a valid substitution θ_j, respectively the substitution denoting a failure `Fail`, then the corresponding segment β_j ends with the success label l_S, respectively the failure label l_F. In other words, our symbolic execution definition is complete since any real execution is modelled by a symbolic one.

The proof of the theorem is straightforward, since the definition of the execution sequences is built as a generalization of the semantics. In order words, while the semantics defines the execution trace based on an input substitution, the function

Theorem 5.4[Soundness] Let the substitution θ be a solution to the sequence condition C associated to $\overline{S}$, an execution sequence representing a derivation tree for a procedure p. Then the execution of p using θ as input substitution will follow the derivation tree represented by $\overline{S}$.

Proof. We prove the soundness property by contradiction. Let us assume that θ is a solution to the sequence condition C associated to $\overline{S}$ and that the execution of p using θ as input substitution follows *another* derivation tree than the one represented by $\overline{S}$. Since the analysis is complete (Theorem5.3), this other derivation tree is represented by another execution sequence $\overline{S'}$ for p.

Since $\overline{S}$ and $\overline{S'}$ are both execution sequences for p and $\overline{S} \neq \overline{S'}$ we know that, by definition:

- they have a common prefix $\langle l_1, \ldots, l_{i-1} \rangle$ (that contains at least the label l_1 preceding the body of p);
- $\exists i$ such that $l_i \neq l'_i$ (where $l_i \in \overline{S}$ and $l'_i \in \overline{S'}$)

By definition, the only divergences between execution sequences for the same procedure are introduced by the fact that an atom can either succeed or fail. Therefore, we have that the pair of labels (l_{i-1}, l_i) represents the constraint c according to the table shown in Section 5.2, while (l_{i-1}, l'_i) represents $not(c)$.

Since θ is a solution to the sequence condition associated to $\overline{S}$, it satisfies in particular the constraint c, and cannot therefore satisfy $not(c)$ at the same time. $\square$

5.4 Constraint Solving

A sequence condition is a set of constraints that is either *satisfiable* or *unsatisfiable*. The latter implies that one or more labels in the path cannot be reached along the path (but may be reached along other paths). A satisfiable sequence condition means that solutions (one or more) exist, and that they will exercise the associated execution sequence. In order to establish satisfiability, we take the usual constraint programming approach of interleaving *propagation* and *search* (Rossi, van Beek, and Walsh 2006).

Propagation We reuse existing (CLP) constraints for most of our base constraints.

- $x = y$ and $x = f(\overline{y})$ are implemented as unification,
- $x \neq y$ is implemented as the standard Herbrand inequality constraint, known as `dif/2` in many Prolog systems, and
- $x = y \oplus z$ is implemented as the corresponding CLP(FD) constraint.

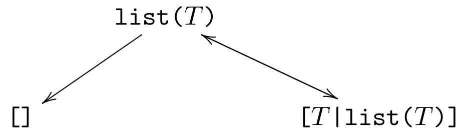
For $x \not\Rightarrow f$ we have our custom constraint propagation rules, implemented in CHR, based on the domain representation of CLP(FD). However, rather than maintaining a set of possible values for a variable, the domain of a variable is the set of possible function symbols. The initial domain of a variable is defined as the set of all the function symbols of the variable's type. For instance, the constraint `domain(X,{[]/0,[|]/2})` expresses that the possible functions symbol for variable `X` with type `list(T)` are `[]/0` and `[|]/2`, which is also its initial domain.
The following CHR rules further define the constraint propagators (and simplifiers) for the `domain/2` constraint:

```
domain(X,∅)       ==> fail.
domain(X,{F/A})  <=> functor(X,F,A).
domain(X,D)       <=> nonvar(X) | functor(X,F,A), F/A ∈ D.
domain(X,D1), domain(X,D2) <=> domain(X,D1 ∩ D2).
domain(X,D), X ⇏ F/A      <=> domain(X,D \ {F/A}).
```

The first rule is obvious; it means that if the domain of the variable X is empty, no solution can be found to the CSP and hence it fails. The second rule is also trivial; if the domain of a variable contains a single functor, that means the variable is bound to this functor. The third rule has a guarded body; the guard $nonvar(X)$ means that the variable X is bound to a non-variable term. The meaning of the whole rule is therefore that the constraint $domain(X, D)$ applied to a variable already bound to a non-variable term requires this term (in our case, a functor) to be in the domain D in order to be satisfied. The last but one rule means that if a variable has two domain constraints, this variable should be bound to a value comprised in the intersection of the two domains. Finally, the last rule means that if a variable has a domain D and if this variable cannot be bound to a functor F/A, then the domain of this variable is D minus $\{F/A\}$.

Search Step During search, we enumerate candidate values for the variables. From all undetermined terms, we choose one x and create a branch in the search tree for each function symbol f_i in its domain. In branch i, we add the constraint $x = f_i(\overline{y})$, where $\overline{y}$ are fresh undetermined terms. Subsequently, we exhaustively propagate again. Then either an (1) inconsistency is found, (2) all terms are determined or (3) some undetermined terms remain. In case (1) we must explore other branches, and in case (2) we have found a solution. In case (3) we simply repeat with another Search Step.
Our search algorithm visits the branches in depth-first order. Hence, we must make sure not to get caught in an infinitely diverging branch of the search tree, e.g. one that generates a list of unbounded length. For this purpose, we order a type's function symbols according to the *type graph*. The nodes of a type graph consist of types and function symbols. There is an edge in the graph from each type to its function symbols, and from each function symbol to its argument types. During search, branches are created for the function symbols that can be associated to a variable according to the order in which they appear in a topologic ordering of the type graph.[1]

Example 5.3 Consider the list type `:- type list(T) ---> [] ; [T|list(T)].` Its type graph is:



A topological order is $\langle$`[]`, `list(`T`)`, `[`T`|list(`T`)]`$\rangle$, which yields the function symbol ordering $\langle$`[]/0`, `[|]/2`$\rangle$. Indeed, if we first try `[|]/2` the search would diverge, whereas if we first try `[]/0` at each step, the search will eventually halt, yielding a finite list.

To conclude this section, note that our approach is the opposite of type inference, as we infer terms from types rather than types from terms. Among the vast amount of work on type inference we note (Demoen, de la Banda, and Stuckey 1999) which employs a similar function symbol domain solver, for resolving ad-hoc overloading.

5.5 Implementation and Evaluation

Note that the symbolic execution of Mercury, as we use it in our approach, has an exponential complexity. Indeed, when symbolically executing a Mercury program, each equality test atom and each deconstruction atom encountered represents a non-deterministic choice to be made, i.e. either the atom succeeds,

[1] We assume that all recursive type definitions are based on a well-founded recursion (Goubault-Larrecq 2001), e.g. the type `:- type stream(T) ---> cons(T,stream(T)).` is not a valid type.

```
test(t1, [member(0,[0])], [success]).
test(t2, [member(0,[1,0])], [success]).
test(t3, [member(0,[1,1,0])], [success]).
test(t4, [member(1,[0,0])], [failure]).
test(t5, [member(1,[0])], [failure]).
test(t6, [member(0,[])], [failure]).
```

```
test(t1, [member(X,[0,1,2])], [true(1,X=0),true(2,X=1),true(3,X=2)]).
test(t2, [member(X,[0,1])], [true(1,X=0),true(2,X=1)]).
test(t3, [member(X,[0])], [true(1,X=0)]).
test(t4, [member(X,[])], [failure]).
```

Table 5.1: Test input generation for `member(X::in,Y::in)` and `member(X::out,Y::in)`

either it fails. It means that the complexity of the symbolic execution is $\mathcal{O}(2^n)$ with n the number of equality tests or deconstructions encountered during the execution.

The described approach for generating test inputs was implemented in Mercury. Our implementation first constructs a control flow graph for the program under test, and computes a set of execution sequences for the procedures in the program. This set of execution sequences is constructed in such a way that the resulting test suite – that will cause each execution sequence of this set to be followed when executed – satisfies one or more given coverage criteria. In our case, we generate the set of execution sequences with respect to both *call-depth-K* and *backtrack-K* coverage criteria (see Section 4.3). Since our implementation is meant to be used as a proof of concept, performance of the tool has not been particularly stressed.

Table 5.1 gives the test cases that are generated for the `member(in,in)` and `member(out,in)` procedures defined in Example 2.7 (page 35), with respect to the call-depth-2 and backtrack-3 coverage criteria.[2] The latter means that the execution of the resulting test suite will exercise all possible executions that don't require performing more than 3 backtrackings and that are such that the number of calls that haven't been returned from yet at any moment of the execution does not exceed 2. As described in Section 5.2, it is up to the user to check whether the obtained result corresponds to the expected result when creating the test suite. The test inputs (and corresponding outputs) presented in Table 5.1 were generated in 20 ms , respectively 10 ms.

Table 5.2 contains the generated test inputs for a procedure implementing the bubble-sort algorithm depicted in Figure 5.1. This well-know algorithm for list sorting uses two recursive sub-procedures. We used the call-depth-5 adequacy criterion, and for each test input we also give the computed output

[2] in the case of `member(out,in)`, we added manually the constraint `all_different/1` which guarantees all the elements of the list to be different.

```
:- pred bubblesort(list(int), list(int)).
:- mode bubblesort(in, out) is det.
bubblesort(List,Sorted):- bsort(List,[],Sorted).

:- pred bsort(list(int), list(int), list(int)).
:- mode bsort(in,in, out) is det.
bsort([],Acc,Acc).
bsort([H|T],Acc,Sorted):-
    bubble(H,T,NT,Max),bsort(NT,[Max|Acc],Sorted).

:- pred bubble(int, list(int),list(int), int).
:- mode bubble(in,in, out,out) is det.
bubble(X,[],[],X).
bubble(X,[Y|T],[Y|NT],Max):- X>Y,bubble(X,T,NT,Max).
bubble(X,[Y|T],[X|NT],Max):- X<=Y,bubble(Y,T,NT,Max).
```

Figure 5.1: The `bubblesort/2` predicate.

Test input	Computed Result	Test input	Computed Result	Test input	Computed Result
`List::in`	`Sorted::out`	`List::in`	`Sorted::out`	`List::in`	`Sorted::out`
[]	[]	[2, 1, 0]	[0, 1, 2]	[1, 2, 1, 0]	[0, 1, 1, 2]
[0]	[0]	[0, 0, 0, 0]	[0, 0, 0, 0]	[1, 1, 0, 0]	[0, 0, 1, 1]
[0, 0]	[0, 0]	[0, 0, 1, 0]	[0, 0, 0, 1]	[2, 2, 1, 0]	[0, 1, 2, 2]
[1, 0]	[0, 1]	[0, 1, 1, 0]	[0, 0, 1, 1]	[1, 0, 0, 0]	[0, 0, 0, 1]
[0, 0, 0]	[0, 0, 0]	[1, 1, 1, 0]	[0, 1, 1, 1]	[2, 0, 1, 0]	[0, 0, 1, 2]
[0, 1, 0]	[0, 0, 1]	[1, 0, 1, 0]	[0, 0, 1, 1]	[2, 1, 1, 0]	[0, 1, 1, 2]
[1, 1, 0]	[0, 1, 1]	[0, 1, 0, 0]	[0, 0, 0, 1]	[2, 1, 0, 0]	[0, 0, 1, 2]
[1, 0, 0]	[0, 0, 1]	[0, 2, 1, 0]	[0, 0, 1, 2]	[3, 2, 1, 0]	[0, 1, 2, 3]

Table 5.2: Test cases generation for `bubblesort(in,out)`

value. The test input generation took 1200 ms. For clarity of the resulting table we don't provide the test cases in their verbose syntax, but we simply provide the test input and the computed result. Of course, a test input [0] with a corresponding computed result [0] can easily be translated in a test case $test(t, [bubblesort([0], Y)], [true(Y = [0])])$.

In Table 5.3, we present the behaviour of our implementation with different procedures, most of them have been chosen from the DPPD library (Leuschel). For each of them, we indicate (1) the mode of the predicate, (2) its determinism, (3) the maximum call depth used, (4) the number of solutions requested (only in the case of non-deterministic and multi-deterministic procedures), (5) the number of test cases generated, and (6) the execution time of the test input generation, given in ms.

Procedures	Deter-minism	Maximum call depth	Solutions requested	Number of test cases	Execution time (in ms)
Partition(in,in,out,out)	det	6	--	126	890
Append(in,in,out)	det	6	--	6	40
Append(out,out,in)	nondet	6	10	10	70
Doubleapp(in,in,in,out)	det	3	--	6	650
Doubleapp(out,out,out,in)	multi	6	8	4	4670
Member(in,in)	semidet	5	--	12	700
Member(out,in)	nondet	5	5	6	1310
Applast(in,in,out)	det	3	--	14	40
Match(in,in)	semidet	3	--	6	960
Matchappend(in,in)	semidet	4	--	20	90
MaxLength(in,out,out)	det	5	--	10	600
Revacctype(in,in,out)	det	4	--	12	500
Transpose(in,out)	det	2	--	9	1370

Table 5.3: Test cases generation for different procedures.

5.6 Automatic completion of test suites

It is possible to use the mechanism described in Section 4.4.3 that enables the computation of the coverage rate of a test suite with respect to given coverage criteria together with the automatic test cases generation mechanism described in this chapter, in order to automatically *complete* a test suite previously written. The completion of a test suite with respect to coverage criteria is the action of adding test cases to the test suite that originally does not satisfy the criteria, such that the resulting test suite satisfies them. The different steps of this automatic completion are the following:

1. The control flow graph of the provided (labelled) program is created as described in Section 4.1;

2. The set of execution sequences corresponding to the provided coverage criteria is created as described in Sections 4.2 and 4.3;

3. The program is instrumented, and the suite is executed using this instrumented program, resulting in the creation of a set of execution traces (see Section 4.4.1);

4. The coverage rate is computed by comparing the set of execution sequences with the set of execution traces as described in Section 4.4.3;

5. The sequence conditions of the execution sequences having no corresponding execution trace – i.e. the execution sequences that were not covered during the execution of the test suite – are solved in order to create new test cases that are added to the test suite such that it satisfies the coverage criterion.

This mechanism has not been implemented but could constitute an interesting lead for further work, as there currently exists no such tool to the best of our knowledge. Such a technique could possibly prove to be an interesting trade-off between the fully automated test suite generation based on an adequacy criterion – which can have the drawback to generate test cases that don't necessarily correspond to real-life values dealt with by the programmer – and the manual creation of test suites that are usually inadequate w.r.t. the chosen criterion.

Chapter 6

TDG for a pointer-based imperative language

In this chapter, we present a technique adapted from the technique presented in the previous chapters, for automatically generating test inputs for programs written in an imperative language dealing with pointer-based data structures. This is especially challenging, as a test input for a procedure in such a program comprises not only a set of atomic values for the procedure's arguments but may also contain data structures build on the heap. Before explaining our approach in details, we first present some existing work in the field of test data generation for imperative languages using complex data structures. Then we introduce a small but representative pointer-based imperative language called ImpL, that we use to define our method and demonstrate how we can achieve test data generation of input data containing pointer-based data structures satisfying test adequacy criteria, using a constraint-based approach. The results presented in this chapter were published in (Degrave, Schrijvers, and Vanhoof 2009).

6.1 Existing work on TDG for imperative languages using complex data structures

In (Khurshid, Pasareanu, and Visser 2003; Visser, Păsăreanu, and Khurshid 2004; Khurshid and Marinov 2003) the authors present a test case generation framework based on symbolic execution and model checking and that is able to handle dynamically allocated structures (e.g. lists and trees), simple (primitive) data (e.g. integers and strings), concurrency and arrays. In this approach, symbolic execution is performed thanks to a model checker applied to a modified source code of the program. Those modifications aim at adding non-determinism and support for manipulating path conditions. They basically enable the model checker to explore the symbolic execution tree of the program. The method is implemented for Java programs thanks to the JavaPathFinder

model checker, but it seems the method could be applied to any (imperative or object oriented) language using any model checker for this language that supports non-deterministic choice. The way the problem of object aliasing is addressed is as follows: when the (symbolic) execution accesses an uninitialized reference, the algorithm non-deterministically initializes it either to the value null, either to a reference to a new object with uninitialized fields, or to a reference of an object created during a prior field initialization. This means that each occurrence of an uninitialized reference will create three different continuations of the symbolic execution, corresponding to three sub-trees in the symbolic execution tree. The way the framework handles concurrency is not very clear; it is simply mentioned that it "uses the model checker to systematically analyse thread interleavings" – no further details are provided. The result of symbolically executing a particular program path using this framework is a heap structure containing the constraints on reference fields, and a path condition containing the constraints on primitive data. The constraints on data structures are solved separately from the ones on primitive data. This approach addresses a number of interesting issue, it requires however to provide formal specifications of the program used during the instrumentation of the code. Also, no details are given on the constraint solver used.

Java is also the main interest in (Müller, Lembeck, and Kuchen 2004), which presents a tool called GlassTT able to create test suites for a given criterion for a Java class file. It uses a Symbolic Java Virtual Machine (SJVM) to generate path conditions; symbolic execution is guided by a user-specified coverage criterion. The authors implemented the all definition-use paths, the branch coverage and the statement coverage criteria within their tool. They did it by adapting the "decision unit" of the SJVM that decides which branch has to be executed. This approach is promising in the perspective of creating a tool parametrized with respect to any adequacy criterion. However, GlassTT seems to be limited to the generation of integer data.

The problem of test data generation using symbolic execution with pointer-based data structures is discussed in (Visvanathan and Gupta 2002). This problem is very similar to the one discussed in Kurshid's work on object aliasing in Java: not only the values in the fields of the input data structure should be determined, but also the shape of the data structure required in order to cause the execution to follow a chosen execution path. The approach of the authors comprises two phases: first a suitable shape for the input data is generated, then the data values in this data structure are generated. This approach is justified by the fact that the constraints on the pointers deal with addresses of memory locations used by the statements whereas the constraints on the data values (integer and real values) deal with the actual values used during the execution. Since the solutions for these two types of constraints are in two different domains, they can be treated separately – similarly as in Kurshid's work. However in their paper, the authors present only the method for generating the data structures (the shapes), the generation of actual values being left to existing techniques – we can however reasonably suppose that existing techniques should be adapted in order to be used on programs dealing with pointer operations.

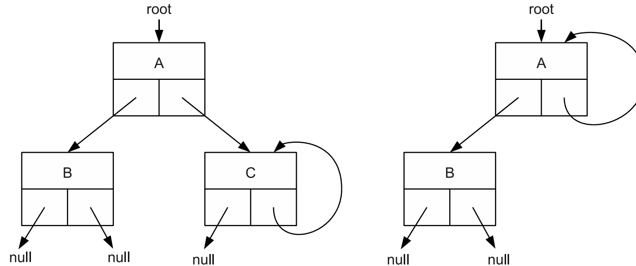


Figure 6.1: Two different pointer-based structures

Of course, for the same execution path in a program, different shapes can be generated. Instead of generating *all* the different shapes, the authors decided to choose the so-called *least restrictive* shape, i.e. the shape containing the maximum number of nodes that can be referenced by the statements along the path. For example, in Figure 6.1, the structure at the left is less restrictive over the other one. The shape at the right is considered more restrictive as it imposes the right child of the root node to be identical to its father. During symbolic execution, the statements along a given path are examined, and a set of constraints is derived on the node addresses. Note that the operations on the pointers supported by the method are the following: assignment, comparison, dereferencing and allocation – no pointer arithmetic is therefore permitted. The constraints are collected using an "address table" which is updated with respect to the statements encountered. This address table is in fact a symbolic representation of the heap and its transformations during the execution. A similar approach has been followed in (Zhao and Li 2007), in which the input shape is first created, then it is filled with generated values.
In (Gotlieb, Denmat, and Botella 2005a; Gotlieb, Denmat, and Botella 2005b; Gotlieb, Botella, and Watel 2006) the authors discuss the same problem of generating test data for programs with pointer variables. They particularly focus on the "conditional aliasing problem" in symbolic execution, which is identical to the one evoked in (Hall 1971) for arrays. In the case of pointers, this problem can be illustrated by the following example:

```
int f(int i, int j, int c){
int* p = &j;
if (c == 1)
 p = &i;
i = 0;
*p = 1;
if (i != 0)...
```

In this example, the value of i in the second condition depends of course on whether p points to its location or not when the assignment $*p = 1$ is performed.

The approach of the authors is the same as in their previous work, namely that the imperative program is translated into a CLP program, using SSA form as intermediate transformation. SSA form is a semantically equivalent form of an imperative program which respects the following rule: each variable has a unique definition (i.e. it is assigned only once). In order to transform a program into its SSA form, the uses and definitions of the variables have to be renamed. For example, $i = i + 1;\ j = j * i$ is transformed into $i_2 = i_1 + 1;\ j_2 = j_1 * i_2$. After a conditional statement, it can happen that the variable is assigned to one value in the first branch, and to another in the second one. For that purpose, the SSA form introduces so called ϕ-functions; therefore, $v_3 = \phi(v_1, v_2)$ assigns the value of v_1 to v_3 if the first branch is traversed, and the value of v_2 otherwise. This SSA form can then easily be transformed into a CLP equivalent, in which each function and each ϕ-function appears under the form of a clause. A special mechanism that deals with pointers is added: first, a pointer analysis is performed in order to determine the set of memory locations that can be accessed through each pointer variable. Then, special ϕ functions are used to model the dereferencing process. Constraint propagation has been improved using Dynamic Linear Relaxations (see Section 1.5.2) – we won't go into details here. This work has been implemented in the Euclide framework (Gotlieb 2009).

The use of Constraint Programming (CP) and its inherent mechanisms facilitate dealing with of a number of important issues. First, representing the heap and environment of the program by means of a symbolic data structure provides a convenient way to describe constraints on those structures. More importantly, we can use the search strategies of CP in order to tackle two essential issues: the first one comprises collecting a finite set of execution paths of the program which satisfies some given adequacy criteria. The second one is the generation, for each such path, of concrete values (a test input) such that when the program is executed with respect to those values, its execution will follow the corresponding path. Therefore, our technique can be seen as parametrised with respect to a coverage criterion or a desired degree of coverage. In order to illustrate the usefulness of this property, let us take an example of a small procedure written in a C-like programming language, supporting pointer-based dynamic data structures. This procedure manipulates a pointer `queu` to a linked list – whose structure is examinated in further details afterwards –, an element `el` of type T (this type has no importance in this example), and two integers `prioD` and `n`.

```
void insert (queu,el,prioD,n) {
        ptr = *queu.next ;
        q = queu ;
        c = 1
        while(ptr.prio >= prioD && c<n){
                ptr = *ptr.next ; c++ }
        r = new(el,max(prioD,*ptr.prio),ptr)
        q.next = r}
```

This procedure basically inserts an element `el` into a priority queue `queu` –

represented as a linked list – with respect to a given priority `prioD`. The element is inserted just after the last element having a higher priority than `prioD` if the number of such elements is less than `n`; otherwise, the element is inserted at the $\mathtt{n}^{th}$ position, and its priority is changed to that of the $\mathtt{n}-1^{th}$ element of the queue.
A test case for a procedure consists of an environment and a heap as they could be at the moment of the procedure's call. For example, a test case for the `insert` procedure could be an environment in which the variables `prioD` and `n` both map to the value 3, `el` maps to an arbitrary value depending on its type, and `queu` maps to a reference, pointing into the heap to the first cell of a linked list, whose cells consist of three fields: 1) a content (whose type and value have no importance in the current example, and is represented as a small shape in Figure 6.2), 2) a priority (an integer value) and 3) a reference to the next cell in the list. Two examples of such test cases are depicted in Figure 6.2.

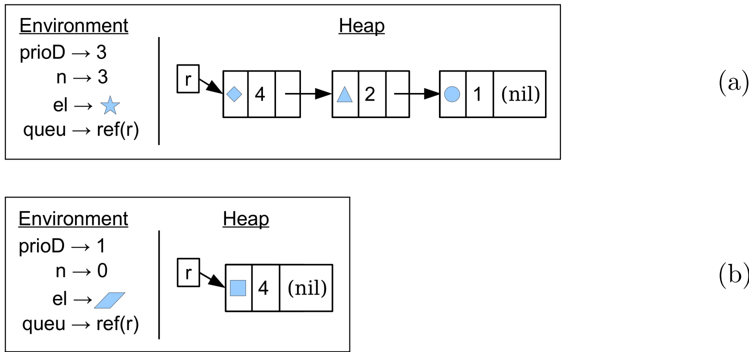


Figure 6.2: Examples of test cases for the `insert` procedure

One major advantage of our framework is that it is parametrised with respect to a given coverage criterion. Different coverage criteria can lead, of course, to different generated test cases. For example, if statement coverage is used as the coverage criterion, our technique might produce the test case (a) depicted in Figure 6.2 as the only testcase. Indeed, execution of the procedure with respect to this data will guarantee that every statement in the procedure's body gets executed. However, if condition coverage is used as the coverage criterion, the test case (a) in itself is not sufficient as the test suite must guarantee that every boolean sub-expression is evaluated both to true and false during testing while with the test case (a), only the subexpression `ptr.prio >= prioD` is evaluated both to true and false. Therefore, instantiated with condition coverage the technique will produce at least one additional testcase, for example the one depicted in Figure 6.2 (b) in which the subexpression `c<n` is guaranteed to be eventually evaluated to false.

Our specific contributions are:

- We show how to extend the semantics of an imperative language to deal with unknown pointer-based input values. (Section 6.3.2)
- We show how concrete test cases satisfying adequacy criteria can be generated by using a suitable CP search strategy. (Section 6.3.5)
- We present a visualization tool and a regression test generator based on our approach. (Section 6.4)

6.2 The ImpL language

In order to focus on the essence of constraint-based test generation for imperative languages, we define a small imperative language supporting dynamic pointer-based data structures and show that our approach is able to generate test cases dealing with in-place updates of variables, pointers and a variety of potentially cyclic data structures – for convenience, we refer to this language as ImpL in what follows. As the definition below shows, we only consider integer values and data structures constructed from simple "cons" cells having two fields that we will name *head* and *tail*. We indicate in Section 6.3.6 how our technique for test case generation can easily be extended to deal with a more involved language having primitive values other than integers and full `struct`-like data structures.

integers	n		
variables	x		
expressions	e	::=	x | n | `nil` | `new cons`(e_1, e_2) | e`.head` | e`.tail`
			| e_1 `==` e_2 | e_1 `/=` e_2 | e_1`+`e_2
statements	s	::=	`skip` | l `:=` e | s_1`;`s_2 | `if` e `then` s_1 `else` s_2
			| `while` e { s }
left-hand sides	l	::=	x | l`.head` | l`.tail`

As usual, expressions are used to syntactically represent values within the source code of a program. Among the possible expressions are program variables, integers, the null-pointer `nil`, a reference to a newly heap-allocated cons cell `new cons(`e_1`,`e_2`)`, the selection of the head ($e.head$), respectively tail ($e.tail$) field of the cons cell referenced by the expression e, equality and inequality tests (`==` and `/=`), and the arithmetic operator for addition `+`.[1] ImpL is simply typed and it only allows comparison of two values belonging to the same type (either integers or references).[2] Moreover, arithmetic is only allowed on integer values; the language does not support pointer arithmetics. Those typing rules are given in Figure 6.3. Recall that an inference rule of the form:

$$\frac{p_1, p_2, ..., p_n}{c}$$

[1]Other arithmetic operators are omitted in order to keep the formal definition of the semantics small, but they can be added at will.

[2]Integers are also used as booleans: 0 denotes false and all other integers denote true.

denotes that given premises $p_1, p_2, ..., p_n$, the specified conclusion c can be taken for granted as well. A judgement of the form $\vdash\ e : T$ denotes that the expression e is a well-typed construct of type T. We denote the type of integers as int and the type of the references as ref. Types T, T_1 and T_2 denote types that are either int or ref. Note that the expression e used as condition in a $if - then - else$ or a $while$ statement must be of type int.

$$(\textsc{Int})\ \frac{n \in \mathbb{Z}}{\vdash\ n : int} \qquad (\textsc{Nil})\ \vdash\ \texttt{nil} : ref$$

$$(\textsc{Cons})\ \frac{\vdash\ e_1 : T_1 \quad \vdash\ e_2 : T_2}{\vdash\ \texttt{new cons}(e_1, e_2) : ref} \qquad (\textsc{Head})\ \frac{\vdash\ e : ref}{\vdash\ e\texttt{.head} : T}$$

$$(\textsc{Tail})\ \frac{\vdash\ e : ref}{\vdash\ e\texttt{.tail} : T} \qquad (\textsc{Equal})\ \frac{\vdash e_1 : T \quad \vdash\ e_2 : T}{\vdash e_1\ \texttt{==}\ e_2 : int}$$

$$(\textsc{NEqual})\ \frac{\vdash\ e_1 : T \quad \vdash\ e_2 : T}{\vdash e_1\ \texttt{/=}\ e_2 : int} \qquad (\text{ADD})\ \frac{\vdash e_1 : int \quad \vdash e_2 : int}{\vdash e_1\texttt{+}e_2 : int}$$

Figure 6.3: Typing rules in ImpL

A program in ImpL is a single statement or a sequence of statements, where a statement is either a no-op (`skip`), an assignment, another sequence, a selection or a while-loop. The left-hand side of an assignment is either a variable or a reference to one of the fields in a cons cell. Consider, for example, the following simple program:

```
while (x.tail.head /= x.head) {
        x := x.tail };
x.tail := nil
```

The above program basically manipulates a simply linked list `x` whose cells consist of two fields: a *head* containing an integer and a *tail* containing a pointer to the following cell or `nil`. It scans the list for two successive identical elements, and severs the list after the first such occurrence. For example, using the notation `[1,2,3]` for the nil-terminated linked list with successive elements 1,2 and 3, the effect of running this program with `x` the list `[1,2,3,3,4]`, is that, upon termination of the program, the list will have the value `[1,2,3]`.

6.3 Generating test inputs

6.3.1 Overview

The execution of an imperative program manipulates an environment E and a heap H. An environment is a finite mapping from variables to *values*, where a value is either an integer, `nil` or a reference to a cons cell represented by $\texttt{ptr}(r)$ with r a unique value denoting the address of the cons cell on the heap.

Likewise, a heap is a finite mapping from such references r to cons cells of the form cons(v_h,v_e) with v_h and v_e values (possibly including references to other cons cells). The operational semantics for the expressions and the statements are respectively provided in Figures 6.4 and 6.5, and describe how the environment and the heap are manipulated in a program written in IMPL. A judgement of the form $\langle E, H_0\rangle\ e \rightsquigarrow v; H_1$ denotes that the expression e evaluates to a value v with respect to an environment E, and transforms the heap from H_0 to H_1. Note that an expression does never update the environment. Similarly, a judgement of the form $\langle E_0, H_0\rangle\ s\ \langle E_1, H_1\rangle$ denotes that the evaluation of statement s transforms an initial environment E_0 and heap H_0 into a final environment E_1 and heap H_1. For the example given above (with x initially the list [1,2,3,3,4]), the environment and heap before and after running the program would look as follows:

	Before		After
$E:$	$x \mapsto$ ptr(r_1)	$E:$	$x \mapsto$ ptr(r_3)
$H:$	$r_1 \mapsto$ cons(1,ptr(r_2))	$H:$	$r_1 \mapsto$ cons(1,ptr(r_2))
	$r_2 \mapsto$ cons(2,ptr(r_3))		$r_2 \mapsto$ cons(2,ptr(r_3))
	$r_3 \mapsto$ cons(3,ptr(r_4))		$r_3 \mapsto$ cons(3,nil)
	$r_4 \mapsto$ cons(3,ptr(r_5))		
	$r_5 \mapsto$ cons(4,nil)		

Now, in order to generate test inputs for a program, the idea is to symbolically execute the program, replacing unknown values by *constraint variables*. During such a symbolic execution, each test in the program (i.e. the *if-then-else* and *while* conditions) represents a choice; the sequence of choices made determines the execution path followed. There are many possible execution paths through the program. Each one of them can be represented by constraints on the introduced variables and on the environment and heap.

Returning to our example, we would replace the concrete value for x by a constraint variable, say V, representing an unknown value. Among the infinite number of possible execution paths, a particular path would execute the *while* condition three times, and the loop body twice. This would imply that the value represented by V is a list of at least 4 elements, and the third and fourth element are identical, whereas the first differs from the second and the second from the third. This information would be represented by constraints on V and the heap collected along the execution. Solving these constraints could get us for instance the concrete input [1,2,3,3,4] proposed above. However, there are many other concrete inputs that satisfy these constraints: [1,2,3,3], [0,1,0,0], or even the cyclic list that starts with [1,2,1] and then points back the first element.

Using our constraint-based approach, we can both capture the many paths and the many solutions for a single path as non-determinism in our constraint-based modelling of test case generation. This allows us to use the search strategies of CP to deal with *both* of them. For instance, we can find all paths up to length 6 using a simple depth-bounded search.

The definition of the semantics is straightforward and very similar to what

$$\text{(Var)}\ \frac{(x \mapsto v) \in E}{\langle E, H\rangle\ x \rightsquigarrow v\langle E, H\rangle}$$

$$\text{(Int)}\ \frac{n \in \mathbb{Z}}{\langle E, H\rangle\ n \rightsquigarrow n\langle E, H\rangle}$$

$$\text{(Nil)}\ \langle E, H\rangle\ \texttt{nil} \rightsquigarrow \texttt{nil}\langle E, H\rangle$$

$$\text{(Cons)}\ \frac{\langle E, H_1\rangle\ e_1 \rightsquigarrow v_1\langle E, H_2\rangle \qquad \langle E, H_2\rangle\ e_2 \rightsquigarrow v_2\langle E, H_3\rangle \qquad r \text{ fresh}}{\langle E, H_1\rangle\ \texttt{new cons}(e_1, e_2) \rightsquigarrow \texttt{ptr}(r)\langle E, H_3 \uplus \{r \mapsto \texttt{cons}(v_1, v_2)\}\rangle}$$

$$\text{(Head)}\ \frac{\langle E, H_1\rangle\ e \rightsquigarrow \texttt{ptr}(r)\langle E, H_2\rangle \qquad (r \mapsto \texttt{cons}(v_h, v_t)) \in H_2}{\langle E, H_1\rangle\ e.\texttt{head} \rightsquigarrow v_h\langle E, H_2\rangle}$$

$$\text{(Tail)}\ \frac{\langle E, H_1\rangle\ e \rightsquigarrow \texttt{ptr}(r)\langle E, H_2\rangle \qquad (r \mapsto \texttt{cons}(v_h, v_t)) \in H_2}{\langle E, H_1\rangle\ e.\texttt{tail} \rightsquigarrow v_t\langle E, H_2\rangle}$$

$$\text{(EqualT)}\ \frac{\langle E, H_1\rangle\ e_1 \rightsquigarrow v_1\langle E, H_2\rangle \qquad \langle E, H_2\rangle\ e_2 \rightsquigarrow v_2\langle E, H_3\rangle \qquad v_1 \equiv v_2}{\langle E, H_1\rangle\ e_1\ \texttt{==}\ e_2 \rightsquigarrow 1\langle E, H_3\rangle}$$

$$\text{(EqualF)}\ \frac{\langle E, H_1\rangle\ e_1 \rightsquigarrow v_1\langle E, H_2\rangle \qquad \langle E, H_2\rangle\ e_2 \rightsquigarrow v_2\langle E, H_3\rangle \qquad v_1 \not\equiv v_2}{\langle E, H_1\rangle\ e_1\ \texttt{==}\ e_2 \rightsquigarrow 0\langle E, H_3\rangle}$$

$$\text{(NEqualT)}\ \frac{\langle E, H_1\rangle\ e_1 \rightsquigarrow v_1\langle E, H_2\rangle \qquad \langle E, H_2\rangle\ e_2 \rightsquigarrow v_2\langle E, H_3\rangle \qquad v_1 \not\equiv v_2}{\langle E, H_1\rangle\ e_1\ \texttt{/=}\ e_2 \rightsquigarrow 1\langle E, H_3\rangle}$$

$$\text{(NEqualF)}\ \frac{\langle E, H_1\rangle\ e_1 \rightsquigarrow v_1\langle E, H_2\rangle \qquad \langle E, H_2\rangle\ e_2 \rightsquigarrow v_2\langle E, H_3\rangle \qquad v_1 \equiv v_2}{\langle E, H_1\rangle\ e_1\ \texttt{/=}\ e_2 \rightsquigarrow 0\langle E, H_3\rangle}$$

Figure 6.4: Semantics of expressions in ImpL

already exists. For example, the evaluation of the creation of a new construction $\texttt{cons}(e_1, e_2)$ (detailed here in the rule CONS) results in the creation of a new cons-cell and returns a pointer to the newly created cell. The values inside the newly created cons-cell are those that are obtained by evaluating e_1 and e_2. As evaluation of e_1 with respect to a heap H_1 can result in a modified heap H_2, e_2 is evaluated with respect to this modified heap H_2.

$$(\textsc{Skip})\ \langle E, H\rangle\ \texttt{skip}\ \langle E, H\rangle$$

$$(\textsc{VarAss})\ \frac{\langle E, H_1\rangle\ e \rightsquigarrow v\langle E, H_2\rangle}{\langle E, H_1\rangle\ \}x\ \texttt{:=}\ e\ \langle E \uplus \{x \mapsto v\}, H_2\rangle}$$

$$(\textsc{HeadAss})\ \frac{\langle E, H_1\rangle\ e \rightsquigarrow v\langle E, H_2\rangle \qquad \langle E, H_2\rangle\ l \rightsquigarrow \texttt{ptr}(r)\langle E, H_2\rangle \qquad (r \mapsto \texttt{cons}(v_h, v_t)) \in H_2}{\langle E, H_1\rangle\ l\texttt{.head := }e\ \langle E, H_2 \uplus \{r \mapsto \texttt{cons}(v, v_t)\}\rangle}$$

$$(\textsc{TailAss})\ \frac{\langle E, H_1\rangle\ e \rightsquigarrow v\langle E, H_2\rangle \qquad \langle E, H_2\rangle\ l \rightsquigarrow \texttt{ptr}(r)\langle E, H_2\rangle \qquad (r \mapsto \texttt{cons}(v_h, v_t)) \in H_2}{\langle E, H_1\rangle\ l\texttt{.tail := }e\ \langle E, H_2 \uplus \{r \mapsto \texttt{cons}(v_h, v)\}\rangle}$$

$$(\textsc{Seq})\ \frac{\langle E_1, H_1\rangle\ s_1\ \langle E_2, H_2\rangle \qquad \langle E_2, H_2\rangle\ s_2\ \langle E_3, H_3\rangle}{\langle E_1, H_1\rangle\ s_1\texttt{;}s_2\ \langle E_3, H_3\rangle}$$

$$(\textsc{IfThen})\ \frac{\langle E_1, H_1\rangle\ e \rightsquigarrow n\langle E_1, H_2\rangle \qquad n \not\equiv 0 \qquad \langle E_1, H_2\rangle\ s_1\ \langle E_2, H_3\rangle}{\langle E_1, H_1\rangle\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \langle E_2, H_4\rangle}$$

$$(\textsc{IfElse})\ \frac{\langle E_1, H_1\rangle\ e \rightsquigarrow n\langle E_1, H_2\rangle \qquad n \equiv 0 \qquad \langle E_1, H_2\rangle\ s_2\ \langle E_2, H_3\rangle}{\langle E_1, H_1\rangle\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \langle E_2, H_4\rangle}$$

$$(\textsc{WhileT})\ \frac{\begin{array}{c}\langle E_1, H_1\rangle\ e \rightsquigarrow n\langle E_1, H_2\rangle \\ n \not\equiv 0 \qquad \langle E_1, H_2\rangle\ s\ \langle E_2, H_3\rangle \qquad \langle E_2, H_3\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_3, H_4\rangle\end{array}}{\langle E_1, H_1\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_3, H_4\rangle}$$

$$(\textsc{WhileF})\ \frac{\langle E_1, H_1\rangle\ e \rightsquigarrow n\langle E_1, H_2\rangle \qquad n \equiv 0}{\langle E_1, H_1\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_1, H_2\rangle}$$

Figure 6.5: Semantics of statements in IMPL

6.3.2 Constraint Generation

In order to represent unknown input data we add logical (or constraint) variables to the semantic domain of values and represent the environment and heap by logical variables as well. In order to model symbolic execution of our language, we introduce a semantics in which program state is represented by a triple $\langle E, H, C\rangle$ where E and H are constraint variables symbolically representing, respectively, the environment and heap, and C is a set of constraints over E and H. Constraints are conjunctions of primitive constraints over *syntactic objects*, that are either constraint variables, pointers $\texttt{ptr}(R)$ (where R is a constraint variable) or integer values. Our primitive constraints take the following form:

- $o_1 = o_2$, equality of two syntactic objects,
- $o_1 \neq o_2$, inequality of two syntactic objects,

- $(o_1 \mapsto o_2) \in M$, membership of a mapping M, and
- $M_1 \uplus \{o_1 \mapsto o_2\} = M_2$, update of a mapping M_1.

where a mapping M denotes a constraint variable representing an environment or a heap. Constraint solvers for these constraints are defined in Section 6.3.4. The symbolic semantics is depicted in Figures 6.6 and 6.7. In these figures and in the remainder of the text, we use uppercase characters to syntactically distinguish constraint variables from ordinary program variables (represented by lowercase characters). A judgement of the form $\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v\langle E_0, H_1, C_1\rangle$ denotes that given a program state $\langle E_0, H_0, C_0\rangle$, the expression e evaluates to value v and transforms the program state into a state represented by $\langle E_0, H_1, C_1\rangle$. Note that H_1 is a *fresh* constraint variable[3] that represents the possibly modified heap whose content is defined by the constraints in C_1. Likewise, a judgement of the form $\langle E_0, H_0, C_0\rangle s \langle E_1, H_1, C_1\rangle$ denotes the fact that a statement s transforms a program state represented by $\langle E_0, H_0, C_0\rangle$ into the one represented by $\langle E_1, H_1, C_1\rangle$. Since a newly added constraint can introduce inconsistencies in the set of collected constraints, we define the *conditional evaluation* of an expression and a statement as follows: judgements of the form $\{E, H_0, C_0\}\ e \rightsquigarrow v\langle E, H_1, C_1\rangle$ and $\{E_0, H_0, C_0\} s\langle E_1, H_1, C_1\rangle$ denote, respectively, $\langle E, H_0, C_0\rangle\ e \rightsquigarrow v\langle E, H_1, C_1\rangle$ and $\langle E_0, H_0, C_0\rangle s \langle E_1, H_1, C_1\rangle$ under the condition that C_0 is *consistent* (represented by $\mathcal{T} \models C_0$, where $\mathcal{T}$ is the constraint theory).[4] Formally:

$$(\textsc{Cond-e})\ \frac{\mathcal{T} \models C_0 \qquad \langle E, H_0, C_0\rangle\ e \rightsquigarrow v\langle E, H_1, C_1\rangle}{\{E, H_0, C_0\}\ e \rightsquigarrow\ v\langle E, H_1, C_1\rangle}$$

$$(\textsc{Cond-s})\ \frac{\mathcal{T} \models C_0 \qquad \langle E_0, H_0, C_0\rangle\ s\ \langle E_1, H_1, C_1\rangle}{\{E_0, H_0, C_0\}\ s\ \langle E_1, H_1, C_1\rangle}$$

The use of conditional evaluation avoids adding further constraints to an already inconsistent set. This implies that search strategies (see Section 6.3.5) will only explore execution paths that can model a real execution.

During the collection of constraints, we represent the sequence of transformations of the (unknown) environment, respectively heap, using a sequence of subscripted variables $E_0,\ E_1, \ldots$, respectively $H_0,\ H_1, \ldots$. In the semantics depicted in Figures 6.6 and 6.7, the variables E_n and H_n denote the variables representing the most recently transformed environment and the heap (i.e. the variables having the highest subscript) appearing in the current set of constraints. If the latter doesn't contain any variable representing the environment or heap, E_n and H_n denote the initial environment variable E_0 and initial heap variable H_0. A solution to a set of constraints is an assignment for each variable $E_0, ..., E_n$, respectively $H_0, ..., H_n$ to sets of mappings from variables to values (integers or pointers), respectively from references to `cons` cells. Some example of such solutions are presented in Section 6.3.5.

[3]A *fresh* constraint variable is a variable different from all the variables used before; in this particular case case, H_1 is different from E_0 and H_0, and does not appear in C_0

[4]In practice, the consistency check may be incomplete. Then unreachable execution paths may be explored.

$$(\textsc{Var})\ \frac{V \text{ fresh}}{\langle E,H,C\rangle\ x \rightsquigarrow V\langle E,H,C \wedge \{x \mapsto V\} \in E\rangle}$$

$$(\textsc{Int})\ \frac{n \in \mathbb{Z}}{\langle E,H,C\rangle\ n \rightsquigarrow n\langle E,H,C\rangle} \qquad (\textsc{Nil})\ \langle E,H,C\rangle\ \texttt{nil} \rightsquigarrow \texttt{nil}\langle E,H,C\rangle$$

$$(\textsc{Cons})\ \frac{\langle E,H_0,C_0\rangle\ e_1 \rightsquigarrow v_1\langle E,H_1,C_1\rangle \qquad \{E,H_1,C_1\}\ e_2 \rightsquigarrow v_2\langle E,H_2,C_2\rangle \qquad H_3, r \text{ fresh}}{\langle E,H_0,C_0\rangle\ \texttt{new cons}(e_1,e_2) \rightsquigarrow \texttt{ptr}(r)\langle E,H_3,C_2 \wedge H_3 = H_2 \uplus \{r \mapsto \texttt{cons}(v_1,v_2)\}\rangle}$$

$$(\textsc{Head})\ \frac{\langle E,H_0,C_0\rangle\ e \rightsquigarrow v\langle E,H_1,C_1\rangle \qquad R, V_h, V_t \text{ fresh}}{\langle E,H_0,C_0\rangle\ e\texttt{.head} \rightsquigarrow V_h\langle E,H_1,C_1 \wedge v = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h,V_t)) \in H_1\rangle}$$

$$(\textsc{Tail})\ \frac{\langle E,H_0,C_0\rangle\ e \rightsquigarrow v\langle E,H_1,C_1\rangle \qquad R, V_h, V_t \text{ fresh}}{\langle E,H_0,C_0\rangle\ e\texttt{.tail} \rightsquigarrow V_t\langle E,H_1,C_1 \wedge v = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h,V_t)) \in H_1\rangle}$$

$$(\textsc{EqualT})\ \frac{\langle E,H_0,C_0\rangle\ e_1 \rightsquigarrow v_1\langle E,H_1,C_1\rangle \qquad \{E,H_1,C_1\}\ e_2 \rightsquigarrow v_2\langle E,H_2,C_2\rangle}{\langle E,H_0,C_0\rangle\ e_1\ \texttt{==}\ e_2 \rightsquigarrow 1\langle E,H_2,C_2 \wedge v_1 = v_2\rangle}$$

$$(\textsc{EqualF})\ \frac{\langle E,H_0,C_0\rangle\ e_1 \rightsquigarrow v_1\langle E,H_1,C_1\rangle \qquad \{E,H_1,C_1\}\ e_2 \rightsquigarrow v_2\langle E,H_2,C_2\rangle}{\langle E,H_0,C_0\rangle\ e_1\ \texttt{==}\ e_2 \rightsquigarrow 0\langle E,H_2,C_2 \wedge v_1 \neq v_2\rangle}$$

$$(\textsc{NEqualT})\ \frac{\langle E,H_0,C_0\rangle\ e_1 \rightsquigarrow v_1\langle E,H_1,C_1\rangle \qquad \{E,H_1,C_1\}\ e_2 \rightsquigarrow v_2\langle E,H_2,C_2\rangle}{\langle E,H_0,C_0\rangle\ e_1\ \texttt{/=}\ e_2 \rightsquigarrow 1\langle E,H_2,C_2 \wedge v_1 \neq v_2\rangle}$$

$$(\textsc{NEqualF})\ \frac{\langle E,H_0,C_0\rangle\ e_1 \rightsquigarrow v_1\langle E,H_1,C_1\rangle \qquad \{E,H_1,C_1\}\ e_2 \rightsquigarrow v_2\langle E,H_2,C_2\rangle}{\langle E,H_0,C_0\rangle\ e_1\ \texttt{/=}\ e_2 \rightsquigarrow 0\langle E,H_2,C_2 \wedge v_1 = v_2\rangle}$$

$$(\textsc{Add})\ \frac{\langle E,H_0,C_0\rangle\ e_1 \rightsquigarrow v_1\langle E,H_1,C_1\rangle \qquad \{E,H_1,C_1\}\ e_2 \rightsquigarrow v_2\langle E,H_2,C_2\rangle \qquad v \text{ fresh}}{\langle E,H_0,C_0\rangle\ e_1 + e_2 \rightsquigarrow v\langle E,H_2,C_2 \wedge v = v_1 + v_2\rangle}$$

Figure 6.6: Symbolic evaluation of expressions.

$$(\textsc{Skip})\ \langle E, H, C\rangle\ \texttt{skip}\ \langle E, H, C\rangle$$

$$(\textsc{VarAss})\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v\langle E_0, H_1, C_1\rangle \qquad E_1 \text{ fresh}}{\langle E_0, H_0, C_0\rangle\ x\ \texttt{:=}\ e\ \langle E_1, H_1, C_1 \wedge E_1 = E_0 \uplus \{x \mapsto v\}\rangle}$$

$$(\textsc{HeadAss})\ \frac{\begin{array}{c}\langle E, H_0, C_0\rangle\ e \rightsquigarrow v\langle E, H_1, C_1\rangle \qquad \{E, H_1, C_1\}\ l \rightsquigarrow v_r\langle E, H_1, C_2\rangle \\ R, V_h, V_t, H_2 \text{ fresh} \qquad C_3 \equiv C_2 \wedge v_r = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H_1\end{array}}{\langle E, H_0, C_0\rangle\ l\texttt{.head := }e\ \langle E, H_2, C_3 \wedge H_2 = H_1 \uplus \{R \mapsto \texttt{cons}(v, V_t)\}\rangle}$$

$$(\textsc{TailAss})\ \frac{\begin{array}{c}\langle E, H_0, C_0\rangle\ e \rightsquigarrow v\langle E, H_1, C_1\rangle \qquad \{E, H_1, C_1\}\ l \rightsquigarrow v_r\langle E, H_1, C_2\rangle \\ R, V_h, V_t, H_2 \text{ fresh} \qquad C_3 \equiv C_2 \wedge v_r = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H_1\end{array}}{\langle E, H_0, C_0\rangle\ l\texttt{.tail := }e\ \langle E, H_2, C_3 \wedge H_2 = H_1 \uplus \{R \mapsto \texttt{cons}(V_h, v)\}\rangle}$$

$$(\textsc{Seq})\ \frac{\langle E_0, H_0, C_0\rangle\ s_1\ \langle E_1, H_1, C_1\rangle \qquad \{E_1, H_1, C_1\}\ s_2\ \{E_2, H_2, C_2\}}{\langle E_0, H_0, C_0\rangle\ s_1\texttt{;}s_2\ \langle E_2, H_2, C_2\rangle}$$

$$(\textsc{IfThen})\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v\langle E_0, H_1, C_1\rangle \qquad \{E_0, H_1, C_1 \wedge v \neq 0\}\ s_1\ \{E_1, H_2, C_2\}}{\langle E_0, H_0, C_0\rangle\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \langle E_1, H_2, C_2\rangle}$$

$$(\textsc{IfElse})\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v\langle E_0, H_1, C_1\rangle \qquad \{E_0, H_1, C_1 \wedge v = 0\}\ s_2\ \{E_1, H_2, C_2\}}{\langle E_0, H_0, C_0\rangle\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \langle E_1, H_2, C_2\rangle}$$

$$(\textsc{WhileT})\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v\langle E_0, H_1, C_1\rangle \qquad \{E_0, H_1, C_1 \wedge v \neq 0\}\ s\texttt{;while}\ e\ \{\ s\ \}\ \langle E_1, H_2, C_2\rangle}{\langle E_0, H_0, C_0\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_1, H_2, C_2\rangle}$$

$$(\textsc{WhileF})\ \frac{\langle E_0, H_0, C_0\rangle\ e \rightsquigarrow v\langle E, H_1, C_1\rangle}{\langle E_0, H_0, C_0\rangle\ \texttt{while}\ e\ \{\ s\ \}\ \langle E_0, H_1, C_1 \wedge v = 0\rangle}$$

Figure 6.7: Symbolic execution of statements.

Example 6.1 Consider again the example program of Section 6.3.1:

```
while (x.tail.head /= x.head) {
    x := x.tail
};
x.tail := nil
```

The derived constraints for `x.tail.head` are:

$$\begin{array}{rcllll} C_1 & \equiv & (x \leadsto V) \in E_0 & \wedge & & \\ & & V = \mathtt{ptr}(R) & \wedge & (R \leadsto \mathtt{cons}(V_h, V_t)) \in H_0 & \wedge \\ & & V_t = \mathtt{ptr}(R_2) & \wedge & (R_2 \leadsto \mathtt{cons}(V_{h2}, V_{t2})) \in H_0 & \end{array}$$

The constraints for the success of the condition `x.tail.head /= x.head`:

$$\begin{array}{rcllll} C_2 & \equiv & C_1 & \wedge & (x \leadsto V_2) \in E_0 & \wedge \;\; V_2 = \mathtt{ptr}(R_3) \\ & & & \wedge & (R_3 \leadsto \mathtt{cons}(V_{h3}, V_{t3})) & \wedge \;\; V_{h2} \neq V_{h3} \end{array}$$

The constraints for the loop body `x := x.tail`:

$$\begin{array}{rclll} C_3 & \equiv & (x \leadsto V_3) \in E_0 & \wedge \;\; V_3 = \mathtt{ptr}(R_4) & \wedge \\ & & (R_4 \leadsto \mathtt{cons}(V_{h4}, V_{t4})) \in H_0 & \wedge \;\; E_1 = E_0 \uplus \{x \leadsto V_{t4}\} & \end{array}$$

Finally, the constraints for the first iteration of the `while` loop are:

$$C \equiv C_2 \wedge C_3$$

6.3.3 Properties

Given environments E, E' and heaps H, H', we use $\langle E, H\rangle \cong \langle E', H'\rangle$ to denote the fact that E and E' define the same program variables and that each such variable either has the same primitive value (integer or `nil`) in both environments or points to identical data structures in both heaps.
More formally, this means that there must exist a bijective mapping σ between (a subset of) the references used in H and (a subset $\mathcal{S}$ of) those used in H' such that $\forall x \in dom(E) = dom(E') : E(x) =_\sigma E'(x)$ and $\forall a \in \mathcal{S} :\; H(a) =_\sigma H'(\sigma(a))$ where $=_\sigma$ is defined as follows:

$$\begin{array}{rcl} \mathtt{nil} & =_\sigma & \mathtt{nil} \\ n & =_\sigma & n,\; n \in \mathbb{Z} \\ cons(v_1, v_2) & =_\sigma & cons(v_1', v_2') \text{ iff } v_1 =_\sigma v_1' \;\wedge\; v_2 =_\sigma v_2' \\ ptr(r) & =_\sigma & ptr(r') \text{ iff } r' = \sigma(r) \end{array}$$

The set $\mathcal{S}$ of references is defined as $\mathcal{S} = fix(ext, \mathcal{S}_0)$ where:

$$\begin{array}{rcl} \mathcal{S}_0 & = & \{r \mid E(x) = ptr(r),\; \forall x \in dom(E)\} \\ ext(e) & = & e \,\cup\, \{r \mid H(a) = cons(v_1, v_2) \;\wedge\; v_1 = ptr(r) \vee v_2 = ptr(r), \\ & & \quad \forall a \in e\},\; \text{where } e \subseteq dom(H) \\ fix(ext, e) & = & \begin{cases} e & \text{if } e = ext(e) \\ fix(ext, ext(e)) & \text{otherwise} \end{cases} \end{array}$$

The set S is thus the set of references in $dom(H)$ that are used in any (possibly cyclic) data structure assigned to a variable in $dom(E)$. It is defined as the fixed point of the function ext applied to the set $\mathcal{S}_0$ of references assigned to variables in $dom(E)$. The function ext extends a set of references e to all the references that are used in the cells pointed to by the references of e.
We can easily prove that this fixed point exists by observing that (1) the function ext is monotonic and (2) the set $dom(H)$ is bounded.

Theorem 6.1[Completeness]
Let E and H be an environment and a heap, and s a statement manipulating the variables in E. If $\langle E, H\rangle\ s\ \langle E', H'\rangle$ then there exists a satisfiable set of constraints C such that $\langle E_v, H_v, true\rangle\ s\ \langle E'_v, H'_v, C\rangle$ with ρ a solution for C such that

$$\langle E, H\rangle \cong \langle \rho(E_v), \rho(H_v)\rangle$$
$$\langle E', H'\rangle \cong \langle \rho(E'_v), \rho(H'_v)\rangle$$

where $(E_v,\ H_v)$ and $(E'_v,\ H'_v)$ are constraint variables representing the initial, respectively final environment and heap during the execution of s.

The completeness property states that any concrete execution of a program s with respect to an initial environment E and heap H is modelled by some abstract derivation represented by a set of constraints C such that there exists a solution to C that models both the initial and final environment and heap. In other words, our method is able to capture *all* executions of a program fragment s. In addition, the soundness property given below states the inverse, namely that our method does not model spurious executions.

Theorem 6.2[Soundness]
Let s be a statement. If $\langle E_v, H_v, true\rangle\ s\ \langle E'_v, H'_v, C\rangle$ and if there exists a solution ρ for the set of constraints C then $\langle \rho(E_v), \rho(H_v)\rangle\ s\ \langle E, H\rangle$ such that

$$\langle E, H\rangle \cong \langle \rho(E'_v), \rho(H'_v)\rangle.$$

where $(E_v,\ H_v)$ and $(E'_v,\ H'_v)$ are constraint variables representing the initial, respectively final environment and heap during the execution of s.

Proofs for Theorems 6.1 and 6.2 will be respectively given in Sections 6.5 and 6.6.

6.3.4 Constraint Propagation

Among the four types of primitive constraints (Section 6.3.2), the equality and inequality constraints are easily defined as Herbrand equality and inequality, and appropriate implementations can be found in Prolog systems as, respectively, unification and the `dif/2` inequality constraint. The constraints on the environment and heap (membership and update of a mapping) on the other

hand are specific to our purpose. We define them in terms of the following propagation rules, that allow us to infer additional constraints:

$$\begin{aligned} (o \mapsto o_1) \in M \wedge (o \mapsto o_2) \in M &\implies o_1 = o_2 \\ M_1 \uplus \{o \mapsto o_1\} = M_2 &\implies (o \mapsto o_1) \in M_2 \\ o \neq o' \wedge M_1 \uplus \{o \mapsto o_1\} = M_2 \wedge (o' \mapsto o_2) \in M_2 &\implies (o' \mapsto o_2) \in M_1 \end{aligned}$$

The above rules are easily implemented as Constraint Handling Rules (Frühwirth 1998). The first rule means that if the same syntactic object o maps to two syntactic object o_1 and o_2 in the same memory (heap or environment), then those two objects o_1 and o_2 are the same object. The second rule is trivial; it mean that if a new state of the memory is obtained by updating it with a mapping $\{o \mapsto o_1\}$, this mapping must necessarily be in the new version of the memory. The last rule is a bit more complicated. It means that if a new state of the memory is obtained by updating it with a mapping $\{o \mapsto o_1\}$, then if a mapping originating from another syntactic object o' is present in the new state of the memory, it was necessarily already present in the previous state.

6.3.5 Search

In order to obtain concrete test cases, our constraint solver has to overcome two forms of non-determinism: 1) the non-determinism inherent to the extended operational semantics, and 2) the non-determinism associated to the selection of concrete values for the program's input. Traditionally, in Constraint Programming a problem with non-deterministic choices is viewed as a (possibly infinite) tree, where each choice is represented by a fork in the tree. Each path from the root of the tree to a leaf represents a particular set of choices, and has zero or one solution. In our context, a solution is of course a concrete test case. As the tree does not imply a particular order on the solutions, we are free to choose any *search strategy*, which specifies how the tree is navigated in search of the solutions. Moreover, since the problem tree can be infinite, we may select an incomplete search strategy, i.e. one that only visits a finite part of the tree. Let us have a more detailed look at these two forms of non-determinism and how they can be handled by a solver.

Non-Deterministic semantics. Several of the language constructs have multiple overlapping rules in the definition of the symbolic semantics. In particular those for if-then-else ((IfThen) and (IfElse)) and while ((WhileT) and (WhileF)) constructs imply alternate execution paths through the program. Also, observe that the while-construct is a possible source of infinity in the problem tree as the latter must in general contain a branch for each possible number of iterations of the loop body. This means that a solver is usually forced to use an *incomplete* search strategy; for example a *depth-bounded* search strategy which does not explore the tree beyond a given depth.
Recall the example in Section 6.3.1 where the while-loop may iterate an arbitrary number of times. A depth-bounded search only considers test cases that involve iterations up to a given bound.

Non-Deterministic Values As the following example shows, even a single execution path can introduce non-determinism in the solving process. Consider the program `y := x.tail`, which has only one execution path. This execution path merely restricts the initial environment and heap to $E_0 = \{\texttt{x} \rightsquigarrow \texttt{ptr}(A), \texttt{y} \rightsquigarrow V_y\}$ and $(A \rightsquigarrow \texttt{cons}(V_h, V_t)) \in H_0$. There are an infinite number of concrete test cases that satisfy these restrictions. Here are just a few:

E_0	H_0
$\{\texttt{x} \rightsquigarrow \texttt{ptr}(\texttt{a1}), \texttt{y} \rightsquigarrow \texttt{nil}\}$	$\{\texttt{a1} \rightsquigarrow \texttt{cons}(0, \texttt{nil})\}$
$\{\texttt{x} \rightsquigarrow \texttt{ptr}(\texttt{a1}), \texttt{y} \rightsquigarrow \texttt{nil}\}$	$\{\texttt{a1} \rightsquigarrow \texttt{cons}(0, \texttt{a1})\}$
$\{\texttt{x} \rightsquigarrow \texttt{ptr}(\texttt{a1}), \texttt{y} \rightsquigarrow \texttt{nil}\}$	$\{\texttt{a1} \rightsquigarrow \texttt{cons}(1, \texttt{nil})\}$
$\{\texttt{x} \rightsquigarrow \texttt{ptr}(\texttt{a1}), \texttt{y} \rightsquigarrow \texttt{ptr}(\texttt{a1})\}$	$\{\texttt{a1} \rightsquigarrow \texttt{cons}(0, \texttt{nil})\}$
$\{\texttt{x} \rightsquigarrow \texttt{ptr}(\texttt{a1}), \texttt{y} \rightsquigarrow \texttt{nil}\}$	$\{\texttt{a1} \rightsquigarrow \texttt{cons}(0, \texttt{ptr}(\texttt{a2})), \texttt{a2} \rightsquigarrow \texttt{cons}(0, \texttt{nil})\}$

There are two kinds of unknown values: unknown integers V_i and unknown references V_r. Integers are easy: non-deterministically assign any natural number to an unknown integer: $\bigvee_{n \in \mathbb{N}} V_i = n$.
For the references the story is more involved. Assume that R is the set of references created so far, r' is a fresh reference, and V_i' and V_r' are fresh unknown integer and reference values. Then there are three assignments for an unknown reference V_r: 1) `nil`, 2) one of the previously created references R, or 3) a new reference r'. In the last case, the heap must contain an additional cell with fresh unknown components. This observation can be formally modelled by the following proposition:

$$V_r = \texttt{nil} \vee (\bigvee_{r \in R} V_r = \texttt{ptr}(r)) \vee (V_r = \texttt{ptr}(r') \wedge (r' \mapsto \texttt{cons}(V_i', V_r')) \in H_0)$$

In practice, we must again restrict ourselves to a finite number of alternatives. We may be interested in only a single solution: either an arbitrary one, one that satisfies additional constraints or one that is minimal according to some criterion. Alternatively, multiple solutions may be desired, each of which *differs sufficiently* from the others based on some measure. All of these preferences can be expressed in terms of suitable search strategies.

6.3.6 Generalized Data Structures

So far we have only considered data structures composed of simple `cons` cells, for the sake of simplicity and concision in the definitions. However, our constraint-based approach can easily be extended to cope with arbitrary structures. Consider for instance this C-like struct for binary trees:

```
struct tree { int value;
              tree left;
              tree right; }
```

In order to deal with the `tree` type defined above , it suffices to extend both the concrete and the constraint semantics of ImpL with 1) a new `tree` constructor

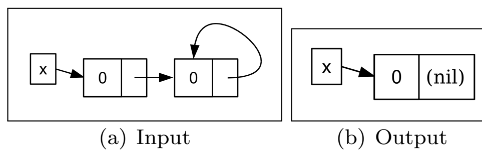
representing a triple and 2) three field selectors (e.g. `value`, `left`, and `right`) similar to the `cons` constructor and the `head` and `tail` selectors. In addition, the search process employed by the solver needs to be adjusted in order to generate arbitrary tree values. An unknown tree value V_t is assigned as follows:

$$V_t = \texttt{nil} \vee (\bigvee_{r \in R_t} V_t = \texttt{ptr}(r)) \vee (V_r = r' \wedge (r' \mapsto \texttt{tree}(V_i', V_l', V_r') \in H_0)$$

where R_t is the set of previously created tree references, r' is a fresh tree reference, and V_i', V_l' and V_r' are respectively a fresh unknown integer value and fresh unknown tree values. It should be clear to the reader that the above approach is easily generalized to arbitrary structures in a datatype-generic manner.
Also, other primitive types such as reals and booleans are easily supported by integrating additional off-the-shelf constraint solvers for them.
Moreover, note that invariants on the data structures, such as acyclicness, *can* be imposed on the unknown input in terms of additional constraints, e.g. provided by the programmer. This allows to seamlessly incorporate specification-level constraints into our method – similarly to (Visser, Păsăreanu, and Khurshid 2004; Offutt and Liu 1999).
The problem of generating arbitrary heap-allocated data structures have been further studied in (Gómez-Zamalloa, Albert, and Puebla 2010), in which the authors generalize the structure of a cell.

6.4 Applications

In this section we propose two applications of our method for test case generation. The first one consists in providing the programmer with (a visualization of) input/output pairs for the program under test satisfying a certain coverage criterion. We have developed a tool that allows to visualise such input/output pairs involving heap-allocated data structures based on Graphviz.[5] This allows the programmer to visually inspect them and verify that the program behaves as expected. For example, Fig. 6.8 depicts an input/output pair for the example program of Section 6.3.1.



(a) Input (b) Output

Figure 6.8: Visualization of an input/output pair for the example program

A second application is the automatic creation of a test suite that can be repeatedly evaluated during regression testing, for example after certain parts of

[5] `http://www.graphviz.org/`

the code have been refactored. The main problem is to translate the data structures originating from a solution to a constraint set into executable code that 1) creates the data structures that are input to the program, and that 2) verifies whether the data structures output by the code correspond to the expected output. Hence, a concrete test case for a program P looks like

$$Setup\,; P\,; Check$$

where *Setup* sets up the initial environment and heap, and *Check* inspects the final environment and heap.

Example 6.2 Consider the simple program `x := nil`. One test configuration consists of an initial environment $E_0 = \{\texttt{x} \rightsquigarrow \texttt{ptr(r1)}\}$ and an initial heap $H_0 = \{\texttt{r1} \rightsquigarrow \texttt{cons}(7, \texttt{ptr(r1)})\}$. The final environment is $E_1 = \{\texttt{x} \rightsquigarrow \texttt{nil}\}$ and the final heap $H_1 = H_0$. The concrete test case for this test configuration looks like the code represented on the left of Figure 6.9. After running this test case, the variable `accept` contains `1` iff the test succeeds; otherwise it contains `0`.

```
  // setup phase
x := new cons(7,nil);
x.tail := x;
  // program under test
x := nil;
  // check phase
if (x == nil) then {
  accept := 1
} else {
  accept := 0
}
```

```
  // setup phase
x := new cons(7,nil);
x.tail := x;
  // program under test
if (x == nil) then {
  foundnil := 1;
  x := nil
} else {
  foundnil := 0;
  x := nil
}
  // check phase
if (x == nil) then {
  accept := 1
} else {
  accept := 0
}
```

Figure 6.9: Testcase for the original (left) and refactored (right) code of Example 6.2.

If the program is changed, e.g. due to refactoring, the existing test case can be used to test the *modified* source code (regression testing). If we replace the program of Example 6.2 above by the refactored version `if (x == nil) { foundnil := 1; x := nil } else { foundnil := 0; x := nil }`, the above test case looks as the code on the right of Figure 6.9. Observe that we consider neither *garbage*, i.e. the parts of the heap H_1 that are unreachable from the environment E_1, nor newly introduced variables such as `foundnil` in the example.

Setup Phase

The inference rules depicted in Figure 6.10 explain how to construct the setup code of the test case from an initial environment E and heap H. The judgement

$H, \emptyset \vdash_s E : s$ expresses that s is the setup code for environment E and heap H. The set of rules basically defines an algorithm that constructs the setup code by generating code for one element of the environment at a time. Note the role of the set A containing the references generated so far.

$$\text{(S-DONE)} \quad \frac{}{H, A \vdash_s \emptyset : \texttt{skip}}$$

$$\text{(S-INT)} \quad \frac{H, A \vdash_s E : s}{H, A \vdash_s \{l \mapsto n\} \cup E : l \texttt{ := } n\texttt{;}s}$$

$$\text{(S-NIL)} \quad \frac{H, A \vdash_s E : s}{H, A \vdash_s \{l \mapsto \texttt{nil}\} \cup E : l \texttt{ := nil;}s}$$

$$\text{(S-NREF)} \quad \frac{a \notin domain(A) \qquad (a \mapsto \texttt{cons}(v_h, v_t)) \in H \qquad H, A \cup \{a \mapsto l\} \vdash_s \{l\texttt{.tail} \mapsto v_t\} \cup E : s}{H, A \vdash_s \{l \mapsto \texttt{ptr}(a)\} \cup E : l \texttt{ := new cons}(v_h, \texttt{nil})\texttt{;}s}$$

$$\text{(S-OREF)} \quad \frac{(a \mapsto l') \in A \qquad H, A \vdash_s E : s}{H, A \vdash_s \{l \mapsto \texttt{ptr}(a)\} \cup E : l \texttt{ := } l'\texttt{;}s}$$

Figure 6.10: Setup Phase Algorithm

The rule (S-DONE) means that if E is empty, then the setup code is `skip` and there is nothing left to be executed. The rules (S-INT) and (S-NIL) mean that if a variable l is bound to an integer n, respectfully the null-pointer `nil` in E, then the setup code is l `:=` n`;`s respectfully l `:= nil;`s where s is the setup code for what remains in E. The rule (S-NREF) means that if a variable l is bound to a pointer in E, and if the reference of the pointer has not been generated yet ($a \notin domain(A)$), then the setup code is l `:= new cons`$(v_h,$ `nil`$)$`;`s where $(a \mapsto \texttt{cons}(v_h, v_t)) \in H$, s is the setup code for what remains in E plus the assignment of the tail of the new cons cell l`.tail` $\mapsto v_t$, and the newly generated reference $\{a \mapsto l\}$ is added to A. Finally, the rule (S-OREF) means that if a variable l is bound to a pointer in E, and if the reference of the pointer has already been generated ($(a \mapsto l') \in A$), then the setup code is l `:=` l'`;`s where l' is the variable that was bound to this reference in the first place.

Check Phase

Likewise, the algorithm given by the inference rules in Figure 6.11 explains how to construct the check code of the test case from the final environment E and heap H. The judgement $H, \emptyset \vdash_c E : s$ expresses that s is the check code for environment E and heap H.
The rule (C-DONE) means that if E is empty, then the check code is `accept` `:=` 1, meaning the success of the check. The rules (C-INT) and (C-NIL) mean that if a variable l is bound to an integer n, respectfully the null-pointer `nil` in E, then the check code is `if` l `==` n `then` s `else accept :=` 0 respectfully `if` l `== nil then` s `else accept :=` 0 where s is the check code for what remains in E. The rule (C-NREF) means that if a variable l is bound to a

$$\text{(C-DONE)}\ \frac{}{H, A \vdash_c \emptyset : \texttt{accept := 1}}$$

$$\text{(C-INT)}\ \frac{H, A \vdash_c E : s}{H, A \vdash_c \{l \mapsto n\} \cup E : \texttt{if}\ l\ \texttt{==}\ n\ \texttt{then}\ s\ \texttt{else accept := 0}}$$

$$\text{(C-NIL)}\ \frac{H, A \vdash_c E : s}{H, A \vdash_c \{l \mapsto \texttt{nil}\} \cup E : \texttt{if}\ l\ \texttt{== nil then}\ s\ \texttt{else accept := 0}}$$

$$\text{(C-NREF)}\ \frac{\begin{array}{c} a \notin domain(A) \qquad (a \mapsto \texttt{cons}(v_h, v_t)) \in H \\ H, A \cup \{a \mapsto l\} \vdash_c \{l\texttt{.head} \mapsto v_h, l\texttt{.tail} \mapsto v_t\} \cup E : s_1 \\ l, s_1 \vdash_n range(A) : s_2 \end{array}}{H, A \vdash_c \{l \mapsto \texttt{ptr}(a)\} \cup E : \texttt{if}\ l\ \texttt{/= nil then}\ s_2\ \texttt{else accept := 0}}$$

$$\text{(C-OREF)}\ \frac{(a \mapsto l') \in A \qquad H, A \vdash_c E : s}{H, A \vdash_c \{l \mapsto \texttt{ptr}(a)\} \cup E : \texttt{if}\ l\ \texttt{==}\ l'\ \texttt{then}\ s\ \texttt{else accept := 0}}$$

$$\text{(N-BASE)}\ l, s \vdash_n \emptyset : s$$

$$\text{(N-REC)}\ \frac{l, s \vdash_n R : s'}{l, s \vdash_n \{l'\} \cup R : \texttt{if}\ l\ \texttt{/=}\ l'\ \texttt{then}\ s'\ \texttt{else accept := 0}}$$

Figure 6.11: Check Phase Algorithm

pointer in E, and the reference of this pointer has not been encountered yet, then we check that (1) l is different than `nil`, (2) the head and the tail of l are bound to the values predicted by H, and (3) l is indeed bound to a new pointer, not another one (already encountered) representing the same data structure. Finally, the rule (C-OREF) means that if a variable l is bound to a pointer in E, and the reference of this pointer has already been encountered, assigned to another variable l', then we check that $l == l'$.

6.5 Proof of Completeness Theorem

Recall the Theorem 6.1 from Section 6.3.3:

Theorem [Completeness]
Let E and H be an environment and a heap, and s a statement manipulating the variables in E. If $\langle E, H\rangle\ s\ \langle E', H'\rangle$ then there exists a satisfiable set of constraints C such that $\langle E_v, H_v, true\rangle\ s\ \langle E'_v, H'_v, C\rangle$ with ρ a solution for C such that

$$\langle E, H\rangle \cong \langle \rho(E_v), \rho(H_v)\rangle$$
$$\langle E', H'\rangle \cong \langle \rho(E'_v), \rho(H'_v)\rangle$$

The proof of this completeness theorem results directly from the proofs of the following lemmas:

Lemma 1

Let E and H be an environment and a heap, and e an expression. If $\langle E, H\rangle\ e \rightsquigarrow v\langle E, H'\rangle$, then if θ is a solution for a satisfiable set of constraints C_1 such that $\langle E, H\rangle \cong \langle \theta(E_v), \theta(H_v)\rangle$ for constraint variables E_v and H_v then there exists a satisfiable set of constraints C_2 such that $\langle E_v, H_v, C_1\rangle\ e \rightsquigarrow v_v\langle E_v, H'_v, C_2\rangle$ with $\theta\rho$ a solution for C_2 such that

$$\langle E, H'\rangle \cong \langle \theta\rho(E_v), \theta\rho(H'_v)\rangle \text{ and } v = \theta\rho(v_v)$$

Proof. The proof is by induction on the structure of the expression e.

The *base cases* are VAR, INT and NIL rules of the expressions. Since neither case updates the heap, we have $H' = H$. In the two last cases (INT and NIL), the proof is direct since those expressions do not imply any constraint on v_v, E_v or H_v. That is, $C_2 = C_1$, $H'_v = H_v$ and $v_v = v$. That means we can simply choose $\rho = \{\}$ and we obtain a solution $\theta\rho$ for C_2 verifying $\theta\rho(v_v) = v$ and $\langle \theta\rho(E_v), \theta\rho(H'_v)\rangle = \langle \theta(E_v), \theta(H_v)\rangle \cong \langle E, H\rangle = \langle E, H'\rangle$. Moreover, for the VAR case, we have $v_v = V$ with V a fresh variable and $C_2 \equiv C_1 \wedge \{(x \mapsto V) \in E_v\}$. If we choose $\rho = \{V/E(x)\}$, we have $\theta\rho$ a solution for C_2 such that $\langle \theta\rho(E_v), \theta\rho(H'_v)\rangle \cong \langle E, H'\rangle$.

We arbitrarily choose the HEAD rule as single *inductive case* for the proof, for convenience. The proof for the other cases can easily be deducted from this one. Suppose

$$\langle E, H\rangle e \rightsquigarrow ptr(r)\langle E, H_2\rangle \text{ and } r \mapsto \texttt{cons}(v_h, v_t) \in H_2$$

By induction hypothesis, we have

$$\langle E_v, H_{v0}, C_0\rangle\ e \rightsquigarrow v_v\langle E_v, H_{v1}, C_1\rangle$$

with a solution ρ_1 such that

$$\langle E, H_2\rangle \cong \langle \rho_1(E_v), \rho_1(H_{v1}\rangle \text{ and } \rho_1(v_v) = ptr(r)$$

We can construct a solution ρ for C_2 where

$$C_2 \equiv C_1 \wedge v = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H_{v1}$$

as follows:

$$\rho = \rho_1 \cup \{R/r, V_t/v_t, V_h/v_h\}.$$

We can easily verify that $\rho(V_h) = v_h$ and $\langle E, H_2\rangle \cong \langle \rho(E_v), \rho(H_{v1})\rangle$. □

Lemma 2

Let E and H be an environment and a heap, and s a statement manipulating the variables in E. If $\langle E, H\rangle\ s\ \langle E', H'\rangle$, then if θ is a solution for a satisfiable set of constraints C_1 such that $\langle E, H\rangle \cong \langle \theta(E_v), \theta(H_v)\rangle$ for constraint variables E_v and H_v then there exists a satisfiable set of constraints C_2 such that $\langle E_v, H_v, C_1\rangle\ s\ \langle E'_v, H'_v, C_2\rangle$ with $\theta\rho$ a solution for C_2 such that

$$\langle E', H'\rangle \cong \langle \theta\rho(E'_v), \theta\rho(H'_v)\rangle$$

Proof. The proof is by induction on the structure of the statement s.

The *base cases* are the SKIP, VARASS, HEADASS and TAILASS rules; the property is only proved for the VARASS rule, for convenience reasons. The proof for the other cases can easily be deducted from this one.
Suppose $\langle E, H_1\rangle e \rightsquigarrow v\langle E, H_2\rangle$. From Lemma 1 we have $\langle E_{v0}, H_{v0}, C_0\rangle\ e \rightsquigarrow v_v\langle E_{v0}, H_{v1}, C_1\rangle$ with a solution ρ_1 such that $\langle E, H_2\rangle \cong \langle \rho_1(E_v), \rho_1(H_{v1}\rangle$ and $\rho_1(v_v) = v$. We can construct a solution ρ for $C_1 \wedge E_{v1} = E_{v0} \uplus \{x \mapsto v_v\}$ as follows:

$$\rho = \rho_1 \cup \{E_{v1}/E_{v0} \uplus \{x \mapsto \rho_1(v_v)\}\}$$

We can easily verify that $\langle E, H_2\rangle \cong \langle \rho(E_v), \rho(H_{v1})\rangle$.

We arbitrarily choose the SEQ rule as single *inductive case* for the proof, for convenience reasons. The proof for the other cases can easily be deducted from this one.
Suppose $\langle E_1, H_1\rangle\ s_1\ \langle E_2, H_2\rangle$ and $\langle E_2, H_2\rangle\ s_2\ \langle E_3, H_3\rangle$. By induction hypothesis, we have $\langle E_{v0}, H_{v0}, \mathit{true}\rangle\ s_1\ \langle E_{v1}, H_{v1}, C_1\rangle$ with a solution ρ_1 such that $\langle E_2, H_2\rangle \cong \langle \rho_1(E_{v1}), \rho_1(H_{v1}\rangle$. We also have $\langle E_{v1}, H_{v1}, C_1\rangle\ s_2\ \langle E_{v2}, H_{v2}, C_2\rangle$ with a solution ρ_2 such that $\langle E_3, H_3\rangle \cong \langle \rho_2(E_{v2}), \rho_2(H_{v2}\rangle$. We can construct a solution $\theta\rho$ for C_2 verifying the property as follows:

$$\rho = \rho_1\rho_2$$

We can directly notice that $\langle E_3, H_3\rangle \cong \langle \rho_2(E_{v2}), \rho_2(H_{v2}\rangle$. □

6.6 Proof of Soundness Theorem

Recall the Theorem 6.2:

Theorem [Soundness]
Let s be a statement. If $\langle E_v, H_v, \mathit{true}\rangle\ s\ \langle E'_v, H'_v, C\rangle$ and if there exists a solution ρ for the set of constraints C then $\langle \rho(E_v), \rho(H_v)\rangle\ s\ \langle E, H\rangle$ such that

$$\langle E, H\rangle \cong \langle \rho(E'_v), \rho(H'_v)\rangle.$$

Before proving the soundness theorem itself, we define and prove the following lemma:

Lemma 3

Let e be an expression. If $\langle E_v, H_v, \mathit{true}\rangle\ e \rightsquigarrow V\ \langle E'_v, H'_v, C\rangle$ and if there exists a solution ρ for the set of constraints C then $\langle \rho(E_v), \rho(H_v)\rangle\ e \rightsquigarrow v\ \langle E, H\rangle$ such that

$$\langle E, H\rangle \cong \langle \rho(E'_v), \rho(H'_v)\rangle$$

and

$$v = \rho(V)$$

Proof. The proof is by induction on the structures of the expressions.

The *base cases* are VAR, INT and NIL rules of the expressions. In those cases, the proof is direct since those expressions do not modify anything among E, H or C. Therefore, if ρ is a valuation for C, we have that $\langle \rho(E_v), \rho(H_v)\rangle\ e \rightsquigarrow v\ \langle \rho(E_v), \rho(H_v)\rangle$. In particular, if e is a variable x, we have $\{x \mapsto V\} \in E$ a constraint of C, with V a fresh variable. Consequently, whatever the valuation for V in ρ, the property holds.

We arbitrary choose the HEAD rule as single *inductive case* for the proof. The proof for the other cases can easily be deducted from this one. By induction hypothesis, we have

$$\langle \rho(E_v), \rho(H_v)\rangle\ e \rightsquigarrow v\ \langle E, H\rangle$$

with $\langle E, H\rangle \cong \langle \rho(E'_v), \rho(H'_v)\rangle$ and ρ is *any* solution for C where

$$\langle E_v, H_v, \mathit{true}\rangle\ e \rightsquigarrow V\ \langle E'_v, H'_v, C\rangle \text{ and } \rho(V) = v$$

Now let us examine

$$\langle E_v, H_v, \mathit{true}\rangle\ e\texttt{.head} \rightsquigarrow V_h\langle E'_v, H'_v, C'\rangle$$

Let ρ' be a solution for C', which (by definition) is equal to $C \wedge v = \texttt{ptr}(R) \wedge (R \mapsto \texttt{cons}(V_h, V_t)) \in H'_v$. Since ρ' is also a solution for C, the induction hypothesis is verified for ρ'. We note that the evaluation of $e\texttt{.head}$ does not change the environment and heap more than the evaluation of e, and therefore we have

$$\langle \rho'(E_v), \rho'(H_v)\rangle\ e\texttt{.head} \rightsquigarrow v_h\ \langle E, H\rangle$$

with $\langle E, H\rangle \cong \langle \rho'(E'_v), \rho'(H'_v)\rangle$. We can also note that whatever the valuation for the fresh variable V_h in ρ', the property $\rho(V_h) = v_h$ holds; the value assigned to R is of no importance here, by definition of $\cong$.

□

Proof of the Theorem

Proof. The proof is by induction on the structures of the statements.

The *base cases* are the Skip, VarAss, HeadAss and TailAss rules; the property is only proved for the VarAss rule, as the other cases can easily be deducted from this one. From the lemma 3, we have

$$\langle \rho(E_v), \rho(H_v) \rangle \ e \rightsquigarrow v \ \langle E, H \rangle$$

with $\langle E, H \rangle \cong \langle \rho(E_v), \rho(H'_v) \rangle$ (note that it is E_v instead of E'_v since the environment is not modified by the evaluation) and ρ is *any* solution for C where

$$\langle E_v, H_v, true \rangle \ e \rightsquigarrow V \ \langle E_v, H'_v, C \rangle \text{ and } \rho(V) = v$$

Let ρ' be a solution for C', which is equal to $C \wedge E'_v = E_v \uplus \{x \mapsto v\}$. Since ρ' is also a solution for C, the lemma is verified for ρ'. We note that according to the semantics of the language, the execution of x := e transforms the environment E to $E \uplus \{x \mapsto v\}$. Since $\langle E, H \rangle \cong \langle \rho'(E_v), \rho'(H'_v) \rangle$ and since $E'_v = E_v \uplus \{x \mapsto v\}$, we have that $\langle E, H \rangle \cong \langle \rho'(E'_v), \rho'(H'_v) \rangle$ and the theorem is therefore verified for x := e.

We arbitrary choose the Seq rule as single *inductive case* for the proof. The proof for the other cases can easily be deducted from this one. The sequence of statements is defined as $s_1;s_2$ where s_1 and s_2 are statements. By induction hypothesis, the proof is verified for $\langle E_v, H_v, true \rangle \ s_1 \ \langle E'_v, H'_v, C_1 \rangle$ and $\langle E'_v, H'_v, true \rangle \ s_2 \ \langle E''_v, H''_v, C_2 \rangle$. Note that the symbolic execution of a statement only *adds* new constraints to the original set of constraints (it never removes or modifies existing ones). Therefore, the symbolic execution

$$\langle E_v, H_v, true \rangle \ s_1;s_2 \ \langle E''_v, H''_v, C \rangle$$

is such that $C = C_1 \wedge C_2$, and thus a solution ρ for C is also a solution for C_1 and C_2. It implies that

$$\langle \rho(E'_v), \rho(H'_v) \rangle \ s_2 \ \langle E, H \rangle$$

is such that

$$\langle E, H \rangle \cong \langle \rho(E''_v), \rho(H''_v) \rangle$$

and thus the property is verified. □

Chapter 7

Mercury normal form

7.1 Introduction and motivation

The problem of deciding whether two code fragments are equivalent, in the sense that they implement the same functionality, is well-known to be undecidable. Nevertheless, there seems to be an interest in developing analyses that are capable to detect such equivalence under particular circumstances and within a certain error margin (Kontogiannis, Demori, Merlo, Galler, and Bernstein 1996; Chen, Francia, Li, McKinnon, and Seker 2004; Wise 1996). Applications can be found in plagiarism detection and tools for program refactoring. Work in this area can be based on parametrised string matching, an example being the MOSS system (Schleimer, Wilkerson, and Aiken 2003), or perform a more involved analysis on a graph representation of a program (Horwitz 1990; Yang 1991). Most of these latter works, including the more recent (Winstead and Evans 2003), concentrate on finding behavioral *differences* between strongly related programs and are often limited to (subsets of) imperative programs.

In (Vanhoof 2005), the professor Vanhoof studied the conditions under which two (fragments of) logic programs can be considered equivalent. The main motivation of that and the current work is to develop an analysis capable of detecting program fragments that are susceptible for refactoring, aiming in particular to the removal of duplicated code or to the generalisation of two related predicates into a new (higher-order) one. The basic idea is as follows: two code fragments (be they goals, clauses or complete predicate definitions) are equivalent if they are *isomorphic* in the sense that one can be considered to be a renaming of the other modulo a permutation of the body goals and the arguments of the

predicate. Take for example the definitions of `app1` and `conc1` below:

```
app1([],Y,Y).
app1([Xe|Xs],Y,[XN|Zs]):- XN is Xe + 1, app1(Xs,Y,Zs).

conc1(A,[],A).
conc1([NB|As],[Be|Bs],C):- conc1(As,Bs,C), NB is Be + 1.
```

Both definitions basically implement the same ternary relation in which one argument is the result of concatenating both other arguments and incrementing each element by one. This can easily be deduced from the source code, since the definition of `conc1` can be obtained from that of `app1` by variable renaming, goal reordering and a permutation of the argument positions. Note that our notion of equivalence is limited to the syntactical equivalence of predicates. Other characteristics like computational complexity etc. are not taken into account. As a second example, let us consider two predicates that do *not* implement the same relation but that nevertheless share a common functionality.

```
rev_all([],[]).
rev_all([X|Xs],[Y|Ys]):- reverse(X,Y), rev_all(Xs,Ys).

add_and_square([],[]).
add_and_square([X|Xs],[Y|Ys]):- N=X+X, Y=N*N, add_and_square(Xs,Ys).
```

The definitions above implement two different relations: `rev_all` reverses all the elements of an input list, while `add_and_square` transforms each element x of an input list into $4x^2$. They nevertheless have a common core which consists of traversing a list and transforming each of its elements. As such, both definitions can be generalised into a single new definition (namely the `map/3` predicate):

```
map([],_,[]).
map([X|Xs],P,[Y|Ys]):- P(X,Y), map(Xs,Ys).
```

and calls to `rev_all` and `add_and_square` can be replaced by calls to `map` with the second argument instantiated to, respectively, `reverse` and a lambda expression `pred(X::in,Y::out) is det :- N=X+X,Y=N*N` (in Mercury syntax). In (Vanhoof 2005) the authors defined an analysis that basically searches for isomorphisms between each possible pair of subgoals in a given program. As outlined above, two goals are isomorphic if they are syntactically identical modulo a renaming and a permutation of the atoms involved. While the analysis *can* be used to search for duplication within two predicate definitions, its complexity – mainly due to the fact that one needs to consider every possible permutation of the predicate's body atoms – renders it hard if not impossible to use in practice. The work we report on in this chapter is motivated by the desire to port the concepts and the analysis of (Vanhoof 2005) to the functional/logic programming language Mercury while, at the same time, rendering such an analysis more practical. This work has been published in (Degrave and Vanhoof 2007b) and (Vanhoof and Degrave 2008). The basic idea is to define a program transformation that reorders clauses, body atoms and predicate arguments in a *unique* and predefined way such that 1) the operational characteristics (well-modedness

and determinism) of the program remain unchanged, but 2) the number of permutations to perform during predicate comparison is substantially reduced.

7.2 Mercury Core Syntax

Recall the superhomogeneous form of Mercury programs, defined in Section 2.1.4. We now define a slightly modified syntax of Mercury programs, derived from that superhomogeneous form, that we call the *Mercury core syntax*[1]. It defines a program as a set of predicate definitions, with each predicate definition consisting of a set of clauses. This core syntax is defined as follows:

Definition 7.1

$$\begin{array}{lll} \mathit{Atom} & ::= & Y = X \mid Y = f(\overline{X}) \mid Y = p(\overline{X}) \mid Z = Y(\overline{X}) \mid p(\overline{X}) \mid Y(\overline{X}) \mid \\ & & \mathit{true} \mid \mathit{fail} \\ \mathit{Goal} & ::= & A \mid (G_1, \ldots, G_n) \mid (G_1; \ldots; G_n) \mid \mathit{not}(G) \mid \mathit{if}(G_1, G_2, G_3) \mid \\ \mathit{Clause} & ::= & p(\overline{X})\,\texttt{:-}\, G. \end{array}$$

where $A \in \mathit{Atom}$, G, $G_i (\forall i) \in \mathit{Goal}$, and X, Y, Z represent variables, $\overline{X}$ a sequence of distinct variables, and f and p respectively a functor and predicate symbol. $\diamond$

The syntax in Definition 7.1 defines a program as a set of predicate definitions, with each predicate definition consisting of a set of clauses. The arguments in the head of each clause and in predicate calls in the body are all distinct variables.
The full Mercury language contains a number of additional constructs, such as function definitions, record syntax, state variables, DCG notation, etc. (Henderson, Conway, Somogyi, Jeffery, Schachte, Taylor, Speirs, Dowd, Becket, and Brown 1996, Mercury reference manual). However, each of these constructions can be translated into the above syntax by introducing new predicates, adding arguments to existing predicates and introducing new unifications (Henderson, Conway, Somogyi, Jeffery, Schachte, Taylor, Speirs, Dowd, Becket, and Brown 1996, Mercury reference manual). Note that these transformations are in principle reversible.

Example 7.1 Let us reconsider the example from above, this time transformed to core syntax:

```
app1(X,Y,Z):-    X=[], Z=Y.
app1(X,Y,Z):-    E=1, Z=[Xn|Zs], X=[Xe|Xs], Xn=(Xe + E), app1(Xs,Y,Zs).

conc1(A,B,C):-   B=[], A=C.
conc1(A,B,C):-   B=[Be|Bs], E=1, Bn=(Be + E), conc1(As,Bs,C), A=[Bn|As].
```

[1]The most important difference being that we still allow for predicates to be defined by multiple clauses rather than by a single clause.

From a programmer's point of view, the order in which the individual goals in a conjunction are written is of no importance. While this is one of the main characteristics that makes the language more declarative than other (logic) programming languages, it clearly renders the search for code isomorphisms in the sense outlined above even more dependent on the need to consider all permutations of the goals within a conjunction.
The fact that Mercury is a strongly moded language provides us with a starting point for our transformation into normal form. As explained in Chapter 2, in Mercury, each predicate has an associated mode declaration[2] that classifies each argument as either input to the call, denoted by `in` (the argument is a ground term before and after the call) or output by the call which is denoted by `out` (the argument is a free variable that will be instantiated to a ground term at the end of the call). Given a predicate's mode declaration it is possible to derive how the instantiation of each variable changes over the execution of each individual goal in the predicate's body. In what follows we will use $\mathbf{in}(G)$ and $\mathbf{out}(G)$ to denote, for a goal G, the set of its input, respectively output variables. As such $\mathbf{in}(G)$ refers to the variables whose values are *consumed* by the goal G, whereas $\mathbf{out}(G)$ refers to the variables whose values are *produced* by G. When appropriate, we will write, for a goal G, $\underline{\mathbf{in}}(G)$ and $\underline{\mathbf{out}}(G)$ to denote the *sequence* of input, respectively output, variables in the order they are occurring in the goal G. For more details about modes and mode analysis in Mercury, we refer to (Overton, Somogyi, and Stuckey 2002b).

Example 7.2 If we consider the `app1` predicate (see Example 7.1) for the mode `app1(in,in,out)` – reflecting the fact that the two first arguments are considered input whereas the third is considered output – we have:

G	$\mathbf{in}(G)$	$\mathbf{out}(G)$
`X = []`	$\{X\}$	$\emptyset$
`Z = Y`	$\{Y\}$	$\{Z\}$
`X = [Xe|Xs]`	$\{X\}$	$\{Xe, Xs\}$
`E = 1`	$\emptyset$	$\{E\}$

G	$\mathbf{in}(G)$	$\mathbf{out}(G)$
`Xn = Xe + E`	$\{Xe, E\}$	$\{Xn\}$
`app1(Xs,Y,Zs)`	$\{Xs, Y\}$	$\{Zs\}$
`Z = [Xn|Zs]`	$\{Xn, Zs\}$	$\{Z\}$

In order to be accepted by the compiler, Mercury programs must be *well-moded*; as explained in Section 2.1.3, that intuitively means that the goals in a predicate's body can be rearranged in such a way that values are produced before they are consumed when the predicate is executed by a left-to-right selection rule (Overton, Somogyi, and Stuckey 2002a). More formally, the well-modedness constraint of a conjunction could be defined as follows:

Definition 7.2 A conjunction $(G_1, ..., G_n)$ *verifies the well-modedness constraint* if

$$\forall 1 \leq i \leq n, \forall k > i : \mathbf{in}(G_i) \cap \mathbf{out}(G_k) = \emptyset.$$

Furthermore, we say that a conjunction is *well-moded* if there exists a reordering of its goals that verifies the well-modedness constraint. ⋄

[2]In general, a predicate may have more than one mode declaration, but these can easily be converted into separate predicate (or, in Mercury terminology, *procedure* definitions.

Example 7.3 When considering the mode `app1(in,in,out)`, the second disjunct of the `app1` definition in Example 7.1 does not verify the well-modedness constraint since the goal `Z=[Xn|Zs]` consumes variables `Xn` and `Zs`, which are both produced by goals further to the right in the conjunction. However, the following reordering *does*:

```
app1(X,Y,Z):- X=[Xe|Xs], E=1, Xn=(Xe + E), app1(Xs,Y,Zs), Z=[Xn|Zs].
```

It is the task of the compiler to rearrange conjunctions in a program such that they verify the well-modedness constraint, thanks to the information provided by the mode analyser.
Note however that well-modedness in itself does not suffice to obtain a *unique* reordering. In the example above one could, e.g. switch the atoms `XN=(Xe + E)` and `app1(Xs,Y,Zs)` while the conjunction would remain well-moded. Consequently, well-modedness can be used as a starting point for our normalization, but it needs to be further constrained in order to obtain a unique reordering.

7.3 Transformation to Normal Form

As a first step in our transformation to normal form, we will assume that programs are converted to disjunctive normal form, in which every clause body is considered to be a conjunction of literals. That is, we restrict the syntax of goals to

$$\begin{array}{lcl} Goal & ::= & (L_1, \ldots, L_n) \\ Literal & := & A \mid not(A) \end{array}$$

where A denotes an atom $\in$ *Atom*, and $L_1,\ldots,L_n$ denote literals in *Literal*.
Note that this transformation can easily be accomplished by flattening conjunctions and disjunctions, replacing if-then-else goals by disjunctions and replacing explicit disjunctions and non-atomic goals within a negation by calls to newly generated predicates.
As a second step in the transformation, we redistribute the atoms of each clause body into a sequential structure, based on a reinforcement of the well-modedness constraint.

Definition 7.3 We define a *proper rearrangement* of a conjunction $L_1, \ldots, L_n$ to be a sequence of multisets $\langle S_1, \ldots, S_k \rangle$ such that

$$\bigcup_{i \in \{1,\ldots,k\}} S_i = \{L_1, \ldots, L_n\}$$

and such that $\forall S_i$ we have

1. $\forall L, L' \in S_i : \mathbf{in}(L) \cap \mathbf{out}(L') = \emptyset$.
2. $\forall L \in S_i, \forall L' \in S_k$ for $k > i : \mathbf{in}(L) \cap \mathbf{out}(L') = \emptyset$.
3. $\forall L \in S_i,\ i > 1 : \exists L' \in S_{i-1} : \mathbf{in}(L) \cap \mathbf{out}(L') \neq \emptyset$.

◇

Intuitively, a conjunction is properly arranged if its components can be partitioned into a sequence of sets of goals such that: (1) there are no dataflow dependencies between the goals in a single set; (2) a goal belonging to a set S_i does not consume values that are produced by a goal belonging to a set S_k that is placed *after* S_i in the sequence; and (3) each goal in a set S_i consumes at least one value that was produced by a goal placed in the previous set S_{i-1}. There are two main points of difference between our notion of a proper arrangement and that of well-modedness. First, we impose an order between *sets* of independent goals and, secondly and more importantly, consumers are pushed forward in the sequence as much as possible.

Example 7.4 Consider the definition of `app1` of Example 7.1. We have that

$$\langle\{\texttt{X} = [], \texttt{Z} = \texttt{Y}\}\rangle$$

is a proper rearrangement of the body of the first clause, whereas

$$\langle\{\texttt{X} = [\texttt{Xe}|\texttt{Xs}], \texttt{E} = \texttt{1}\}, \{\texttt{XN} = (\texttt{Xe} + \texttt{E}), \texttt{app1}(\texttt{Xs}, \texttt{Y}, \texttt{Zs})\}, \{\texttt{Z} = [\texttt{XN}|\texttt{Zs}]\}\rangle$$

is a proper rearrangement of the body of the second clause.

Note that there always exists a proper rearrangement of a well-moded conjunction. Also note that the required partitioning into sets is unique. This observation is captured formally by the following result:

Theorem 7.4 Let $p(\overline{X}) \leftarrow L_1, \ldots, L_m$ be a clause. Then there exists exactly one proper rearrangement of the conjunction $L_1, \ldots, L_m$.

Proof. We split the proof in two parts.

1. We will first proof that there exists a proper rearrangement of the clause body $L_1, \ldots, L_m$. The proof is by construction. Let us define

$$S_1 = \left\{ L \mid \mathbf{in}(L) \subseteq \{\overline{X}\} \right\}$$

and, for $j > 1$,

$$S_j = \left\{ L \mid \mathbf{in}(L) \subseteq \{\overline{X}\} \ \cup \bigcup_{i=1}^{j-1} \mathbf{out}(S_i) \right\} \setminus \bigcup_{i=1}^{j-1} S_i.$$

These sets are well defined. Indeed:

(a) If the clause body is not empty ($m \neq 0$), then $S_1 \neq \emptyset$. Indeed, since the clause is well-moded, we have that if the clause body is not empty, then it should contain at least one literal that either does not consume any values, or that consumes only values provided as argument to the predicate.

(b) Furthermore, for $k > 1$, we have that if $\bigcup_{i=1}^{k-1} S_i \neq \{L_1, ..., L_m\}$, then $S_k \neq \emptyset$. Again, due to well-modedness, among the atoms that are not in $\bigcup_{i=1}^{k-1} S_i$, there is at least one that consumes only values produced before.

From 1a and 1b we can conclude that there exists a finite sequence of non-empty sets $S_1, \ldots, S_n$ (for some $n \geq 1$) such that $\bigcup_{i=1}^{n} S_i = \{L_1, \ldots, L_m\}$. Moreover, by construction we have that

(a) $\forall L, L' \in S_i : \mathbf{in}(L) \cap \mathbf{out}(L') = \emptyset$. It is obviously the case, since a set is constructed by collecting the goals consuming *only* values produced in the sets already constructed.

(b) $\forall L \in S_i, \forall L' \in S_k$ for $k > i : \mathbf{in}(L) \cap \mathbf{out}(L') = \emptyset$. This is obvious for the same reason as the previous point.

(c) $\forall L \in S_j,\ j > 1\ \exists L' \in S_{j-1} :\ \mathbf{in}(L) \cap \mathbf{out}(L') \neq \emptyset$. Indeed, if that was not the case, L would have been integrated into S_{j-1} instead of S_j

2. We will now prove uniqueness of the proper rearrangement. The proof is by contradiction. Let us assume that for a given clause, there exists two different proper rearrangements of the clause body, $PA_1 = Seq_1 = \langle S_1, ..., S_n \rangle$, and $PA_2 = \langle S'_1, ..., S'_m \rangle$. Since $PA_1 \neq PA_2$, we have that $\exists 1 \leq i \leq min\{m, n\},\ S_i \neq S'_i\ and\ S_j = S'_j,\ \forall j < i$. In other words, we take S_i to be the first subset different from S'_i.

   Since $S_i \neq S'_i$, there exists $L \in S'_i$ such that $L \notin S_i$ (or the other way round, in what case the proof is similar). Since PA_1 and PA_2 are proper rearrangements of the same conjunction, the literal L must also occur in PA_1, in a set to the right of $S_i : \exists k > i$ such that $L \in S_k$.

   The fact that the literal L belongs to different sets in both proper rearrangements leads to a contradiction. Since $L \in S_k$, we have that $\exists L' \in S_{k-1},\ \mathbf{in}(L) \cap \mathbf{out}(L') \neq \emptyset$ (in other words, L consumes a value produced by L', which is the second condition for a proper rearrangement). The literal L' necessarily appears in PA_2 as well, in a set S'_k with $k \geq i$, since, again, $S_j = S'_j,\ \forall j < i$. There are two possibilities:

   (a) Either $k = i$. In that case we have that $L' \in S'_i\ and\ L \in S'_i$ which contradicts the fact that PA_2 is a proper rearrangement (first condition: there should be no dataflow dependencies between literals in the same set).

   (b) Or $k > i$, but in that case the literal $L \in S'_i$ consumes a value produced by a literal in L' in a later set ($L' \in S'_k$ with $k > i$) which contradicts the second condition of a proper rearrangement.

□

The above result is important in our setting of constructing a normal form. Intuitively, the fact that a clause has a unique proper rearrangement implies that if two clauses are isomorphic (always in the sense that one being a renaming of the other modulo a permutation of its body literals), then they have the *same* proper rearrangements (modulo renaming).

Example 7.5 Reconsider the definition of `conc1` from Example 7.1. One can easily verify that

$$\langle\{\mathtt{B} = [], \mathtt{A} = \mathtt{C}\}\rangle$$

is a proper rearrangement of the body of the first clause, whereas

$$\langle\{\mathtt{B} = [\mathtt{Be}|\mathtt{Bs}], \mathtt{E} = \mathtt{1}\}, \{\mathtt{Bn} = (\mathtt{Be} + \mathtt{E}), \mathtt{conc1}(\mathtt{As}, \mathtt{Bs}, \mathtt{C})\}, \{\mathtt{A} = [\mathtt{Bn}|\mathtt{As}]\}\rangle$$

is a proper rearrangement of the body of the second clause.

When considering Examples 7.4 and 7.5, it is clear that for verifying whether two predicates implement the same relation, the search for isomorphisms can be limited to a pairwise comparison of the corresponding sets of goals in the predicate's proper rearrangements.
As such our notion of proper rearrangement seems a good starting point for a transformation that aims at rearranging predicate definitions in a unique way. All that remains, is to impose an order on the goals within the individual sets of a proper rearrangement. Since these goals share no dataflow dependencies, we can use any order without influencing well-modedness. We choose lexicographic ordering on goals in tree representation. Formally:

Definition 7.5 Given a literal L, we define its tree representation, denoted $tr(L)$ as a tree over strings defined as follows:

$$\begin{array}{ll} tr(not(L)) = (not, tr(L)) & \\ tr(Y = X) = (unifv, \underline{\mathbf{in}}(Y = X)) & tr(true) = (true) \\ tr(Y = f(\overline{X})) = (unifc, f, \underline{\mathbf{in}}(Y = f(\overline{X}))) & tr(fail) = (fail) \\ tr(Y = p(\overline{X})) = (closc, p, \underline{\mathbf{in}}(Y = p(\overline{X}))) & tr(p(\overline{X})) = (call, p, \underline{\mathbf{in}}(p(\overline{X}))) \\ tr(Z = Y(\overline{X})) = (closv, \underline{\mathbf{in}}(Z = Y(\overline{X}))) & tr(Y(\overline{X}) = (hocall, Y, \underline{\mathbf{in}}(Y(\overline{X})) \end{array}$$

Given two literals L and L', we will write $L < L'$ if and only if $tr(L) <_l tr(L')$ where $<_l$ represents the lexicographic ordering over trees of strings. $\diamond$

Example 7.6 Reconsider the `app1` predicate from Example 7.1 with the mode information as in Example 7.2.
We have $tr(\mathtt{X} = []) = (unifc, [], X)$ and $tr(\mathtt{Z} = \mathtt{Y}) = (unifv, Y)$. Consequently, we have $\mathtt{X} = [] < \mathtt{Z} = \mathtt{Y}$. Likewise, one can easily verify that we have $\mathtt{X} = [\mathtt{Xe}|\mathtt{Xs}] < \mathtt{E} = \mathtt{1}$ and $\mathtt{app1}(\mathtt{Xs}, \mathtt{Y}, \mathtt{Zs}) < \mathtt{Xn} = (\mathtt{Xe} + \mathtt{E})$.

The main idea of imposing an order on the literals of a conjunction, is to be able to limit the search for isomorphisms between two conjunctions to a pairwise comparison of the corresponding literals. As such, when verifying whether two predicates implement the same relation (by verifying whether the two definitions

are isomorphic), there would be no more need to consider all permutations of the body atoms since *if* the two predicate definitions are isomorphic, they should have the *same* normal form (modulo a renaming of the variables). In order to have this characteristic, the order relation $<$ defined on the literals must be total and hence it must take the variables into account. However, since the variable names used in different predicate definitions are usually unrelated, using them might make that the order we get is not the order wanted, as illustrated by the following example.

Example 7.7 Consider the two conjunctions:
$C_1 \equiv$ `A = a, B = b, C = f(A), D = f(B)`
$C_2 \equiv$ `X = b, Y = a, R = f(X), S = f(Y)`
and suppose that the associated mode information is such that the first half of C_1 produces the values for A and B that are consumed in the second half of C_1. Likewise, we assume that the values Y and X are produced in the first half of C_2 and consumed in its second half. In other words, all variables are output variables and the clauses' proper rearrangements are as follows:

$$PA_1 = \langle\{\texttt{A = a,B = b}\}, \{\texttt{C = f(A),D = f(B)}\}\rangle$$
$$PA_2 = \langle\{\texttt{X = b,Y = a}\}, \{\texttt{R = f(X),S = f(Y)}\}\rangle$$

When we use the order relation $<$ defined above to order the individual atoms in each set of the proper rearrangements, we obtain
$C'_1 \equiv$ `A = a, B = b, C = f(A), D = f(B)`
$C'_2 \equiv$ `Y = a, X = b, R = f(X), S = f(Y)`
Even-though the two clauses *are* isomorphic, there does not exist a renaming ρ such that $C'_1\rho = C'_2$. The problem is that the last two literals of C'_2 are in the wrong order with respect to the order chosen for C'_1 due to choice of the variable names.

The example above suggests that rather than basing the order relation on the variable names chosen by the programmer, it would be better to rename the variables in each clause in a consistent way reflecting the data flow within the clause. This is precisely what our transformation to normal form will do. Before we can define the transformation itself, we need one more concept though.

Definition 7.6 Let p/n be a predicate defined in the program. An *argument permutation* for p/n is a bijection over $\{1, \ldots, n\}$. For an argument permutation π, we define the result of *permuting by* π the arguments of a call $p(X_1, \ldots, X_n)$ as the call $p(X_{\pi^{-1}(1)}, \ldots, X_{\pi^{-1}(n)})$. $\diamond$

Example 7.8 The permutation $\pi = \{(1,3), (2,1), (3,2)\}$ is an argument permutation for `conc1`. The result of permuting the arguments of a call $\texttt{conc1(}X_1\texttt{,}X_2\texttt{,}X_3\texttt{)}$ by π is $\texttt{conc1(}X_2\texttt{,}X_3\texttt{,}X_1\texttt{)}$.

We will use the notion of an argument permutation to rearrange the arguments of each predicate in such a way that the arguments are regrouped by their mode and type. For the types, we assume an ordering $<_\tau$ that is defined on all types occurring in the program that is being normalized.

Definition 7.7 Let π be an argument permutation for a predicate p/n. We call π *suitable* if the following conditions hold: let $p(X_{\pi_1}, \ldots, X_{\pi_n})$ denote the result of permuting by π the arguments in a call $c \equiv p(X_1, \ldots, X_n)$, then

1. $\exists k \geq 0$ such that $\{X_{\pi_1}, \ldots, X_{\pi_k}\} = \mathbf{in}(c)$ and $\{X_{\pi_{k+1}}, \ldots, X_{\pi_n}\} = \mathbf{out}(c)$
2. if we denote by τ_i the type of variable X_i in the call, then $\forall 1 \leq i, j \leq k$ and $\forall k+1 \leq i, j \leq n$, if $\pi_i < \pi_j$ then $\tau_{\pi_i} <_\tau \tau_{\pi_j}$.

In other words, an argument permutation is suitable if it places all input arguments in front of the output arguments, and if the input, respectively output, arguments are ordered according to a given ordering on their types. $\diamond$

It is easy to see that the following proposition holds:

Proposition Let p/n be a predicate; then there exists at least one suitable argument permutation π – as described in Definition 7.7 – for this predicate.

Note that the ordering on the predicate arguments defined by a suitable argument permutation is not necessarily unique, if there are multiple arguments having the same type and mode.

Example 7.9 Let us consider the type and mode declarations for the predicates `app1` and `conc1` defined in Example 7.1:

```
:- pred app1(list(int),list(int),list(int)).
:- mode app1(in,in,out) is det.

:- pred conc1(list(int),list(int),list(int)).
:- mode conc1(out,in,in) is det.
```

The argument permutation π given in example 7.8 is a suitable argument permutation for the `conc1` predicate.
The argument permutation $\pi' = \{(1,3),(2,2),(3,1)\}$ is also a suitable argument permutation for `conc1`. The identity function and the permutation $\{(1,2),(2,1),(3,3)\}$ are suitable argument permutations for `app1`.

We are now in a position to define our transformation to normal form. We use the following notation: for a clause c, we use $head(c)$ and $body(c)$ to denote, respectively the head atom and body goal of the clause. If S represents a set of literals and ρ a renaming, then $S\rho$ represents the set of literals obtained by renaming every literal in S by ρ. For a renaming ρ, we represents by $codom(\rho)$ the co-domain of ρ, i.e. $\{V \mid X/V \in \rho\}$. During the transformation, we use a special kind of renaming ρ, in which $codom(\rho)$ is a set of variables of the form V_i for subsequent values of i and V a fresh variable symbol.

Definition 7.9 Let p/n be a predicate and π a suitable argument permutation for p. The *normal form of p w.r.t. π* is obtained by repeatedly applying the following transformation to each clause in the definition of p.

For a clause c, let $h = p(X_{\pi_1}, \ldots, X_{\pi_n})$ denote the result of permuting *head*(c) by π and let $\langle S_1, \ldots, S_m \rangle$ denote the proper rearrangement of *body*(c) in which every recursive call of the form $p(Y_1, \ldots, Y_n)$ is replaced by the atom $rec(Y_{\pi_1}, \ldots, Y_{\pi_n})$.[3] The clause c is transformed into a clause

$$p(X_{\pi_1}, \ldots, X_{\pi_n}) \leftarrow C_1, \ldots, C_m$$

where (for $1 \leq i \leq m$) C_i is a conjunction of literals obtained from S_i in the following way:

$$(C_i, \rho_i) \leftarrow \mathbf{reorder}(S_i, \rho_{i-1})$$

where

1. ρ_0 is a variable renaming for the input arguments of the clause, that is if $\underline{\mathbf{in}}(h) = \langle X_{\pi_1}, \ldots, X_{\pi_k} \rangle$ then
$$\rho_0 = \{X_{\pi_1}/V_1, \ldots, X_{\pi_k}/V_k\}.$$
2. Given a set of literals S and a renaming ρ, the function **reorder** is defined as follows:
$$\mathbf{reorder}(S, \rho) = (C\sigma, \rho \cup \sigma)$$
where the conjunction C is obtained by ordering the literals in $S\rho$ by $<_l$ and if $codom(\rho) = \{V_1, \ldots, V_i\}$ then
$$\sigma = \{O_1/V_{i+1}, \ldots, O_l/V_{i+l}\}$$
for $\langle O_1, \ldots, O_l \rangle = \underline{\mathbf{out}}(C)$.

$\diamond$

Note that the transformation to normal form is such that a unique order is imposed on the body literals of each clause, first by computing the proper rearrangement of the body, and then imposing the lexicographic ordering on the literals in each set of the rearrangement. During the process, variables are systematically and consistently renamed into variables of the form V_i in which the index i represents the order in which the variable is introduced in the (reordered) clause. This renaming scheme allows to abstract from the variable names as they have been introduced by the programmer. As a result, the ordering $<$ (see Definition 7.5) orders identical literals (up to a variable renaming) according to the order in which their respective input arguments appear in the clause.

Example 7.10 Let us reconsider the two clauses C_1 and C_2 from Example 7.7 and their proper rearrangements

$$PA_1 = \langle \{\texttt{A = a}, \texttt{B = b}\}, \{\texttt{C = f(A)}, \texttt{D = f(B)}\} \rangle$$
$$PA_2 = \langle \{\texttt{X = b}, \texttt{Y = a}\}, \{\texttt{R = f(X)}, \texttt{S = f(Y)}\} \rangle$$

[3] We use `rec` to denote a special name, not used in the program being normalized.

It can easily be verified that the transformation as defined above transforms PA_1 into

$C_1' \equiv$ `V1 = a, V2 = b, V3 = f(V1), V4 = f(V2)`

The transformation of PA_2 proceeds as follows. The set {`X = b`, `Y = a`} is transformed into the conjunction `V1 = a, V2 = b` as such creating the renaming $\rho_1 = \{Y/V1, X/V2\}$. This renaming is applied to the second set of literals: {`R = f(X)`, `S = f(Y)`}, giving {`R = f(V2)`, `S = f(V1)`} which is subsequently reordered into the conjunction `S = f(V1), R = f(V2)` and, finally, renamed into `V3 = f(V1), V4 = f(V2)`. As a result, both clauses have an identical normal form.

Also note that recursive calls are replaced by a call to a special (predicate) symbol `rec` and that the arguments are permuted according to π. The use of the symbol `rec` for *each* recursive call, regardless the predicate, makes sure that recursive calls are ordered in a consistent way, regardless the predicate being normalized.
As a final note, observe that the normal form of a predicate is unique *for a given suitable permutation*. Indeed, the choice of another permutation induces another ordering on the renamed variables and, thus, another normal form, as is illustrated by the following examples.

Example 7.11 The `app1` predicate of Example 7.1 has two normal forms, the first one with respect to the identity argument permutation, the second with respect to the permutation $\{1, 2), (2, 1), (3, 3)\}$.

```
app1(V1, V2, V3) :-   V3 = V2, V1 = [].
app1(V1, V2, V8) :-   V3 = 1, V1 = [|](V4, V5), V6 = +(V4, V3),
                      rec(V2, V5, V7), V8 = [|](V6, V7).

app1(V1, V2, V3) :-   V3 = V1, V2 = [].
app1(V1, V2, V8) :-   V3 = 1, V2 = [|](V4, V5), V6 = +(V4, V3),
                      rec(V1, V5, V7), V8 = [|](V6, V7).
```

Likewise, the `conc1` predicate of Example 7.1 has two normal forms, the first one with respect to the argument permutation $\{(1, 3), (2, 2), (3, 1)\}$, the second one with respect to the permutation $\{(1, 3), (2, 1), (3, 2)\}$.

```
conc1(V1, V2, V3) :-   V3 = V1, V2 = [].
conc1(V1, V2, V8) :-   V3 = 1, V2 = [|](V4, V5), V6 = +(V4, V3),
                       rec(V1, V5, V7), V8 = [|](V6, V7).

conc1(V1, V2, V3) :-   V3 = V2, V1 = [].
conc1(V1, V2, V8) :-   V3 = 1, V1 = [|](V4, V5), V6 = +(V4, V3),
                       rec(V2, V5, V7), V8 = [|](V6, V7).
```

The examples above illustrate that the transformation to normal form offers a substantial help for detecting duplicated functionality. Indeed, the transformation makes the existence of isomorphisms explicit in the code, by reordering

and renaming corresponding clause bodies in exactly the same way. In other words, detecting duplication between predicates in normal form does not require to consider permutations of the conjunctions (nor of the predicate arguments), thereby removing a layer of complexity.

7.4 Detecting duplicated functionality and experimental results

The described transformation to normal form was implemented in Mercury. In order to perform some experiments and to provide us with a proof of concept, we have also implemented a number of algorithms for searching for duplicated functionality:

1. **Naïve search**. Basically an implementation of the analysis described in (Vanhoof 2005). The predicates are *not* transformed to normal form, and search is performed by computing all possible permutations of the clauses body atoms.

2. **Identical search**. Predicate definitions are transformed to normal form, possibly resulting in different versions if multiple suitable argument permutations exist. In a next step, each such version of a predicate is compared with each version of the other predicates. Given that the predicates are in normal form, the comparison is a simple check for identity. Consequently, this algorithm is able to detect *duplication* between relations (such as `app1` and `conc1`), but it has no means to detect *similarity* between relations (such as `reverse_all` and `add_and_square`.

3. **Similar search**. Predicate definitions are transformed to normal form as in the **identical search** algorithm. However, in a next step the normal forms are compared using a more involved algorithm that checks whether the corresponding clauses of two predicate definitions in normal form are identical *modulo* 1) variable renaming, and 2) a set of adjacent body atoms (a so-called *gap*). The **similar search** algorithm is capable of detecting duplication between relations (in what case there are *no* gaps) *and* certain forms of similarity.

The **similar search** algorithm basically works as follows:

1. It selects two clauses from different predicates for comparison;

2. It walks the two clauses bodies from the left, comparing each pair of corresponding atoms;

3. Once (and if) an inconsistency (that is two atoms that don't match) is encountered, it starts walking the two bodies from the right;

4. Once an inconsistency is encountered, it then considers the two "gaps", i.e. the sub-conjunction of atoms in each clause body starting, respectively

Program (500 executions)	Naïve search	Normal form	Identical search	Similar search
`app1` and `conc1`	590	180	10	140
`rev_all` and `add_and_sqr`	4250	340	80	420
`member`	809	170	10	140

Table 7.1: Execution time for identical and similar predicates detection.

ending with the first atom involved in the inconsistency encountered when walking each body from the left, respectively right;

5. It then compares the n first atoms of the first gap with the n last atoms of the second one, and vice-versa (n is initialized to the length of the smallest gap);
6. if no match is found, n is decreased by one and point 5 is repeated until $n = 0$ or until a match is found.

This algorithm is therefore able to detect two identical clauses bodies, modulo two sequences of atoms (one in each body) having no match in the other body, and placed *anywhere* in each body.
Table 7.1 provides timings for some basic examples. All times are in milliseconds, and experiments were performed on a Pentium 4 running at 3.06GHz with 1GB of memory. Three examples were tested:

1. The `app1` and `conc1` predicates from Example 7.1.
2. The `rev_all` and `add_and_square` predicates from the introduction.
3. Two different implementations of the `member` predicates, one using an if-then-else, the other using disjunction and negation.

The column labeled **Normal form** represents the time needed for the normal form transformation, the other columns represent the times needed for executing the mentioned algorithms (naïve, identical or similar search). Since only individual predicate definitions are compared, each algorithm was repeatedly executed 500 times in order to obtain a measurable timing.
In the case of the `rev_all`, `add_and_sqr` and `member` examples, the execution times given for the naïve and identical search algorithms represent the times needed to conclude that the given examples do *not* implement duplicated relations, since these algorithms are only able to detect similarities. The similar search algorithm is able to detect that these predicates are identical modulo a *gap*.
Table 7.1 shows that even for these small examples, the transformation to normal form followed by either the identical search or similar search algorithm easily outperforms the analysis of (Vanhoof 2005). This justifies the viability of our current approach.

Program (100 executions)	Naïve search	Normal form	Identical search	**Total**	**Speedup**
2 predicates	**890**	95	10	**105**	**8.5**
5 predicates	**3769**	120	10	**130**	**29.0**
10 predicates	**10,970**	160	60	**320**	**34.3**
20 predicates	**54,860**	310	150	**460**	**119.3**
40 predicates	**243,520**	580	910	**1,490**	**163.4**

Table 7.2: Comparison of the algorithms for identical predicates detection.

In what follows, we compare the performance of our algorithm for detecting duplication with that of the naïve search algorithm when dealing with programs containing several predicate definitions. Note that even if not much duplication is present, all definitions must be pairwise compared in order to drop to conclusions. Table 7.2 contains the execution times of all the algorithms on programs with a different number of predicate definitions. All predicates have 1 or 2 clauses, and each of these clauses has between 3 and 6 atoms. The given execution times represent the total time needed for 100 repeated executions of each algorithm. In this table, the column labeled **Total** represent the time needed for the normalization followed by the identical search algorithm. The column labeled **Speedup** represents the speedup of identical search (with normalization) with respect to the naïve algorithm.
As shown in this table, the transformation of a program into its normal form enables an important speedup when searching for identical predicates across a program. Moreover, this speedup increases strongly when the size of the program to explore increases, showing a reduction in complexity due to dealing with programs in normal form.
Table 7.3 compares the performance of the identical search and similar search algorithms when used on programs containing several predicate definitions. The table shows that the similar search algorithm is substantially slower (about a factor 10). This is as one would expect, given that this algorithm needs to consider renamings and still needs to perform a number of permutations in order to find the smallest *gaps* in two predicate definitions such that their code (these gaps aside) is duplicated.

Program (100 executions)	Identical search	Similar search
2 predicates	10	115
5 predicates	10	240
10 predicates	60	520
20 predicates	150	2090
40 predicates	910	9780

Table 7.3: Comparison of the algorithms for identical and similar predicates detection.

Chapter 8

Conclusion

In this final chapter, we discuss some of our achievements and present different possible directions for future research. In Chapter 2, we have defined a labelled syntax for the Mercury programming language, i.e. a syntax in which the different program points traversed during an execution are represented by labels. Using that syntax, we were able to define a semantics for the Mercury language that provides not only the solutions computed by the program (in the right order), but also captures the exact operational behaviour of a Mercury execution, i.e. provides a representation of the exact execution path followed during the computation. This semantics is the basis on which our framework for the automatic generation of test data is built.

In Chapter 3, we have presented a formalism for the notion of *test case* for Mercury (and subsequently, the notion of *test suite*), in such a way as to make it possible for a tool to automatically process them. The representation we defined for a test case is quite general, since it is represented as a triple containing an identifier for the test case, a Mercury code fragment, and a list of assertions about the expected result of this code fragment. We provided a variety of assertions that would allow one to deal with different kinds of results. One can for example assert the success or the failure of the code fragment (which is particularly useful for *semidet* procedures), as well as the number of solutions (substitutions) returned, or the success of a test involving all/any/a particular solution returned, allowing one to deal with *multi* and *nondet* procedures. Using this formalism, we created a tool able to automatically execute each test case of a test suite, and produce a report about what test case failed and why. Such a framework is widely used for imperative programming languages, yet very few work has been dedicated to this domain in the area of declarative languages and none for Mercury (to the best of our knowledge). The techniques developed to create such frameworks for Prolog or Haskell cannot be reused directly for Mercury because of the particularities of the latter, such as strict type- and mode-checking mechanisms. The tool we wrote transforms a test case into a deterministic procedure returning an indication about the success or failure of the test; this procedure is able to deal with exceptions and I/O operations (in

case the tested procedure is *det* or *cc-multi*).

In Chapter 4 we have defined how one can build a control flow graph for programs written in Mercury. Widely used in the domain of imperative programming as a basis for multiple program analyses and optimizations or as a visual aid for debugging and test generation, control flow graphs are however rarely used in a declarative programming setting. The reason is probably that in most declarative languages, the control structure is much more difficult to apprehend and less relevant than in imperative languages. Though, the fact that Mercury is a *moded* language makes it easier to extract control flow information. We decided therefore to define such a control flow graph for a Mercury procedure in which nodes are constituted by the labels appearing in labelled syntax of this procedure. This was more than a trivial adaptation of the usual notion of control flow graph, since we had to deal with many logic-programming-specific features such as failing goals (which don't necessarily cause the whole procedure to fail) and backtracking. The graph helped us achieve different purposes: first, it obviously provides us with a visual aid that can be particularly useful in the scope of a feature-rich testing framework. Second, it eases the adaptation of concepts existing in the imperative languages area but not defined for declarative languages, such as a range of test coverage criteria. Finally and most importantly, the graph was used as a basis to derive *execution sequences*. We defined the notion of execution sequence as a sequence of *execution segments*, each of which represents a single derivation for a call to the procedure under concern with respect to some (unknown) input values in which for each encountered choicepoint an arbitrary choice is made. The order of the execution segments within an execution sequence is such that it effectively represents a complete derivation tree of the procedure for some input. We formally defined the symbolic execution of a procedure as a set of execution sequences, an proved that for each possible semantics trace (defined in Chapter 2), there exists a bijective correspondence with an execution sequence in this set. Using the notions of labelled syntax and semantics traces, we enhanced our test framework for Mercury with a complementary module that computes the coverage rate with respect to some of the coverage criteria at our disposal – some of which had already been defined in other works (such as the *block count-K criterion* defined in (Albert, Gómez-Zamalloa, and Puebla 2009)), some others were defined by us or adapted from existing coverage criteria based on the control flow of programs written in an imperative programming language (e.g. *call-depth-K*, *backtrack-K*, *procedure*, ... coverage criteria). By automatically instrumenting the source code under test, i.e. adding counters in the superhomogeneous form of the source code placed with respect to the labels in the labelled Mercury syntax defined in Section 2.1.5, we collected a set of complete execution traces (as defined in Chapter 2) for each test case. Indeed, the use of the instrumented code instead of the original one during the execution of a test suite allows us to associate each test case to the sequence of labels encountered during its execution, which corresponds to a semantics trace in which the original substitution is the input

substitution used in the test case. Since there also exists a correspondence between a trace and an execution sequence derived from the control flow graph, we can therefore compare the set of traces collected during the execution of a whole test suite with a set of execution sequences derived from the control flow graph of the procedure under concern, that should be followed in order to satisfy a given coverage criterion.
It should be noted here that the choice to define and use a control flow graph for Mercury programs to serve our purposes has its advantages and its disadvantages. As we stated here, it can be used for a number of applications. However it is not as obvious to use and understand as its counterparts defined on programs written in imperative languages. Indeed, the control flow graph for Mercury is defined not on the original source code of the program but on a transformed code, the so-called superhomogeneous form of Mercury. While this transformation remains easily readable, it is nevertheless different from the original code, and could confuse the programmer willing to use the framework to test his program, since the atoms are transformed and reordered, and the predicates are duplicated in as many procedures as there are modes defined.

The automatic generation of test cases for Mercury was treated in Chapter 5. It starts with the definitions of *segment condition* and *sequence condition*. The latter is the Mercury equivalent of the *path condition* used for symbolic execution of programs written in imperative programming languages. A segment, respectively sequence condition is a set of constraints over (input) variables; input values satisfying those formulas would cause the program to follow the branch, respectively the derivation tree represented by the chosen execution segment, or sequence. The collected constraints are of two types: symbolic constraints and numerical constraints. The latter can be dealt with by existing solvers. Symbolic constraints however need to be solved by a solver able to deal with symbolic types. Therefore, we wrote a solver in CHR using custom constraint propagation rules. And rather than maintaining a set of possible values for a variable, the domain of a variable is the set of possible function symbols. The initial domain of a variable is defined as the set of all the function symbols of the variable's type. This method enables the possibility to generate data of any (even user-defined) symbolic type, since the solver can be very easily and automatically adapted to deal with each necessary type, based on its definition. Note that a side-effect of our approach is that not only test inputs are computed, but also the corresponding outputs (the fact that the predicate fails or succeeds and, in the latter case, what output values are produced). All the programmer has to do in order to construct a test suite is then to check whether the generated output corresponds to what is expected. We have evaluated a prototype implementation that computes a finite set of execution sequences and the associated test inputs (and outputs).

However, if we can generate sets of execution sequences satisfying criteria such as *call-depth-K* and *backtrack-K* coverage criteria, interesting further work

would be the adaptation of some of the many other adequacy criteria existing for imperative languages to fit the declarative languages particularities, as well as the investigation on algorithms that, for each such adequacy criterion, would be able to derive a set execution sequences that should be followed in order to satisfy a given coverage criterion, and therefore the automatic generation of test suites satisfying this criterion.

Another interesting topic for further work would be the extension of the testing framework with a module allowing it to interact with an integrated debugger, in order to try to identify the code fragments that are likely to contain errors, based on the sets of succeeded and failed test cases of an automatically generated test suite. Existing work about automatic debugging such as (Ducassé and Emde 1988) could be a good starting point for this research topic.

Also note that several improvements can be administered to our prototype in order to improve its performance, which has not been particularly stressed since our implementation was meant to be used as a proof of concept only. In particular, one could argue that the decoupling of the derivation of execution sequences and the solving of the associated constraints induces two different search phases, the complexity of which can be particularly expensive in computation time. Moreover, the question of the scalability of the approach can be raised as the number of possible execution paths increases exponentially when dealing with the non-determinism of the approach. We actually have to deal with two forms of non-determinism. The first one is due to the fact that each time an equality test or a deconstruction atom is encountered along an execution path followed, we have to choose whether the atom fails or succeeds. The second form of non-determinism is associated to the selection, during the solving of the constraints, of concrete values for the variables representing the input values of the program.

One possible way to tackle this issue would possibly be the reuse of test suites already generated for a procedure in order to generate test suites for another procedure that calls the first one. One could also take advantage of the mechanism of "modules" featured by Mercury. This feature was created in order to facilitate the creation of large systems, enable separated compilation and support code encapsulation (Henderson, Conway, Somogyi, Jeffery, Schachte, Taylor, Speirs, Dowd, Becket, and Brown 1996). It seems interesting to examine the propagation of the modifications of test suites associated to the different module whenever changes are applied to one of them, instead of restarting the process from scratch.

Another lead for further work is to examine the case of higher-order predicates, and the possibility to generate other procedures as input values for the procedure under test. Some work has already been published in this area, such as (Koopman and Plasmeijer 2006) which examines the problem for functional programming languages; in this work, a transformation is applied to represent input functions as instances of regular algebraic data types, and then use existing test generation data frameworks to generate the functions.

It would also be interesting to study the possibility of generating external data sources, in order to automate the testing process of programs manipulating

databases for example. Such techniques have recently been studied in (Marcozzi, Vanhoof, and Hainaut 2012).

We visited an slightly different approach in Chapter 6 dedicated to the automatic generation of test data for a pointer-based imperative language. In that work, we first defined a small but representative imperative language using data structures based on pointers. We represented the heap and environment of the program by means of symbolic data structures, and used the search strategies of constraint programming to deal with both the collection of the (finite) set of execution paths of the program and the generation of concrete test data. In that configuration, a symbolic execution of the program is defined by sets of constraints on variables representing the successive states of the environment and the heap. Each test (i.e. the *if-then-else* and *while* conditions) encountered in a program execution represents a choice, and the sequence of choices made during the execution determines the path followed. A *path condition* associated to such a path can possibly have an infinity of concrete solutions. Both non-deterministic choices of what path to follow and what concrete value to assign to a constraint variable are depend on the search strategy chosen within the solver used. That is why we can consider this approach as being "parametrized" with the chosen adequacy criteria expressed under the form of CP search strategies. We have proposed two applications of this approach; the first one is to provide the programmer a visualization of input-output pairs for his program. The second one is the creation of a concrete test case for checking a refactored version of the original program. Interesting future work would be the investigation of the exact relation between the use of particular search strategies for constraint solving and different test adequacy criteria. We could also study the possibility of extending the ImpL language to other features of imperative and object-oriented languages.

In the last chapter, Chapter 7, we have presented work that deviates slightly from the topics covered in the other chapters. Still, the treated topic remains in the domain of program analysis, and in particular the analysis of programs written in the Mercury language. We present a program transformation that normalizes a Mercury program by reordering body goals and predicate arguments. The transformation, which preserves the well-modedness and determinism characteristics of the program, aims at reducing the complexity of performing a search for duplicated or similar code fragments between programs. The defined normal form is unique (with respect to a suitable argument permutation). The transformation is implemented and some basic algorithms for the detection of duplicated and similar code have been implemented and evaluated. Topics for further work include the development of more evolved algorithms for the detection of similarities between predicate definitions, based on our normal form. The similar search algorithm that was used to evaluation is a first step in this direction. It is able to detect the similarity between the `reverse_all` and `add_and_square` predicates (showing they can be generalised into `map`) but

a more involved algorithm is needed if more interesting cases of similar code have to be detected.
The normal form as we have defined it is meant to be an *internal* representation that is not shown to the programmer. If the normal form is to be used as a basis for developing tools for program refactoring, it must be investigated if and how the proposed transformation to normal form can be reversed, such that the results of the analysis (indications of what code fragments are identical) can be displayed on the *original* source code rather than the normalized code.
As a last topic for further research, we mention the normalization of type definitions. It needs to be investigated whether such normalization is possible, and in what cases it might be interesting to detect similarities between type definitions, and possibly generalise them into a single more general type definition.

Appendix

Proof of the completeness property (Section 5.3)

In order to ease the notation within the following proof, we define a function $\mathcal{T}race$ which, when applied to the result of the semantics function, provides the complete semantics trace:

$$\mathcal{T}race(\mathcal{S}[\![(G)]\!]\theta) = \langle t_1, \ldots, t_n \rangle$$
$$\text{where } \mathcal{S}[\![(G)]\!]\theta = \langle (t_1, \theta_1), \ldots, (t_n, \theta_n) \rangle$$

We also define a very simple function $\mathcal{R}$ which, when applied to an execution sequence, simply removes the success and failure labels l_S and l_F.

Proof. The proof is by induction on the structure of the goal to execute symbolically.

The *base cases* are $X == Y$, $X \Rightarrow Y$, $X := Y$ and $X \Leftarrow Y$. Here we provide the proof only for $X \Rightarrow Y$ and $X \Leftarrow Y$, since the proves for the respective cases $X == Y$ and $X := Y$ are (nearly) identical.

$$\begin{array}{rcl} \mathcal{S}[\![X \Leftarrow f(\overline{Y})]\!]\theta & = & \langle (\langle\rangle, \theta\{X/f(t_1, \ldots, t_n)\}) \rangle \\ & & \quad \text{where } \forall i : t_i = \theta(Y_i) \\ \Phi(X \Leftarrow f(\overline{Y})) & = & \{\langle\langle l_S \rangle\rangle\} \end{array}$$

We can observe that since the construction $\Leftarrow$ cannot fail, its semantics is a single (empty) trace associated to a substitution. The function Φ returns a single tree with a single branch containing only l_S, as stated by the theorem. Moreover, $\mathcal{R}(\langle\langle l_S \rangle\rangle) = \langle\langle\rangle\rangle$ which corresponds to the semantics trace.

$$\begin{array}{rcl} \mathcal{S}[\![X \Rightarrow f(\overline{Y})]\!]\theta & = & \left\{ \begin{array}{ll} \langle (\langle\rangle, \theta\{\overline{Y}/\overline{t}\}) \rangle & \text{if } \theta(X) = f(\overline{t}) \\ \langle (\langle\rangle, \texttt{Fail}) \rangle & \text{otherwise} \end{array} \right. \\ \Phi(X \Rightarrow f(\overline{Y})) & = & \{\mathcal{T}_1, \mathcal{T}_2\} \\ \text{where } \mathcal{T}_1 & = & \langle\langle l_S \rangle\rangle \\ \mathcal{T}_2 & = & \langle\langle l_F \rangle\rangle \end{array}$$

Since the construction $\Rightarrow$ can succeed or fail, its semantics is defined as two (empty) traces, one being associated to a valid substitution, the other one to the

substitution representing a failure of the derivation. The function Φ returns two trees with a single branch each, containing only l_S respectively l_F, as stated by the theorem. Moreover, $\mathcal{R}(\langle\langle l_S\rangle\rangle) = \langle\langle\rangle\rangle$ and $\mathcal{R}(\langle\langle l_F\rangle\rangle) = \langle\langle\rangle\rangle$ which correspond to the semantics traces.

The inductive cases of the proof are the following:

Goal preceded by a label

$$\begin{array}{rcl} \mathcal{S}[\![({}_lG)]\!]\theta & = & \langle((\langle l\rangle \cdot t_1, \theta_1)\rangle \cdot (\bullet_{i=2}^{n}\langle(t_i,\theta_i)\rangle) \\ & & \text{where } \mathcal{S}[\![G]\!]\theta = \langle(t_1,\theta_1),\ldots,(t_n,\theta_n)\rangle \end{array}$$

$$\begin{array}{rcl} \Phi({}_lG) & = & \langle\mathcal{T}'_1,\ldots,\mathcal{T}'_m\rangle \\ \text{where } \Phi(G) & = & \langle\mathcal{T}_1,\ldots,\mathcal{T}_m\rangle \\ \mathcal{T}'_i & = & \langle\langle l\rangle \cdot \beta^i_1,\ldots,\langle l\rangle \cdot \beta^i_n\rangle \\ \mathcal{T}_i & = & \langle\beta^i_1,\ldots,\beta^i_n\rangle \end{array}$$

Using inductive hypothesis, we have that

$$\exists\mathcal{T}_k \in \Phi(G) \text{ such that } \mathcal{T}race(\mathcal{S}[\![G]\!]\theta) = \mathcal{R}(\mathcal{T}_k)$$

and

$$\begin{array}{l} \text{if } last_{t_j} = l_S, \text{ then } \theta_j \text{ is a valid substitution} \\ \text{if } last_{t_j} = l_F, \text{ then } \theta_j \text{ is } \texttt{Fail} \\ \text{where } \mathcal{T}_k = \langle t_1,\ldots,t_n\rangle \\ \text{and } \mathcal{S}[\![G]\!]\theta = \langle(t'_1,\theta_1),\ldots,(t'_n,\theta_n)\rangle \end{array}$$

We easily observe that:

$$\begin{array}{l} \mathcal{R}(\langle\langle l\rangle \cdot \beta_1,\ldots,\langle l\rangle \cdot \beta_m\rangle) = \langle\langle l\rangle \cdot \beta'_1,\ldots,\langle l\rangle \cdot \beta'_m\rangle \\ \text{where } \mathcal{R}(\langle\beta_1,\ldots,\beta_m\rangle) = \langle\beta'_1,\ldots,\beta'_m\rangle \end{array}$$

Therefore:

$$\begin{array}{l} \mathcal{R}(\langle\langle l\rangle \cdot \beta^k_1,\ldots,\langle l\rangle \cdot \beta^k_n\rangle) = \langle\langle l\rangle \cdot t_1,\ldots,\langle l\rangle \cdot t_i\rangle = \mathcal{T}race(\mathcal{S}[\![({}_lG)]\!]\theta) \\ \text{where } \mathcal{T}_k = \langle\beta^k_1,\ldots,\beta^k_n\rangle \end{array}$$

and since the end of each trace is not transformed, w.r.t. the inductive hypothesis, the following property still holds:

$$\begin{array}{l} \text{if } last_{t_j} = l_S, \text{ then } \theta_j \text{ is a valid substitution} \\ \text{if } last_{t_j} = l_F, \text{ then } \theta_j \text{ is } \texttt{Fail} \\ \text{where } \langle t_1,\ldots,t_n\rangle = \mathcal{T}_k \end{array}$$

Goal surrounded by labels

$$\begin{array}{rcl}\mathcal{S}[\![({}_lG_{l'})]\!]\theta & = & \bullet_{i=1}^{n}\langle(t'_i,\theta_i)\rangle \\ & & \text{where } \mathcal{S}[\![({}_lG)]\!]\theta = \langle(t_1,\theta_1),\ldots,(t_n,\theta_n)\rangle \\ & & \qquad t'_i = \begin{cases} t_i & \text{if } \theta_i = \texttt{Fail} \\ t_i\cdot\langle l'\rangle & \text{otherwise}\end{cases}\end{array}$$

$$\begin{array}{rcl}\Phi({}_lG_{l'}) & = & \langle\mathcal{T}'_1,\ldots,\mathcal{T}'_m\rangle \\ \text{where } \Phi({}_lG) & = & \langle\mathcal{T}_1,\ldots,\mathcal{T}_m\rangle \\ \mathcal{T}'_i & = & \langle\beta'^i_1,\ldots,\beta'^i_n\rangle \\ \mathcal{T}_i & = & \langle\beta^i_1,\ldots,\beta^i_n\rangle \\ \beta'^i_j & = & \begin{cases}\langle l,l_1,\ldots,l_p,l',l_S\rangle & \text{if } \beta^i_j = \langle l,l_1,\ldots,l_p,l_S\rangle \\ \langle l,l_1,\ldots,l_p,l_F\rangle & \text{if } \beta^i_j = \langle l,l_1,\ldots,l_p,l_F\rangle\end{cases}\end{array}$$

From the previous point we have that

$$\begin{array}{rrcl} & \mathcal{T}race(\mathcal{S}[\![({}_lG)]\!]\theta) & = & \mathcal{R}(\mathcal{T}),\ \mathcal{T}\in\Phi({}_lG) \\ \Leftrightarrow & \mathcal{T}race(\langle(t'_1,\theta_1),\ldots,(t'_n,\theta_n)\rangle) & = & \mathcal{R}(\langle\beta_1,\ldots,\beta_n\rangle) \\ \Leftrightarrow & \langle t'_1,\ldots,t'_n\rangle & = & \langle\beta'_1,\ldots,\beta'_n\rangle\end{array}$$

Let us apply a transformation f to both terms of the equality, which is

$$f(x_i) \quad = \quad \begin{cases} x_i & \text{if } \theta_i = \texttt{Fail} \\ x_i\cdot\langle l'\rangle & \text{otherwise}\end{cases}$$

By hypothesis, this transformation can also be defined as follows

$$f(x_i) \quad = \quad \begin{cases} x_i & \text{if } last_{\beta_i} = l_F \\ x_i\cdot\langle l'\rangle & \text{if } last_{\beta_i} = l_S\end{cases}$$

And thus

$$\begin{array}{rrcl} & \langle f(t_1),\ldots,f(t_n)\rangle & = & \langle f(\beta'^1),\ldots,f(\beta'_n)\rangle \\ \Leftrightarrow & \mathcal{T}race(\mathcal{S}[\![({}_lG_{l'})]\!]\theta) & = & \mathcal{R}(\mathcal{T}),\ \mathcal{T}\in\Phi({}_lG_{l'})\end{array}$$

Moreover, since $last_{\beta'_j} = last_{\beta_j}$ by definition, the property

$$\begin{array}{l}\text{if } last_{\beta'_j} = l_S,\ \text{then } \theta_j \text{ is a valid substitution} \\ \text{if } last_{\beta'_j} = l_F,\ \text{then } \theta_j \text{ is } \texttt{Fail} \\ \text{where } \langle\beta'_1,\ldots,\beta'_n\rangle = \mathcal{R}(\mathcal{T})\end{array}$$

still holds.

Conjunction preceded by a label

$$\begin{array}{rcl}\mathcal{S}[\![({}_lG\ ,\ C)]\!]\theta & = & \bullet_{i=1}^{n}\langle(t_{Gi}\cdot t_{Ci1},\theta_{Ci1})\rangle\cdot\langle(t_{Ci2},\theta_{Ci2}),\ldots,(t_{Cim},\theta_{Cim})\rangle \\ & & \text{where } \mathcal{S}[\![({}_lG)]\!]\theta = \langle(t_{G1},\theta_{G1}),\ldots,(t_{Gn},\theta_{Gn})\rangle \\ & & \qquad \mathcal{S}[\![C]\!]\theta_{Gi} = \langle(t_{Ci1},\theta_{Ci1}),\ldots,(t_{Cim},\theta_{Cim})\rangle\end{array}$$

$$\Phi({}_lG\ ,\ C) \quad = \quad \Phi({}_lG)\boxplus\Phi(C)$$

Disjunction

$$\begin{array}{rcl} \mathcal{S}[\![(C\,;\,C')]\!]\theta & = & \mathcal{S}[\![C]\!]\theta \cdot \mathcal{S}[\![C']\!]\theta \\ \mathcal{S}[\![(D\,;\,C)]\!]\theta & = & \mathcal{S}[\![D]\!]\theta \cdot \mathcal{S}[\![C]\!]\theta \end{array}$$

$$\begin{array}{rcl} \Phi(C\,;\,C') & = & \Phi(C) \otimes \Phi(C') \\ \Phi(D\,;\,C) & = & \Phi(D) \otimes \Phi(C) \end{array}$$

We demonstrate the proposition for $(D\,;\,C)$, as the exact same reasoning can be applied to $(C\,;\,C')$. By inductive hypothesis

$$\begin{array}{rcl} \mathcal{T}race(\mathcal{S}[\![D]\!]\theta) & = & \mathcal{R}(\mathcal{T}_k^D),\ \mathcal{T}_k^D \in \Phi(D) \\ \mathcal{T}race(\mathcal{S}[\![C]\!]\theta) & = & \mathcal{R}(\mathcal{T}_p^C),\ \mathcal{T}_p^C \in \Phi(C) \end{array}$$

By definition, $\otimes$ is the "cartesian concatenation" which applies to two sets of sequences, and its result is the result of a cartesian product in which the members of each tuples are concatenated in a single sequence. Therefore we know that

$$\begin{array}{l} \forall \mathcal{T}_D \in \Phi(D),\ \forall \mathcal{T}_C \in \Phi(C): \\ (\mathcal{T}_D \cdot \mathcal{T}_C) \in \Phi(D) \otimes \Phi(C) \end{array}$$

In particular

$$(\mathcal{T}_k^D \cdot \mathcal{T}_p^C) \in \Phi(D) \otimes \Phi(C)$$

By construction, we can easily see that

$$\begin{array}{rcl} \mathcal{T}race(\mathcal{S}[\![D]\!]\theta \cdot \mathcal{S}[\![C]\!]\theta) & = & \mathcal{T}race(\mathcal{S}[\![D]\!]\theta) \cdot \mathcal{T}race(\mathcal{S}[\![C]\!]\theta) \\ \mathcal{R}(\mathcal{T}_k^D \cdot \mathcal{T}_p^C) & = & \mathcal{R}(\mathcal{T}_k^D) \cdot \mathcal{R}(\mathcal{T}_p^C) \end{array}$$

Thus finally

$$\begin{array}{lrcl} & \mathcal{T}race(\mathcal{S}[\![D]\!]\theta \cdot \mathcal{S}[\![C]\!]\theta) & = & \mathcal{R}(\mathcal{T}_k^D \cdot \mathcal{T}_p^C) \\ \Leftrightarrow & \mathcal{T}race(\mathcal{S}[\![(D\,;\,C)]\!]\theta) & = & \mathcal{R}(\mathcal{T}) \text{ with } \mathcal{T} \in \Phi(D\,;\,C) \end{array}$$

And obviously, since no execution segment or substitution has been modified, the property

$$\begin{array}{l} \text{if } last_{\beta_j} = l_S,\ \text{then } \theta_j \text{ is a valid substitution} \\ \text{if } last_{\beta_j} = l_F,\ \text{then } \theta_j \text{ is } \mathtt{Fail} \\ \text{where } \langle \beta_1, \ldots, \beta_n \rangle = \mathcal{T} \\ \text{and } \langle (t_1, \theta_1), \ldots, (t_n, \theta_n) \rangle = \mathcal{S}[\![(D\,;\,C)]\!]\theta \end{array}$$

still holds.

Negation of a conjunction

$$\mathcal{S}[\![not(C)]\!]\theta = \begin{cases} \langle (t_1, \texttt{Fail}), ..., (t_n, \emptyset)\rangle & \text{if } \theta_i = \texttt{Fail},\ \forall 1 \leq i \leq n \\ \langle (t_1, \texttt{Fail}), ..., (t_k, \texttt{Fail})\rangle & \text{if } \theta_1, \ldots, \theta_{k-1} = \texttt{Fail} \\ & \quad \wedge\ \theta_k \neq \texttt{Fail},\ k \leq n \end{cases}$$
$$\text{where } \mathcal{S}[\![C]\!]\theta = \langle (t_1, \theta_1), ..., (t_n, \theta_n)\rangle$$

$$\begin{aligned}
\Phi(\texttt{not}(C)) &= \langle \mathcal{N}(\mathcal{T}_1), \ldots, \mathcal{N}(\mathcal{T}_r)\rangle \\
\text{where } \Phi(C) &= \langle \mathcal{T}_1, \ldots, \mathcal{T}_r\rangle \\
\mathcal{N}(\langle \beta_1, \ldots, \beta_n\rangle) &= \begin{cases} \langle \beta_1, \ldots, \beta_{k-1}, \beta'_k\rangle & \text{if } \exists 1 \leq k \leq n \text{ such that } last_{\beta_k} = l_S \\ & \text{and } last_{\beta_i} \neq l_S\ \forall 1 \leq i \leq k \\ \langle \beta_1, \ldots, \beta_{n-1}, \beta''_n\rangle & \text{otherwise} \end{cases} \\
\beta'_k &= \langle l_1, \ldots, l_m, l_F\rangle \\
\text{where } \beta_k &= \langle l_1, \ldots, l_m, l_S\rangle \\
\beta''_n &= \langle l_1, \ldots, l_p, l_S\rangle \\
\text{where } \beta_n &= \langle l_1, \ldots, l_p, l_F\rangle
\end{aligned}$$

$\square$

Bibliography

Abdennadher, S., T. Frühwirth, and H. Meuss (1999). Confluence and semantics of constraint simplification rules. *Constraints 4*(2), 133–165.

Abrial, J.-R. (1996). *The B-book: assigning programs to meanings.* New York, NY, USA: Cambridge University Press.

Albert, E., M. Gómez-Zamalloa, and G. Puebla (2009). Test data generation of bytecode by clp partial evaluation. pp. 4–23.

Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not. 5*(7), 1–19.

Allen, R. and K. Kennedy (2002). *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers.

Alshraideh, M., L. Bottaci, and B. A. Mahafzah (2010, March). Using program data-state scarcity to guide automatic test data generation. *Software Quality Journal 18*(1), 109–144.

Ambler, S. (1995). Reduce development cost with use-case scenario testing. *Softw. Dev. 3*(7), 53–61.

Apt, K. R. (1990). Logic programming. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pp. 493–574. Elsevier Science Publishers B.V.

Apt, K. R. and M. H. van Emden (1982, July). Contributions to the theory of logic programming. *J. ACM 29*(3), 841–862.

Bailey, M. A. and R. J. Whiddett (1996). Systems maintenance and development methodologies. In *ISCNZ*, pp. 178.

Baldan, D., B. Le Charlier, C. Leclre, and I. Pollet (1999). A step towards a methodology for mercury program construction: A declarative semantics for mercury. In P. Flener (Ed.), *Logic-Based Program Synthesis and Transformation*, Volume 1559 of *Lecture Notes in Computer Science*, pp. 21–40. Springer Berlin - Heidelberg.

Beizer, B. (1990). *Software testing techniques (2nd ed.).* New York, NY, USA: Van Nostrand Reinhold Co.

Beizer, B. (1995). *Black-box testing: techniques for functional testing of software and systems.* New York, NY, USA: John Wiley & Sons, Inc.

Benhamou, F., D. A. McAllester, and P. V. Hentenryck (1994). Clp(intervals) revisited. In *SLP*, pp. 124–138.

Benhamou, F. and W. J. Older (1997). Applying interval arithmetic to real, integer and boolean constraints. *JOURNAL OF LOGIC PROGRAMMING 32*(1), 1–24.

Bently, W. G. and E. F. Miller (1993). Ct coverage – initial results. *Software Quality Journal 2*(1), 29–47.

Berge, C. (1958). *Théorie des graphes et ses applications.* Collection Univesitaire des Mathématiques, Dunod, Paris.

Berztiss, A. T. (1994). Non-functional requirements in the design of software. In J. L. Díaz-Herrera (Ed.), *CSEE*, Volume 750 of *Lecture Notes in Computer Science*, pp. 375–386. Springer.

Bessiere, C. (2006). Constraint propagation. *Foundations of Artificial Intelligence 2*, 29–83.

Bessière, C., J.-C. Régin, R. H. C. Yap, and Y. Zhang (2005, July). An optimal coarse-grained arc consistency algorithm. *Artif. Intell. 165*, 165–185.

Biener, P., F. Degrave, and W. Vanhoof (2010). A test automation framework for mercury. *Proceedings of CoRR 2010*.

Bird, D. L. and C. U. Munoz (1983). Automatic generation of random self-checking test cases. *IBM Syst. J. 22*(3), 229–245.

Botella, B., A. Gotlieb, and C. Michel (2006). Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability journal*. to appear.

Boyapati, C., S. Khurshid, and D. Marinov (2002). Korat: automated testing based on java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, New York, NY, USA, pp. 123–133. ACM.

Boyer, R. S., B. Elspas, and K. N. Levitt (1975). Select—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, New York, NY, USA, pp. 234–245. ACM.

Brayshaw, M. and M. Eisenstadt (1991). A practical graphical tracer for Prolog. *International Journal of Man-Machine Studies 35*(5), 597–631.

Brown, J. R. and M. Lipow (1975). Testing for software reliability. In *Proceedings of the international conference on Reliable software*, New York, NY, USA, pp. 518–527. ACM.

Cadar, C. and D. Engler (2008). Rwset: Attacking path explosion in constraint-based test generation. In *In TACAS08: International Conference on Tools and Algorithms for the Constructions and Analysis of Systems.*

Cameron, M., M. G. de la Banda, K. Marriott, and P. Moulder (2003). Vimer: A visual debugger for Mercury. In *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming.*

Chen, H. Y., T. H. Tse, F. T. Chan, and T. Y. Chen (1998). In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Trans. Softw. Eng. Methodol. 7*(3), 250–295.

Chen, X., B. Francia, M. Li, B. McKinnon, and A. Seker (2004). Shared information and program plagiarism detection. *IEEE Transactions on Information Theory 50(7)*, 1545–1551.

Claessen, K. and J. Hughes (2000). Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, pp. 268–279. ACM.

Clarke, E. M., D. E. Long, and K. L. McMillan (1989). Compositional model checking. In *LICS*, pp. 353–362. IEEE Computer Society.

Clarke, L. A. (1976). A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng. 2*(3), 215–222.

Clarke, L. A., A. Podgurski, D. J. Richardson, and S. J. Zeil (1989). A formal evaluation of data flow path selection criteria. *IEEE Trans. Softw. Eng. 15*(11), 1318–1332.

Conway, T., F. Henderson, and Z. Somogyi (1995). Code generation for Mercury. In J. Lloyd (Ed.), *Proceedings of the International Symposium on Logic Programming*, Cambridge, pp. 242–256. MIT Press.

Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck (1988). An efficient method of computing static single assignment form. Technical report, Providence, RI, USA.

Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare (1972). *Structured programming.* London, UK, UK: Academic Press Ltd.

Davis, C., D. Chirillo, D. Gouveia, F. Saracevic, J. B. Bocarsley, L. Quesada, L. B. Thomas, and M. v. Lint (2009). *Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource.* IBM Press.

Davis, M., G. Logemann, and D. Loveland (1962, July). A machine program for theorem-proving. *Commun. ACM 5*, 394–397.

Davis, M. and H. Putnam (1960). A computing procedure for quantification theory. *Journal of the ACM (JACM) 7*(3), 201–215.

de la Banda, M. G. and E. Pontelli (Eds.) (2008). *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, Volume 5366 of *Lecture Notes in Computer Science.* Springer.

Debray, S. K., W. Evans, R. Muth, and B. De Sutter (2000). Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst. 22*(2), 378–415.

Dechter, R. (1986). *Learning while searching in constraint-satisfaction-problems*. University of California, Computer Science Department, Cognitive Systems Laboratory.

Dechter, R. and D. Frost (1998). Backtracking algorithms for constraint satisfaction problems - a tutorial survey. Technical report.

Degrave, F. (2008). Development of an automatic testing environment for Mercury. See de la Banda and Pontelli (2008), pp. 805–806.

Degrave, F., T. Schrijvers, and W. Vanhoof (2008). Automatic generation of test inputs for mercury. In M. Hanus (Ed.), *LOPSTR*, Volume 5438 of *Lecture Notes in Computer Science*, pp. 71–86. Springer.

Degrave, F., T. Schrijvers, and W. Vanhoof (2009). Towards a framework for constraint-based test case generation. In D. D. Schreye (Ed.), *LOPSTR*, Volume 6037 of *Lecture Notes in Computer Science*, pp. 128–142. Springer.

Degrave, F. and W. Vanhoof (2007a). A control flow graph for Mercury. In *Proceedings of CICLOPS 2007*.

Degrave, F. and W. Vanhoof (2007b). Towards a normal form for mercury programs. In A. King (Ed.), *LOPSTR*, Volume 4915 of *Lecture Notes in Computer Science*, pp. 43–58. Springer.

DeMarco, T. (1996). The role of software development methodologies: Past, present, and future. In *ICSE*, pp. 2–4.

DeMillo, R., D. Guindi, W. McCracken, A. Offutt, and K. King (1988, jul). An extended overview of the mothra software testing environment. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pp. 142 –151.

DeMillo, R., R. Lipton, and F. Sayward (1978). Hints on test data selection: Help for the practicing programmer. *Computer 11*, 34–41.

DeMillo, R. and A. Offutt (1991, Sep). Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on 17*(9), 900–910.

DeMillo, R. A. and A. J. Offutt (1993). Experimental results from an automatic test case generator. *ACM Trans. Softw. Eng. Methodol. 2*(2), 109–127.

Demoen, B., M. G. de la Banda, and P. Stuckey (1999, January). Type constraint solving for parametric and ad-hoc polymorphism. In J. Edwards (Ed.), *the 22nd Australian Computer Science Conference*, pp. 217–228. Springer-Verlag.

Dijkstra, E. W. (1972). Notes on structured programming. pp. 1–82.

Ducassé, M. and A.-M. Emde (1988). A review of automated debugging systems: knowledge, strategies and techniques. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, Los Alamitos, CA, USA, pp. 162–171. IEEE Computer Society Press.

Duran, J. W. and S. Ntafos (1981). A report on random testing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, Piscataway, NJ, USA, pp. 179–183. IEEE Press.

Electrical, I. O. and E. E. (ieee) (1990). *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology.*

Ferguson, R. and B. Korel (1995, oct). Software test data generation using the chaining approach. pp. 703 –709.

Ferguson, R. and B. Korel (1996). The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol. 5*(1), 63–86.

Fine, M. R. (2002). *Beta Testing for Better Software.* New York, NY, USA: John Wiley & Sons, Inc.

Frankl, P. G. and E. J. Weyuker (1988). An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng. 14*(10), 1483–1498.

Frühwirth, T. (1992). Constraint simplification rules. *Constraint Programming: Basics and Trends, Lecture Notes in Computer Science 910.*

Frühwirth, T. (1998). Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming 37*(1–3), 95–138.

Frühwirth, T. (2009). *Constraint handling rules.* Cambridge University Press.

Fujita, H. and I. A. Zualkernan (Eds.) (2008). *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Seventh SoMeT 2008, October 15-17, 2008, Sharjah, United Arab Emirates*, Volume 182 of *Frontiers in Artificial Intelligence and Applications.* IOS Press.

Gargantini, A. and C. Heitmeyer (1999). Using model checking to generate tests from requirements specifications. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, London, UK, pp. 146–162. Springer-Verlag.

Gaschnig, J. (1974). A constraint satisfaction method for inference making. In *Proceedings of the Twelfth Annual Allerton Conference on Circuit Systems Theory*, pp. 866–874.

Gill, A. and C. Runciman (2007). Haskell program coverage. In G. Keller (Ed.), *Haskell*, pp. 1–12. ACM.

Gill, P. E. and W. Murray (1974). *Numerical methods for constrained optimization / edited by P. E. Gill and W. Murray.* Academic Press, London ; New York :.

Ginsberg, M. (1993). Dynamic backtracking. *Arxiv preprint cs/9308101*.

Glass, R. (1997). Software runaways : lessons learned from massive software project failures. Prentice Hall.

Golomb, S. W. and L. D. Baumert (1965, October). Backtrack programming. *J. ACM 12*, 516–524.

Gómez-Zamalloa, M., E. Albert, and G. Puebla (2010). Test case generation for object-oriented imperative languages in clp. *Theory and Practice of Logic Programming 10*(4-6), 659–674.

Goodenough, J. B. and S. L. Gerhart (1975). Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, New York, NY, USA, pp. 493–510. ACM.

Gotlieb, A. (2009). Euclide: A constraint-based testing framework for critical c programs. *Software Testing, Verification, and Validation, 2008 International Conference on 0*, 151–160.

Gotlieb, A., B. Botella, and M. Rueher (1998). Automatic test data generation using constraint solving techniques. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, New York, NY, USA, pp. 53–62. ACM.

Gotlieb, A., B. Botella, and M. Rueher (2000). A clp framework for computing structural test data. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey (Eds.), *Computational Logic*, Volume 1861 of *Lecture Notes in Computer Science*, pp. 399–413. Springer.

Gotlieb, A., B. Botella, and M. Watel (2006, Dec.). Inka: Ten years after the first ideas. In *19th International Conference on Software & Systems Engineering and their Applications (ICSSEA'06)*, Paris, France.

Gotlieb, A., T. Denmat, and B. Botella (2005a). Constraint-based test data generation in the presence of stack-directed pointers. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, New York, NY, USA, pp. 313–316. ACM.

Gotlieb, A., T. Denmat, and B. Botella (2005b). Goal-oriented test data generation for programs with pointer variables. *Computer Software and Applications Conference, Annual International 1*, 449–454.

Gotlieb, A. and M. Petit (2006). Path-oriented random testing. In *RT '06: Proceedings of the 1st international workshop on Random testing*, New York, NY, USA, pp. 28–35. ACM.

Goubault-Larrecq, J. (2001). Well-founded recursive relations. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, London, UK, pp. 484–497. Springer-Verlag.

Gourlay, J. S. (1983). A mathematical framework for the investigation of testing. *IEEE Trans. Softw. Eng. 9*(6), 686–709.

Gravell, A. M. (1990). What is a good formal specification? In J. E. Nicholls (Ed.), *Z User Workshop*, Workshops in Computing, pp. 137–150. Springer.

Gupta, A. and P. Jalote (2008). An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *STTT 10*(2), 145–160.

Gupta, N., A. P. Mathur, and M. L. Soffa (1998). Automated test data generation using an iterative relaxation method. In *In SIGSOFT 98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 231–244. ACM Press.

Gupta, N., A. P. Mathur, and M. L. Soffa (2000). Generating test data for branch coverage. In *In Proc. of the International Conference on Automated Software Engineering*, pp. 219–227. IEEE.

Hager, J. (1989). Software cost reduction methods in practice. *IEEE Transactions on Software Engineering 15*, 1638–1644.

Hall, Jr., A. D. (1971). The altran system for rational function manipulation — a survey. *Commun. ACM 14*(8), 517–521.

Harrold, M. J. (2008). Testing evolving software: Current practice and future promise. In *ISEC '08: Proceedings of the 1st India software engineering conference*, New York, NY, USA, pp. 3–4. ACM.

Henderson, F., T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, C. Speirs, T. Dowd, R. Becket, and M. Brown (1996). The mercury language reference manual. *URL: http://www. cs. mu. oz. au/research/mercury/information/doc/reference manual toc.html*.

Hentenryck, P. V. (1997). Numerica: a modeling language for global optimization. MIT Press.

Hentenryck, P. V. and V. Saraswat (1997, April). Constraint programming: Strategic directions. *Constraints 2*(1), 7–33.

Herington, D. (2002). *HUnit User's Guide.* http://hunit.sourceforge.net/HUnit-1.0/Guide.html.

Herman, P. M. (1976). A data flow analysis approach to program testing. *Australian Computer Journal 8*(3), 92–96.

Hetzel, W. C. and B. Hetzel (1991). *The Complete Guide to Software Testing.* New York, NY, USA: John Wiley & Sons, Inc.

Hong, H. (1992). Non-linear real constraints in constraint logic programming. In *Algebraic and Logic Programming*, Volume 632 of *Lecture Notes in Computer Science*, pp. 201–212. Springer Berlin / Heidelberg.

Horwitz, S. (1990). Identifying the semantic and textual differences between two versions of a program. *ACM SIGPLAN Notices 25*(6), 234–245.

Howden, W. (1975). Methodology for the generation of program test data. *IEEE Transactions on Computers 24*, 554–560.

Howden, W. E. (1977). Symbolic testing and the dissect symbolic evaluation system. *IEEE Trans. Softw. Eng. 3*(4), 266–278.

Hunt, A. and D. Thomas (2003). Pragmatic unit testing in java with junit. Pragmatic Bookshelf.

Hutcheson, M. L. (2003). *Software Testing Fundamentals: Methods and Metrics.* New York, NY, USA: John Wiley & Sons, Inc.

Jaffar, J. and J.-L. Lassez (1987). Constraint logic programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, pp. 111–119. ACM.

Jaffar, J. and M. J. Maher (1994). Constraint logic programming: A survey. *J. Log. Program. 19/20*, 503–581.

Jessop, W. H., J. R. Kane, S. Roy, and J. M. Scanlon (1976). Atlas-an automated software testing system. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, Los Alamitos, CA, USA, pp. 629–635. IEEE Computer Society Press.

Kan, S. H. (2002). *Metrics and Models in Software Quality Engineering.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Khurshid, S. and D. Marinov (2003). Testera: A novel framework for testing java programs. In *In IEEE International Conference on Automated Software Engineering (ASE*, pp. 22–31.

Khurshid, S., C. S. Pasareanu, and W. Visser (2003). Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff (Eds.), *TACAS*, Volume 2619 of *Lecture Notes in Computer Science*, pp. 553–568. Springer.

Kim, H., S. Kang, J. Baik, and I. Ko (2007). Test cases generation from uml activity diagrams. *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, ACIS International Conference on 3*, 556–561.

King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM 19*(7), 385–394.

Kontogiannis, K. A., R. Demori, E. Merlo, M. Galler, and M. Bernstein (1996). Pattern matching for clone and concept detection. *Reverse engineering*, 77–108.

Koopman, P. and R. Plasmeijer (2006). Automatic testing of higher order functions. In *Proceedings of the 4th Asian conference on Programming Languages and Systems*, APLAS'06, Berlin, Heidelberg, pp. 148–164. Springer-Verlag.

Korel, B. (1990). Automated software test data generation. *IEEE Trans. Software Eng. 16*(8), 870–879.

Korel, B. (1992). Dynamic method of software test data generation. *Softw. Test., Verif. Reliab. 2*(4), 203–213.

Korel, B. (1996). Automated test data generation for programs with procedures. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, New York, NY, USA, pp. 209–215. ACM.

Krishnamurthy, D., J. Rolia, and S. Majumdar (2006). A synthetic workload generation technique for stress testing session-based systems. *Software Engineering, IEEE Transactions on 32*(11), 868–882.

Leivant, D. (1983). Structural semantics for polymorphic data types (preliminary report). In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, pp. 155–166. ACM.

Leung, H. K. N. (1992). *A framework for regression testing.* Ph. D. thesis, Edmonton, Alta., Canada.

Leuschel, M. The dppd library of benchmarks. Available at `http://www.ecs.soton.ac.uk/~mal/systems/dppd.html`.

Li, K. and M. Wu (2004). Effective software test automation. Sybex.

Lindgren, T. (1995). Control flow analysis of Prolog. In *International Logic Programming Symposium*, pp. 432–446.

Lindquist, T. E. and J. L. Facemire (1985). Ada-based abstract machine specification of cais to generate validation tests. In *WADAS '85: Proceedings of the second annual Washington Ada symposium on Ada*, New York, NY, USA, pp. 173–178. ACM.

Lindquist, T. E. and J. R. Jenkins (1988). Test-case generation with iogen. *IEEE Software 5*, 72–79.

Linzhang, W., Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang (2004). Generating test cases from uml activity diagram based on gray-box method. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference*, Washington, DC, USA, pp. 284–291. IEEE Computer Society.

Lloyd, J. W. (1987a). Declarative error diagnosis. *New Generation Computing 5*, 133–154.

Lloyd, J. W. (1987b). *Foundations of Logic Programming.* Springer-Verlag.

Mackworth, A. (1977). Consistency in networks of relations. *Artificial intelligence 8*(1), 99–118.

Maclarty, I. D. (2005). Practical declarative debugging of mercury programs. Technical report.

Majumdar, R. and K. Sen (2007). Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, Washington, DC, USA, pp. 416–426. IEEE Computer Society.

Malaiya, Y. K. and Y. K. Malaiya (1996). Antirandom testing: Getting the most out of black-box testing.

Marcozzi, M., W. Vanhoof, and J.-L. Hainaut (2012). Test input generation for database programs using relational constraints. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, DBTest '12, New York, NY, USA, pp. 6:1–6:6. ACM.

Marriott, K. and P. J. Stuckey (1998). *Programming with constraints : an introduction / Kim Marriott and Peter J. Stuckey*. MIT Press, Cambridge, Mass. :.

Massol, V. and T. Husted (2003). *JUnit in Action*. Greenwich, CT, USA: Manning Publications Co.

McMahon, C. (2009). History of a large test automation project using selenium. In *AGILE '09: Proceedings of the 2009 Agile Conference*, Washington, DC, USA, pp. 363–368. IEEE Computer Society.

Meek, B. and K. K. Siu (1989). The effectiveness of error seeding. *SIGPLAN Not. 24*(6), 81–89.

Miller, Jr., E. F. and R. A. Melton (1975). Automated generation of testcase datasets. In *Proceedings of the international conference on Reliable software*, New York, NY, USA, pp. 51–58. ACM.

Miller, W. and D. Spooner (1976). Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering 2*, 223–226.

Mohagheghi, P. (2008). Evaluating software development methodologies based on their practices and promises. See Fujita and Zualkernan (2008), pp. 14–35.

Muchnick, S. (1997). *Compiler Design and Implementation*. Morgan Kaufmann Publishers.

Müller, R. A., C. Lembeck, and H. Kuchen (2004). A symbolic java virtual machine for test case generation. In *IASTED Conf. on Software Engineering*, pp. 365–371.

Mycroft, A. and R. A. O'Keefe (1984). A polymorphic type system for PROLOG. *Artificial Intelligence 23*(3), 295–307.

Myers, G. J. (1979). *Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc.

Offutt, A. J. (1989). The coupling effect: fact or fiction. *SIGSOFT Softw. Eng. Notes 14*(8), 131–140.

Offutt, A. J. (1991). An integrated automatic test data generation system. *Journal of Systems Integration 1*, 391–409.

Offutt, A. J. and S. Liu (1999). Generating test data from sofl specifications. *The Journal of Systems and Software 49*, 49–62.

Omar, F. and S. Ibrahim (2010). Designing test coverage for grey box analysis. In *QSIC '10: Proceedings of the 2010 10th International Conference on Quality Software*, Washington, DC, USA, pp. 353–356. IEEE Computer Society.

Overton, D., Z. Somogyi, and P. Stuckey (2002a). Constraint-based mode analysis of Mercury. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, New York, NY, USA, pp. 109–120. ACM Press.

Overton, D., Z. Somogyi, and P. J. Stuckey (2002b). Constraint-based mode analysis of Mercury. In C. Kirchner (Ed.), *Proceedings of the Fourth International Conference on Principles and Practice of Declarative Programming*, pp. 109–120. ACM.

Parrish, A. S. and S. H. Zweben (1995). On the relationships among the all-uses, all-du-paths, and all-edges testing criteria. *IEEE Trans. Softw. Eng. 21*(12), 1006–1009.

Pesant, G. and M. Gendreau (1996). A view of local search in constraint programming. In *Principles and Practice of Constraint ProgrammingCP96*, pp. 353–366. Springer.

Pfenning, F. (Ed.) (1992). *Types in Logic Programming.* MIT Press.

Pressman, R. S. (2001). *Software Engineering: A Practitioner's Approach.* McGraw-Hill Higher Education.

Ramamoorthy, C., S.-B. Ho, and W. Chen (1976, dec.). On the automated generation of program test data. *Software Engineering, IEEE Transactions on SE-2*(4), 293 – 300.

Rapps, S. and E. J. Weyuker (1982). Data flow analysis techniques for test data selection. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, Los Alamitos, CA, USA, pp. 272–278. IEEE Computer Society Press.

Rossi, F., P. v. van Beek, and T. Walsh (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* New York, NY, USA: Elsevier Science Inc.

Royce, W. W. (1987). Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, Los Alamitos, CA, USA, pp. 328–338. IEEE Computer Society Press.

Runeson, P. (2006). A survey of unit testing practices. *IEEE Softw. 23*(4), 22–29.

Samuel, P., R. Mall, and P. Kanth (2007). Automatic test case generation from uml communication diagrams. *Inf. Softw. Technol. 49*(2), 158–171.

Scheid, F. (1968). *Outline of Theory and Problems of Numerical Analysis.* New York, NY, USA: McGraw-Hill.

Schiex, T. and G. Verfaillie (1993). Nogood recording for static and dynamic constraint satisfaction problems. In *Tools with Artificial Intelligence, 1993. TAI'93. Proceedings., Fifth International Conference on*, pp. 48–55. IEEE.

Schimpf, J. test_util library for ECLiPSe Prolog. http://www.win.tue.nl/~setalle/lp/eclipse-doc/bips/lib/test_util/.

Schleimer, S., D. Wilkerson, and A. Aiken (2003). Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of Data*, San Diego, CA.

Shan, J.-H., J. Wang, and Z.-C. Qi (2001). On path-wise automatic generation of test data for both white-box and black-box testing. In *APSEC '01: Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, Washington, DC, USA, pp. 237. IEEE Computer Society.

Sneyers, J., P. Van Weert, T. Schrijvers, and L. De Koninck (2010). As time goes by: Constraint handling rules. *TPLP 10*(1), 1–47.

Somogyi, Z. et al. The Melbourne Mercury compiler, release 0.9.

Somogyi, Z., F. Henderson, and T. Conway (1994). The implementation of Mercury, an efficient purely declarative logic programming language. In *Proceedings of the ILPS'94 Postconference Workshop on Implementation Techniques for Logic Programming Languages.*

Somogyi, Z., F. Henderson, and T. Conway (1995). Logic programming for the real world. In *Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming.*

Somogyi, Z., F. Henderson, and T. Conway (1996). The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 29*(1–3), 17–64.

Somogyi, Z., F. Henderson, T. Conway, A. Bromage, T. Dowd, D. Jeffery, P. Ross, P. Schachte, and S. Taylor (1996). Status of the Mercury system. In *Proceedings of the JICSLP'96 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages.*

Stallman, R. and G. Sussman (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial intelligence 9*(2), 135–196.

Sy, N. T. and Y. Deville (2001). Automatic test data generation for programs with integer and float variables. *Automated Software Engineering, International Conference on 0*, 13.

Sy, N. T. and Y. Deville (2003). Consistency techniques for interprocedural test data generation. In *In Proc. Joint 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE03*, pp. 03.

Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project* (7007.011).

Tsang, E. (1993). *Foundations of Constraint Satisfaction.* Academic Press.

van Dongen, M. R. C. (2003). Domain-heuristics for arc-consistency algorithms. In *Proceedings of the 2002 Joint ERCIM/CologNet international conference on Constraint solving and constraint logic programming*, ERCIM'02/CologNet'02, Berlin, Heidelberg, pp. 62–75. Springer-Verlag.

Van Hentenryck, F. (1987). A theoretical framework for consistency techniques in logic programming. In *Proceedings of the 10th international joint*

conference on Artificial intelligence - Volume 1, San Francisco, CA, USA, pp. 2–8. Morgan Kaufmann Publishers Inc.

Van Hentenryck, P. (1989). *Constraint satisfaction in logic programming.*

Vanhoof, W. (2005). Searching semantically equivalent code fragments in logic programs. In S. Etalle (Ed.), *Proceedings of LOPSTR 2004*, Volume 3573 of *Lecture Notes in Computer Science*, pp. 1–18. Springer-Verlag.

Vanhoof, W. and F. Degrave (2008). An algorithm for sophisticated code matching in logic programs. See de la Banda and Pontelli (2008), pp. 785–789.

Visser, W., C. S. Păsăreanu, and S. Khurshid (2004). Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes 29*(4), 97–107.

Visvanathan, S. and N. Gupta (2002). Generating test data for functions with pointer inputs. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, Washington, DC, USA, pp. 149. IEEE Computer Society.

Voges, U., L. Gmeiner, and A. V. Mayrhauser (1980). Sadat-an automated testing tool. *IEEE Transactions on Software Engineering 6*, 286–290.

Weyuker, E. J. (1982). On testing non-testable programs. *Comput. J. 25*(4), 465–470.

Wielemaker, J. (2006). *Prolog Unit Tests. Manual.* http://www.swi-prolog.org/packages/plunit.html.

Winstead, J. and D. Evans (2003). Towards differential program analysis. In *Proceedings of the 2003 Workshop on Dynamic Analysis.*

Wise (1996). YAP3: Improved detection of similarities in computer program and other texts. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education) 28.*

Woodward, M. R., D. Hedley, and M. A. Hennell (1980). Experience with path analysis and testing of programs. *IEEE Trans. Softw. Eng. 6*(3), 278–286.

Wu, S. H., S. Jandhyala, Y. K. Malaiya, and A. P. Jayasumana (2008). Antirandom testing: a distance-based approach. *VLSI Des. 2008*(2), 1–9.

Yang, W. (1991). Identifying syntactic differences between two programs. *Software Practice and Experience 21*(7), 739–755.

Yin, H., Z. Lebne-Dengel, and Y. Malaiya (1997, 2-5). Automatic test generation using checkpoint encoding and antirandom testing. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pp. 84 –95.

Yourdon, E. (1977). The choice of new software development methodologies for software development projects. In *AFIPS National Computer Conference*, Volume 46 of *AFIPS Conference Proceedings*, pp. 261–265. AFIPS Press.

Zhao, R. and Q. Li (2007). Automatic test generation for dynamic data structures. *Software Engineering Research, Management and Applications, ACIS International Conference on 0*, 545–549.

Zhu, H., P. A. V. Hall, and J. H. R. May (1997). Software unit test coverage and adequacy. *ACM Comput. Surv. 29*(4), 366–427.