

Constraint-based Test Generation for Pointer-based Data Structures

Tom Schrijvers^{*1}, François Degraeve^{**2}, and Wim Vanhoof²

¹ Department of Computer Science
Katholieke Universiteit Leuven

² Faculty of Computer Science
University of Namur

Abstract. In this paper, we propose an approach for automated test case generation based on techniques from Constraint Programming. We show how an imperative language can be extended with constraint variables and its operational semantics with constraints. By running a program in this extended language, we obtain constraints on the initial environment and heap. Solving these constraints, we obtain concrete test cases for the program. Our technique can handle primitive data such as integers, as well as arbitrary pointer-based data structures allocated on the heap. Standard CP search strategies can be used to express preferences on the generated test cases and to obtain the desired degree of coverage.

1 Introduction

It is a well-known fact that a substantial part of a software development budget (estimates range from 50% to 75% (1)) is spent on *corrective maintenance* – the act of correcting errors in the software under development. Arguably the most commonly applied strategy for finding errors and thus producing (more) reliable software is testing. Testing refers to the activity of running a software component with respect to a well-chosen set of inputs and comparing the outputs that are produced with the expected results in order to find errors. While the fact that the system under test successfully passes a large number of tests does not prove correctness of the software, it nevertheless increases confidence in its correctness and reliability (2).

In testing terminology, a *test case* for a software component refers to the combination of a single test input and the expected result whereas a *test suite* refers to a collection of individual test cases. Evaluating a test suite is a process that can easily be automated by running the software component under test once for each test input from the suite and comparing the obtained result with the expected result as recorded in the test case. The hard part of the testing process is *constructing* a test suite, which comprises finding a suitable set of test

* Post-doctoral researcher of the Fund for Scientific Research - Flanders.

** Supported by a grant FRiA - Belgium.

inputs either based on the specification (*blackbox* testing) or on the source code of program (*whitebox* or *structural* testing). In the latter approach, which we follow in this work, the objective is to create a set of test inputs satisfying a set of *adequacy criteria*. According to (3), an adequacy criterion is considered to be a stopping rule that determines whether sufficient testing has been done. In practice, the most used criteria are different *coverage* criteria, such as *statement*, *branch* or *path coverage* criteria.

In this work, we present a technique for automatically generating test inputs for programs written in an imperative language dealing with pointer-based data structures. It is based on our prior work (4) for the declarative language Mercury. Most existing work on the subject concentrates on generating numerical values – sometimes including floating point and boolean values – and possibly simple data types such as lists. A notable exception is (5) which is capable of generating complex data structures, but which requires the user to introduce complex preconditions for the methods defined in the program. In our work, the use of Constraint Programming (CP) and its inherent mechanisms facilitate dealing with 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. Secondly, 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. Our specific contributions are:

- We show how to extend the semantics of an imperative language to deal with unknown pointer-based input values. (Section 4.1)
- We show how concrete test cases satisfying adequacy criteria can be generated by using a suitable CP search strategy. (Section 4.4)
- We generalize from (possibly cyclic) linked lists to general pointer-based data structures. (Section 4.5)
- We present a visualization tool and a regression test generator based on our approach. (Section 5)

2 Overview

Let us consider a simple example in order to illustrate our constraint-based approach. The following program is written in our prototype imperative language IMPL whose syntax and semantics will be introduced in a following section. The 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.

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

```

    };
x.tail := nil

```

For example, the effect of running this program with x the list $[1, 2, 3, 3, 4]$, is that after the while loop x will have the value $[1, 2, 3]$.³ Now, in our constraint-based approach, we do not run the program with a concrete value for x , but rather with an unknown value, represented by a *constraint variable* V .

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 under the form of constraints on the input V . Among the infinite number of execution paths of our example program, a particular path would execute the *while* condition three times, and the loop body twice. This would imply that the input 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 third. 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 depth-bounded search. Figure 1 illustrates the search tree for the example.

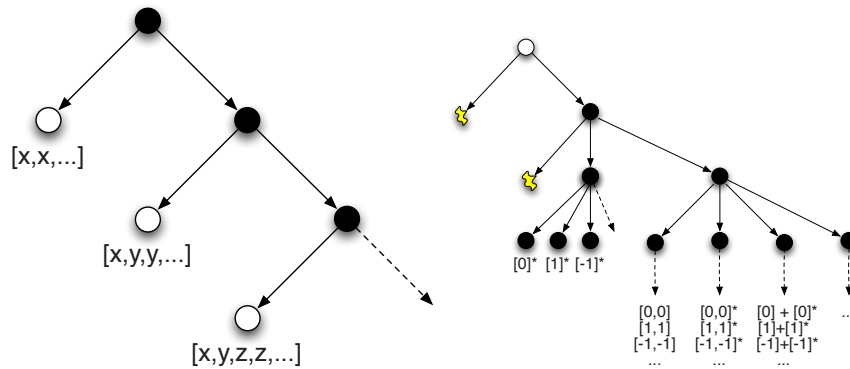


Fig. 1. Tree of paths (left) and tree of value assignments for left-most path (right).

³ We use the notation $[1, 2, 3]$ for the nil-terminated linked list with successive elements 1, 2, 3.

3 The Impl language

In order to focus on the essence of constraint-based test generation for imperative languages, we define a small imperative language IMPL. Programs in IMPL can manipulate integer values and pointer-based data structures constructed from simple “cons” cells having two fields that we will name *head* and *tail*. We show in Section 4.5 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. The syntax of the IMPL language is summarized in the following table:

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

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 ($e.\mathit{head}$), respectively tail ($e.\mathit{tail}$) field of the cons cell referenced by e , equality and inequality tests ($==$ and \neq), and the arithmetic operator for addition $+$.⁴ We will assume that Impl is simply typed and only allows comparison of two values belonging to the same type (either integers or references). Moreover, arithmetic is only allowed on integer values; the language does not support pointer arithmetics.

A program in IMPL is 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.

Figures 2 and 3 capture the semantics of the IMPL language. Execution of a 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 **ptr**(r) with r a unique value denoting the 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).

A judgment of the form $\langle E, H_0 \rangle e \rightsquigarrow v; H_1$ denotes that expression e evaluates to value v with respect to 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

⁴ Other arithmetic operators are omitted in order to keep the formal definition of the semantics small, but they can be added at will.

(VAR)	$\frac{(x \mapsto v) \in E}{\langle E, H \rangle x \rightsquigarrow v; H}$
(INT)	$\frac{n \in \mathbb{Z}}{\langle E, H \rangle n \rightsquigarrow n; H}$
(NIL)	$\langle E, H \rangle \text{nil} \rightsquigarrow \text{nil}; H$
(CONS)	$\frac{\langle E, H_1 \rangle e_1 \rightsquigarrow v_1; H_2 \quad \langle E, H_2 \rangle e_2 \rightsquigarrow v_2; H_3 \quad r \text{ fresh}}{\langle E, H_1 \rangle \text{new cons}(e_1, e_2) \rightsquigarrow \text{ptr}(r); H_3 \uplus \{r \mapsto \text{cons}(v_1, v_2)\}}$
(HEAD)	$\frac{\langle E, H_1 \rangle e \rightsquigarrow \text{ptr}(r); H_2 \quad (r \mapsto \text{cons}(v_h, v_t)) \in H_2}{\langle E, H_1 \rangle e.\text{head} \rightsquigarrow v_h; H_2}$
(TAIL)	$\frac{\langle E, H_1 \rangle e \rightsquigarrow \text{ptr}(r); H_2 \quad (r \mapsto \text{cons}(v_h, v_t)) \in H_2}{\langle E, H_1 \rangle e.\text{tail} \rightsquigarrow v_t; H_2}$
(EQUALT)	$\frac{\langle E, H_1 \rangle e_1 \rightsquigarrow v_1; H_2 \quad \langle E, H_2 \rangle e_2 \rightsquigarrow v_2; H_3 \quad v_1 \equiv v_2}{\langle E, H_1 \rangle e_1 == e_2 \rightsquigarrow 1; H_3}$
(EQUALF)	$\frac{\langle E, H_1 \rangle e_1 \rightsquigarrow v_1; H_2 \quad \langle E, H_2 \rangle e_2 \rightsquigarrow v_2; H_3 \quad v_1 \neq v_2}{\langle E, H_1 \rangle e_1 == e_2 \rightsquigarrow 0; H_3}$
(NEQUALT)	$\frac{\langle E, H_1 \rangle e_1 \rightsquigarrow v_1; H_2 \quad \langle E, H_2 \rangle e_2 \rightsquigarrow v_2; H_3 \quad v_1 \neq v_2}{\langle E, H_1 \rangle e_1 /= e_2 \rightsquigarrow 1; H_3}$
(NEQUALF)	$\frac{\langle E, H_1 \rangle e_1 \rightsquigarrow v_1; H_2 \quad \langle E, H_2 \rangle e_2 \rightsquigarrow v_2; H_3 \quad v_1 \equiv v_2}{\langle E, H_1 \rangle e_1 /= e_2 \rightsquigarrow 0; H_3}$

Fig. 2. Semantics of expressions in IMPL

(VARASS)	$\frac{\langle E, H_1 \rangle e \rightsquigarrow v; H_2}{\langle E, H_1 \rangle \{x := e \langle E \uplus \{x \mapsto v\}, H_2 \}}$
(HEADASS)	$\frac{\langle E, H_1 \rangle e \rightsquigarrow v; H_2 \quad \langle E, H_2 \rangle l \rightsquigarrow \text{ptr}(r); H_2 \quad (r \mapsto \text{cons}(v_h, v_t)) \in H_2}{\langle E, H_1 \rangle l.\text{head} := e \langle E, H_2 \uplus \{r \mapsto \text{cons}(v, v_t)\}}$
(TAILASS)	$\frac{\langle E, H_1 \rangle e \rightsquigarrow v; H_2 \quad \langle E, H_2 \rangle l \rightsquigarrow \text{ptr}(r); H_2 \quad (r \mapsto \text{cons}(v_h, v_t)) \in H_2}{\langle E, H_1 \rangle l.\text{tail} := e \langle E, H_2 \uplus \{r \mapsto \text{cons}(v_h, v)\}}$
(SEQ)	$\frac{\langle E_1, H_1 \rangle s_1 \langle E_2, H_2 \rangle \quad \langle E_2, H_2 \rangle s_2 \langle E_3, H_3 \rangle}{\langle E_1, H_1 \rangle s_1; s_2 \langle E_3, H_3 \rangle}$
(IFTHEN)	$\frac{\langle E_1, H_1 \rangle e \rightsquigarrow n; H_2 \quad n \neq 0 \quad \langle E_1, H_2 \rangle s_1 \langle E_2, H_3 \rangle}{\langle E_1, H_1 \rangle \text{if } e \text{ then } s_1 \text{ else } s_2 \langle E_2, H_4 \rangle}$
(IFELSE)	$\frac{\langle E_1, H_1 \rangle e \rightsquigarrow n; H_2 \quad n \equiv 0 \quad \langle E_1, H_2 \rangle s_2 \langle E_2, H_3 \rangle}{\langle E_1, H_1 \rangle \text{if } e \text{ then } s_1 \text{ else } s_2 \langle E_2, H_4 \rangle}$
(WHILET)	$\frac{n \neq 0 \quad \langle E_1, H_2 \rangle s \langle E_2, H_3 \rangle \quad \langle E_2, H_3 \rangle \text{while } e \{ s \} \langle E_3, H_4 \rangle}{\langle E_1, H_1 \rangle \text{while } e \{ s \} \langle E_3, H_4 \rangle}$
(WHILEF)	$\frac{\langle E_1, H_1 \rangle e \rightsquigarrow n; H_2 \quad n \equiv 0}{\langle E_1, H_1 \rangle \text{while } e \{ s \} \langle E_1, H_2 \rangle}$
(SKIP)	$\langle E, H \rangle \text{skip} \langle E, H \rangle$

Fig. 3. Semantics of statements in IMPL

environment E_1 and heap H_1 . For the purpose of this paper, we assume that no derivations get stuck.

4 Generating test inputs for ImpL

In this section we formally develop our method for generating test inputs for a given ImpL program. As explained before, a first step consists of generating constraints on the program input, environment and heap by symbolic execution of the program.

4.1 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. We modify the semantics of ImpL such that 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 that 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 4.3.

The modified semantics of ImpL is depicted in Figures 4 and 5. 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 $\langle E_0, H_0, C_0 \rangle e \rightsquigarrow v; H_1; C_1$ 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 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 $\{E_0, H_0, C_0\} e \rightsquigarrow v; H_1; C_1$ and $\{E_0, H_0, C_0\} s \langle E_1, H_1, C_1 \rangle$ denote, respectively, $\langle E_0, H_0, C_0 \rangle e \rightsquigarrow v; H_1; C_1$ and $\langle E_0, H_0, C_0 \rangle s \langle E_1, H_1, C_1 \rangle$ under the condition that C is *consistent* (represented by $T \models C$). Formally:

$$\begin{array}{c}
(\text{COND-E}) \frac{\mathcal{T} \models C_0 \quad \langle E_0, H_0, C_0 \rangle e \rightsquigarrow v; H_1; C_1}{\{E_0, H_0, C_0\} e \rightsquigarrow v; H_1; C_1} \\
\\
(\text{COND-S}) \frac{\mathcal{T} \models C_0 \quad \langle E_0, H_0, C_0 \rangle s \langle E_1, H_1, C_1 \rangle}{\{E_0, H_0, C_0\} s \{E_1, H_1, C_1\}}
\end{array}$$

The use of conditional evaluation avoids adding further constraints to an already inconsistent set. This implies that search strategies (see Section 4.4) will only explore execution paths that can model a real execution.

$ (\text{VAR}) \frac{V \text{ fresh}}{\langle E, H, C \rangle x \rightsquigarrow V; H; C \wedge \{x \mapsto V\} \in E} $	$ (\text{INT}) \frac{n \in \mathbb{Z}}{\langle E, H, C \rangle n \rightsquigarrow n; H; C} $
$ (\text{NIL}) \langle E, H, C \rangle \text{ nil} \rightsquigarrow \text{nil}; H; C $	
$ (\text{CONS}) \frac{\langle E, H_0, C_0 \rangle e_1 \rightsquigarrow v_1; H_1; C_1 \quad \{E, H_1, C_1\} e_2 \rightsquigarrow v_2; H_2; C_2 \quad H_3, r \text{ fresh}}{\langle E, H_0, C_0 \rangle \text{ new cons}(e_1, e_2) \rightsquigarrow \text{ptr}(r); H_3; C_2 \wedge H_3 = H_2 \uplus \{r \mapsto \text{cons}(v_1, v_2)\}} $	
$ (\text{HEAD}) \frac{\langle E, H_0, C_0 \rangle e \rightsquigarrow v; H_1; C_1 \quad R, V_h, V_t \text{ fresh}}{\langle E, H_0, C_0 \rangle e.\text{head} \rightsquigarrow V_h; H_1; C_1 \wedge v = \text{ptr}(R) \wedge (R \mapsto \text{cons}(V_h, V_t)) \in H_1} $	
$ (\text{TAIL}) \frac{\langle E, H_0, C_0 \rangle e \rightsquigarrow v; H_1; C_1 \quad R, V_h, V_t \text{ fresh}}{\langle E, H_0, C_0 \rangle e.\text{tail} \rightsquigarrow V_t; H_1; C_1 \wedge v = \text{ptr}(R) \wedge (R \mapsto \text{cons}(V_h, V_t)) \in H_1} $	
$ (\text{EQUALT}) \frac{\langle E, H_0, C_0 \rangle e_1 \rightsquigarrow v_1; H_1; C_1 \quad \{E, H_1, C_1\} e_2 \rightsquigarrow v_2; H_2; C_2}{\langle E, H_0, C_0 \rangle e_1 == e_2 \rightsquigarrow 1; H_2; C_2 \wedge v_1 = v_2} $	
$ (\text{EQUALF}) \frac{\langle E, H_0, C_0 \rangle e_1 \rightsquigarrow v_1; H_1; C_1 \quad \{E, H_1, C_1\} e_2 \rightsquigarrow v_2; H_2; C_2}{\langle E, H_0, C_0 \rangle e_1 == e_2 \rightsquigarrow 0; H_2; C_2 \wedge v_1 \neq v_2} $	
$ (\text{NEQUALT}) \frac{\langle E, H_0, C_0 \rangle e_1 \rightsquigarrow v_1; H_1; C_1 \quad \{E, H_1, C_1\} e_2 \rightsquigarrow v_2; H_2; C_2}{\langle E, H_0, C_0 \rangle e_1 == e_2 \rightsquigarrow 1; H_2; C_2 \wedge v_1 \neq v_2} $	
$ (\text{NEQUALF}) \frac{\langle E, H_0, C_0 \rangle e_1 \rightsquigarrow v_1; H_1; C_1 \quad \{E, H_1, C_1\} e_2 \rightsquigarrow v_2; H_2; C_2}{\langle E, H_0, C_0 \rangle e_1 == e_2 \rightsquigarrow 0; H_2; C_2 \wedge v_1 = v_2} $	

Fig. 4. Semantics of expressions in IMPL extended with logical variables

Example 1. Consider again the example program of Section 2:

```

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

```

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

$$\begin{aligned}
C_1 &\equiv (x \rightsquigarrow V) \in E_0 \wedge \\
&V = \text{ptr}(R) \quad \wedge \quad (R \rightsquigarrow \text{cons}(V_h, V_t)) \in H_0 \quad \wedge \\
&V_t = \text{ptr}(R_2) \quad \wedge \quad (R_2 \rightsquigarrow \text{cons}(V_{h2}, V_{t2})) \in H_0
\end{aligned}$$

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

(VARASS)	$\frac{\langle E_0, H_0, C_0 \rangle e \rightsquigarrow v; H_1; C_1 \quad E_1 \text{ fresh}}{\langle E_0, H_0, C_0 \rangle x := e \langle E_1, H_1, C_1 \wedge E_1 = E_0 \uplus \{x \mapsto v\} \rangle}$
(HEADASS)	$\frac{R, V_h, V_t, H_2 \text{ fresh} \quad C_3 \equiv C_2 \wedge v_r = \text{ptr}(R) \wedge (R \mapsto \text{cons}(V_h, V_t)) \in H_1}{\langle E, H_0, C_0 \rangle e \rightsquigarrow v; H_1; C_1 \quad \{E, H_1, C_1\} l \rightsquigarrow v_r; H_1; C_2}$
(TAILASS)	$\frac{R, V_h, V_t, H_2 \text{ fresh} \quad C_3 \equiv C_2 \wedge v_r = \text{ptr}(R) \wedge (R \mapsto \text{cons}(V_h, V_t)) \in H_1}{\langle E, H_0, C_0 \rangle l.\text{head} := e \langle E, H_2, C_3 \wedge H_2 = H_1 \uplus \{R \mapsto \text{cons}(v, V_t)\} \rangle}$
(SEQ)	$\frac{\langle E_0, H_0, C_0 \rangle s_1 \langle E_1, H_1, C_1 \rangle \quad \{E_1, H_1, C_1\} s_2 \{E_2, H_2, C_2\}}{\langle E_0, H_0, C_0 \rangle s_1; s_2 \langle E_2, H_2, C_2 \rangle}$
(IFTHEN)	$\frac{\langle E_0, H_0, C_0 \rangle e \rightsquigarrow v; H_1; C_1 \quad \{E_0, H_1, C_1 \wedge v \neq 0\} s_1 \{E_1, H_2, C_2\}}{\langle E_0, H_0, C_0 \rangle \text{if } e \text{ then } s_1 \text{ else } s_2 \langle E_1, H_2, C_2 \rangle}$
(IFELSE)	$\frac{\langle E_0, H_0, C_0 \rangle e \rightsquigarrow v; H_1; C_1 \quad \{E_0, H_1, C_1 \wedge v = 0\} s_2 \{E_1, H_2, C_2\}}{\langle E_0, H_0, C_0 \rangle \text{if } e \text{ then } s_1 \text{ else } s_2 \langle E_1, H_2, C_2 \rangle}$
(WHILET)	$\frac{\langle E_0, H_0, C_0 \rangle e \rightsquigarrow v; H_1; C_1 \quad \{E_0, H_1, C_1 \wedge v \neq 0\} s; \text{while } e \{ s \} \{E_1, H_2, C_2\}}{\langle E_0, H_0, C_0 \rangle \text{while } e \{ s \} \langle E_1, H_2, C_2 \rangle}$
(WHILEF)	$\frac{\langle E_0, H_0, C_0 \rangle e \rightsquigarrow v; H_1; C_1}{\langle E_0, H_0, C_0 \rangle \text{while } e \{ s \} \langle E_0, H_1, C_1 \wedge v = 0 \rangle}$
	$\text{(SKIP)} \langle E, H, C \rangle \text{skip } \langle E, H, C \rangle$

Fig. 5. Semantics of statements in IMPL extended with logical variables

$$C_2 \equiv C_1 \wedge (x \rightsquigarrow V_2) \in E_0 \quad \wedge V_2 = \text{ptr}(R_3) \\ \wedge (R_3 \rightsquigarrow \text{cons}(V_{h3}, V_{t3})) \wedge V_{h2} \neq V_{h3}$$

The constraints for the loop body $x := x.\text{tail}$:

$$C_3 \equiv (x \rightsquigarrow V_3) \in E_0 \quad \wedge V_3 = \text{ptr}(R_4) \quad \wedge \\ (R_4 \rightsquigarrow \text{cons}(V_{h4}, V_{t4})) \in H_0 \wedge E_1 = E_0 \uplus \{x \rightsquigarrow V_{t4}\}$$

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

$$C \equiv C_2 \wedge C_3$$

4.2 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 of) those used in H' such that $\forall x \in \text{dom}(E) = \text{dom}(E') : E(x) =_\sigma E'(x)$ where $=_\sigma$ is defined as follows: $\text{nil} =_\sigma \text{nil}$, $n =_\sigma n$ for all integer constants n and $\text{ptr}(r) =_\sigma \text{ptr}(r')$ if $r' = \sigma(r)$, $H(r) = \text{cons}(v_1, v_2)$ and $H'(r') = \text{cons}(v'_1, v'_2)$ and $v_1 =_\sigma v'_1$ and $v_2 =_\sigma v'_2$.

Theorem 1 (Completeness).

Let E and H be an environment and a heap, and s an instruction 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

$$\begin{aligned} \langle E, H \rangle &\cong \langle \rho(E_v), \rho(H_v) \rangle \\ \langle E', H' \rangle &\cong \langle \rho(E'_v), \rho(H'_v) \rangle \end{aligned}$$

The completeness property states that any concrete execution of a program s with respect to an initial environment E and heap H is modeled 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 2 (Soundness).

Let s be an instruction. 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

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

4.3 Constraint Propagation

Among the four types of primitive constraints (Section 4.1), 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:

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

The above rules are easily implemented in the Constraint Handling Rules (CHR) language (6).

4.4 Search

In order to obtain concrete test cases, our constraint solver have 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 as 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 extended operational 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.

Example 2. Recall the example in Section 2 where the while-loop may iterate an arbitrary number of times. A depth-bounded search only considers test cases that involve iterations upto a given bound.

Non-Deterministic Values As the following example shows, even a single execution path can introduce non-determinism in the solving process.

Example 3. Consider the program $y := x.\text{tail}$, which has only one execution path. This execution path merely restricts the initial environment and heap to $E_0 = \{x \rightsquigarrow \text{ptr}(A), y \rightsquigarrow V_y\}$ and $(A \rightsquigarrow \text{cons}(V_h, V_i)) \in H_0$. There are an infinite number of concrete test cases that satisfy these restrictions. Here are just a few:

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

There are two kinds of unknown values: unknown integer 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 previous references R , or 3) a new reference r' . In the last case, the

heap must contain an additional cell with fresh unknown components.

$$V_r = \text{nil} \vee \left(\bigvee_{r \in R} V_r = \text{ptr}(r) \right) \vee (V_r = \text{ptr}(r') \wedge (r' \mapsto \text{cons}(V'_i, V'_r)) \in H_0)$$

In practice, we must again restrict ourselves to a finite number of alternatives. We may be interested in only a single solution: 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. For instance, the minimality criterion is captured by a branch-and-bound optimization strategy.

4.5 Generalized Data Structures

So far we have only considered data structures composed of simple `cons` cells. 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:

$$V_t = \text{nil} \vee \left(\bigvee_{r \in R_t} V_t = \text{ptr}(r) \right) \vee (V_r = r' \wedge (r' \mapsto \text{tree}(V'_i, V'_l, V'_r)) \in H_0)$$

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. *[[WIM: Is this similar to some other work on black-box testing?]]*

5 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 depicts an input/output pair for the example program of Section 2.

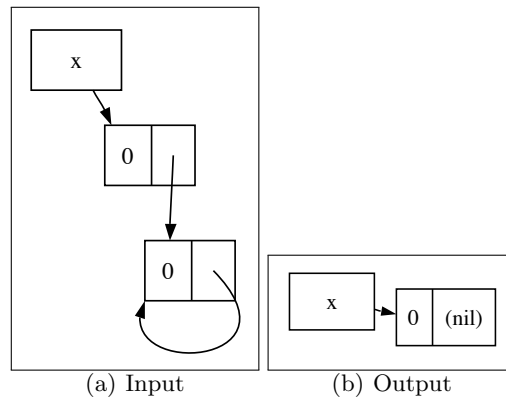


Fig. 6. Visualization of an input/output pair for the example program

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 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 4. Consider the simple program $x := \text{nil}$. One test configuration consists of an initial environment $E_0 = \{x \rightsquigarrow \text{ptr}(r1)\}$ and an initial heap $H_0 = \{r1 \rightsquigarrow \text{cons}(7, \text{ptr}(r1))\}$. The final environment is $E_1 = \{x \rightsquigarrow \text{nil}\}$ and the

⁵ <http://www.graphviz.org/>

final heap $H_1 = H_0$. The concrete test case for this test configuration looks like the code represented on the left of Figure 7. 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
if (x == nil) then {
  foundnil := 1;
  x := nil
} else {
  foundnil := 0;
  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
}

```

Fig. 7. Testcase for the original (left) and refactored (right) code of Example 4.

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 4 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 7. 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 8 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.

Check Phase Likewise, the algorithm given by the inference rules in Figure 9 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 setup code for environment E and heap H .

(S-DONE)	$\frac{}{H, A \vdash_s \emptyset : \mathbf{skip}}$
(S-INT)	$\frac{H, A \vdash_s E : s}{H, A \vdash_s \{l \mapsto n\} \cup E : l := n; s}$
(S-NIL)	$\frac{H, A \vdash_s E : s}{H, A \vdash_s \{l \mapsto \mathbf{nil}\} \cup E : l := \mathbf{nil}; s}$
(S-NREF)	$\frac{a \notin \mathit{domain}(A) \quad (a \mapsto \mathbf{cons}(v_h, v_t)) \in H}{H, A \cup \{a \mapsto l\} \vdash_s \{l.\mathbf{tail} : v_t\} \cup E : s}$
(S-OREF)	$\frac{H, A \vdash_s \{l \mapsto \mathbf{ptr}(a)\} \cup E : l := \mathbf{new\ cons}(v_h, \mathbf{nil}); s \quad (a \mapsto l') \in A \quad H, A \vdash_s E : s}{H, A \vdash_s \{l \mapsto \mathbf{ptr}(a)\} \cup E : l := l'; s}$

Fig. 8. Setup Phase Algorithm

(C-DONE)	$\frac{}{H, A \vdash_c \emptyset : \mathbf{accept} := 1}$
(C-INT)	$\frac{H, A \vdash_c E : s}{H, A \vdash_c \{l \mapsto n\} \cup E : \mathbf{if } l == n \mathbf{ then } s \mathbf{ else } \mathbf{accept} := 0}$
(C-NIL)	$\frac{H, A \vdash_c E : s}{H, A \vdash_c \{l \mapsto \mathbf{nil}\} \cup E : \mathbf{if } l == \mathbf{nil} \mathbf{ then } s \mathbf{ else } \mathbf{accept} := 0}$
(C-NREF)	$\frac{a \notin \mathit{domain}(A) \quad (a \mapsto \mathbf{cons}(v_h, v_t)) \in H \quad H, A \cup \{a \mapsto l\} \vdash_c \{l.\mathbf{head} : v_h, l.\mathbf{tail} : v_t\} \cup E : s_1 \quad l, s_1 \vdash_n \mathit{range}(A) : s_2}{H, A \vdash_c \{l \mapsto \mathbf{ptr}(a)\} \cup E : \mathbf{if } l \neq \mathbf{nil} \mathbf{ then } s_2 \mathbf{ else } \mathbf{accept} := 0}$
(C-OREF)	$\frac{(a \mapsto l') \in A \quad H, A \vdash_c E : s}{H, A \vdash_c \{l \mapsto \mathbf{ptr}(a)\} \cup E : \mathbf{if } l == l' \mathbf{ then } s \mathbf{ else } \mathbf{accept} := 0}$
(N-BASE)	$l, s \vdash_n \emptyset : s$
(N-REC)	$\frac{l, s \vdash_n R : s'}{l, s \vdash_n \{l'\} \cup R : \mathbf{if } l \neq l' \mathbf{ then } s' \mathbf{ else } \mathbf{accept} := 0}$

Fig. 9. Check Phase Algorithm

6 Related Work and Conclusion

A large amount of work exists in the field of automatic test case generation for imperative programs. The arguably simplest method is *random generation* of test data (7; 8). However, the approach turns out to be quite inefficient when trying to generate test suites that satisfy a given coverage criterion, due to its blackbox nature. The related *antirandom* testing (9) improves on the previous technique by creating a dependence between each pair of successively generated test values, but it still fails to ensure the satisfaction of a given coverage criterion.

Most related to our work are the approaches based *symbolic evaluation* (10; 11; 5; 12; 13; 14) that use symbolic values instead of actual data as input values, in order to derive a symbolic expression representing the values of a program's variables. Symbolic evaluation is particularly well-suited to generate test suites satisfying given coverage criterion since they typically allow to select a desired subset of execution paths that need to be followed. In so-called *dynamic* approaches, the program is actually executed on input data that is arbitrarily chosen from a given domain. The input data is then iteratively refined to obtain a final test input such that the execution follows a chosen path, or reaches a chosen statement (15; 16). These approaches are only used to generate numerical values, similarly to those using *optimization* techniques (17; 18; 19; 20; 21). These techniques reformulate the generation of test cases as an optimization problem of a function representing a chosen coverage criterion. Some enable monitoring of the execution flow when the program is executed; if an undesired branch is followed, the execution is suspended and function minimization search algorithms are used to automatically alter the execution path at the choicepoint under concern (17). Still other approaches make use of *genetic algorithms* (22) (limited to programs without procedure calls), the *narrowing* feature provided by the functional/logic language Curry (23) (for test data generation for the Curry and Haskell languages), or *interval arithmetics* (24), the latter limited to numerical values.

In this paper, we have presented a constraint-based approach for generating white-box test cases for a small but representative imperative programming language. Our technique is able to generate complex heap-allocated and pointer-based data structures without any user intervention. 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. Our prototype and example programs are available at <http://www.cs.kuleuven.be/~toms/Testing/>. In future work, we will investigate the exact relation between the use of particular search strategies for constraint solving and the generation of *interesting* sets of test cases according to different adequacy criteria. Other topics of further work include extending the IMPL language towards more involved imperative and object-oriented languages.

Bibliography

- [1] Glass, R.: Software runaways : lessons learned from massive software project failures. Prentice Hall (1997)
- [2] Kaner, C., Falk, J., Nguyen, H.Q.: Testing computer software. John Wiley and Sons (1993)
- [3] Zhu, H., Hall, P., May, J.: Software unit test coverage and adequacy. *ACM Computing Surveys* **29**(4) (1997)
- [4] Degrave, F., Schrijvers, T., Vanhoof, W.: Automatic generation of test inputs for Mercury. In: *Logic-Based Program Synthesis and Transformation*, Springer (2008)
- [5] Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java pathfinder. *SIGSOFT Softw. Eng. Notes* **29**(4) (2004) 97–107
- [6] Frühwirth, T.: Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming* **37**(1–3) (1998) 95–138
- [7] Bird, D.L., Munoz, C.U.: Automatic generation of random self-checking test cases. *IBM Syst. J.* **22**(3) (1983) 229–245
- [8] Duran, J.W., Ntafos, S.C.: An evaluation of random testing. *IEEE Trans. Software Eng.* **10**(4) (1984) 438–444
- [9] Yin, H., Yin, H., Lebne-dengely, Z., Lebne-dengely, Z., Malaiya, Y.K., Malaiya, Y.K.: Automatic test generation using checkpoint encoding and antirandom testing. In: *Proc. Int. Symp. Software Reliability Engineering*. (1997) 84–95
- [10] Clarke, L.A.: A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.* **2**(3) (1976) 215–222
- [11] King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**(7) (1976) 385–394
- [12] Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: *In Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer (2003) 553–568
- [13] Sy, N.T., Deville, Y.: Automatic test data generation for programs with integer and float variables. In: *Proceedings of ASE 01*. (2001)
- [14] Müller, R.A., Lembeck, C., Kuchen, H.: A symbolic java virtual machine for test case generation. In: *IASTED Conf. on Software Engineering*. (2004) 365–371
- [15] Gupta, N., Mathur, A.P., Soffa, M.L.: Automated test data generation using an iterative relaxation method. *SIGSOFT Softw. Eng. Notes* **23**(6) (1998) 231–244
- [16] Ferguson, R., Korel, B.: The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.* **5**(1) (1996) 63–86
- [17] Korel, B.: Automated software test data generation. *IEEE Trans. Softw. Eng.* **16**(8) (1990) 870–879

- [18] Tracey, N., Clark, J., Mander, K.: The way forward for unifying dynamic test-case generation: The optimisation-based approach. In: International Workshop on Dependable Computing and Its Applications (DCIA), IFIP (January 1998) 169–180
- [19] Ferguson, R., Korel, B.: The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.* **5**(1) (1996) 63–86
- [20] Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. *Software Engineering, IEEE Transactions on* **SE-2**(3) (1976) 223–226
- [21] et al., C.A.: Combining test case generation and runtime verification. *Theoretical Computer Science* **336**(2-3) (2005)
- [22] Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability* **9** (1999) 263–282
- [23] Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: *PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, New York, NY, USA, ACM (2007) 63–74
- [24] Sy, N.T., Deville, Y.: Automatic test data generation for programs with integer and float variables. In: *In 16th IEEE International Conference on Automated Software Engineering(ASE01)*. (2001) 3–21

A (For referees) Proof of Completeness Theorem

The proof of the completeness theorem of Section 4.3 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 $\langle E, H \rangle e \rightsquigarrow v; H'$, 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 $\langle E_v, H_v, C_1 \rangle e \rightsquigarrow v_v; H'_v; C_2$ 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 $\langle E, H \rangle e \rightsquigarrow ptr(r); H_2$ and $r \mapsto \mathbf{cons}(v_h, v_t) \in H_2$. By induction hypothesis, we have $\langle E_v, H_{v0}, C_0 \rangle e \rightsquigarrow v_v; H_{v1}; C_1$ with a solution ρ_1 such that $\langle E, H_2 \rangle \cong \langle \rho_1(E_v), \rho_1(H_{v1}) \rangle$ and $\rho_1(v_v) = ptr(r)$. We can construct a solution ρ for C_2 where

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

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 an instruction 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 $\langle E, H_1 \rangle e \rightsquigarrow v; H_2$. From Lemma 1 we have $\langle E_{v0}, H_{v0}, C_0 \rangle e \rightsquigarrow v_v; H_{v1}; C_1$ 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 $\langle E_{v0}, H_{v0}, C_0 \rangle s_1 \langle E_{v1}, H_{v1}, C_1 \rangle$ 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 $\langle E_3, H_3 \rangle \cong \langle \rho_2(E_{v2}), \rho_2(H_{v2}) \rangle$.

The soundness property can be proved in an analogous way.