Towards a framework for constraint-based test case generation

François Degraeve^{*2}, Tom Schrijvers^{**1}, 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 (CP). We advocate the use of standard CP search strategies in order to express preferences on the generated test cases and to obtain the desired degree of coverage. We develop our framework in the concrete context of an imperative language and show that the technique is sufficiently powerful to deal with arbitrary pointer-based data-structures allocated on the heap.

1 Introduction

It is a well-known fact that a substantial part of a software development budget is spent on 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: 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. In previous work [4], we have developed a technique that allows to automatically generate a set of test inputs for a given program written in the logic programming language Mercury. The basic idea of that technique, itself inspired by [20], is as follows: first, a path through the control-flow graph of a predicate is transformed into a set of constraints on the (input) variables of the predicate such that when the predicate is called with input values satisfying these constraints, its execution is guaranteed to follow the given path. Next, a dedicated constraint solver is used to compute such concrete input values. By repeating the process for a carefully constructed finite subset of all execution paths one can guarantee that the set of generated input values *cover* a substantial part of the source code and can as such be used to perform so-called *structural* or *whitebox* testing.

In the current paper, we generalize and extend the approach of [4], reformulating it completely within the framework of constraint programming (CP). This presents several advantages over [4] and similar proposals:

* Supported by a grant FRIA - Belgium.

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

- The whole process – from the construction of an execution path to the generation of concrete test inputs – is now modeled as a single constraint problem, rather than as a pair of complementary processes. This presents a cleaner and more uniform formalisation of the technique.
- More importantly, since the selection of an execution path is modeled within the constraint problem, one can obtain different degrees of coverage, or even coverage with respect to different criteria, by using different search strategies within the solver. In other words, the technique can – at least to a certain extent – be seen as parametrised with respect to a coverage criterion or a desired degree of coverage.
- The resulting technique is also more efficient, since it avoids the construction of constraint sets representing paths that do not correspond to a real execution.

In order to show the power of our generalized approach, we cast it in the setting of a (small) deterministic imperative language with 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. 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 2.6 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	$e ::= x \mid n \mid \mathbf{nil} \mid \mathbf{new\ cons}(e_1, e_2) \mid e.\mathbf{head} \mid e.\mathbf{tail}$ $\mid e_1 == e_2 \mid e_1 \neq e_2 \mid e_1 + e_2$
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 ($e.\mathbf{head}$), respectively tail ($e.\mathbf{tail}$) field of the cons cell referenced by e , equality and inequality tests ($==$ and $\neq$), and the arithmetic operator for addition $+$.³ We will assume that **Impl** is simply typed and only allows comparison of two values belonging to the same type (either integers or references).⁴ Moreover, arithmetic is only allowed on integer values; the language does not support pointer arithmetics.

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

³ Other arithmetic operators are omitted in order to keep the formal definition of the semantics small, but they can be added at will.

⁴ Integers are also used as booleans: 0 denotes false and all other integers denote true.

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, after the while loop, the list will have the value $[1,2,3]$.

2 Generating test inputs

2.1 Overview

As usual, 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 $\text{ptr}(r)$ 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 $\text{cons}(v_h, v_e)$ with v_h and v_e values (possibly including references to other cons cells). 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:

$$\begin{array}{l|l} E : x \mapsto \text{ptr}(r_1) & E : x \mapsto \text{ptr}(r_1) \\ H : r_1 \mapsto \text{cons}(1, \text{ptr}(r_2)) \quad r_4 \mapsto \text{cons}(3, \text{ptr}(r_5)) & H : r_1 \mapsto \text{cons}(1, \text{ptr}(r_2)) \\ \quad r_2 \mapsto \text{cons}(2, \text{ptr}(r_3)) \quad r_5 \mapsto \text{cons}(4, \text{nil}) & \quad r_2 \mapsto \text{cons}(2, \text{ptr}(r_3)) \\ \quad r_3 \mapsto \text{cons}(3, \text{ptr}(r_4)) & \quad r_3 \mapsto \text{cons}(3, \text{nil}) \end{array}$$

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*

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. 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).

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 that take the following form:

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

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

Fig. 2. Semantics of expressions in IMPL

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

Fig. 3. Semantics of statements in IMPL

- $(o_1 \mapsto o_2) \in M$, 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 2.4.

The symbolic semantics 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 $\langle E_0, H_0, C_0 \rangle e \rightsquigarrow v \langle E_0, H_1 \rangle C_1$ 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 $\{E, H_0, C_0\} e \rightsquigarrow v \langle E, H_1, C_1 \rangle$ 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).⁵ Formally:

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

The use of conditional evaluation avoids adding further constraints to an already inconsistent set. This implies that search strategies (see Section 2.5) will only explore execution paths that can model a real execution.

2.3 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 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 $v_2 =_\sigma v'_2$.

⁵ In practice, the consistency check may be incomplete. Then unreachable execution paths may be explored.

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

Fig. 4. Symbolic evaluation of expressions.

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 $\langle E_v, H_v, \text{true} \rangle s \langle E'_v, H'_v, C \rangle$ 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 $\langle E_v, H_v, \text{true} \rangle s \langle E'_v, H'_v, C \rangle$ 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

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

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

Fig. 5. Symbolic execution of statements.

2.4 Constraint Propagation

Among the four types of primitive constraints (Section 2.2), 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 (CHR) [9].

2.5 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 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 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 2.1 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 $y := x.\text{tail}$, 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 $(A \rightsquigarrow \text{cons}(V_h, V_t)) \in H_0$. There are an infinite number of concrete test cases that satisfy these restrictions. Here are just a few:

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

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.

2.6 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 acyclicity, *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 [21, 18].

3 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.⁶ 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 2.1.

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

$$\text{Setup}; P; \text{Check}$$

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

Example 1. Consider the simple program $\mathbf{x} := \text{nil}$. One test configuration consists of an initial environment $E_0 = \{\mathbf{x} \rightsquigarrow \text{ptr}(\mathbf{r1})\}$ and an initial heap $H_0 = \{\mathbf{r1} \rightsquigarrow \text{cons}(7, \text{ptr}(\mathbf{r1}))\}$. The final environment is $E_1 = \{\mathbf{x} \rightsquigarrow \text{nil}\}$ and the

⁶ <http://www.graphviz.org/>

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

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 1.

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 1 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.

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

Fig. 8. Setup Phase Algorithm

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 .

4 Ongoing Work

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 [1, 7]. In *symbolic evaluation* techniques (e.g. [3, 14, 16]), the input parameters are replaced by symbolic values, in order to derive a symbolic expression representing the values of a program's variables. This approach is notably used in the ATGen tool for structural coverage of Spark ADA programs [15]. 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

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

Fig. 9. Check Phase Algorithm

refined to obtain a final test input such that the execution follows a chosen path, or reaches a chosen statement [13, 8].

Constraint-based test data generation was originally introduced in [6] in the context of mutation testing [5] and aims at transforming the automatic test data generation problem into a CLP problem over finite domains. This approach has been used in many works, including [11, 12]. It is also used in two different testing tools, Godzilla [17] and InKA [10]. The latter notably generates test data satisfying different criteria such as statement coverage, branches coverage and MC/DC⁷.

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. The selection of both execution paths and concrete test inputs are modelled uniformly as a single constraint problem. An interesting advantage of our technique over most existing work is that we use parametrizable CP search strategies within the solver in order to express (coverage) criteria the generated test suites must satisfy. Our framework would therefore be able to generate test cases accordingly to any (coverage) criterion, instead of proposing a predefined set of built-in criteria.

We have developed an initial prototype in SWI-Prolog, which is available at <http://www.cs.kuleuven.be/~toms/Testing/> together with example programs. As we've already said, Prolog's advantage is that it provides most of the required constraints for free, and the missing ones are easily implemented in

⁷ Modified condition/decision coverage [2]

CHR. However, Prolog’s disadvantage is that depth-first search is built in, which makes experimenting with alternative search strategies hard – if not impossible.

We are currently working on a new implementation of our approach within the Monadic Constraint Programming framework [19], which is an open framework for building complex search strategies as a composition of simpler ones. This system will enable us to 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.

References

1. D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22(3):229–245, 1983.
2. J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, Sep 1994.
3. L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.
4. F. Degraeve, T. Schrijvers, and W. Vanhoof. Automatic generation of test inputs for Mercury. In *Logic-Based Program Synthesis and Transformation*. Springer, 2008.
5. R.A. DeMillo. Test adequacy and program mutation. In *Software Engineering, 1989. 11th International Conference on*, pages 355–356, May 1989.
6. R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on*, 17(9):900–910, Sep 1991.
7. J. W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, 10(4):438–444, 1984.
8. Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.
9. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
10. Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):53–62, 1998.
11. Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A clp framework for computing structural test data. In *Proceedings of the First International Conference on Computational Logic*, pages 399–413, London, UK, 2000. Springer-Verlag.
12. Arnaud Gotlieb, Tristan Denmat, and Bernard Botella. Goal-oriented test data generation for programs with pointer variables. *Computer Software and Applications Conference, Annual International*, 1:449–454, 2005.
13. Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. *SIGSOFT Softw. Eng. Notes*, 23(6):231–244, 1998.
14. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
15. C. Meudec. Atgen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing Verification and Reliability*, 11(2):81–96, 2001.
16. Roger A. Müller, Christoph Lembeck, and Herbert Kuchen. A symbolic java virtual machine for test case generation. In *IASTED Conf. on Software Engineering*, pages 365–371, 2004.

17. A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation: Design and algorithms. *Software - Practice and Experience*, 29:167–193, 1994.
18. A. Jefferson Offutt and Shaoying Liu. Generating test data from soft specifications. *The Journal of Systems and Software*, 49:49–62, 1999.
19. Tom Schrijvers, Peter Stuckey, and Phil Wadler. Monadic Constraint Programming, 2009. <http://www.cs.kuleuven.be/~toms/Haskell/>.
20. Nguyen Tran Sy and Y. Deville. Automatic test data generation for programs with integer and float variables. In *Proceedings of ASE 01*, 2001.
21. W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.

A (For referees) Proof of Completeness Theorem

The proof of the completeness theorem of Section 2.4 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 $\langle E, H \rangle e \rightsquigarrow ptr(r) \langle E, H_2 \rangle$ and $r \mapsto \text{cons}(v_h, v_t) \in H_2$. 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 $\langle E, H_2 \rangle \cong \langle \rho_1(E_v), \rho_1(H_{v1}) \rangle$ and $\rho_1(v_v) = ptr(r)$. We can construct a solution ρ for C_2 where

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

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 deduced 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 deduced 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.