Data Structures in the Succinct Solver (V1.0)

Authors	: Hongyan Sun, Hanne Riis Nielson and Flemming Nielson
Date	: November 11, 2002
Number	: SECSAFE-IMM-005-1.0
Classification	: Public

Abstract.

This report documents our recent study and experiment with data structures used in the succinct solver. The succinct solver incorporates state-of-the-art approaches to constraint solving and solves static analysis problems specified in Alternation-free Least Fixed Point Logic (ALFP). In previous studies and experiments with the solver, we have observed that minor syntactical variations of formulas have a strong impact on the efficiency of computation, and tuning of the analysis can be achieved by tuning of formulas in many circumstances. We have also gained some insights into *which* formulations are better than the others in the sense that the solver can give a better performance. In this study, we aim on one hand at gaining insights into the internal behaviour of the solver to explain *why* some formulations are better than the others, and to enhance our expertise in tuning clauses to use the solver efficiently. On the other hand, we aim at being capable to enrich and improve the solver with desired features so that real applications such as the security analysis of Carmel can be solved efficiently considering both time and space efficiency. In this report, we use an example, reaching definitions analysis of a factorial program written in the WHILE language, to discuss and illustrate how the data structures are constructed and expanded with the evolution of the solving processes of the solver.

1 Introduction

Program analysis can often be carried out in a two-phase process [1]. In the first phase, the focus is on the *specification* of the analysis, and where the analyzed program is transformed into a suitable set of constraints. In the second phase, the main concern is the *computation* of the analysis, and where the constraints are solved by employing an appropriate constraint solver. Here, we consider the succinct solver developed by Nielson and Seidl [2] as such a constraint solver.

The succinct solver uses the Alternation-free Least Fixed Point Logic (ALFP) in clausal form as the constraint specification language. This specification logic is more expressive than that either in BANE or in Datalog as pointed in [2]. Formulas in ALFP naturally arise in the specification of static analyses of programs (c.f. [3] and [4]). On the other hand, the algorithm in the solver allows to be formulated in a succinct manner due to the use of continuation and memoisation. Thus the behaviour of the solver can be characterized precisely and the complexity analysis can be developed formally and automatically as shown in [3] and [5]. I addition, computing the solution in the solver boils down to computing the desired model of a formula. Previous experiences with the solver in [2, 3, 6] report that minor syntactical variations of formulas have a strong impact on the efficiency of computation, and tuning of the analysis can be achieved by tuning of formulas in many circumstances.

By the experiments with various transformations of formulas together with the associated time complexities as described in [2, 6], we have gained some insights into *which* formulations are better than the others in the sense that the solver can give a better performance. In this report, we document our recent study and experiment with data structures implemented in the succinct solver of version 1.0[1]. We aim on one hand at gaining insights into the internal behaviour of the solver to explain *why* some formulations are better than the others, and to enhance our expertise in tuning clauses to use the solver efficiently. On the other hand, we aim at being capable to enrich and improve the solver with our (or users) desired features so that real applications such as the security analysis of Carmel [7] can be solved efficiently considering both time and space efficiency.

We have focused so far on how the data structures are constructed and expanded with the evolution of the solving processes of the solver. The preliminary data obtained so far from the study bring us some interesting discussions on improving the solver, which will be reported in this document.

The remainder of the report is organized as follows: in section 2, we briefly give the syntax and the semantics of the ALFP logic that is used by the solver as the specification logic. In section 3, we give an overview of the solver by sketching the main program structure and data structures. We explain and illustrate, in section 4, how the solver processes clauses and manipulates the data structures by means of an example, reaching definitions analysis of a factorial program written in the WHILE language [8]. Finally, in section 5, we conclude the report with some discussions on the further improvement of the solver.

2 ALFP in brief

The specification logic used in the succinct solver is the alternation-free fragment of Least Fixed Point Logic (ALFP), which is an extension of Horn Clauses. In this section we give a brief introduction to ALFP in terms of the syntax and the semantics.

2.1 Syntax

Assume we are given a fixed countable set $\mathcal{X}$ of (auxiliary) variables and a finite ranked alphabet $\mathcal{R}$ of predicate symbols. Then the set of clauses, *cl*, is given by the following grammar

$$\begin{array}{lcl} pre & ::= & R(x_1, \cdots, x_k) \quad | \quad \neg R(x_1, \cdots, x_k) \quad | \quad pre_1 \wedge pre_2 \\ & | & pre_1 \vee pre_2 \quad | \quad \exists x : pre \quad | \quad \forall x : pre \\ cl & ::= & R(x_1, \cdots, x_k) \quad | \quad \mathbf{1} \quad | \quad cl_1 \wedge cl_2 \\ & | & pre \implies cl \quad | \quad \forall x : cl \end{array}$$

where $R \in \mathcal{R}$ is a k-ary predicate symbol for $k \geq 1$, $x, x_1, \cdots \in \mathcal{X}$ denote arbitrary variables, and $\mathbf{1}$ is the always true clause. Occurrences of $R(\cdots)$ and $\neg R(\cdots)$ in pre-conditions are also called *queries* and *negative queries*, respectively, whereas the other occurrences are called *assertions* of the predicate R.

[1]The version as of September 2002. We refer to it throughout this report.

$(\rho,\sigma) \models R(x_1,\cdots,x_k)$	iff	$(\sigma(x_1),\cdots,\sigma(x_k)) \in \rho(R)$
$(\rho,\sigma) \models \neg R(x_1,\cdots,x_k)$	iff	$(\sigma(x_1),\cdots,\sigma(x_k)) \notin \rho(R)$
$(\rho,\sigma) \models pre_1 \wedge pre_2$	iff	$(\rho,\sigma) \models pre_1$ and $(\rho,\sigma) \models pre_2$
$(\rho,\sigma) \models pre_1 \vee pre_2$	iff	$(\rho,\sigma) \models pre_1$ or $(\rho,\sigma) \models pre_2$
$(\rho,\sigma) \models \exists x : pre$	iff	$(\rho,\sigma[x \mapsto a]) \models pre$ for some $a \in \mathcal{U}$
$(\rho,\sigma) \models \forall x : pre$	iff	$(\rho,\sigma[x \mapsto a]) \models pre$ for all $a \in \mathcal{U}$
$(\rho,\sigma) \models R(x_1,\cdots,x_k)$	iff	$(\sigma(x_1),\cdots,\sigma(x_k)) \in \rho(R)$
$(\rho,\sigma) \models \mathbf{1}$		always
$(\rho,\sigma) \models cl_1 \wedge cl_2$	iff	$(\rho,\sigma) \models cl_1$ and $(\rho,\sigma) \models cl_2$
$(\rho,\sigma) \models pre \implies cl$	iff	$(\rho,\sigma) \models cl$ whenever $(\rho,\sigma) \models pre$
$(\rho,\sigma) \models \forall x : cl$	iff	$(\rho,\sigma[x \mapsto a]) \models cl$ for all $a \in \mathcal{U}$

Table 1: Semantics of pre-conditions and clauses

In order to deal with negations conveniently, we restrict ourselves to *alternation-free* formulas. We introduce a notion of stratification similar to the one which is known from *Datalog* [9, 10]. A clause cl is an *alternation-free Least Fixpoint formula* (ALFP formula for short) if it has the form $cl = cl_1 \wedge \cdots \wedge cl_k$, and there is a function $rank : \mathcal{R} \to \mathbb{N}$ such that for all $j = 1,\cdots,k$, the following properties hold:

- all predicates of assertions in cl_j have rank j;
- all predicates of queries in cl_j have ranks at most j; and
- all predicates of negated queries in cl_j have ranks strictly less than j.

2.2 Semantics

Given a non-empty and countable universe $\mathcal{U}$ of atomic values (or atoms) together with interpretations ρ and σ for predicate symbols and free variables, respectively, we define the satisfaction relations

$$(\rho,\sigma) \models pre \quad \text{and} \quad (\rho,\sigma) \models cl$$

for pre-conditions and clauses as in Table 1. Here we write $\rho(R)$ for the set of k-tuples $(a_1,\cdots,a_k)$ from $\mathcal{U}$ associated with the k-ary predicate R, we write $\sigma(x)$ for the atom of $\mathcal{U}$ bound to x and finally $\sigma[x \mapsto a]$ stands for the mapping that is as σ except that x is mapped to a.

In the sequel, we view the free variables occurring in a formula as constant symbols or atoms from the finite universe $\mathcal{U}$. Thus, given an interpretation σ_0 of the constant symbols, in the clause cl, we call an interpretation ρ of the predicate symbols $\mathcal{R}$ a *solution* to the clause provided $(\rho,\sigma_0) \models cl$.

3 Overview of the succinct solver

The succinct solver is implemented using NJ/SML featured with modular structures, continuations, and memoisations. In this section, we give an overview of

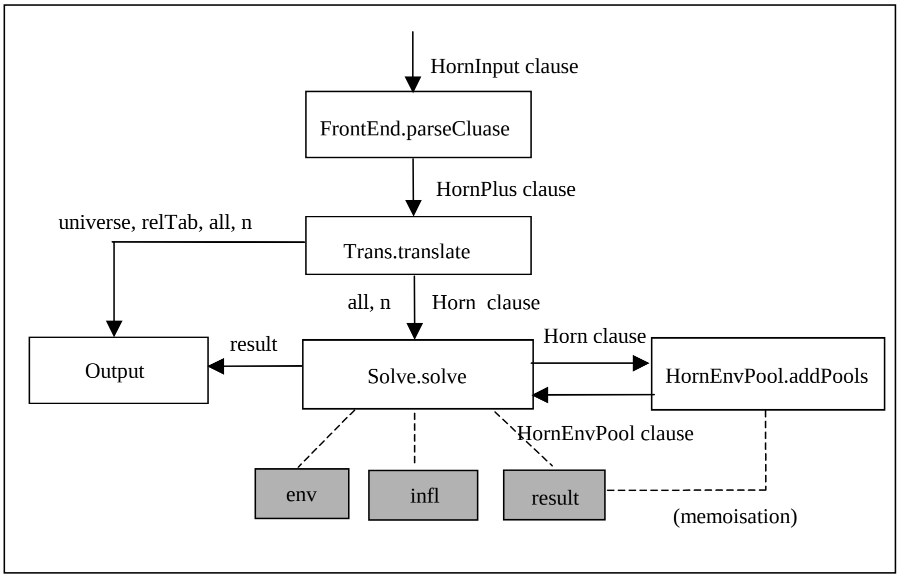

Fig. 1: The program structure

the solver by sketching the program structure and data structures, and explaining briefly functionalities of the main functions.

3.1 Program structure

The program structure of the solver is sketched in Fig. 1. The various stages will be exemplified in section 4.

In Fig. 1, the function *parseClause* in the module *FrontEnd* parses the ALFP clause from the text file (we shall call it as *HornInput clause* hereafter), and transforms it into an internal representation, *HornPlus clause*. The function *translate* in the *Trans* module translates the *HornPlus clause* into another internal representation, *Horn clause*. The main goal of this is to transform the atoms and predicate symbols into integers. At the same time, it extracts the useful static information into the internal data structures, i.e. *universe*, *relTab*, *all* and *n* respectively. Here, *universe* contains the information about the finite universe of atoms, and *relTab* holds the information about the predicates. Each integer in the integer list *all* represents an atom from the universe, whereas integer *n* represents the number of the predicates in *relTab*.

The information in *universe*, *relTab*, *all* and *n* are used by the *Output* module to produce the final output. The information in *all* and *n* together with *Horn clause* are used by the *solve* function in the *Solve* module to compute the solution to the clause.

The function *solve* first transforms *Horn clause* into the internal representation, *HornEnvPool clause*, for the purpose of the *memoisation* (c.f. [2]) in the case that disjunctions or existential quantifications are used in preconditions. It then processes the *HornEnvPool clause* and computes the solution by manipulating three main data structures, i.e. *env* for the partial environment, *result* for the solution, and *infl* for the *consumer* registration. We will discuss the *solve* function in more detail in section 4.3.

3.2 Data structures

The data structures *env*, *result*, and *infl* in the solver are abstracted as SML data types as follows:

```
type env = (var * (univ option)) list

type result = univ list option stack * table
type 'a stack = (int * 'a array) ref
type table = {buckets: ((loc, univ) * loc) list array ref,
                 hash: ((loc, univ) → idx) ref,
                count: int ref}

type infl = consumer list option stack * table
type consumer = univ list → unit
```

Where, *var* corresponds to variables, *univ* corresponds to the universe of atoms, *loc* the locations in the stack, and *idx* the locations in the *buckets* (i.e. the hash table). These are all of *int* type in the implementation (c.f. Appendix A). The *int* in *stack* corresponds to the size of the stack, while *int* in *count* the number of elements in the *buckets* of the table.

The type declarations for *env*, *result* and *infl* in the implementation of the solver are given in Appendix A.

Definition 1. A prefix tree is a rooted tree. It is used to represent an n-ary relation R on a given finite universe $\mathcal{U}$. Each path of the tree represents a tuple $(a_1, ..., a_n) \in R$. Along a path from the root node to the leaf node, each edge between any two nodes (i.e. a parent node and its child node) is respectively labeled with a_1,..., a_n for $(a_1, ..., a_n) \in R$. Given a node v_i and its child node v_j in the prefix tree, if the edge between them is labeled with a, then we say v_i prefixes the subtree rooted on v_j by a, and shortly, v_i prefixes v_j by a.

Example 1. A 2-*ary* relation $R = \{(a, a), (a, b), (b, c)\}$ is represented by a prefix tree shown in Fig. 2.

3.2.1 The *result* data structure.

The *result* data structure implements a set of prefix trees as illustrated in Fig. 3.

In Fig. 3, the stack associates with an attribute m, which denotes the size of the stack. The table associates with two attributes *hash* and *count*, which are respectively the hash function and the number of elements in the *buckets* of the table.

A slot in the stack corresponds to a node in a prefix tree. The first n slots in the stack corresponds to the root nodes of n prefix trees. The content of the slot can be NONE, SOME[] or SOME $[b_1, ..., b_i]$. Here, NONE denotes an uninitialized node, SOME [] denotes a leaf node, and SOME $[b_1, ..., b_i]$ denotes a node that prefixes its i $(i \geq 1)$ child nodes by b_1,...,and b_i respectively. In the

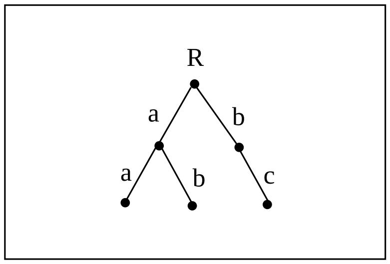

Fig. 2: A prefix tree representing a 2-*ary* relation R

case of Fig. 3, SOME $[a,\ b]$ in slot 0 denotes that the root node of the prefix tree for R prefixes its two children by a and b respectively.

An element $((v_1,\ a),\ v_2)$ in the buckets corresponds to an edge between two nodes v_1 and v_2 such that v_1 prefixes v_2 by a. Here, v_1 and v_2 are the slot locations in the stack.

The buckets constitute a hash table. The hash function takes the pair $(v_1,\ a)$ as the input and produces the hash value as the index of the buckets. Therefore, each slot in the buckets may be hashed into more than one elements. To resolve the collisions, we define that a slot of the buckets contains a list of elements.

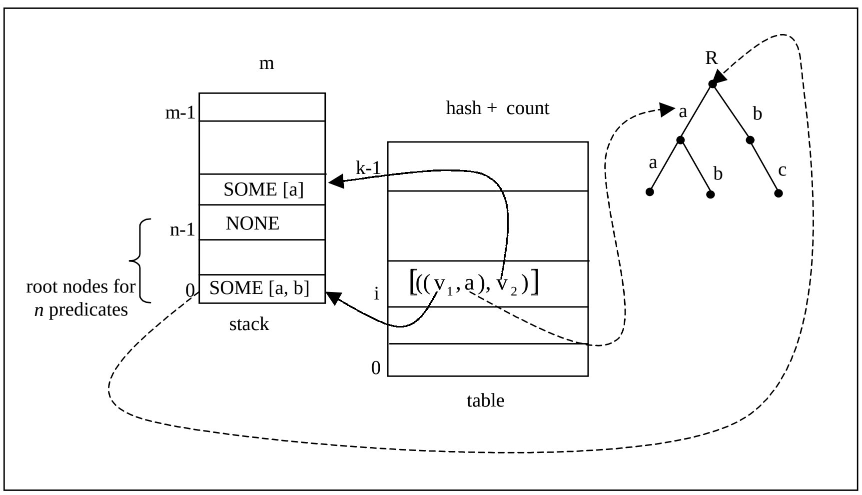

Fig. 3: The *result* data structure

Example 2. Three prefix trees representing three relations, i.e. 1-*ary* relation R, 2-*ary* relation P, and undefined relation Q, are implemented by *result* as shown in Fig. 4.

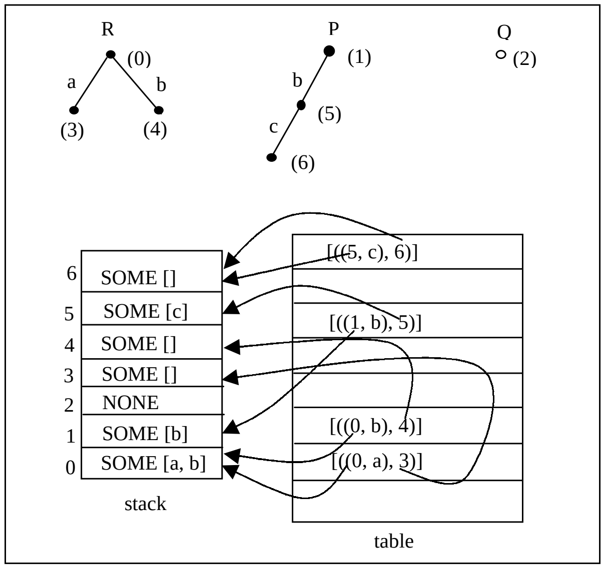

Fig. 4: The three prefix trees in *result*

3.2.2 The *infl* data structure.

Fig. 5 illustrates the data structure *infl*, which again implements a set of prefix trees as that in *result*. But it differs from *result* in that each slot of the stack contains information about *consumers* (c.f. [2]). A consumer is constructed when the current computation can not be completed for the lack of information. The solver suspends the computation by saving the necessary context as the consumer, and resumes the computation when the expected information is obtained. In Fig. 5, SOME [*csm*] in slot 0 in the stack means that one consumer (denoted by *csm*) is registered in the root node of the prefix tree for R, whereas NONE in a slot means that no consumer is registered in the corresponding node.

4 Reaching definitions analysis using the solver

In this section we use an example to explain and illustrate how the solver works, in particular in manipulating data structures, to derive the final solution. The example we are using is a reaching definitions (RD for short) analysis of a small program taken from the book [8] as:

$$[y := x]^1;\ [z := 1]^2;\ \text{while } [y > 1]^3 \text{ do } ([z := z * y]^4;\ [y := y - 1]^5);\ [y := 0]^6$$

This program is written in the WHILE language. It calculates the factorial of the number stored in x and saves the result in z. The number outside the brackets [] is the label of the elementary block embraced in the brackets.

In reaching definitions analysis, we are interested in which assignment may reach which program point (namely *entry* point and *exit* point of each elementary block). The analysis for the WHILE language is given in Table 2.2 in the book [8]. What we do here is to transform the analysis in terms of ALFP clausal form (i.e. *HornInput clause*) as shown in Fig. 6.

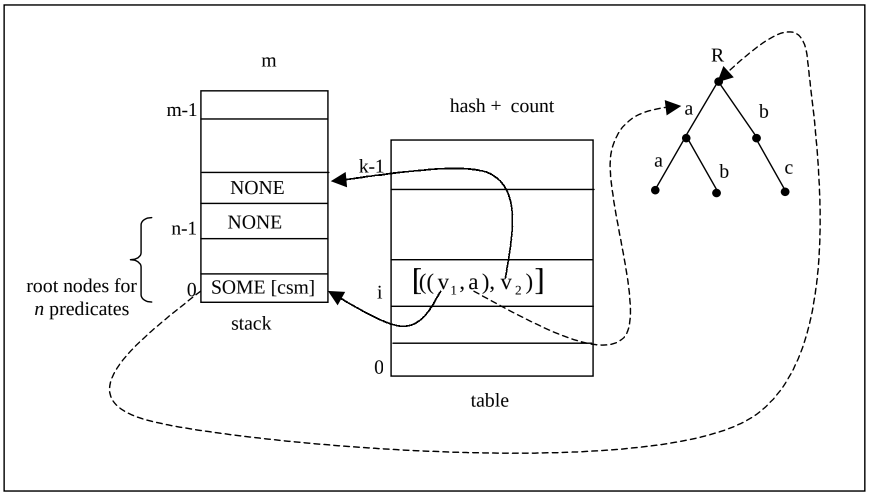

Fig. 5: The *infl* data structure

INIT(L1) &

FVAR(x1) & FVAR(z1) & FVAR(y1) &

FLOW(L4,L5) & FLOW(L3,L4) & FLOW(L5,L3) & FLOW(L3,L6) & FLOW(L2,L3) &
FLOW(L1,L2) &

RDKILL(L1,y1,labQ) & RDKILL(L1,y1,L1) & RDKILL(L1,y1,L5) & RDKILL(L1,y1,L6) &
RDKILL(L2,z1,labQ) & RDKILL(L2,z1,L2) & RDKILL(L2,z1,L4) & RDKILL(L4,z1,labQ) &
RDKILL(L4,z1,L2) & RDKILL(L4,z1,L4) & RDKILL(L5,y1,labQ) & RDKILL(L5,y1,L1) &
RDKILL(L5,y1,L5) & RDKILL(L5,y1,L6) & RDKILL(L6,y1,labQ) & RDKILL(L6,y1,L1) &
RDKILL(L6,y1,L5) & RDKILL(L6,y1,L6) &

RDGEN(L1,y1,L1) & RDGEN(L2,z1,L2) & RDGEN(L4,z1,L4) & RDGEN(L5,y1,L5) &
RDGEN(L6,y1,L6) &

(A l. INIT(l) => (A x. FVAR(x) => RDIN(l, x, labQ))) &

(A l. A x. A l1. A l2. (! INIT(l) => (RDOUT(l1, x, l2) & FLOW(l1, l) => RDIN(l, x, l2)))) &

(A l. A x. A l1. (RDIN(l, x, l1) & (! RDKILL(l, x, l1)) | RDGEN (l, x, l1)) => RDOUT(l, x, l1))

Fig. 6: RD analysis for the factorial program in *HornInput clause*

Where, the predicate INIT defines the initial block, FVAR gives all the variables used in the program, and FLOW defines all the pairs of blocks that have an edge connected in the flow graph [8]. RDKILL defines all the assignments that are killed by a block, e.g. a tuple $(l_1, x, l_2) \in$ RDKILL means that at block l_1, an assignment to x at block l_2 is destroyed. RDKILL defines all the assignments that are generated by a block, e.g. a tuple $(l_1, x, l_2) \in$ RDGEN means that at block l_1, an assignment to x at block l_2 is generated.

The syntactical constructs for *HornInput clause* with comparison to that for *HornPlus clause* will be given in Table 2.

In the following subsections, we will explain and illustrate how the solver processes this clause in different stages as sketched in Fig. 1 and manipulates *result* and *infl* to compute the final solution.

4.1 FrontEnd.parseClause

The function *FrontEnd.parseClause* parses the clause given in Fig. 6 and transforms the clause into the internal representation, *HornPlus clause*, as shown in Fig. 7.

```
Both (R(INIT,[Const L1]),

Both (R(FVAR,[Const x1]),Both (R(FVAR,[Const z1]), Both (R(FVAR,[Const y1]),

Both (R(FLOW,[Const L4,Const L5]),Both (R(FLOW,[Const L3,Const L4]),
Both (R(FLOW,[Const L5,Const L3]),Both (R(FLOW,[Const L3,Const L6]),
Both (R(FLOW,[Const L2,Const L3]),Both (R(FLOW,[Const L1,Const L2]),

Both (R(RDKILL,[Const L1,Const y1,Const labQ]),Both (R(RDKILL,[Const L1,Const y1,Const L1]),
Both (R(RDKILL,[Const L1,Const y1,Const L5]),Both (R(RDKILL,[Const L1,Const y1,Const L6]),
Both (R(RDKILL,[Const L2,Const z1,Const labQ]),Both (R(RDKILL,[Const L2,Const z1,Const L2]),
Both (R(RDKILL,[Const L2,Const z1,Const L4]),Both (R(RDKILL,[Const L4,Const z1,Const labQ]),
Both (R(RDKILL,[Const L4,Const z1,Const L2]),Both (R(RDKILL,[Const L4,Const z1,Const L4]),
Both (R(RDKILL,[Const L5,Const y1,Const labQ]),Both (R(RDKILL,[Const L5,Const y1,Const L1]),
Both (R(RDKILL,[Const L5,Const y1,Const L5]),Both (R(RDKILL,[Const L5,Const y1,Const L6]),
Both (R(RDKILL,[Const L6,Const y1,Const labQ]),Both (R(RDKILL,[Const L6,Const y1,Const L1]),
Both (R(RDKILL,[Const L6,Const y1,Const L5]),Both (R(RDKILL,[Const L6,Const y1,Const L6]),

Both (R(RDGEN,[Const L1,Const y1,Const L1]),Both (R(RDGEN,[Const L2,Const z1,Const L2]),
Both (R(RDGEN,[Const L4,Const z1,Const L4]),Both (R(RDGEN,[Const L5,Const y1,Const L5]),
Both (R(RDGEN,[Const L6,Const y1,Const L6]),

Both (Forall (0,Implies (U (INIT,[Var 0]),Forall (1,Implies (U (FVAR,[Var 1]),R(RDIN,[Var 0,Var 1,Const labQ]))))),

Both (Forall (2,Forall (3,Forall (4,Forall (5,Implies (N (INIT,[Var 2]), Implies (And (U (RDOUT,[Var 4,Var 3,Var 5]),
U (FLOW,[Var 4,Var 2])),R(RDIN,[Var 2,Var 3,Var 5]))))))),

Forall (6,Forall (7,Forall (8,Implies (Or (And (U (RDIN,[Var 6,Var 7,Var 8]),N (RDKILL,[Var 6,Var 7,Var 8])),
U (RDGEN,[Var 6,Var 7,Var 8])),R(RDOUT,[Var 6,Var 7,Var 8]))))))))))))))))))))))))))))))))))))))))))
```

Fig. 7: RD analysis for the factorial program in *HornPlus clause*

The syntactical comparison between *HornPlus clause* (i.e. Fig. 7) and *HornInput clause* (i.e. Fig. 6) is sketched in Table 2.

In Table 2, prd denotes a predicate symbol; $args$ denotes a tuple of arguments associated with a predicate; cl_1, cl_2 or cl denotes a clause; pre_1, pre_2, or pre denotes a precondition; and x denotes a bounded variable.

4.2 Trans.translate

The function *Trans.translate* translates the clause given in Fig. 7 into the internal representation, *Horn clause*, where all the variables and atoms are repre-

HornInput clause	*HornPlus clause*	legend
$prd(args)$	$R(prd,\ args)$	assertions
$cl_1\ \&\ cl_2$	$Both\ (cl_1,\ cl_2)$	conjunctions in clauses
$Ax.\ cl$	$Forall(x,\ cl)$	universal quantifications in clauses
$pre \Rightarrow cl$	$Implies(pre,\ cl)$	implications
$prd(args)$	$U(prd,\ args)$	queries
$!prd(args)$	$N(prd,\ args)$	negative queries
$pre_1\ \&\ pre_2$	$And(pre_1,\ pre_2)$	conjunctions in preconditions
$pre_1\ \vert\ pre_2$	$Or(pre_1,\ pre_2)$	disjunctions in preconditions

Table 2: Syntactical comparison between *HornInput clause* and *HornPlus clause*

sented by integers as shown in Fig. 8[2].

```
Both (Both (R(0,[Const 0]),

Both (R(1,[Const 1]), Both (R(1,[Const 2]),Both (R(1,[Const 3]),

Both (R(2,[Const 4,Const 5]),Both (R(2,[Const 6,Const 4]),Both (R(2,[Const 5,Const 6]),
Both (R(2,[Const 6,Const 7]),Both (R(2,[Const 8,Const 6]), Both (R(2,[Const 0,Const 8]),

Both (R(3,[Const 0,Const 3,Const 9]), Both (R(3,[Const 0,Const 3,Const 0]),
Both (R(3,[Const 0,Const 3,Const 5]), Both (R(3,[Const 0,Const 3,Const 7]),
Both (R(3,[Const 8,Const 2,Const 9]), Both (R(3,[Const 8,Const 2,Const 8]),
Both (R(3,[Const 8,Const 2,Const 4]), Both (R(3,[Const 4,Const 2,Const 9]),
Both (R(3,[Const 4,Const 2,Const 8]), Both (R(3,[Const 4,Const 2,Const 4]),
Both (R(3,[Const 5,Const 3,Const 9]), Both (R(3,[Const 5,Const 3,Const 0]),
Both (R(3,[Const 5,Const 3,Const 5]), Both (R(3,[Const 5,Const 3,Const 7]),
Both (R(3,[Const 7,Const 3,Const 9]), Both (R(3,[Const 7,Const 3,Const 0]),
Both (R(3,[Const 7,Const 3,Const 5]), Both (R(3,[Const 7,Const 3,Const 7]),

Both (R(4,[Const 0,Const 3,Const 0]), Both (R(4,[Const 8,Const 2,Const 8]),
Both (R(4,[Const 4,Const 2,Const 4]), Both (R(4,[Const 5,Const 3,Const 5]),
Both (R(4,[Const 7,Const 3,Const 7]),

Both (Forall (0,Implies (U(0,[Var 0]),Forall (1,Implies (U(1,[Var 1]),R(5,[Var 0,Var 1,Const 9]))))),

Both (Forall (2,Forall (3,Forall (4,Forall (5,Implies (N(0,[Var 2]),
Implies (And (U(6,[Var 4,Var 3,Var 5]),U(2,[Var 4,Var 2])),R(5,[Var 2,Var 3,Var 5]))))))),

Forall (6,Forall (7,Forall (8,Implies (Or (And (U(5,[Var 6,Var 7,Var 8]),N(3,[Var 6,Var 7,Var 8])),
U(4,[Var 6,Var 7,Var 8])),R(6,[Var 6,Var 7,Var 8]))))))))))))))))))))))))))))))))))))))))))),

One)
```

Fig. 8: RD analysis for the factorial program in *Horn clause*

At the same time, it extracts the static information as given in Fig. 9. In the figure, the $relTab$ gives the information about predicates by means of the predicate symbol and the associated arity, e.g. INIT/1 denotes the 1-ary predicate INIT.

[2]The true clause *One* is added in the last line to ease the construction of clauses in the implementation. It is removed later on as shown in Fig. 11.

universe:
0: L1
1: x1
2: z1
3: y1
4: L4
5: L5
6: L3
7: L6
8: L2
9: labQ

relTab:
0: INIT/1
1: FVAR/1
2: FLOW/2
3: RDKILL/3
4: RDGEN/3
5: RDIN/3
6: RDOUT/3

all: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
n: 7

Fig. 9: Extracted static information

4.3 Solve.solve

The function *Solve.solve* does mainly three tasks. It first initializes the two data structures *result* and *infl*. It secondly calls the function *HornEnvPool.addPools* to add pools for the memoisation of the disjunctions and existential quantifications in preconditions. It then processes all the sub-clauses and preconditions of the input clause to compute the solution.

4.3.1 Initializing *result* and *infl*.

Fig. 10 illustrates the initial states of *result* and *infl*. In both cases, the length of either the stack or the buckets is 7, as there are 7 predicates (as indicated in n in Fig. 9) in the clause. Each slot in the stack is initialized as NONE, while each slot in the buckets is initialized as empty list []. The number of elements indicates the number of edges in the set of prefix trees implemented by either *result* or *infl*.

4.3.2 Calling HornEnvPool.addPools.

To achieve efficiency, the solver uses the memoisation technique to deal with disjunctions or existential quantifications in preconditions. The function *addPools* transforms the *Horn clause* into an internal representation, *HornEnvPool clause*, as illustrated in Fig. 11 where the *Or* precondition has one more argument pointing to location 7 in the stack. The slot in location 7 is created by a push operation, and is used to save the partial environment (i.e. *env*) for the memoisation purpose. Fig. 12 shows the modified *result*.

result: Stack length: 7 Stack: NONE NONE NONE NONE NONE NONE NONE Buckets length: 7 Buckets: [] [] [] [] [] [] [] Number of elements: 0	*infl*: Stack length: 7 Stack: NONE NONE NONE NONE NONE NONE NONE Buckets length: 7 Buckets: [] [] [] [] [] [] [] Number of elements: 0

Fig. 10: Initial states of *result* and *infl*

```
Both (R(0,[Const 0]),

Both (R(1,[Const 1]), Both (R(1,[Const 2]), Both (R(1,[Const 3]),

Both (R(2,[Const 4,Const 5]), Both (R(2,[Const 6,Const 4]), Both (R(2,[Const 5,Const 6]),
Both (R(2,[Const 6,Const 7]), Both (R(2,[Const 8,Const 6]), Both (R(2,[Const 0,Const 8]),

Both (R(3,[Const 0,Const 3,Const 9]), Both (R(3,[Const 0,Const 3,Const 0]),
Both (R(3,[Const 0,Const 3,Const 5]), Both (R(3,[Const 0,Const 3,Const 7]),
Both (R(3,[Const 8,Const 2,Const 9]), Both (R(3,[Const 8,Const 2,Const 8]),
Both (R(3,[Const 8,Const 2,Const 4]), Both (R(3,[Const 4,Const 2,Const 9]),
Both (R(3,[Const 4,Const 2,Const 8]), Both (R(3,[Const 4,Const 2,Const 4]),
Both (R(3,[Const 5,Const 3,Const 9]), Both (R(3,[Const 5,Const 3,Const 0]),
Both (R(3,[Const 5,Const 3,Const 5]), Both (R(3,[Const 5,Const 3,Const 7]),
Both (R(3,[Const 7,Const 3,Const 9]), Both (R(3,[Const 7,Const 3,Const 0]),
Both (R(3,[Const 7,Const 3,Const 5]), Both (R(3,[Const 7,Const 3,Const 7]),

Both (R(4,[Const 0,Const 3,Const 0]), Both (R(4,[Const 8,Const 2,Const 8]),
Both (R(4,[Const 4,Const 2,Const 4]), Both (R(4,[Const 5,Const 3,Const 5]),
Both (R(4,[Const 7,Const 3,Const 7]),

Both (Forall (0,Implies (U(0,[Var 0]),Forall (1,Implies (U(1,[Var 1]),R(5,[Var 0,Var 1,Const 9]))))),

Both (Forall (2,Forall (3,Forall (4,Forall (5,Implies (N(0,[Var 2]),
Implies (And (U(6,[Var 4,Var 3,Var 5]),U(2,[Var 4,Var 2])),R(5,[Var 2,Var 3,Var 5]))))))),

Forall (6,Forall (7,Forall (8,Implies (Or (And (U(5,[Var 6,Var 7,Var 8]),N(3,[Var 6,Var 7,Var 8])),
U(4,[Var 6,Var 7,Var 8]), 7),R(6,[Var 6,Var 7,Var 8]))))))))))))))))))))))))))))))))))))))))))
```

Fig. 11: RD analysis for the factorial program in *HornEnvPool clause*

4.3.3 Solving clauses.

The main algorithm that the *solve* function uses to solve clauses is given in Appendix B. The *solve* function is primarily composed of two functions i.e. *execute* and *check*. The *execute* function deals with clauses, while the *check* function deals with preconditions. In the followings we shall focus on those clauses and preconditions occurred in Fig. 11 that are sensitive to the data structures *result* and *infl*, meaning that by dealing with them the *solve* function modifies the two data structures. We describe briefly the other clauses occurred

result:
Stack length: 8

Stack:
NONE NONE NONE NONE NONE NONE NONE NONE

Buckets length: 7

Buckets: [] [] [] [] [] [] []

Number of elements: 0

Fig. 12: The state of *result* after the call to *addPools*

in Fig. 11 when we come across them. For more detail descriptions about the algorithm, we refer to [2].

execute(Both(cl_1, cl_2)) env. In the case of the conjunction in clauses i.e. with the syntactical form $Both(cl_1, cl_2)$, the *execute* function executes two sub-clauses cl_1 and cl_2 respectively under the same partial environment *env*. For example, consider the first conjunction in line 1 of Fig. 11, the assertion $R(0,$ $[Const\ 0])$ is executed first, then the second conjunction in line 2 is executed.

execute(R(r, args)) env. To deal with an assertion $R(r, args)$, where r is an integer representing a predicate, and $args$ is a list of arguments associated with the predicate, the *execute* function considers two cases based on $args$:

1) All arguments in $args$ are either constant values indicated by $Const$ or variables indicated by Var that have been evaluated in *env*. The *execute* function derives a tuple of integers (each represents an atom from the universe) corresponding to $args$, and inserts the tuple into *result* (and at the same time *infl* is also modified). For example, when $R(0, [Const\ 0])$ in line 1 of the clause given in Fig. 11 is executed, the tuple [0] is derived and then inserted into *result* as illustrated in Fig. 13.

Stack length: 9

Stack:
SOME[] NONE NONE NONE NONE NONE NONE NONE **SOME [0]**

Buckets length: 7

Buckets:
[(0/0,8)] [] [] [] [] [] []

Number of elements: 1

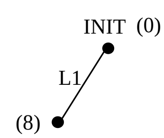

Fig. 13: *result* after the execution of $R(0, [Const\ 0])$

In Fig. 13, the texts in bold font constitute the prefix tree (on the right corner) for the predicate INIT. Where, again, NONE denotes the node uninitialized so far. SOME [] denotes the leaf node, and SOME [0] denotes the node that prefixes its child node by atom 0 (i.e. L1) from the universe. It needs to mention that, hereafter, we shall use the notation $(v_1/a,\, v_2)$ as an alternation of the notation $((v_1,\, a),\, v_2)$ (c.f. Fig. 3) to denote an element in the buckets (simply to get rid of many parentheses), e.g. (0/0,8) means the same as ((0, 0), 8). When the execution of $R(1,\, [Const\ 1])$ in line 2 of Fig. 11 is done, *result* is modified as shown in Fig. 14.

Stack length: 10

Stack:
SOME [] SOME [] NONE NONE NONE NONE NONE NONE **SOME [1]** SOME [0]

Buckets length: 7

Buckets:
[(0/0,8)] [] [] [] **[(1/1, 9)]** [] []

Number of elements: 2

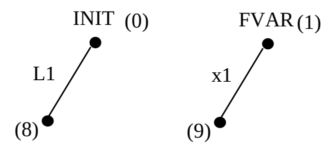

Fig. 14: *result* after the execution of $R(1,\, [Const\ 1])$

When $R(4,\, [Const\ 7,\, Const\ 3,\, Const\ 7])$ in line 16 in Fig. 11 is executed, *result* is expanded as shown in Fig. 15, which corresponds to a set of prefix trees illustrated in Fig. 16.

Stack length: 66
Stack:
SOME [] SOME [7] SOME [3] SOME [] SOME [5] SOME [3] SOME [] SOME [4] SOME [2] SOME [] SOME [8] SOME [2] SOME [] SOME [0] SOME [3] SOME [] SOME [] SOME [] SOME [] SOME [7,5,0,9] SOME [3] SOME [] SOME [] SOME [] SOME [] SOME [7,5,0,9] SOME [3] SOME [] SOME [] SOME [] SOME [4,8,9] SOME [2] SOME [] SOME [] SOME [] SOME [4,8,9] SOME [2] SOME [] SOME [] SOME [] SOME [] SOME [7,5,0,9] SOME [3] SOME [] SOME [8] SOME [] SOME [6] SOME [] SOME [] SOME [6] SOME []SOME [7,4] SOME [] SOME [5] SOME [] SOME [] SOME [] SOME [] NONE NONE NONE SOME [7,5,4,8,0] SOME [7,5,4,8,0] SOME [0,8,5,6,4] SOME [3,2,1] SOME [0]
Buckets length: 112
Buckets:
[] [(40/0,42)] [] [(58/4,59)] [] [(4/0,51)] [(40/5,43)] [] [(40/7,44)] [(51/3,52)(4/4,57)] [(40/9,41)(4/5,60)] [] [(4/7,63)] [(4/8,54)] [] [] [] [] [] [] [] [] [] [] [] [(1/1,9)] [(1/2,10)(55/8,56)] [(1/3,11)] [(19/6,20)] [] [] [(30/4,33)] [] [] [(12/5,13)] [(30/8,32)] [(30/9,31)] [(23/3,24)(52/0,53)] [] [] [] [(34/2,35)] [] [] [] [(63/3,64)] [] [(16/6,17)(45/3,46)] [] [] [] [] [] [] [] [(2/0,21)] [] [] [] [(2/4,12)] [(2/5,16)] [(2/6,14)] [] [(2/8,19)] [(60/3,61)] [(24/0,26)] [] [] [] [] [(24/5,27)] [] [(24/7,28)] [] [(24/9,25)(35/4,38)] [(46/0,48)] [] [] [(35/8,37)] [(35/9,36)] [(46/5,49)(64/7,65)] [] [(46/7,50)(57/2,58)] [] [(46/9,47)] [(39/3,40)] [(3/0,23)] [] [] [] [(3/4,34)] [(3/5,39)] [(21/8,22)] [(3/7,45)] [(3/8,29)] [(14/4,15)] [] [(61/5,62)] [(14/7,18)] [] [] [(54/2,55)] [] [] [] [(0/0,8)] [] [] [] [] [(29/2,30)] []
Number of elements: 58

Fig. 15: *result* after the execution of $R(4,\, [Const\ 7,\, Const\ 3,\, Const\ 7])$

2)The arguments in *args* contain variables that have not been evaluated in *env*. The *execute* function constructs a list of tuples of integers by evaluating such a variable with each atom of the universe respectively, and inserts each

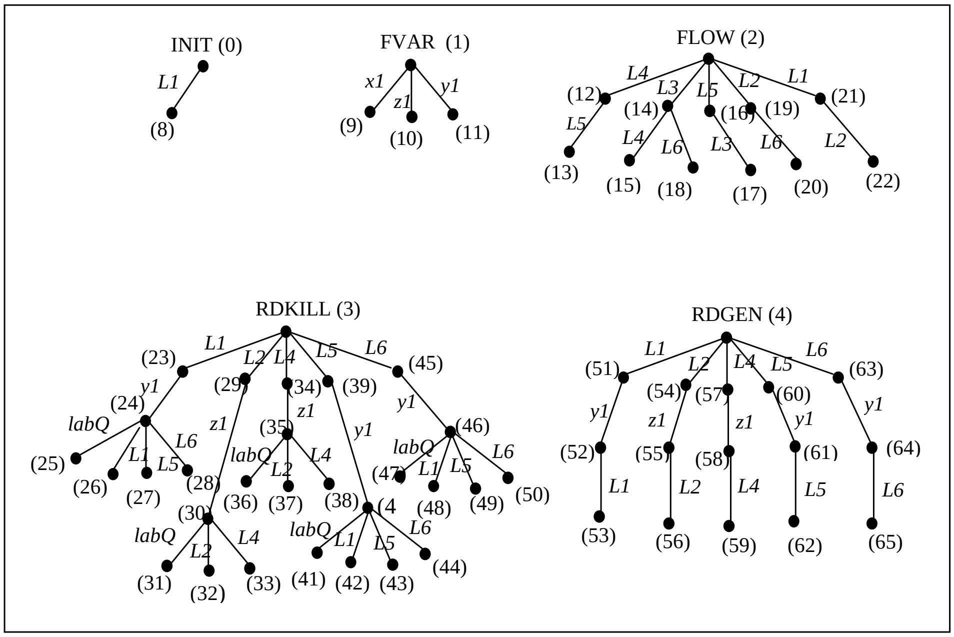

Fig. 16: Prefix trees after the execution of $R(4, [Const\ 7, Const\ 3, Const\ 7])$

tuple of atoms into *result* as in case 1. For example, if we have an assertion like $R(0, [Var\ 0])$, which corresponds to INIT(x), and env = [], then the *execute* function constructs a list of tuples as: [0], [1], [2], [3], [4], [5], [6], [7], [8], [9]. Each element in the tuples denotes an atom from the universe. Each of the tuple is respectively inserted into *result*.

execute(Forall(x, cl)) env. In the case of the universal quantification in the clause, i.e. $Forall(x, cl)$, the *execute* function introduces the new variable x by extending *env* as (x, NONE)::*env*, here, NONE means that x is not binded by any value yet. It then executes *cl* in the extended *env*. For example, when the *execute* function processes the the first universal quantification in line 17 in Fig. 11, i.e. $Forall(0, Implies(U(0, [Var\ 0]), Forall(...)))$, here, we use ... to denote the rest of the clause, it updates the *env* from *env* = [] to *env*=(x, NONE). Afterwards, it processes the implication clause $Implies(U(0, [Var\ 0]), Forall(...))$ under the new *env*.

execute(Implies(pre, cl)) env. To deal with the implication, i.e. $Implies(pre, cl)$, the *execute* function first calls the *check* function to check the precondition *pre*. At the same time, it passes *execute*(*cl*) to the *check* function as the continuation. If the precondition is true, the conclusion *cl* is executed. Otherwise, the current execution terminates. Consider the same example $Implies(U(0, [Var\ 0]), Forall(...))$ as above, when the precondition $U(0, [Var\ 0])$ is true, the conclusion $Forall(...)$ is executed. Otherwise, there is no further execution.

check(U(r, args), next) env. To deal with the query $U(r,\ args)$, the *check* function looks up the data structure *result* and derives a list of tuples associated with the predicate r, and then unifies each tuple with $args$. If the unification is successful, the solver will continue to work with the $next$ clause or precondition accordingly. To be efficient, the *check* function proceeds in three steps:

- Split $args$ into two parts: $prefix$ and $rest$. The $prefix$ contains the first part of $args$, in which all arguments are either of $Const$ or Var that have been evaluated in env. The $rest$ contains the remains of $args$. For example, the $args$ in $U(2,\ [Const\ 4,\ Var\ 0])$, is split into $prefix = [4]$, and $rest = [Var\ 0]$.
- The list $prefix$ is then used to look up *result* for a list of tuples, that have such a prefix, associated with predicate r, and then each of the tuple is used to unify with $rest$. For the previous example, if the prefix tree for predicate 2 (i.e. FLOW) as shown in Fig. 16 contains only the most left path, i.e. (L4, L5) $\in$ FLOW, in the current *result*, then $list = [5]$, and the atom 5 (i.e. $L5$) is used to unify with $Var\ 0$.
- In some situations, the query may not be satisfied with the current tuples in *result* under the current env, but it may be satisfied when some new tuples add into *result*. In this case, the solver adds a consumer into *infl* so that the current computation can be resumed later when a new tuple associated with the predicate is inserted into *result*.

Example 3. After the check of two queries $U(0,\ [Var\ 0])$ and $U(1,\ [Var\ 1])$ in line 17 in Fig. 11, the modified *infl* together with the three lists, i.e. $prefix$, $rest$ and $list$, are illustrated in Fig. 17 and Fig. 18 respectively.

prefix: []
rest: [Var 0]
list of tuples with the prefix: [0]
Stack length: 65
Stack:
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE **SOME [c]**
Buckets length: 112
Buckets:
[(29/4,32)] [] [] [(11/5,12)] [(29/8,31)] [(29/9,30)(4/0,50)] [(22/3,23)(51/0,52)] [] [] [(4/4,56)]
[(33/2,34)(4/5,59)] [] [(4/7,62)] [(4/8,53)] [(62/3,63)] [] [(15/6,16)(44/3,45)] [] [] [] [] [] [] [] [] [(1/1,8)]
[(1/2,9)] [(1/3,10)] [] [] [] [] [] [(59/3,60)] [(23/0,25)] [] [] [] [] [(23/5,26)] [] [(23/7,27)] []
[(23/9,24)(34/4,37)] [(45/0,47)] [] [] [(34/8,36)] [(34/9,35)] [(45/5,48)(63/7,64)] [] [(45/7,49)(56/2,57)] []
[(45/9,46)] [(38/3,39)] [(2/0,20)] [] [] [] [(2/4,11)] [(2/5,15)] [(2/6,13)(20/8,21)] [] [(2/8,18)] [(13/4,14)]
[] [(60/5,61)] [(13/7,17)] [] [] [(53/2,54)] [] [] [] [] [] [] [] [] [(28/2,29)] [] [] [(39/0,41)] [] [(57/4,58)] []
[(3/0,22)] [(39/5,42)] [] [(39/7,43)] [(3/4,33)(50/3,51)] [(3/5,38)(39/9,40)] [] [(3/7,44)] [(3/8,28)] [] [] []
[] [] [] [] [] [] [] [(0/0,7)] [] [(54/8,55)] [] [(18/6,19)] [] []
Number of elements: 58

Fig. 17: *infl* and the three lists after the check of $U(0,\ [Var\ 0])$

In Fig.17, as in *result*, each slot in the stack corresponds to a node in a prefix tree. The content in a slot is either NONE denoting a node without a consumer, or SOME $[c_1, ..., c_i]$ denoting a node with i $(i \geq 1)$ consumers, i.e. $c_1, ..., c_i$. Since $c_1, ..., c_i$ each is merely used to denote the existence of a consumer, we shall use the same notation c to denotes each of them. The solver will distinguish them. The contents in the buckets are essentially the same as that in *result.* The main difference is that in the stack of *result*, there are pools for the memoisation, thus, the same node v in the prefix tree may be located in slot i in the stack of *result* but in slot j in the stack of *infl*, assuming that $i \neq j$. Therefore, the contents in the buckets in Fig. 17 are different from that in Fig. 15 although they contain the same prefix trees. It is also noticeable that the size of the stack is 65 in Fig. 17 whereas 66 in Fig. 15 where slot 7 is used for the memoisation as discussed in section 4.3.2.

prefix: []
rest: [Var 1]
list of tuples with the prefix: [3], [2], [1]
Stack length: 65
Stack:
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE **SOME [c]** SOME [c]
Buckets length: 112
Buckets:
[(29/4,32)] [] [] [(11/5,12)] [(29/8,31)] [(29/9,30)(4/0,50)] [(22/3,23)(51/0,52)] [] [] [(4/4,56)]
[(33/2,34)(4/5,59)] [] [(4/7,62)] [(4/8,53)] [(62/3,63)] [] [(15/6,16)(44/3,45)] [] [] [] [] [] [] [] [] [(1/1,8)]
[(1/2,9)] [(1/3,10)] [] [] [] [] [] [(59/3,60)] [(23/0,25)] [] [] [] [] [(23/5,26)] [] [(23/7,27)] []
[(23/9,24)(34/4,37)] [(45/0,47)] [] [] [(34/8,36)] [(34/9,35)] [(45/5,48)(63/7,64)] [] [(45/7,49)(56/2,57)] []
[(45/9,46)] [(38/3,39)] [(2/0,20)] [] [] [] [(2/4,11)] [(2/5,15)] [(2/6,13)(20/8,21)] [] [(2/8,18)] [(13/4,14)] []
[(60/5,61)] [(13/7,17)] [] [] [(53/2,54)] [] [] [] [] [] [] [] [] [(28/2,29)] [] [] [(39/0,41)] [] [(57/4,58)] []
[(3/0,22)] [(39/5,42)] [] [(39/7,43)] [(3/4,33)(50/3,51)] [(3/5,38)(39/9,40)] [] [(3/7,44)] [(3/8,28)] [] [] [] []
[] [] [] [] [] [] [(0/0,7)] [] [(54/8,55)] [] [(18/6,19)] [] []
Number of elements: 58

Fig. 18: *infl* and three lists after the check of $U(1, [Var\ 1])$

check(N(r, args), next) env. To deal with an negative query $N(r,\ args)$, the *check* function does three things:

- Construct a list $vars$ from $args$ which are not evaluated in env.
- For each $x \in vars$, and each atom $a \in all$, update env with a pair $(x,\ a)$, and eventually obtain a list $envList$ of env.
- For each $env \in envList$, construct a tuple of atoms by binding the variables in $args$ with the values of the same variables in env, and check whether such a tuple has already been in *result.* If the tuple is not in *result*, the corresponding env is propagated to the $next$ clause or precondition accordingly.

Example 4. Consider the negative query $N(0,\ [Var\ 2])$, in line 18 in Fig. 11, the $vars$ and $envList$ are illustrated in Fig. 19. Where, $envList$ contains 10 envs by binding the variable 2 with value 0 to 9 (i.e. atom 0 to 9 from the universe) respectively. The 10 tuples [0], [1], ..., [9] will be checked against the prefix tree for predicate 0 (i.e. INIT) in *result* to see whether any of them has been in. It finds out that [0] is in *result*, so that the last 9 *env*s will be propagated except the first one.

```
vars: [2]

envList:
[(5,NONE), (4,NONE), (3,NONE), (2,SOME 0)], [(5,NONE), (4,NONE), (3,NONE), (2,SOME 1)],
[(5,NONE), (4,NONE), (3,NONE), (2,SOME 2)], [(5,NONE), (4,NONE), (3,NONE), (2,SOME 3)],
[(5,NONE), (4,NONE), (3,NONE), (2,SOME 4)], [(5,NONE), (4,NONE), (3,NONE), (2,SOME 5)],
[(5,NONE), (4,NONE), (3,NONE), (2,SOME 6)], [(5,NONE), (4,NONE), (3,NONE), (2,SOME 7)],
[(5,NONE), (4,NONE), (3,NONE), (2,SOME 8)], [(5,NONE), (4,NONE), (3,NONE), (2,SOME 9)]
```

Fig. 19: Outcomes from the check of $N(0, [Var\ 2])$

check(And(pre_1, pre_2), next) env. In the case of the conjunction in the precondition i.e. $And(pre_1, pre_2)$, the *check* function checks pre_1 under the environment *env*. At the same time it passes $check(pre_2, next)$ as the continuation of the *check* function that checks pre_1. If pre_1 is true, pre_2 is checked under the environment propagated by the first check. Otherwise, no check on pre_2 is needed, i.e. the current check terminates.

check(Or(pre_1, pre_2, pool), next) env. In the case of the disjunction in the precondition i.e. $Or(pre_1, pre_2, pool)$, the *check* function checks preconditions pre_1 and pre_2 respectively, and propagates the new *env* to the *next* clause or precondition. To be efficient, if both checks produce the same *env*, the second check does not need to consider the *next*, since it has been done once in checking the first precondition. This can be achieved by the memoisation, which saves the first *env* in the *pool* that is created in *result* previously, and when the second *env* is generated, if it is the same as the first one, the check terminates.

Example 5. Consider the *Or* precondition in line 19 in Fig. 11: $Or(And(U(5, [Var\ 6, Var\ 7, Var\ 8]), N(3, [Var\ 6, Var\ 7, Var\ 8])), U(4, [Var\ 6, Var\ 7, Var\ 8]), \mathbf{7})$, where the third argument, integer 7, points to the location in the stack of *result*. When $U(5, [Var\ 6, Var\ 7, Var\ 8])$ is checked, the prefix tree for predicate 5 (i.e. RDIN) in *result* is as illustrated in Fig. 20.

After the check is done, the modified *infl* together with three lists i.e. $prefix$, $rest$ and $list$ are as shown in Fig. 21.

Thus, when the first tuple in $list$ unifies with $rest$, the environment $env = [(8, 9), (7, 1), (6, 0)]$, and it will not be modified by checking $N(3,[Var\ 6, Var\ 7, Var\ 8]))$. This *env* is then saved in the *pool* that is rooted in location 7 of the stack in *result*. When this *env* is added, *result* is modified as illustrated in Fig. 22.

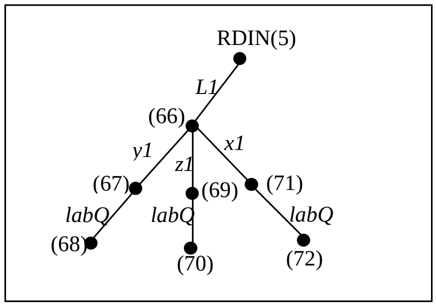

Fig. 20: The current prefix tree in *result* for predicate RDIN

prefix: []
rest: [Var 6,Var 7,Var 8]
list of tuples with the prefix:
[0,1,9], [0,2,9], [0,3,9]
Stack length: 72
Stack:
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE SOME [c,c,c,c,c,c,c,c,c] SOME [c]
NONE NONE NONE SOME [c] SOME [c]
Buckets length: 112
Buckets:
[(29/4,32)] [] [] [(11/5,12)] [(29/8,31)] [(29/9,30)(4/0,50)] [(22/3,23)(51/0,52)] [] [] [(4/4,56)]
[(33/2,34)(4/5,59)] [] [(4/7,62)] [(4/8,53)] [(62/3,63)] [] [(15/6,16)(44/3,45)] [] [] [] [] [] [] [] [] [(1/1,8)]
[(1/2,9)] [(1/3,10)] [] [] [] [] [(66/9,67)] [(59/3,60)] [(23/0,25)] [] [(5/0,65)] [] [] [(23/5,26)] [] [(23/7,27)]
[] [(23/9,24)(34/4,37)] [(45/0,47)(70/9,71)] [] [] [(34/8,36)] [(34/9,35)] [(45/5,48)(63/7,64)] []
[(45/7,49)(56/2,57)] [] [(45/9,46)] [(38/3,39)] [(2/0,20)] [] [] [] [(2/4,11)] [(2/5,15)] [(2/6,13)(20/8,21)] []
[(2/8,18)] [(13/4,14)] [] [(60/5,61)] [(13/7,17)] [] [] [(53/2,54)] [] [] [] [] [] [] [] [] [(28/2,29)] [] []
[(39/0,41)] [] [(57/4,58)] [] [(3/0,22)] [(39/5,42)] [] [(39/7,43)] [(3/4,33)(50/3,51)] [(3/5,38)(39/9,40)] []
[(3/7,44)] [(3/8,28)(68/9,69)] [] [] [] [] [] [] [] [] [] [] [(0/0,7)(65/1,70)] [(65/2,68)] [(54/8,55)(65/3,66)] []
[(18/6,19)] [] []
Number of elements: 65

Fig. 21: *infl* and three lists after the check of $U(5, [Var\ 6, Var\ 7, Var\ 8])$

More on execute(R(r, args)) env. Previously, when we discussed the execution of the assertion $R(r, args)$, we hided one important thing that the *execute* function does. It is to resume the corresponding consumers when a new tuple is added in *result*. If we consider the query $U(6, [Var\ 4, Var\ 3, Var\ 5])$ in line 19 in Fig. 11 as an example, when this query is checked, *result* contains nothing about predicate 6 (i.e. RDOUT), therefore the *check* function suspends the current computation and adds the consumer to *infl* for each *env* that is propagated by checking $N(0, [Var\ 2])$ as we discussed before. Thus, there are 9 consumers registered in *infl* at location 6 in the stack, as already shown in Fig. 21.

When the assertion $R(6,[Var\ 6, Var\ 7, Var\ 8])$ in the last line in Fig. 11 is processed, the first tuple [0, 1, 9] is inserted into *result* as illustrated in Fig. 23.

```
Stack length: 76
Stack:
SOME [] SOME [0] SOME [1] SOME [] SOME [9] SOME [] SOME [9] SOME [] SOME [9]
SOME [1,2,3] SOME [] SOME [7] SOME [3] SOME [] SOME [5] SOME [3] SOME [] SOME [4]
SOME [2] SOME [] SOME [8] SOME [2] SOME [] SOME [0] SOME [3] SOME [] SOME []
SOME [] SOME [] SOME [7,5,0,9] SOME [3] SOME [] SOME [] SOME [] SOME [] SOME [7,5,0,9]
SOME [3] SOME [] SOME [] SOME [] SOME [4,8,9] SOME [2] SOME [] SOME [] SOME []
SOME [4,8,9] SOME [2] SOME [] SOME [] SOME [] SOME [] SOME [7,5,0,9] SOME [3] SOME []
SOME [8] SOME [] SOME [6] SOME [] SOME [] SOME [6] SOME [] SOME [7,4] SOME []
SOME [5] SOME [] SOME [] SOME [] SOME [] SOME [9] NONE SOME [0] SOME [7,5,4,8,0]
SOME [7,5,4,8,0] SOME [0,8,5,6,4] SOME [3,2,1] SOME [0]
Buckets length: 112
Buckets:
[] [(40/0,42)] [] [(58/4,59)] [] [(4/0,51)] [(40/5,43)] [] [(40/7,44)] [(51/3,52)(4/4,57)] [(40/9,41)(4/5,60)]
[] [(4/7,63)] [(4/8,54)(69/9,70)] [] [] [] [(73/1,74)] [] [] [] [] [] [] [(66/1,71)] [(1/1,9)(66/2,69)]
[(1/2,10)(55/8,56)(66/3,67)] [(1/3,11)] [(19/6,20)] [] [] [(30/4,33)] [] [] [(12/5,13)] [(30/8,32)]
[(30/9,31)(5/0,66)] [(23/3,24)(52/0,53)] [] [] [] [(34/2,35)] [] [] [] [(63/3,64)] []
[(16/6,17)(45/3,46)(74/0,75)] [] [] [] [] [] [] [] [(2/0,21)] [] [] [] [(2/4,12)] [(2/5,16)] [(2/6,14)] []
[(2/8,19)(67/9,68)] [(60/3,61)] [(24/0,26)] [] [] [] [] [(24/5,27)] [] [(24/7,28)] [] [(24/9,25)(35/4,38)]
[(46/0,48)(71/9,72)] [] [] [(35/8,37)] [(35/9,36)] [(46/5,49)(64/7,65)] [] [(46/7,50)(57/2,58)] [] [(46/9,47)]
[(39/3,40)] [(3/0,23)] [] [] [] [(3/4,34)] [(3/5,39)] [(21/8,22)] [(3/7,45)] [(3/8,29)] [(14/4,15)] [] [(61/5,62)]
[(14/7,18)] [] [] [(54/2,55)] [] [] [] [(0/0,8)] [] [(7/9,73)] [] [] [(29/2,30)] []
Number of elements: 68
```

Fig. 22: Memoisation for the *Or* precondition

```
Stack length: 79
Stack:
SOME [] SOME [9] SOME [1] SOME [] SOME [0] SOME [1] SOME [] SOME [9] SOME []
SOME [9] SOME [] SOME [9] SOME [1,2,3] SOME [] SOME [7] SOME [3] SOME [] SOME [5]
SOME [3] SOME [] SOME [4] SOME [2] SOME [] SOME [8] SOME [2] SOME [] SOME [0]
SOME [3] SOME [] SOME [] SOME [] SOME [] SOME [7,5,0,9] SOME [3] SOME [] SOME []
SOME [] SOME [] SOME [7,5,0,9] SOME [3] SOME [] SOME [] SOME [] SOME [4,8,9] SOME [2]
SOME [] SOME [] SOME [] SOME [4,8,9] SOME [2] SOME [] SOME [] SOME [] SOME []
SOME [7,5,0,9] SOME [3] SOME [] SOME [8] SOME [] SOME [6] SOME [] SOME [] SOME [6]
SOME [] SOME [7,4] SOME [] SOME [5] SOME [] SOME [] SOME [] SOME [] SOME [9]
SOME [0] SOME [0] SOME [7,5,4,8,0] SOME [7,5,4,8,0] SOME [0,8,5,6,4] SOME [3,2,1] SOME [0]
Buckets length: 112
Buckets:
[] [(40/0,42)] [] [(58/4,59)] [] [(4/0,51)] [(40/5,43)] [] [(40/7,44)] [(51/3,52)(4/4,57)]
[(40/9,41)(4/5,60)] [] [(4/7,63)] [(4/8,54)(69/9,70)] [] [] [] [(73/1,74)] [] [] [] [] [] [] [(66/1,71)]
[(1/1,9)(66/2,69)] [(1/2,10)(55/8,56)(66/3,67)] [(1/3,11)] [(19/6,20)] [] [] [(30/4,33)] [] [] [(12/5,13)]
[(30/8,32)] [(30/9,31)(5/0,66)] [(23/3,24)(52/0,53)(77/9,78)] [] [] [] [(34/2,35)] [] [] [] [(63/3,64)] []
[(16/6,17)(45/3,46)(74/0,75)] [] [] [] [] [] [] [] [(2/0,21)] [] [] [] [(2/4,12)] [(2/5,16)] [(2/6,14)] []
[(2/8,19)(67/9,68)] [(60/3,61)] [(24/0,26)] [] [(6/0,76)] [] [] [(24/5,27)] [] [(24/7,28)] []
[(24/9,25)(35/4,38)] [(46/0,48)(71/9,72)] [] [] [(35/8,37)] [(35/9,36)] [(46/5,49)(64/7,65)] []
[(46/7,50)(57/2,58)] [] [(46/9,47)] [(39/3,40)] [(3/0,23)] [] [] [] [(3/4,34)] [(3/5,39)] [(21/8,22)]
[(3/7,45)] [(3/8,29)] [(14/4,15)] [] [(61/5,62)] [(14/7,18)] [] [] [(54/2,55)] [] [] [] [(0/0,8)] [] [(7/9,73)]
[] [] [(29/2,30)(76/1,77)] []
Number of elements: 71
```

Fig. 23: *result* after inserting a tuple associated with predicate RDOUT

When the insertion is done, the *execute* function constructs a list of consumers associated with predicate 6 (i.e. RDOUT), and resumes all the computations that were suspended before.

The solution. The solution generated by the solver to the reaching definitions analysis for the factorial program is exactly the same as that given in the book [8]. The *Output* module prints out the solution as shown in Appendix C. The final data structures *result* and *infl* are given in Appendix D and E respectively.

5 Conclusion

We have gained some insights into both the internal data structures and the internal behaviour of the succinct solver, which enhances our confidence in tuning clauses for using the solver efficiently. Concerning the solver as a general tool for solving static analysis problems specified in ALFP, we consider the future work to improve the solver in the following aspects:

- Provide more flexible API (application programmer interface) to facilitate both new users and experts in various analyses phases.
- Support directly a universe of terms in a free algebra to release the current restrictions on the universe of atoms and ground terms.
- Enhance the space efficiency of the solver so that some of the design choices in favour of time efficiency over space efficiency can be improved.
- Reuse the *result* from the previous solving to deal with a notion of *iterative program analysis* developed for security analyses in the context of mobility.

We are working towards these improvements of the solver.

References

[1] H. Riis Nielson and F. Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. To appear in the book *The Essence of Computation: Complexity, Analysis, Transformation*, published as LNCS 2566, Springer Verlag, 2002.

[2] F. Nielson, H. Seidl, and H. Riis Nielson. Succinct Solvers. To *Nordic Journal of Computing*, 1997.

[3] F. Nielson and H. Seidl. Control-Flow Analysis in Cubic Time. In *The 10th European Symposium on Programming (ESOP)*. LNCS 2028, Springer Verlag, 2001.

[4] David McAllester. On the Complexity Analysis of Static Analysis. In *The 6th Static Analysis Symposium (SAS)*. LNCS 1694, Springer Verlag, 1999.

[5] F. Nielson, H. Riis Nielson, and H. Seidl. Automatic Complexity Analysis. In *The 11th European Symposium on Programming (ESOP)*. LNCS 2305, Springer Verlag, 2002.

[6] M. Buchholtz, H. Riis Nielson, and F. Nielson. Experiments with Succinct Solvers. Technical Report IMM-TR-2002-4, IMM, DTU, 2002.

[7] René Rydhof Hansen, F. Nielson, and H. Riis Nielson. Security Analysis for Carmel. In *VeriSafe Workshop*, 2002.

[8] F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[9] A. Chandra and D. Harel. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, 25(2), 1980.

[10] K. Apt, H. Blair, and A. Walker. Towards A Theory of Declarative Programming. In J. Minsker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan-Kaufman, 1988.

Appendix A. Type declarations for *env*, *result* and *infl*

```
type env = (int * (int option)) list

type forest = list option Stack.stack * Table.table
type 'a stack = (int * 'a array) ref
type table = {buckets: bucket array ref,
                 hash: Item.item → int) ref,
                count: int ref}

val result: forest    (* abstraction of value result *)

type bucket = (Item.item * value) list
type Item.item = IntPairItem.item  (* see Note 1 *)
type IntPairItem.item = (int * int)  (* see Note 2 *)
type value = int

type infl = consumer list ForestMap.map
type 'a map = 'a option Stack.stack * Table.table
type consumer = int list → unit
```

Note 1: Item.item denotes item in structure Item

Note 2: IntPairItem.item denotes item in structure IntPairItem

Appendix B. Main algorithm

```
fun check (R(x⃗), K) η         =  let fun K' a⃗ =  case unify(η, x⃗, a⃗) of
                                                     NONE -> ()
                                                   | SOME η' -> K(η')
                                   in (infl.register(R, K');
                                       app K' (rho.sub R))
                                   end
  | check (¬R(x⃗), K) η        =  let fun K' a⃗ =  if rho.has(R, a⃗)
                                                   then ()
                                                   else K (unify(η, x⃗, a⃗))
                                   in app K' (unifiable (η, x⃗))
                                   end
  | check (pre₁ ∧ pre₂, K) η   =  check (pre₁, check (pre₂, K)) η
  | check (pre₁ ∨ pre₂, K) η   =  check (pre₁, K) η; check (pre₂, K) η
  | check (∃x : pre, K) η      =  check (pre, K ∘ tl) ((x, NONE)::η)
  | check (∀x : pre, K) η      =  let fun check' [] ((x, _) :: η') = K(η')
                                          | check' (a :: U) ((x, _) :: η')
                                            = check (pre, check' U) ((x, SOME a) :: η')
                                   in check' 𝒰 ((x, NONE) :: η)
                                   end
----------------------------------------------------------------------------------------
fun execute (R(x⃗)) η           =  let fun K a⃗ =  if rho.has(R, a⃗)
                                                  then ()
                                                  else (rho.add(R, a⃗);
                                                        app (fn K' => K' a⃗)
                                                            (infl.consumers R))
                                   in app K (unifiable (η, x⃗))
                                   end
  | execute 1 η                 =  ()
  | execute (cl₁ ∧ cl₂) η       =  execute cl₁ η; execute cl₂ η
  | execute (pre ⇒ cl) η        =  check (pre, execute cl) η
  | execute (∀x : cl) η         =  execute cl ((x, NONE) :: η)
```

Note: *rho* is called *result* in the implementation

Appendix C. Output from the solver

```
The Universe:
(L1, x1, z1, y1, L4, L5, L3, L6, L2, labQ)

Relation INIT/1:
(L1),

Relation FVAR/1:
(y1),  (z1), (x1),

Relation FLOW/2:
(L1, L2),  (L2, L3),  (L5, L3),  (L3, L6),  (L3, L4),  (L4, L5),

Relation RDKILL/3:
(L6, y1, L6),  (L6, y1, L5),  (L6, y1, L1),  (L6, y1, labQ),
(L5, y1, L6),  (L5, y1, L5),  (L5, y1, L1),  (L5, y1, labQ),
(L4, z1, L4),  (L4, z1, L2),  (L4, z1, labQ),
(L2, z1, L4),  (L2, z1, L2),  (L2, z1, labQ),
(L1, y1, L6),  (L1, y1, L5),  (L1, y1, L1),  (L1, y1, labQ),

Relation RDGEN/3:
(L6, y1, L6),  (L5, y1, L5),  (L4, z1, L4),  (L2, z1, L2),  (L1, y1, L1),

Relation RDIN/3:
(L6, z1, L2),  (L6, z1, L4),  (L6, y1, L1),  (L6, y1, L5),  (L6, x1, labQ),
(L5, z1, L4),  (L5, y1, L1),  (L5, y1, L5),  (L5, x1, labQ),
(L4, z1, L2),  (L4, z1, L4),  (L4, y1, L1),  (L4, y1, L5),  (L4, x1, labQ),
(L3, z1, L2),  (L3, z1, L4), (L3, y1, L1), (L3, y1, L5), (L3, x1, labQ),
(L2, y1, L1),  (L2, z1, labQ),  (L2, x1, labQ),
(L1, x1, labQ),  (L1, z1, labQ),  (L1, y1, labQ),

Relation RDOUT/3:
(L6, z1, L2),  (L6, z1, L4),  (L6, y1, L6),  (L6, x1, labQ),  (L5, z1, L4),
(L5, y1, L5),  (L5, x1, labQ),
(L4, z1, L4),  (L4, y1, L1),  (L4, y1, L5),  (L4, x1, labQ),
(L3, z1, L2),  (L3, z1, L4),  (L3, y1, L1),  (L3, y1, L5),  (L3, x1, labQ),
(L2, y1, L1),  (L2, z1, L2),  (L2, x1, labQ),
(L1, y1, L1),  (L1, z1, labQ),  (L1, x1, labQ),
```

Appendix D. Final *result*

```
Stack length: 196
Stack:
SOME [] SOME [] SOME [] SOME [] SOME [] SOME [] SOME [] SOME [] SOME [] SOME [0] SOME []
SOME [] SOME [0] SOME [] SOME [0] SOME [] SOME [4,6,8,0] SOME [3] SOME [] SOME [] SOME [] SOME []
SOME [] SOME [] SOME [] SOME [] SOME [8] SOME [] SOME [7,6,8] SOME [2] SOME [] SOME [8,4] SOME []
SOME [] SOME [8,4] SOME [] SOME [8,4] SOME [] SOME [8,4] SOME [] SOME [] SOME [8,4] SOME []
SOME [4] SOME [] SOME [] SOME [4] SOME [] SOME [4] SOME [] SOME [7,6,5,4] SOME [2] SOME []
SOME [0,5] SOME [] SOME [0,5] SOME [] SOME [0,5] SOME [] SOME [] SOME [0,5] SOME [] SOME [0,5]
SOME [] SOME [] SOME [0,5] SOME [] SOME [5] SOME [] SOME [4,6,5] SOME [3] SOME [] SOME [7]
SOME [] SOME [7] SOME [3] SOME [] SOME [9] SOME [] SOME [9] SOME [] SOME [0] SOME [] SOME [9]
SOME [2,3,1] SOME [] SOME [] SOME [9] SOME [2,3,1] SOME [] SOME [9] SOME [2,3,1] SOME [] SOME []
SOME [9] SOME [2,3,1] SOME [] SOME [9] SOME [2,3,1] SOME [] SOME [] SOME [9] SOME [2,3,1] SOME []
SOME [9] SOME [2,3,1] SOME [] SOME [] SOME [9] SOME [2,3,1] SOME [] SOME [9] SOME [3,2,1] SOME []
SOME [] SOME [9] SOME [3,2,1] SOME [] SOME [9] SOME [3,2,1] SOME [] SOME [7,5,4,6,8,0] SOME [2,1]
SOME [] SOME [9] SOME [] SOME [9] SOME [] SOME [9] SOME [1,2,3] SOME [] SOME [7] SOME [3]
SOME [] SOME [5] SOME [3] SOME [] SOME [4] SOME [2] SOME [] SOME [8] SOME [2] SOME [] SOME [0]
SOME [3] SOME [] SOME [] SOME [] SOME [] SOME [7,5,0,9] SOME [3] SOME [] SOME [] SOME [] SOME []
SOME [7,5,0,9] SOME [3] SOME [] SOME [] SOME [] SOME [4,8,9] SOME [2] SOME [] SOME [] SOME []
SOME [4,8,9] SOME [2] SOME [] SOME [] SOME [] SOME [] SOME [7,5,0,9] SOME [3] SOME [] SOME [8]
SOME [] SOME [6] SOME [] SOME [] SOME [6] SOME [] SOME [7,4] SOME [] SOME [5] SOME [] SOME []
SOME [] SOME [] SOME [0,8,4,5,7,9] SOME [7,5,4,6,8,0] SOME [7,5,4,6,8,0] SOME [7,5,4,8,0] SOME [7,5,4,8,0]
SOME [0,8,5,6,4] SOME [3,2,1] SOME [0]
Buckets length: 448
Buckets:
[] [] [] [(58/4,59)] [(181/0,182)] [(123/7,124)] [] [] [] [(51/3,52)] [] [] [] [] [(116/9,117)] [] [] [] [] [] [] [] [] []
[(167/6,172)] [(167/7,176)] [(167/8,168)] [] [] [] [] [(30/4,33)] [] [] [] [(30/8,32)] [(30/9,31)] [(23/3,24)] [] [] [] [] []
[] [] [] [(16/6,17)(74/0,75)] [] [] [] [(74/4,96)] [(74/5,103)] [(74/6,89)] [(74/7,110)] [(2/0,21)(74/8,82)] [] [] []
[(2/4,12)] [(2/5,16)] [(2/6,14)] [] [(2/8,19)(67/9,68)(125/3,126)] [(60/3,61)] [] [(183/0,184)] [] [] [] [] [] [] [] []
[(46/0,48)(111/1,112)] [(118/9,119)(111/2,164)] [(111/3,123)] [] [] [(46/5,49)] [] [(46/7,50)(104/1,105)] [(104/2,152)]
[(46/9,47)(104/3,128)] [(39/3,40)] [] [] [(169/8,170)] [(97/1,98)] [(97/2,147)] [(97/3,138)] [] [] [] [] [(90/1,91)]
[(90/2,157)] [(90/3,133)] [] [] [] [] [(83/1,84)] [(83/2,169)] [(83/3,186)] [] [] [] [] [(76/1,77)] [(76/2,116)] [(76/3,181)]
[] [] [] [] [(4/0,51)] [] [] [] [(4/4,57)] [(4/5,60)] [] [(4/7,63)] [(4/8,54)(69/9,70)] [] [] [] [] [] [] [(120/3,121)] [] [] [] [] []
[(55/8,56)(178/3,179)] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [(34/2,35)(164/4,165)] [] [] [] [(164/8,177)] [] [] [(157/4,158)]
[] [] [] [(157/8,173)] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [(6/0,76)] [] [] [] [(6/4,97)] [(6/5,104)] [(6/6,90)] [(6/7,111)]
[(71/9,72)(6/8,83)] [] [] [] [] [(64/7,65)] [] [(57/2,58)] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []  [(166/2,167)]
[(108/9,109)] [] [] [] [] [] [] [(101/9,102)] [(29/2,30)(159/4,160)] [] [] [] [(159/8,174)] [] [(94/9,95)] [(152/4,153)] [] []
[] [] [] [(87/9,88)] [(145/4,146)] [(145/5,151)] [(145/6,156)] [(145/7,163)(138/0,193)] [] [(73/1,74)]
[(80/9,81)(73/2,114)] [] [(138/5,139)] [] [] [] [(66/1,71)] [(1/1,9)(66/2,69)] [(1/2,10)(66/3,67)] [(1/3,11)] [] [] [] [] [] []
[] [] [] [(52/0,53)] [] [] [] [] [] [] [] [] [] [(45/3,46)] [] [] [] [] [] [] [] [] [] [] [] [] [(161/4,162)] [] [] [] [(161/8,175)]
[(24/0,26)] [] [(154/4,155)] [] [] [(24/5,27)] [(154/8,171)] [(24/7,28)] [] [(24/9,25)(147/4,148)] [] [] [(140/0,194)] [] []
[] [] [(140/5,141)] [] [(133/0,190)] [] [(3/0,23)] [] [] [(133/5,134)] [(3/4,34)] [(3/5,39)] [] [(3/7,45)] [(3/8,29)]
[(126/4,137)] [(126/5,127)] [(61/5,62)(126/6,132)] [] [] [] [(54/2,55)] [] [] [] [] [] [] [] [] [] [] [] [(40/0,42)]
[(112/9,113)] [] [] [] [(40/5,43)] [] [(40/7,44)] [(105/9,106)] [(40/9,41)] [] [] [] [] [] [(98/9,99)] [] [] [] [] [] []
[(91/9,92)] [(149/4,150)] [] [] [(142/0,195)] [(19/6,20)] [] [(84/9,85)] [] [(142/5,143)] [] [(12/5,13)(135/0,191)] []
[(5/0,66)] [(77/9,78)] [] [(135/5,136)] [(5/4,93)] [(5/5,100)] [(5/6,86)] [(5/7,107)] [(5/8,79)] [(63/3,64)] [(128/5,129)]
[(186/0,187)] [] [] [] [] [] [] [(179/0,180)] [(114/0,115)(121/7,122)] [] [] [(179/4,192)] [] [(179/6,189)] [] [(179/8,185)]
[(107/1,108)] [(107/2,161)] [(107/3,142)] [] [] [] []  [(100/1,101)] [(100/2,149)] [(100/3,140)] [] [(35/4,38)] [] []
[(93/1,94)] [(35/8,37)(93/2,159)] [(35/9,36)(93/3,135)] [] [] [] [] [(86/1,87)] [(86/2,154)] [(86/3,130)] [] [] [] []
[(79/1,80)(144/2,145)] [(21/8,22)(79/2,118)] [(79/3,183)] [] [(14/4,15)] [] []  [(14/7,18)(7/0,178)] [] [] [] [(7/4,144)]
[(7/5,125)(130/0,188)] [] [(0/0,8)(7/7,120)] [(7/8,166)] [(7/9,73)] [(130/5,131)] [] [] []
Number of elements: 188
```

Appendix E. Final *infl*

```
Stack length: 209
Stack:
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE SOME [c,c,c] SOME [c,c,c]
SOME [c,c,c] SOME [c,c,c] SOME [c,c,c,c,c] SOME [c,c,c,c,c] SOME [c,c,c,c] SOME [c,c,c,c] SOME [c,c,c,c]
SOME [c,c,c,c] SOME [c,c,c,c] SOME [c,c,c,c] SOME [c,c,c,c] SOME [c,c,c,c] SOME [c,c,c,c] NONE NONE
NONE NONE NONE NONE NONE SOME [c,c,c,c,c] SOME [c,c,c,c,c] SOME [c,c,c,c] SOME [c,c,c,c]
SOME [c,c,c,c] SOME [c,c,c,c] SOME [c,c,c] SOME [c,c,c] SOME [c,c,c] SOME [c,c,c] SOME [c,c,c]
SOME [c,c,c] SOME [c,c,c] SOME [c,c,c] NONE NONE NONE NONE NONE NONE SOME [c,c,c,c]
SOME [c,c,c,c] SOME [c,c,c,c] SOME [c,c,c,c] NONE NONE NONE NONE NONE NONE SOME [c,c,c,c,c]
SOME [c,c,c,c,c] SOME [c,c,c,c,c] NONE NONE NONE NONE NONE NONE SOME [c,c,c] SOME [c,c,c]
 SOME [c,c,c] SOME [c,c,c] SOME [c,c,c] NONE NONE NONE NONE NONE NONE SOME [c,c,c]
SOME [c,c,c] SOME [c,c,c] SOME [c,c,c] SOME [c,c,c] SOME [c,c,c] SOME [c,c,c] NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE NONE
NONE NONE NONE NONE SOME [c,c,c,c,c,c,c,c,c] SOME [c] SOME [c] NONE NONE SOME [c] SOME [c]
Buckets length: 448
Buckets:
[] [(188/4,189)] [] [] [] [(188/8,196)] [(51/0,52)] [] [] [] [] [] [] [] [(116/9,117)] [(174/4,175)] [(44/3,45)] [] [] []
[(102/1,103)] [(102/2,184)] [(102/3,166)] [] [] [] [] [] [] [] [(160/5,161)] [] [] [] [(23/0,25)] [] [] [] [] [(23/5,26)] []
[(23/7,27)] [] [(23/9,24)] [] [] [] [] [] [] [] [] [(197/0,198)] [] [(132/1,133)] [(2/0,20)(132/2,186)] [(132/3,172)] [] []
[(2/4,11)] [(2/5,15)] [(2/6,13)] [(2/7,138)] [(2/8,18)] [] [] [(60/5,61)] [(190/8,191)] [] [] [(53/2,54)] [] [] [] [] [] []
[(176/4,177)] [] [] [] [] [(39/0,41)] [] [] [] [] [(39/5,42)(162/0,203)] [] [(39/7,43)] [] [(39/9,40)] [(162/5,163)] [] [] [] []
[(97/9,98)] [] [] [] [] [] [] [(18/1,88)] [(18/2,89)] [(18/3,90)] [(18/4,91)] [(18/5,92)] [(18/6,19)] [(18/7,150)]
[(83/9,84)(11/1,108)(18/8,151)] [(11/2,109)(18/9,152)] [(11/3,110)] [(11/4,111)(199/0,200)] [(11/5,12)] [(11/6,126)]
[(4/0,50)(11/7,127)] [(11/8,128)] [(11/9,129)] [] [(4/4,56)] [(4/5,59)] [] [(4/7,62)] [(4/8,53)] [(62/3,63)] [] [] [] [] [] []
[] [] [] [] [] [] [(178/4,179)] [] [] [] [] [] [(113/9,114)] [] [] [] [(164/0,204)] [] [] [(106/9,107)] [] [(164/5,165)]
[(34/4,37)] [] [] [] [(34/8,36)] [(34/9,35)] [] [] [] [] [(85/1,86)] [(20/1,75)(85/2,190)] [(20/2,76)(85/3,201)] [(20/3,77)]
[(20/4,78)] [(20/5,79)] [(20/6,80)] [(20/7,81)] [(20/8,21)(13/1,99)] [(13/2,100)(20/9,153)] [(13/3,101)]
[(13/4,14)(201/0,202)] [(13/5,130)] [(13/6,131)] [(13/7,17)(6/0,72)] [(13/8,148)] [(13/9,149)] [] [(6/4,105)]
[(6/5,115)] [(6/6,96)] [(6/7,135)(136/9,137)] [(6/8,85)] [] [] [] [] [] [] [] [] [(57/4,58)] [] [] [(115/1,116)] [(115/2,178)]
[(115/3,160)(180/4,181)] [(50/3,51)] [] [] [(180/8,192)] [] [] [] [] [] [(166/0,205)] [] [] [] [] [(166/5,167)] [] [] [] [] [] []
[] [(29/4,32)] [] [] [] [(29/8,31)(94/9,95)] [(29/9,30)] [(22/3,23)] [] [] [] [] [(15/1,118)] [(15/2,119)] [(15/3,120)]
[(15/4,121)] [(15/5,122)] [(15/6,16)(138/1,139)] [(15/7,123)(138/2,140)] [(15/8,124)(138/3,141)]
[(15/9,125)(138/4,142)] [(138/5,143)] [(138/6,144)] [(138/7,145)] [(138/8,146)] [(138/9,147)] [(1/1,8)(73/9,74)]
[(1/2,9)] [(1/3,10)] [] [] [] [] [(66/9,67)] [(59/3,60)] [] [] [] [] [] [(182/4,183)] [] [] [] [(182/8,193)] [(45/0,47)] [] [] []
[] [(45/5,48)(168/0,206)] [] [(45/7,49)] [] [(45/9,46)] [(38/3,39)(168/5,169)] [] [] [] [(96/1,97)]
[(103/9,104)(96/2,182)] [(96/3,164)] [] [] [] [] [] [] [] [] [] [] [] [(82/1,83)(154/9,155)] [(82/2,156)] [(82/3,199)] [] [] []
[] [] [] [] [] [] [] [] [(3/0,22)] [] [] [] [(3/4,33)] [(3/5,38)] [] [(3/7,44)(133/9,134)] [(3/8,28)(68/9,69)] [] [] [] [] [] []
[(184/4,185)] [] [] [] [(184/8,194)] [(112/1,113)] [(54/8,55)(112/2,176)] [(112/3,170)] [] [] [(170/0,207)] []
[(105/1,106)] [(105/2,174)] [(105/3,168)] [(170/5,171)] [] [] [] [] [] [(33/2,34)] [] [] [] [] [] [] [] [] [] [] [] [(156/9,157)]
[] [] [] [] [] [] [] [] [] []  [] [] [(135/1,136)] [(5/0,65)(135/2,188)] [(135/3,158)] [] [] [(5/4,102)] [(5/5,112)] [(5/6,93)]
[(5/7,132)] [(70/9,71)(5/8,82)] [] [] [] [] [(63/7,64)] [] [(56/2,57)(186/4,187)] [] [] [] [(186/8,195)] [] [] [] [] []
[(172/0,208)] [] [] [] [] [(172/5,173)] [] [] [] [] [] [] [] [] [] [] [(93/1,94)] [(93/2,180)] [(28/2,29)(93/3,162)] [] []
[(158/7,159)] [] [] [] [] [] [] [] [] [] [(86/9,87)] [] [] [] [] [] [(72/1,73)] [(72/2,154)] [(72/3,197)] [] [] [] []
[(0/0,7)(65/1,70)] [(65/2,68)] [(65/3,66)] [] [] [] []
Number of elements: 202
```