Extended Features in the Succinct Solver (V2.0)

Authors	: Hongyan Sun, Hanne Riis Nielson and Flemming Nielson
Date	: September 29, 2003
Number	: SECSAFE-IMM-009-1.0
Classification	: Public

Abstract

We describe the new features extended in the Succinct Solver (V2.0) and the techniques developed for supporting these features in this report. The Succinct Solver developed by Nielson and Seidl incorporates state-of-the-art approaches to constraint solving. It is used to solve static analysis problems specified in Alternation-free Least Fixpoint Logic (ALFP). The major new feature in the version V2.0 is to allow using structured terms freely, hence the universe is potentially infinite. To do so we first transform the clause containing structured terms into the equivalent clause with only simple terms as arguments of predicates, via explicit unifications, in the preprocessing stage. In the solving stage we expand the universe dynamically, and we use continuations to resolve the computations that were suspended due to the lack of information about the whole universe. Once a new ground term is added to the universe, the continuations are resumed. Moreover, we extend the solver to allow querying old results while solving new clauses, and provide more flexible user interfaces.

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. The Succinct Solver [2] is developed as such a constraint solver.

The Succinct Solver uses the Alternation-free Least Fixpoint 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].

One thing inconvenient in the Succinct Solver (V1.0) is that it is restricted in using terms, since its solving mechanism is based on the fixed finite universe. Thus essentially, only simple terms[1] are allowed as arguments of predicates. In this report, we document our new extended features in the Succinct Solver

[1] We say a term is a *simple term* if it is either a ground term or a variable.

(V2.0)[2], one major feature of which is to relax the restriction (in V1.0) on using structured terms, hence the universe is potentially infinite. To do so we first transform the clause containing structured terms into the equivalent clause with only simple terms as the arguments of predicates in the preprocessing stage. The transformation is based on explicit unifications, with the similar idea of equational unifications [6, 7] used in the areas of theorem proving and term rewriting [8, 9]. In the solving stage we expand the universe dynamically, and we use continuations to resolve the computations that were suspended due to the lack of information about the whole universe. Once a new ground term is added to the universe, the continuations are resumed. In addition, we extends the solver to allow querying old results while solving new clauses, and provide more flexible user interfaces etc.

The remainder of the report is organized as follows: in Section 2, we give the syntax and the semantics of the ALFP logic that is used by the Succinct Solver (V2.0) as the constraint specification language. In Section 3, we introduce our approach to handling the structured terms, with a brief review of the theoretical foundations. Section 4 describes the data representation in the solver and gives an overview of the solver by sketching the system structure and data structures. We describe and explain, in Section 5, the algorithms and the implementation for the extended features. In Section 6, we describe the user interfaces provided by the Succinct Solver (V2.0). Section 7 concludes the report.

2 ALFP Clauses

The specification language used in the Succinct Solver is the alternation-free fragment of Least Fixpoint Logic (ALFP), which is an extension of Horn Clauses. In this section we give a brief introduction to ALFP clauses in terms of the syntax and the semantics.

2.1 Syntax

Given a fixed countable set $\mathcal{X}$ of variables, a countable set $\mathcal{C}$ of constant symbols, a finite ranked alphabet $\mathcal{R}$ of predicate symbols, and a finite set $\mathcal{F}$ of function symbols, the set of ALFP clauses, cl, is generated by the following grammar

$$
\begin{array}{lcl}
t & ::= & c \quad | \quad x \quad | \quad f\,(t_1, \cdots, t_k) \\
pre & ::= & R\,(t_1, \cdots, t_k) \quad | \quad \neg R\,(t_1, \cdots, t_k) \quad | \quad pre_1 \wedge pre_2 \\
 & | & pre_1 \vee pre_2 \quad | \quad \exists x : pre \quad | \quad t_1 = t_2 \quad | \quad t_1 \neq t_2 \\
 & | & \forall x : pre \\
cl & ::= & R\,(t_1, \cdots, t_k) \quad | \quad \mathbf{1} \quad | \quad cl_1 \wedge\ cl_2 \quad | \quad \forall x : cl \\
 & | & pre \Rightarrow cl \quad | \quad pre \Longrightarrow cl
\end{array}
$$

where t is called a term which is generated either by a constant $c \in \mathcal{C}$, a variable $x \in \mathcal{X}$ or applying a function $f \in \mathcal{F}$ over $\mathcal{C}$ and $\mathcal{X}$. $R \in \mathcal{R}$ is a k-ary predicate symbol for $k \geq 1$, and $t_1, \cdots, t_k$ denote arbitrary terms. $\mathbf{1}$ is the always true clause. Occurrences of $R(\cdots)$ and $\neg R(\cdots)$ in preconditions are also called *queries* and *negative queries*, respectively, whereas the other occurrences are

[2]This is in cooperation with Prof. Hemult Seidl at the University of Trier.

called *assertions* of the predicate R. In literatures $R(\cdots)$ is often called an atomic formula.

It needs to mention that "=" and "$\neq$" are pre-defined predicate symbols as infix operators for *equality* (=, also called explicit unification here) and *inequality* ($\neq$) respectively.

One additional extension of the syntax in the Succinct Solver (V2.0) is the clause $pre \Longrightarrow cl$ which makes a *breakpoint* in front of clause cl and is only used for the debugging purpose. In the solver we use a counter, namely an integer variable *cnt*, to count the number of the environments (represented by *env*) passing through the *breakpoint* of clause cl. We will discuss how to use this clause in Section 6.

Stratified ALFP. In order to deal with negations conveniently, we restrict ourselves to *alternation-free* formulae. We introduce a notion of stratification similar to the one which is known from *Datalog* and logic programming [10, 11].

Definition 1 *A clause cl is stratified 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 $j = 0, \cdots, k$, 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.*

The Succinct Solver uses the stratified ALFP as the constraint specification language. In the version V2.0, the universal quantifier for preconditions is not available any longer due to the fact that the potentially infinite universe is dynamically expanded.

2.2 Semantics

Let $\mathcal{U}$ denote an infinite universe of ground terms[3] formed only from $c \in \mathcal{C}$ and $f \in \mathcal{F}$, and $\mathcal{U}^*$ denote a subset of $\mathcal{U}$. Given interpretations ρ and σ for predicate symbols and terms, respectively, we define the satisfaction relations

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

for preconditions and clauses as in Table 1. Here we write $\rho(R)$ for the set of k-tuples $(a_1, \cdots, a_k)$ from $\mathcal{U}^k$ associated with the k-ary predicate R, and we write $\sigma(t)$ for the ground term of $\mathcal{U}$ bound to t with $\sigma(c) = c$ for the constants, and $\sigma(f(t_1, ..., t_k)) = f(\sigma(t_1), ..., \sigma(t_k))$ for the functional terms. Further $\sigma[x \mapsto a]$ stands for the mapping that is as σ except that x is mapped to a, and as a particular case $\sigma[cnt \mapsto 1^+]$ stands for the mapping that is as σ except that the value of the integer variable *cnt* is increased by 1. We assume that $\mathcal{U}^* \supseteq \Im_{\rho(R)}$ for all $R \in \mathcal{R}$ given that $\Im_{\rho(R)} = \{a_1, ..., a_k : (a_1, ..., a_k) \in \rho(R)\}$.

We view also the free variables occurring in a formula as ground terms from the universe $\mathcal{U}$. Thus, given an interpretation σ_0 of the ground terms, 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] A *ground term* is a term not containing variables.

$(\rho,\sigma) \models R\,(t_1,\cdots,t_k)$	iff	$(\sigma(t_1),\cdots,\sigma(t_k)) \in \rho(R)$
$(\rho,\sigma) \models \neg R\,(t_1,\cdots,t_k)$	iff	$(\sigma(t_1),\cdots,\sigma(t_k)) \notin \rho(R)$
$(\rho,\sigma) \models pre_1 \wedge pre_2$	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 $a \in \mathcal{U}^*$
$(\rho,\sigma) \models t_1 = t_2$	iff	$\sigma(t_1) = \sigma(t_2)$
$(\rho,\sigma) \models t_1 \neq t_2$	iff	$\sigma(t_1) \neq \sigma(t_2)$
$(\rho,\sigma) \models R\,(t_1,\cdots,t_k)$	iff	$(\sigma(t_1),\cdots,\sigma(t_k)) \in \rho(R)$
$(\rho,\sigma) \models \mathbf{1}$	iff	true
$(\rho,\sigma) \models cl_1 \wedge cl_2$	iff	$(\rho,\sigma) \models cl_1$ and $(\rho,\sigma) \models cl_2$
$(\rho,\sigma) \models \forall x : cl$	iff	$(\rho,\sigma[x \mapsto a]) \models cl$ for all $a \in \mathcal{U}^*$
$(\rho,\sigma) \models pre \Rightarrow cl$	iff	$(\rho,\sigma) \models cl$ whenever $(\rho,\sigma) \models pre$
$(\rho,\sigma) \models pre \Longrightarrow cl$	iff	$(\rho,\sigma[cnt \mapsto 1^+]) \models cl$ whenever $(\rho,\sigma) \models pre$

Table 1: Semantics of preconditions and clauses

It needs to mention that the solver only terminates with a least solution iff the least solution is finite.

Example 1 *Let Nat be a 1-ary predicate defining a natural number. The following formula defines all the natural numbers:*

$$Nat(zero) \wedge \forall x : (Nat(x) \Rightarrow Nat(succ(x)))$$

where function succ *computes the successor of a natural number. The least solution to this formula is the infinite set $\mathbb{N}$ of natural numbers. In this case the solver will not terminate.* □

Existence of the least solution. Let Δ be the set of interpretations ρ of predicate symbols in $\mathcal{R}$ over $\mathcal{U}^*$ (we can also use $\mathcal{U}$ here, but we are more interested in $\mathcal{U}^*$), then $\Delta = (\Delta, \sqsubseteq)$ forms a complete lattice, where the lexicographical ordering $\sqsubseteq$ is defined by $\rho_1 \sqsubseteq \rho_2$ if and only if there is some $0 \leq j \leq N$ such that the following properties hold:

- $\rho_1(R) = \rho_2(R)$ for all $R \in \mathcal{R}$ with $rank(R) < j$
- $\rho_1(R) \subseteq \rho_2(R)$ for all $R \in \mathcal{R}$ with $rank(R) = j$
- either $j = N$ or $\rho_1(R) \subset \rho_2(R)$ for at least one $R \in \mathcal{R}$ with $rank(R) = j$

Proposition 1 *Assume the cl is a stratified ALFP formula and σ_0 is an interpretation of the free variables in cl. Then the set $\Delta' \subseteq \Delta$ containing all ρ such that $(\rho,\sigma_0) \models cl$ forms a Moore family, i.e. it is closed under greatest lower bounds.*

The proof can be found in [2].

3 Essentials for Handling Structured Terms

We recall the notions of *substitution* and *unification* that are fundamentals for handling the structured terms in the Succinct Solver (V2.0). We then develop the transformation that transforms a clause with structured terms into an equivalent clause with only simple terms.

3.1 Basis

Definition 2 (substitution) *A substitution θ is a finite set of the form $\theta = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$ where $x_1, \cdots, x_n$ are distinct variables and $t_1, \cdots, t_n$ are terms such that $t_i \neq x_i$. θ is called a ground substitution if t_i is a ground term for all $i \in \{1, \cdots, n\}$.*

If A is a term or an atomic formula, and $\theta = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$ is a substitution, then $A\theta$ is an *instance* of A by θ obtained from A by simultaneously replacing each occurrence of the variables x_i in A by the terms t_i for $i \in \{1, \cdots, n\}$.

Example 2 *Given A = f(a, g(x)), and $\theta = \{x \mapsto b\}$, we then have $A\theta = f(a, g(b))$.* □

Definition 3 (composition of substitutions) *Let θ and γ be two substitutions, i. e. $\theta = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$ and $\gamma = \{y_1 \mapsto s_1, \cdots, y_m \mapsto s_m\}$. Then the composition of θ and γ denoted by $\theta\gamma$ is the substitution obtained from the set $\{(x_1 \mapsto t_1)\gamma, \cdots, (x_n \mapsto t_n)\gamma, y_1 \mapsto s_1, \cdots, y_m \mapsto s_m\}$ by deleting any binding $(x_1 \mapsto t_1)\gamma$ for which $x_i = t_i\gamma$ and deleting any binding $y_j \mapsto s_j$ for which $y_j \in \{x_1, \cdots, x_n\}$.*

It is known that the composition of substitutions is associative, i.e. $(\theta\gamma)\varsigma = \theta(\gamma\varsigma)$, where θ, γ, and ς are substitutions [12].

Definition 4 (unification) *A unification problem is a finite set of equations $S = \{s_1 = t_1, \cdots, s_k = t_k\}$. A unifier or a solution of S is a substitution γ such that $\gamma s_i = \gamma t_i$. A most general unifier (mgu) is a unifier γ such that any other unifier θ can be obtained from γ by a further substitution ς such that*

$$\theta = \gamma\varsigma$$

We say a unification is *explicit unification* if we explicitly write $t = s$ for t and s are both terms, or both atomic formulas.

Clearly, two atomic formulae are unifiable only if they have the same predicate symbol (with the same arity), and their arguments (i.e. terms) are unifiable [13].

3.2 Transformation via unification

The main idea of handling structured terms in the Succinct Solver (V2.0) is to transform a clause with structured terms into the one with only simple terms as the arguments of the predicates via explicit unifications between the auxiliary variables and the structured terms. This can be illustrated by an example as follows.

$\aleph(R\,(t_1,\cdots,t_k))$	$=$	$\forall y_1,\cdots,y_k : \bigwedge_{i\in\{1,\cdots,k\}} \vartheta(y_i,t_i) \Rightarrow R\,(y_1,\cdots,y_k)$
$\aleph(\mathbf{1})$	$=$	$\mathbf{1}$
$\aleph(cl_1 \wedge cl_2)$	$=$	$\aleph(cl_1) \wedge \aleph(cl_2)$
$\aleph(\forall x : cl)$	$=$	$\forall x : \aleph(cl)$
$\aleph(pre \Rightarrow cl)$	$=$	$\aleph'(pre) \Rightarrow \aleph(cl)$
$\aleph(pre \Longrightarrow cl)$	$=$	$\aleph'(pre) \Longrightarrow \aleph(cl)$
$\aleph'(R\,(t_1,\cdots,t_k))$	$=$	$\exists y_1,\cdots,y_k : R\,(y_1,\cdots,y_k) \bigwedge_{i\in\{1,\cdots,k\}} \vartheta(y_i,t_i)$
$\aleph'(\neg R\,(t_1,\cdots,t_k))$	$=$	$\exists y_1,\cdots,y_k : \neg R\,(y_1,\cdots,y_k) \bigwedge_{i\in\{1,\cdots,k\}} \vartheta(y_i,t_i)$
$\aleph'(pre_1 \wedge pre_2)$	$=$	$\aleph'(pre_1) \wedge \aleph'(pre_2)$
$\aleph'(pre_1 \vee pre_2)$	$=$	$\aleph'(pre_1) \vee \aleph'(pre_2)$
$\aleph'(\exists x : pre)$	$=$	$\exists x : \aleph'(pre)$
$\aleph'(t_1 = t_2)$	$=$	$\exists y_1, y_2 : (y_1 = y_2) \wedge \vartheta(y_1,t_1) \wedge \vartheta(y_2,t_2)$
$\aleph'(t_1 \neq t_2)$	$=$	$\exists y_1, y_2 : (y_1 \neq y_2) \wedge \vartheta(y_1,t_1) \wedge \vartheta(y_2,t_2)$
$\vartheta(y,a)$	$=$	$y = a$
$\vartheta(y,x)$	$=$	$y = x$
$\vartheta(y,f(t_1,\cdots,t_k))$	$=$	$\exists y_1,\cdots,y_k : y = f(y_1,\cdots,y_k) \bigwedge_{i\in\{1,\cdots,k\}} \vartheta(y_i,t_i)$

Table 2: The transformation function $\aleph$

Example 3 *The following clause*

$$pre \Rightarrow R(a,x,f(x))$$

is transformed into

$$pre \Rightarrow \forall y : y = f(x) \Rightarrow R(a,x,y)$$

where y is an auxiliary variable, and it is explicitly *unified with $f(x)$.* □

The transformation is carried out by the function $\aleph$ as defined below.

Definition 5 *We define in Table 2 the function $\aleph$: $\mathcal{CL} \rightarrow \mathcal{CL}$, where $\mathcal{CL}$ is the set of all ALFP clauses.*

The function $\aleph'$ deals with preconditions and ϑ deals with terms. Clearly, ϑ unifies a variable with a term, and if a term is in a nested structure, it is then simplified by the further unifications with the nested terms respectively. The variables $y_1 \cdots y_k$ are distinct auxiliary variables.

Proposition 2 *Given a clause cl, then $\aleph(cl) \Leftrightarrow cl$.*

Proof. It can be proceed by structured induction on *cl*. We show briefly here the case of $R(t_1,\cdots,t_k)$. For all $y_1 \cdots, y_k$, if $\bigwedge_{i\in\{1,\cdots,k\}} \vartheta(y_i,t_i)$ holds, then $\sigma(y_i) = \sigma(t_i)$ for $i \in \{1,\cdots,k\}$. It is then clear whenever $(\sigma(t_1),\cdots,\sigma(t_k)) \in \rho(R)$, we have $(\sigma(y_1),\cdots,\sigma(y_k)) \in \rho(R)$, i.e. $R(t_1,\cdots,t_k) \Leftrightarrow \aleph(R(t_1,\cdots,t_k))$. □

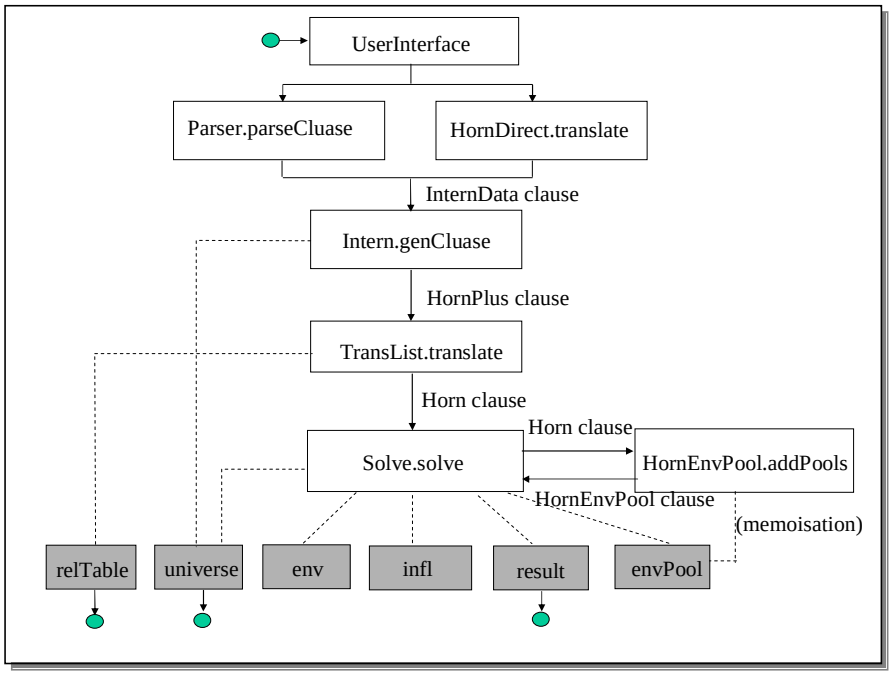

Figure 1: The system structure

4 Data Representation

In this section, we first give an overview of the system structure and briefly introduce the functionality of each component of the system. We then describe the main data structures that are used in the solver to construct the least solution to the given clauses.

4.1 System structure

The system structure of the solver is sketched in Fig. 1. In the figure, the *User-Interface* is composed of a set of functions defined in the functor `FrontBackEnd`. By means of these functions a user can access the solver. We will in Section 6 give a detailed description of each of those functions. The function `parseClause` in the module *Parser* parses the ALFP clause from the text file and transforms it into an internal representation, InternData, which will be used to generate HornPlus clause by the function `genClause` in the module *Intern*. Another way to input the ALFP clause is to encode the clause into the SML data structure, and the clause can be translated to InternData clause through the function `translate` in the *HornDirect* module. At the same time, the function `genClause` modifies the *universe* data structure if it encounters ground terms, and transforms structured terms etc. The function `translate` in the *TransList* module translates the HornPlus clause into another internal representation, i.e. Horn clause. At the same time, it extracts the information for predicates into

the data structure *relTable*.

The function `solve` first transforms the 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 five main data structures, i.e. *env* for the partial environment, *result* for the solution, and *infl* for the *consumer* registration, *envPool* for the memoisation and *universe* for expending the universe. The `solve` function manipulates also some FIFO-queues for the purpose of continuations upon adding a new ground term to the *universe*.

4.2 Data structures

The data structures *env*, *result*, *infl* *universe* and *relTable* in the solver are abstracted as SML data types as given in Table 3.

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

type result = univ list option stack * (loc, univ) table

type 'a stack = (loc * 'a array) ref

type 'a table ={buckets : ('a * loc) list array ref,
                hash : 'a -> idx ref,
                count: int ref}

type infl = consumer list option stack * (loc, consumer) table

type consumer = univ list -> unit

type universe = (termName, args list option) stack *
                (termName, args list option) table

type relTable = (relName, ary) table'

type 'a table' ={buckets : ('a * rel) list array ref,
                hash : 'a -> idx ref,
                count: int ref}
```

Table 3: Abstract data types

Where, *var* corresponds to variables, *univ* corresponds to the ground terms in the universe, *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. 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 *termName* specifies the function symbols that are applied to the list of *args* to form a structured term, and *relName* specifies the relation symbols whose arity is given by *ary* while *rel* gives its integer representation.

4.2.1 The *universe* data structure

The data structure *universe* represents a list of ground terms each is expressed by a *unique tree*.

Definition 6 *A unique tree is an ordered tree where each node expresses a ground term. A leaf node is a constant and a non-leaf node is a functional term with function name as a root node and its children are the arguments.*

Example 4 *Two ground terms $f(g(a, h(b)), f(b, a))$ and $f'(a, b, g(a, b))$ are represented by the unique tree (i) and (ii) respectively in Fig. 2.* □

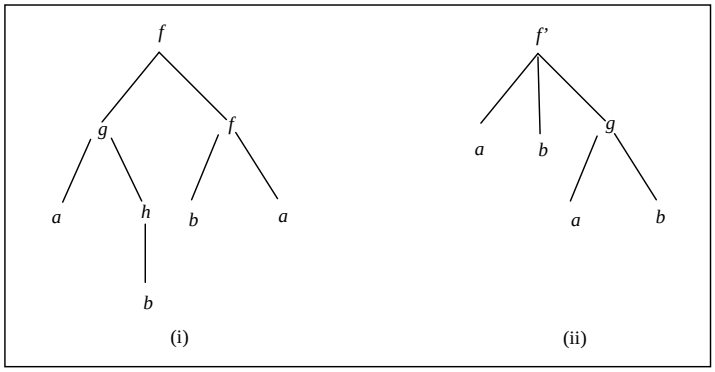

Figure 2: Two examples of unique trees

Fig. 3 illustrates the data structure *universe* which implements a unique tree. In the figure, 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 ground term. The content of the slot is a pair (*name*, *args*), where name is a string, and it can be the name of the constant or the name of the function that constructs a structured term. If it is a structured term, the args expresses a list of arguments in terms SOME $[arg_1, arg_2, ..., arg_n]$, and arg_i is an integer representing a term. If it is a constant term then *args* is NONE. In the case of Fig. 3, a and b are constant terms, $f(a, b)$ and $g(b, h(a))$ are structured terms.

An element in the buckets is also a pair (*term*, *val*) mapping a term to an integer number (which is also the index of the stack). Thus, the argument list of a function can be efficiently constructed.

The buckets constitute a hash table. To resolve the collisions, a slot of a bucket contains a list of elements (c.f. [14]).

4.2.2 The *relTable* data structure

The data structure *relTable* contains information about the relation symbols as illustrated in Fig. 4. It is implemented by a hash table, where each of the

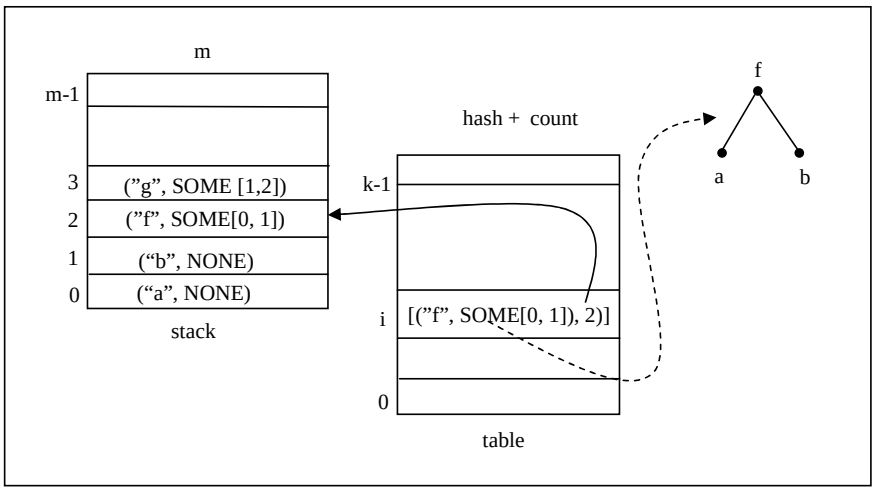

Figure 3: The *universe* data structure

elements in the table is a pair $((name, ary), val)$. The first element in the pair is the relation symbol name and its arity, and the second element in the pair is the integer representation of a relation symbol, which also gives the location of the relation symbol in the *result* data structure (i.e the root node). To make the search efficient we use also an reverse table such that given a val, we can get the corresponding relation symbol and its arity immediately.

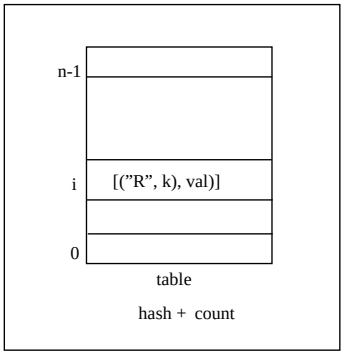

Figure 4: The *relTable* data structure

4.2.3 The *result* data structure

Definition 7 *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*

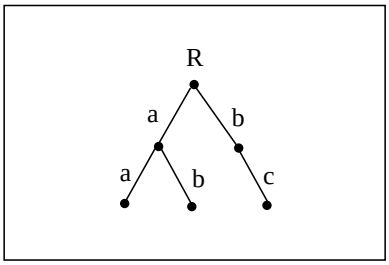

Figure 5: A prefix tree representing a 2-*ary* relation R

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 5 *A 2-ary relation $R = \{(a, a), (a, b), (b, c)\}$ is represented by a prefix tree shown in Fig. 5.* □

The *result* data structure implements a set of prefix trees as illustrated in Fig. 6.

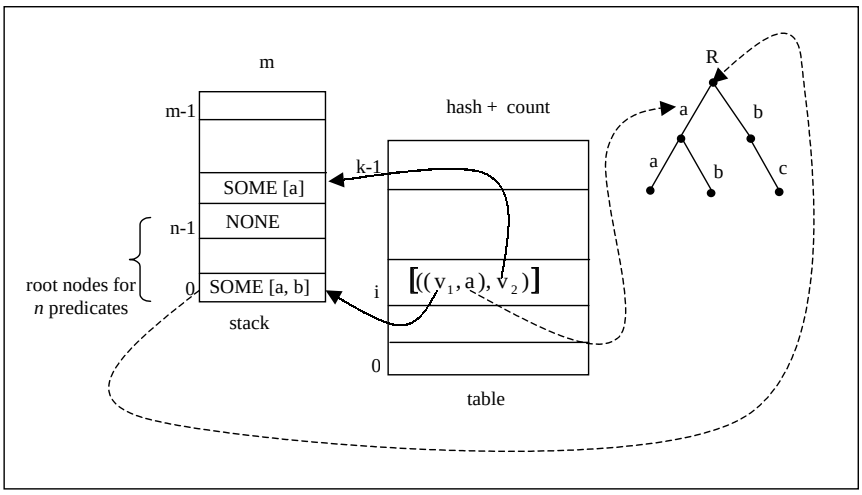

Figure 6: The *result* data structure

In Fig. 6, 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. In version V1.0, the first n slots in the stack correspond to the root nodes of n prefix trees [14]. This holds no longer in version V2.0 since it supports re-run the solver based on the previous *reult*. In stead, the root node is created and indexed dynamically

as the other nodes. 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 $b_1, \cdots, b_i$ respectively. In the case of Fig. 6, 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 a bucket contains a list of elements.

Example 6 *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. 7.* □

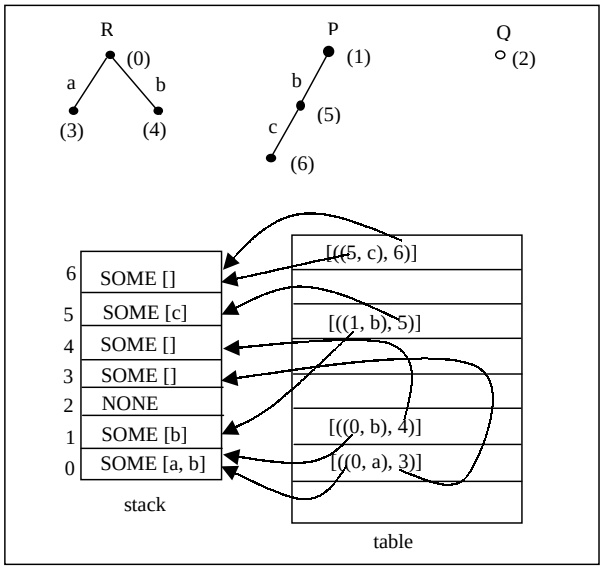

Figure 7: The three prefix trees in *result*

4.2.4 The *infl* data structure

Fig. 8 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. 8, 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.

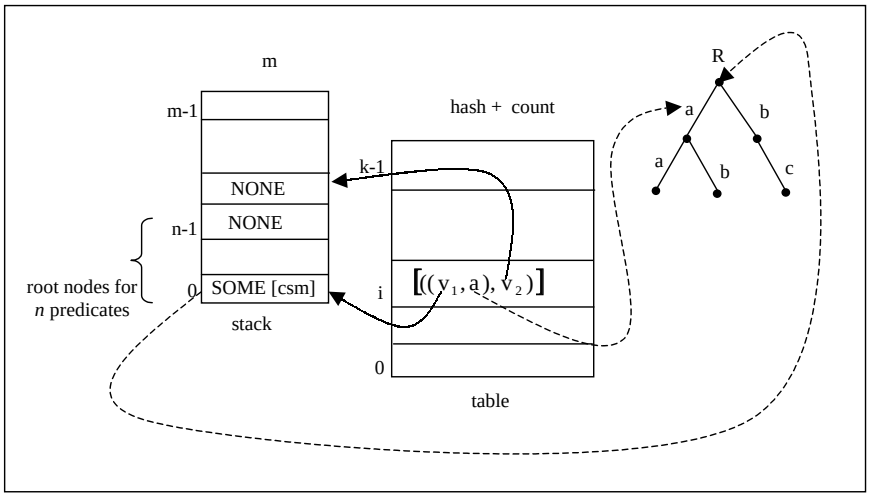

Figure 8: The *infl* data structure

5 Algorithms and Implementation

The algorithms are in fact classified into two major parts: preprocessing and solving. In the preprocessing part, a clause with structured terms is transformed into the clause with only simple terms by implementing the function defined in Table 2. The solving algorithm computes then the least solution to the clauses with only the simple terms.

5.1 Preprocessing

The preprocessing is done by the functions defined in the structure *Intern* which transforms clauses represented by the internal data structure InternData to another data structure called HornPlus. At the same time it implements the transformation function $\aleph$ defined in Table 2. The implementation is shown in Table 4 in SML pseudocode. We adopt the notations used in [2], and explain mainly those related to the extensions in the Succinct Solver (V2.0).

In Table 4, the function `doPre` processes the preconditions, and the function `doClause` processes the clauses. One of the major tasks in the preprocessing is to implement the function $\aleph$ which is facilitated mainly by the function `genTerm` and `wrap`. The function `genTerm` used in the `doPre` is abstracted as follows:

$$\texttt{fun genTerm}(y, f(\vec{x})) = R_f(y :: (\texttt{doArgs}\,\vec{x}))$$

where, R_f is a new predicate which will be described later on.

This function is slightly different from the function `genTerm` used in `doClause`. We discuss the difference through the following example.

```
fun doPre (R(x⃗)) = let
    val pre = R(doArgs genTerm x⃗)
    fun wrap auxPre pre = case auxPre of
          [] -> pre
        | ((y, p) :: rest)-> wrap rest (∃y : (pre ∧ p))
    in wrap auxPre pre  end
  | doPre (¬R(x⃗)) = let
    val pre = ¬R(doArgs genTerm x⃗)
    fun wrap auxPre pre = case auxPre of
          [] -> pre
        | ((y, p) :: rest) -> wrap rest (∃y : (pre ∧ p))
    in wrap auxPre pre  end
  | doPre (pre_1 ∧ pre_2) = doPre pre_1 ∧ doPre pre_2
  | doPre (pre_1 ∧ pre_2) = doPre pre_1 ∨ doPre pre_2
  | doPre (∃x : pre) = ∃x : doPre pre
  |  doPre (t_1 = t_2) = let
    val pre = let  val x = genTerm′ t_1
                   val y = genTerm′ t_2
                   in (x = y)  end
    fun wrap auxPre pre = case auxPre of
          [] -> pre
        | ((y, p) :: rest) -> wrap rest (∃y : (pre ∧ p))
    in wrap auxPre pre  end
  |  doPre (t_1 ≠ t_2) = let
    val pre = let  val x = genTerm′ t_1
                   val y = genTerm′ t_2
                   in (x ≠ y)  end
    fun wrap auxPre pre = case auxPre of
          [] -> pre
        | ((y, p) :: rest) -> wrap rest (∃y : (pre ∧ p))
    in wrap auxPre pre  end
```

```
fun doClause (R(x⃗)) = let
    fun genTerm (y, f(x⃗)) =
        (FunApp(y, U.applyCons f, x⃗), R_f(y :: doArgs x⃗))
    val conclution = R(doArgs genTerm x⃗)
    fun wrap auxClause conclution = case auxClause of
          [] -> conclution
        | ((y, (pre, ass)) :: rest)->
          ∀y : (pre ⇒ ass ∧ (wrap rest conclution))
    in wrap auxCLause conclution
    end
  | doClause 1 = 1
  | doClause (cl_1 ∧ cl_2) = doClause cl_1 ∧ doClause cl_2
  | doClause (pre ⇒ cl) = doPre pre ⇒ doClause cl
  | doClause (pre ⟹ cl) = doPre pre ⟹ doClause cl
  | doClause (∀x : cl) = ∀x : doClause cl
```

Table 4: The preprocessing algorithm in SML pseudocode

Example 7 *Consider $R(f(x), b)$ occurs as an assertion and a query respectively in the following two clauses. Where, b is a constant, and $f(x)$ is a structured term.*

1. *For the clause*

$$pre \Rightarrow R(f(x), b)$$

it is transformed into

$$\forall y : (\texttt{FunApp}\,(y, \texttt{applyCons}\,fx) \wedge pre \Rightarrow R_f(y, x) \wedge R(y, b))$$

2. *For the clause*

$$R(f(x), b) \Rightarrow cl$$

it is transformed into

$$\exists y : (R(y, b) \wedge R_f(y, x)) \Rightarrow cl$$

□

In the example (1), we introduce a new construct `FunApp` for preconditions, which is actually an representation of an explicit unification, e.g. $y = f(x)$ in this case. In addition, the value of y, i.e. the ground instance of $f(x)$ will be inserted into the universe during the solving process. The predicate $R_f(y, x)$ denotes that y and $f(x)$ are unified, thus when the same structured term occurs in the query, i.e. in the case of example (2), we only need to query $R_f(y, x)$ without repeating the unification again. The `genTerm` then generates a list of the ingredients, e.g. `FunApp(···)` and $R_f(\cdots)$, which will be wrapped up into the corresponding clauses by the `wrap` function in both `doPre` and `doClause` cases. Clearly the introduction of new predicate symbol R_f is mainly for efficiency that the same term does not need to be transformed twice in both queries and assertions.

Similar to function `genTerm` in `doPre`, the function `genTerm'`, in the case of equality or inequality, also generates a list of queries of the form $R_f(y, x)$ for each structured term of the form $f(x)$. If a term t is a variable or a ground term, it returns that term without any transformation. Again, `wrap` function wraps the ingredients into a clause.

The function `doArgs` as abstracted below is used to process all the arguments for both predicates and for functions.

```
fun doArgs (t :: aa) =
    if groundOrVar t then (toH t) :: doArgs aa
    else let val y = Aux.newAux()
             val _ = auxPre := (y, genTerm (y, t)) :: (!auxPre)
    in y :: (doArgs aa)
    end
```

where if a term t is a ground term or a variable, captured by function `groundOrVar` then it is transformed to HornPlus format by the function `toH`. Otherwise, an auxiliary variable y is generated for each structured term and then explicitly unifies with the term by calling the `genTerm` function, which generates a list of ingredients represented by `auxPre` in the case of `doPre` or `auxClause` in the case of `doClause`.

When the preprocessing terminates, it returns the HornPlus clause, which is further translated to the Horn clause and then solved by the `solve` function.

5.2 Solving

The solving is done by the function `solve` in the structure *Solve*, which computes the least solution to the clauses and constructs the solution in the data structure *result* (of Forest.forest type).

The solving algorithm is described in the SML pseudocode as given in Table 5.

The `solve` function is composed of the `check` function for checking preconditions and the `execute` function for computing clauses. It constructs the least solution to the given clause by computations and manipulations of the data structures. In Table 5, η corresponds to the data structure *env*, U corresponds to the data structure *universe*, and `rho` corresponds to the data structure *result*. Operations on `rho` include:

```
rho.add :  predicate * univ list -> unit
rho.has :  predicate * univ list -> bool
rho.sub :  predicate * (univ list) list
```

where `rho.add` $(R, \vec{a})$ adds $R(\vec{a})$ into *result*, and `rho.has` $(R, \vec{a})$ checks whether $R(\vec{a})$ is already in *result* or not, while `rho.sub`R returns all the tuples of ground terms associated with the predicate symbol R, which have a prefix $\vec{a}$.

The operations on *infl* data structure include:

```
infl.register :   predicate * consumer -> unit
infl.consumers :  predicate -> consumer list
```

here the function `infl.register` save the consumer to *infl* which saves the context of computation of the given predicate and gives the information about where to start once the required data are available, and the function `infl.consumers` returns a list of consumers related to the specified predicate that are going to be continued.

The operations on U include:

```
U.applyCons :  funConstruct -> univ -> univ
U.all :        unit -> univ list
```

where `U.applyCons` constructs a new ground term and adds the new ground term into the *universe* data structure, and `U.all` gets a list of all ground terms from the current *universe*.

The operations associated with the *env* data structure include:

```
unify :     env * var list * univ list -> env option
update :    env * var * univ -> env
bind :      env * var list * univ list -> env list
getValue :  env * var -> univ option
each :      env list -> env
```

where the `unify` function modifies the given *env* whenever the given tuple $(x_1, \cdots, x_k)$ of variables are unifiable with the given tuple $(a_1, \cdots, a_k)$ of ground terms from the universe under the *env*. Informally, we say that x_i and a_i (for $i \in \{1, \cdots k\}$ are unifiable under *env* whenever $(x_i \mapsto a_i) \in env$ or $(x_i \mapsto \text{NONE}) \in env$, or $(x_i \mapsto a) \notin env$ (meaning that the *env* does not contain variable x_i).

```
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′ η′ = if rho.has (R, eval (x⃗, η′)) then () else K (η′)
    in (ncs.register(x⃗, η, K′); app K′ (bind (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 (FunApp (y, U.applyCons f, x⃗), K) η = let
    val new = U.applyCons f(eval (x⃗, η))
    in (resumeCsms (new);
        case y of Const i -> if i = new then K (η) else ()
                | Var z -> K (update (η, z, new)))
    end
  | check (x1 = x2, K) η = case unify′(η, x1, x2) of
      SOME η′-> K (η′)
    | NONE-> if getVal (η, x1) = SOME a then ()
             else let fun K′ a = K (update′(η, (x1, x2), (a, a)))
                  in (eqcs.register K′; app K′ U.all)
                  end
  | check (x1 ≠ x2, K) η = case (getValue(η, x1), getValue(η, x2)) of
      (SOME a1, SOME a2) -> if a1 = a2 then ()else K (η)
    | (NONE, NONE) -> if x1 = x2 then ()else let
                        fun K′ u a = let
                            fun f b =  if a = b then ()
                                       else K (update′ (η, (x1, x2), (a, b)))
                            in app f u
                            end
                        in(neqcs.register K′; app (K′ U.all) U.all)
                        end
    | _-> let fun K′ b = if a_{x_i} <> b then K (update (η, x_{3−i}, b)) else ()
          in(neqcs.register K′; app K′ U.all)
          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))
                                 fun K′ η′ = eval (x⃗, each η′)
                              in (rcs.register (R, x⃗, η, K′);
                                  app K (K′ bind (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 (pre ⟹ cl) η   =  check (pre, (cnt := cnt + 1; execute cl)) η
  | execute (∀x : cl) η    =  execute cl ((x, NONE) :: η)
```

Table 5: The solving algorithm in SML pseudocode

If each pair of x_i and a_i are unifiable, then the corresponding variable x_i in *env* is modified with a_i and SOME *env*' is returned, meaning that the current *env* is modified as *env*'. Otherwise, it returns NONE meaning that the modification (or unification) is failed.

The function `update` modifies the given *env* by re-binding the given variable, e.g. x with the given value, e.g. ground term a. The function `bind` assigns to the given list of variables (whose values in *env* are all NONE) the lists of tuples of ground terms formed from the current universe, and then generates a list of new environments.

The function `getValue` returns the values of the given variables in *env* if there is any, and the function `each` is an abstract function that returns each time an *env* until the all the given *env* list are enumerated. The function `unify`' is a variant of `unify`, and the function `update`' is a variant of `update`.

The main strategy for handling the dynamic expansion of the universe is to add a new ground term into the universe whenever checking `FunApp` becomes true, and at the same time to resume all the continuations of computations (abstracted as `resumeCsms` in Table 5) that demand the whole universe information, and that were suspended in the previous computations.

We use four FIFO queues, *ncs*, *eqcs*, *neqcs* and *rcs* to maintain the continuations for negative queries, equalities, in-equalities and assertions respectively. We again use the concept of *consumer* and define four types of *consumers*:

```
type consumerN  =  univ list * env * (env -> unit)
type consumerE  =  univ -> unit
type consumerNE =  int * (univ list -> int -> unit)
type consumerR  =  predicate * var list * env * (env list -> univ list)
```

As an example, we consider the case for checking a negative query $\neg R(\vec{x})$, and describe here how the continuations are registered and resumed once a new ground term is added to the universe.

Example 8 *Consider the clause below, we show step by step how the continuation associated with $\neg R(x)$ is registered and resumed upon a new ground term added to the universe.*

$$R(a) \wedge \forall x : ((\neg R(x) \Rightarrow S(x)) \wedge (R(x) \Rightarrow T(f(x))))$$

1. *When $\neg R(x)$ is first met, we have $env = \{x \mapsto NONE\}$, and $universe = \{a\}$. Therefore, the solver would take each of the ground term in the universe to unify with the variable x, or in another word, construct a set of envs such that each x is bound to a ground term from the universe, and then check $\neg R(x)$ under each env. This is carried out by the function* `K'` *as in Table 5. And if $\neg R(x)$ is true under an env, then such an env will propagated to the continuous computation, i.e. computing the rest of the clauses. Since we do not know the whole universe for the moment, and we can only establish $env = \{x \mapsto a\}$, we then register the function* `K'` *to the consumer queue ncs, together with the current env, and at the same time check $\neg R(x)$ under $env = \{x \mapsto a\}$. Since $R(a)$ has been in the result data structure, i.e. checking $\neg R(a)$ returns false, thus the computation for the first implication clause terminates under the current env.*

2. *For the second implication clause, the starting env is also that* $env = \{x \mapsto NONE\}$. *When checking the precondition* $R(x)$, *the variable* x *is unifiable with a under the current env, thus the precondition becomes true, and the new* $env' = \{x \mapsto a\}$ *will propagated to the assertion* $T(f(x))$, *and at the same time a consumer is registered to the* infl *in case that the current information is not fully known.*

3. *When executing* $T(f(x))$ *under* env', *a new ground term* $f(a)$ *is added to the universe, and* $T(f(a))$ *is added to the result data structure.*

4. *Once the ground term* $f(a)$ *is added to the universe the consumer in ncs is resumed, i.e. checking the negative query under* $env' = \{x \mapsto f(a)\}$, *which is a modification of the registered env. Since* $R(f(a))$ *is not in the result data structure, thus* $\neg R(x)$ *is true, and the* env' *is propagated to the the execution of the assertion* $S(x)$.

5. *Then* $S(f(a))$ *is added to the result data structure, and the computation terminates.*

6. *The solution to the given clause is:* $result = \{R : a, T : f(a), S : f(a)\}$ *and the universe is:* $universe = \{a, f(a)\}$.

□

Remark 1 *We claim without proof that the theories for the complexity analysis developed in [2] still hold, but the size of the input clause is expanded within a bounded number due to the transformation* ℵ.

Concerning the manipulations of the internal data structures during the solving process we consider the clause

$$GT(one, zero) \wedge IsOne(one) \wedge$$
$$\forall x : \forall y : (GT(x, y) \wedge IsOne(x) \Rightarrow GT(f(x), y))$$

where *zero* and *one* are constants. $IsOne$ is unary predicate symbol and GT is a binary predicate symbol, and f is a function symbol. The clause is transformed by the preprocessing into the clause

$$GT(one, zero) \wedge IsOne(one) \wedge$$
$$\forall x : \forall y : ((GT(x, y) \wedge IsOne(x)) \Rightarrow$$
$$\forall z : (FunApp(z, applyConsfx) \Rightarrow f(z, x) \wedge GT(z, y)))$$

We now illustrate how the solver establishes the data structure *result* via dynamically expending *universe* while using and updating *env*.

1) When the preprocessing is done, both *env* and *result* are empty, but the data structure *universe* is as shown in Fig. 9.

2) After the first conjunct $GT(one, zero)$ is processed, the *result* data structure contains one tuple $(one, zero)$ associated with predicate GT. When the second conjunct $IsOne(one)$ is processes, the *result* data structure is as shown in Fig. 10. Since no new ground term is added, the universe remains unchanged. Again *env* is empty.

3) When the universal quantifiers $\forall x$ and $\forall y$ are processed the *env* is modified as $env = \{x \mapsto$ NONE, $y \mapsto$NONE$\}$.

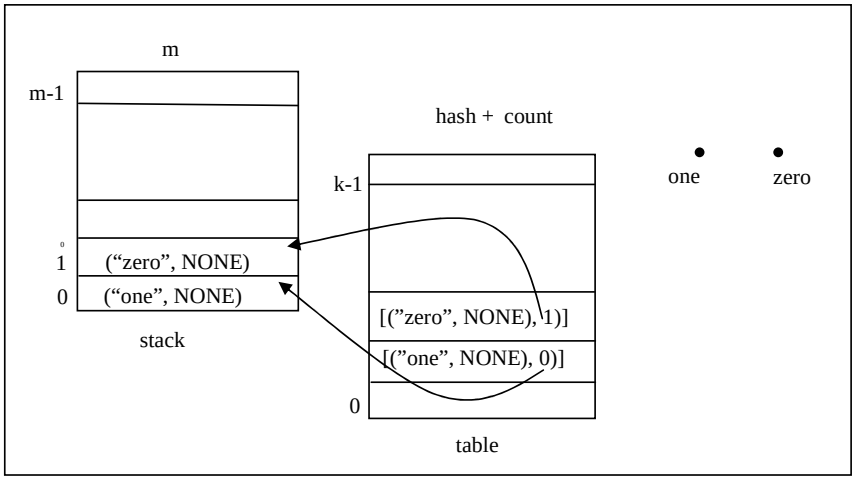

Figure 9: The *universe* data structure (1)

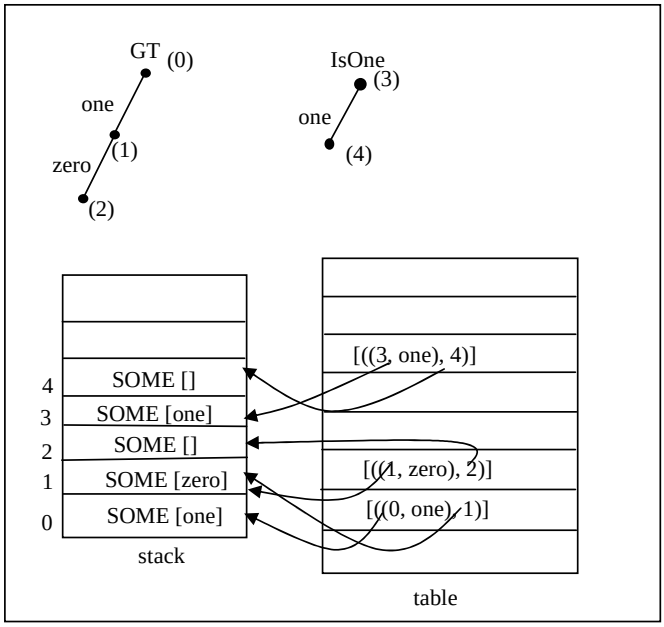

Figure 10: The *result* data structure (1)

4) When the conjunctive precondition $GT(x, y) \wedge IsOne(x)$ is processed, the solver first visits the *result* data structure and unifies $GT(x, y)$ to each of the tuples associated with GT predicate respectively (at the same time adds a consumer in $infl$) to obtain a list of new envs, each reflects one unification. Since now there is only one tuple $(one, zero)$ associated with GT, thus $GT(x, y)$ and $GT(one, zero)$ are unified and new env becomes $env = \{x \mapsto one, y \mapsto zero\}$

which is used for checking $IsOne(x)$. Again there is only one tuple (one) is associated with $IsOne$, and $IsOne(x)$ is unifiable with $IsOne(one)$. The same env will be used for the further computation, i.e. checking the conclusion.

5) When the universal quantification $\forall z$ is met, the env is updated as $env = \{x \mapsto one, y \mapsto zero, z \mapsto \text{NONE}\ \}$.

6) After $FunApp(z, applyCons\ f\ x)$ is checked, the term $f(one)$ is added to the *universe* data structure, as shown in Fig. 11, and the env is updated to $env = \{x \mapsto one, y \mapsto zero, z \mapsto f(one)\}$.

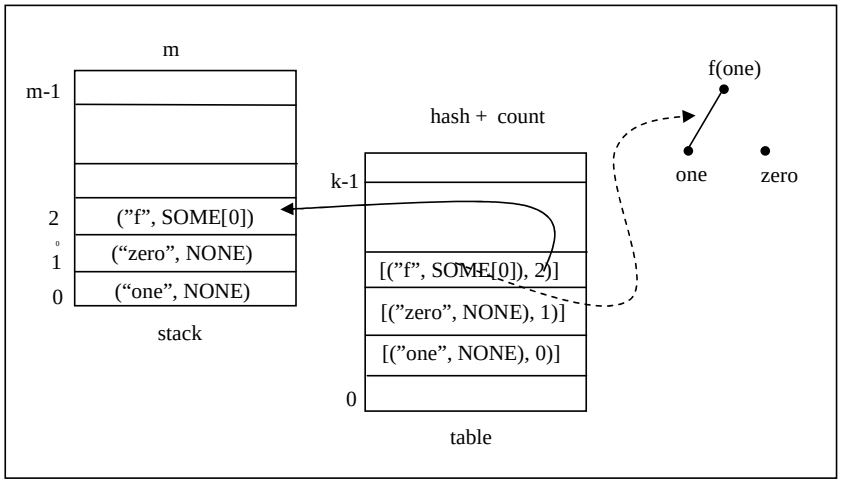

Figure 11: The *universe* data structure (2)

7) The two assertions $f(z, x)$ and $GT(z, y)$ will be instantiated respectively as $f(f(one), one)$ and $GT(f(one), y)$ according to the current bindings to x, y, z. This results in the modification to the *result* data structure which is shown in Fig. 12.

8) After $GT(f(one), zero)$ is added to the *result*, the consumer in $infl$ for GT is resumed, and the computation for checking $GT(x, y)$ is continued, that results in propagating the new $env = \{x \mapsto f(one), y \mapsto zero\}$ to check $IsOne(x)$. Since $IsOne(x)$ is false under the new env, the computation terminates.

5.3 Query the *result* from previous solving

It is often the case that the solution to a clause contains a huge amount of tuples associated with all predicates. It is then hard to read the solution. One way to do it is to formulate the properties into ALFP clauses to query the *result* data structure for verifying the properties.

To enable the access to the result from the previous solution, we implement the data structure *result* and *relTable*, and *universe* as global data structures, which remain unchanged unless an explicit initialization is done by calling the function *init* (c.f. Section 6).

It needs to mention that, it requires that the clauses are stratified, that means previous solved clauses and the clauses to solve are in the different strata.

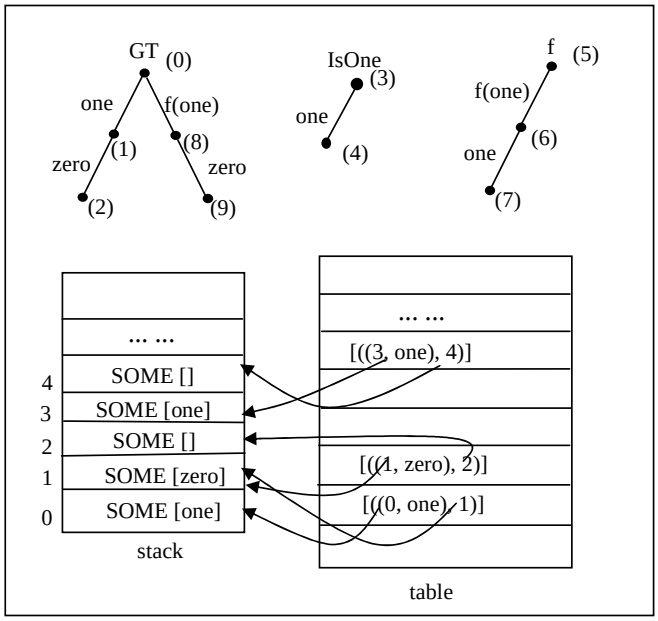

Figure 12: The *result* data structure (2)

Thus, the data structure *infl* for the consumer remains local is sufficient. In [14], we have describe the *infl* which has the same structure as *result*, but each nodes records the consumer that is going to be resumed late on when the new tuples associated with the corresponding predicate symbol is added.

Example 9 *If we continues with the previous clause, and query with the clause*

$$\forall x : \forall y : GT(f(x), y) \Rightarrow OK(x)$$

1. *It is transformed in the pre-precessing as*

$$\forall x : \forall y : (\exists z : f(z, x) \wedge GT(z, y)) \Rightarrow OK(x))$$

2. *When the solving starts, the data structures result and universe remain the same as shown in Fig. 11 and Fig. 12, but the env is empty.*

3. *After the first two quantifications are processed, the env becomes* $env = \{x \mapsto NONE, y \mapsto NONE\}$.

4. *After the precondition is checked, the exists quantifier makes the env changes first to* $env = \{x \mapsto NONE, y \mapsto NONE, z \mapsto NONE\}$, *and after both conjuncts in the precondition are checked, it becomes* $env = \{x \mapsto one, y \mapsto zero, z \mapsto f(one)\}$.

5. *The assertion* $OK(x)$ *will be processed under* $env = \{x \mapsto one, y \mapsto zero\}$ *since variable z is only "local" to the precondition. Then* $OK(one)$ *is added to the result data structure. We know then that* $GT(f(one), zero)$ *is the only one that use the function symbol* f *in the arguments.*

□

6 User Interfaces

The structure `FormulaAnalyzer` described in the file *formulaAnalyzer.sml* in the **Formulas** directory is an application of the functor `FrontBackEnd` which defines the functions for users to access the solver. The functor `FrontBackEnd` is defined in the file *frontBackEnd.sml* in the same directory. The user interface provided by `FrontBackEnd` is summarized in Table 6.

The functions given in the Table 6 can be classified into four groups including the function for initialization, functions for input, functions for solving clauses, and functions for output. The only one initialization function `init` is used to initialize all the global data structures which are used by the solver to construct the result. These data structures should be initialized before calling the functions for solving clauses. In the following subsections, we shall describe the functions of the other three groups.

6.1 Functions for input

There are four ways to input ALFP clauses to the solver:

- input from SML data structures
- input from a text file
- input from stdIn (standard input)
- input from a string

◇ `inputData` *clause*
The function `inputData` receives the ALFP *clause* represented by the SML data structure defined in the structure `HornDirect`. It then transforms the clause into the internal data structure that the solver uses to process the clause.

The structure `HornDirect`, in the file *hornDirect.sml* under the directory **Formulas** defines the following SML data structures to express ALFP clauses:

```
datatype term   =    Cons of string
                |    Var of string
                |    AppF of string * term list

datatype pre    =    Prel of string * term list
                |    Nrel of string * term list
                |    Pconj of pre list
                |    Pdisj of pre list
                |    Exists of string list * pre
                |    Eq of term * term
                |    Neq of term * term

datatype clause =    One
                |    Rel of string * term list
                |    Impl of pre * clause
                |    BImpl of pre * clause
                |    Conj of clause list
                |    Forall of string list * clause
```

```
val init : unit -> Forest.forest

val inputData : HornDirect.clause -> HornPlus.clause list
val inputFile : string -> HornPlus.clause list
val inputStd  : unit -> HornPlus.clause list
val inputStr  : string -> HornPlus.clause list
val fromALFPtoHD : string -> HornDirect.clause list

val outputData : Forest.forest ->
                  (StringItem.item * IntItem.item) list * bool
                     -> string list * (string * string list list) list
val outputFile : Forest.forest * string ->
                  (StringItem.item * IntItem.item) list * bool -> unit
val outputStd : Forest.forest ->
                  (StringItem.item * IntItem.item) list * bool -> unit
val outputStr : Forest.forest ->
                  (StringItem.item * IntItem.item) list * bool -> string

val solve : Forest.forest * HornPlus.clause list -> unit
val solveCount : Forest.forest * HornPlus.clause list -> unit

val solveData : Forest.forest * HornPlus.clause list
                   -> string list * (string * string list list) list
val solveFile : Forest.forest * HornPlus.clause list * string -> unit
val solveStd : Forest.forest * HornPlus.clause list -> unit
val solveStr : Forest.forest * HornPlus.clause list -> string
val selectSolveData : Forest.forest * HornPlus.clause list ->
                       (StringItem.item * IntItem.item) list * bool * bool
                         -> string list * (string * string list list) list
val selectSolveFile : Forest.forest * HornPlus.clause list * string ->
                       (StringItem.item * IntItem.item) list * bool * bool
                         -> unit
val selectSolveStd : Forest.forest * HornPlus.clause list ->
                      (StringItem.item * IntItem.item) list * bool * bool
                         -> unit
val selectSolveStr : Forest.forest * HornPlus.clause list ->
                      (StringItem.item * IntItem.item) list * bool * bool
                         -> string
```

Table 6: Interface provided by `FrontBackEnd`

This corresponds to the following syntax:

$$
\begin{array}{lcl}
t & ::= & a \quad | \quad x \quad | \quad f\,(t_1, \cdots, t_k) \\
pre & ::= & R\,(t_1, \cdots, t_k) \quad | \quad \neg R\,(t_1, \cdots, t_k) \quad | \quad pre_1 \wedge pre_2 \\
 & | & pre_1 \vee pre_2 \quad | \quad \exists x_1, \cdots, x_k : pre \quad | \quad t_1 = t_2 \quad | \quad t_1 \neq t_2 \\
cl & ::= & R\,(t_1, \cdots, t_k) \quad | \quad \mathbf{1} \quad | \quad cl_1 \wedge\ cl_2 \\
 & | & pre \Rightarrow cl \quad | \quad pre \Longrightarrow cl \quad | \quad \forall x_1, \cdots, x_k : cl
\end{array}
$$

This syntax is slightly different from the one described in section 1.1 mainly on that it allows to write a quantifier over a sequence of variables. For example, it allows to write $\forall x, y, z : P(x, y, z)$ instead of writing $\forall x : \forall y : \forall z : P(x, y, z)$.

◇ `inputFile` *fileName*
The function `inputFile` receives the ALFP clause from the text file specified by *fileName*. It then transforms the clause into the internal data structure that the solver uses to process the clause.

The ALFP clauses in the text file requires the following syntax:

$$
\begin{array}{lcl}
t & ::= & c \quad | \quad x \quad | \quad f\,(t_1, \cdots, t_k) \\
pre & ::= & R\,(t_1, \cdots, t_k) \quad | \quad !R\,(t_1, \cdots, t_k) \quad | \quad pre_1 \& pre_2 \\
 & | & pre_1 \mid pre_2 \quad | \quad E\ x.pre \quad | \quad t_1 = t_2 \quad | \quad t_1 != t_2 \\
cl & ::= & R\,(t_1, \cdots, t_k) \quad | \quad \mathbf{1} \quad | \quad cl_1 \& cl_2 \\
 & | & pre \Rightarrow cl \quad | \quad pre \Longrightarrow cl \quad | \quad A\ x.cl
\end{array}
$$

This syntax is slightly different from the one introduced in section 1, mainly on notations used for the operators and quantifiers. Here ! is used for negation, | for disjunction, and & for conjunction. A in $A\ x.$ is the universal quantifier, and E in $E\ x.$ the existential quantifier. The notation != is used for $\neq$.

◇ `inputStd`
The function `inputStd` receives the ALFP clause from the standard input, known as stdIn, and then transforms the clause into the internal data structure.

◇ `inputStr` *str*
The function `inputStr` receives the ALFP clause from the string *str*, and transforms the clause into the internal data structure. This is mainly used for receiving the clause via sockets by means of the SML/NJ Socket structure (cf. [15]).

◇ `fromALFPtoHD` *fileName*
The auxiliary function `fromALFPtoHD` is used to transform the ALFP clause in the text file specified by *fileName* to the SML data structure defined in `HornDirect`.

6.2 Functions for output

There are also four ways to output the result from the solver:

- output to SML data structures

- output to a file
- output to stdOut (standard output)
- output to a string

◇ `outputData` *result, (select, univ)*
The function `outputData` is used to output the result to a SML data structure with the type `string list * (string * string list list) list`. The first list represents the universe, i.e. a list of ground terms, each is represented by a string. The second list represents the result, which contains a list of relations. Each relation is represented by a relation name (of string type), and a list of tuples of ground terms (string list list).

Example 10 *The relation*

$$SISTERS : (Ann, Mary), (Susan, Hellen)$$

is represented by a list:

$$[\text{“SISTERS”}, [[\text{“Ann”}, \text{“Mary”}], [\text{“Susan”}, \text{“Hellen”}]]]$$

□

The function has three parameters: *result*, *select*, and *univ*. The *result* (of type Forest.forest) holds the solution from the solver. The *select* (of type (string * int) list) specifies a list of relations as the output. A relation is represented by a pair of the relation name, and its arity. For example, if one is going to output the 2-ary relation $SISTERS$ from the result, the *select* is specified by $[(\textit{“SISTERS”}, 2)]$. The empty list corresponds to selecting all the relations as the output. The *univ* specifies whether or not the universe is also included as the output. If it is true, the universe is output as well. If it is false the universe is excluded from the output.

◇ `outputFile` *(result, fileName) (select, univ)*
The function `outputFile` is used to output the result to a text file. The function has four parameters: *result*, *fileName*, *select*, and *univ*. The *fileName* specifies the output file name. The *result*, *select* and *univ* are the same as those in `outputData`.

◇ `outputStd` *result (select, univ)*
The function `outputFile` is used to output the result to stdOut, i.e. the standard output. The three parameters: *result*, *select*, and *univ* are the same as those in `outputData`.

◇ `outputStr` *result (select, univ)*
The function `outputFile` is used to output the result to a string. The three parameters: *result*, *select*, and *univ* are the same as those in `outputData`. Again, this function is mainly used for sending the result via sockets by means of the SML/NJ Socket structure (cf. [15]).

6.3 Functions for solving

There are ten functions in this group. Two of them (`solve` and `solveCount`) are used to solve the clause represented by the internal data structure and construct the solution to the clause in the result data structure, without output of the result. The other functions combine these two functions with different output functions to easy the user to output the result.

◇ `solve` *(result, cl)*
The function `solve` is used to solve the clause without the output of the result. This can be used in an iterative solving process where the intermediate result is not of interests, and the final result can be output using the functions for output.

The `solve` function has two parameters: *result* and *cl*. The *result* holds the global data structure containing the solution to the clause, and the *cl* is the clause (generated by an input function) that is going to be solved. These two parameters are used in all the following functions, and we shall not describe them again in the sequel, if there no special purpose.

◇ `solveCount` *(result, cl)*
The function `solveCount` is essentially the same as `solve`, but printing a summary of the result, e.g. the universe, the number of elements for each relation, and etc., onto the screen.

◇ `solveData` *(result, cl)*
The function `solveData` is used to solve the clause and output all the result including the universe in the way that `outputData` does. At the same time it prints a summary of the result onto the screen.

◇ `solveFile` *(result, cl, fileName)*
The function `solveFile` is used to solve the clause and output all the result including the universe to the file specified by *fileName*. At the same time it prints a summary of the result onto the screen.

◇ `solveStd` *(result, cl)*
The function `solveStd` is used to solve the clause and output all the result including the universe to the standard output stdOut. At the same time it prints a summary onto the screen.

◇ `solveStr` *(result, cl)*
The function `solveStd` is used to solve the clause and output all the result including the universe to a string, which will be returned. At the same time it prints a summary onto the screen.

◇ `selectSolveData` *(result, cl) (select, univ, count)*
The function `selectSolveData` is used to solve the clause and output the selected the result in the way that `outputData` does. It has five parameters. The parameters *result* and *cl* are the same as those in `solve`, while *select* and *univ* are the same as those in `outputData`. The *count* is used to specify whether a summary of the result is printed or not. If it is true, the summary of the result

is printed, otherwise the summary is not printed.

◇ `selectSolveFile` *(result, cl, fileName) (select, univ, count)*
The function `selectSolveFile` is used to solve the clause and output the the selected result to the file specified by *fileName*. The other parameters are the same as those in `selectSolveData`.

◇ `selectSolveStd` *(result, cl) (select, univ, count)*
The function `selectSolveStd` is used to solve the clause and output the selected result to the standard output stdOut.

◇ `selectSolveStr` *(result, cl) (select, univ, count)*
The function `selectSolveStr` is used to solve the clause and output the selected result as a string returned.

6.4 Other facilities

We provide also some other auxiliary functions to facilitate the use of the solver. As we have already mentioned that the structure `HornDirect` defines the SML data structure to represent ALFP clauses. It provides also a function called `translate` to translate a clause from `HornDirect` data structure to the solver's internal representation of clauses and thus avoiding the ALFP's parser. Symmetrically, we also provide a structure called `InternToDirect`, in the file *internToDirect.sml* under the directory **Formulas**, where a function called `toHdClause` is to transform a clause resulted from the parser into the `HornDirect` data structure. In addition, we provide the following facilities:

6.4.1 Pretty print

The structure `Pretty`, defined in the *pPrint.sml* file under the directory **Formulas**, provides three functions to print clauses defined in the `HornDirect` structure into a Latex file so that one can include it into the Latex document whenever there is a need. The interface to these functions are:

```
val prettyTable : string * clause * string →  unit
val prettyFrame : string * clause  →  unit
val prettyText : string * clause  →  unit
```

◇`prettyTable` *(fname, cl, capt)*
The function `prettyTable` is used to print the clause *cl* into the Latex file called *fname.tex* as a framed table, and the string *capt* is the contents of the caption of the table. Moreover, *fname* is also used as a label of the table for your reference in your Latex document.

◇ `prettyFrame` *(fname, cl)*
The function `prettyFrame` is used to print the clause *cl* into the Latex file called *fname.tex* as a framed text.

◇ `prettyText` *(fname, cl)*
The function `prettyText` is used to print the clause *cl* into the Latex file called *fname.tex* as a pure text without any decoration.

When the file *fname.tex* is included in the main Latex document, the *alfp.sty* file should also be included. The *alfp.sty* file is in the **Application** directory.

As mentioned in the previous section, the function `fromALFPtoHD` is used to transform the ALFP clause from the text file to the clause represented by the `HornDirect` structure. One can therefore use the pretty print function to print the ALFP clause from the text file as well.

6.4.2 Making a breakpoint

As mentioned, the clause $pre \implies cl$ is used to make a *breakpoint* in front of clause cl in order to print the number of environments passing through the *breakpoint* of clause cl. Intuitively, it gives how many times the clause cl will be executed. In the solver, the (partial) environment is used to map variables to the ground terms from the universe [2]. The experiments with the solver in [16] reports that minor syntactical variations of formulas, e.g. the order of conjuncts in preconditions, have a strong impact on computation efficiency of the solver.

It is often not easy to predicate which order of the conjuncts in preconditions is better than the others, which depends on the current environment that the unification is carried on and the values to be unified.

Example 11 *Consider a fragment $P_1(x,y) \wedge P_2(x,y) \Rightarrow Q(x,y)$ of a clause, where x and y are bounded variables. When the precondition $P_1(x,y) \wedge P_2(x,y)$ is checked, variable x is e.g. bound to the constant a, and y is not yet bound in the current environment. The solver makes first the query to the predicate P_1 and obtains a list of tuples for P_1, and it then unifies each tuple with the current environment. Each successful unification will produce a new environment for P_2 to be checked. If e.g. the list of tuples for predicate P_1 is [(a, b), (a,c), (a,d)], and after the query to P_1 is done, it produces three environments. The query to P_2 is then carried out in three environment. If e.g. the list tuples for predicate P_2 is [(b, b), (b,c), (b,d)], it is easy to see that all the unifications will be failed. Therefore no execution to assertion $Q(x,y)$. In this case, if we could check P_2 first then we would not check P_1 and save unnecessary computation expenses.*

□

In this situation, e.g., one can use the clause $pre \implies cl$ to replace the clause $pre \Rightarrow cl$ in order to do some debugging, e.g. with different orderings in preconditions.

Example 12 *Consider the clause $\forall x : \forall y : P_1(x,y) \wedge P_2(x,y) \Rightarrow Q(x,y)$, we can do the followings:*

1. *Transform the clause into: $\forall x : \forall y : P_1(x,y) \implies (P_2(y,x) \Rightarrow Q(x,y))$, and run the solver. When the computation is done, we will see a message from the screen "Number of env's passing through the breakpoint: n_1", where the number n_1 tells you how many times that the clause $P_2(y,x) \Rightarrow Q(x,y)$ is going to be executed.*
2. *Transform the clause into: $\forall x : \forall y : P_1(x,y) \implies (P_2(x,y) \Rightarrow Q(x,y))$, repeat the same procedure as in (1) and we obtain the number n_2*
3. *We then may be able to choose the order by n_1 and n_2. We shall choose the order that precondition producing the smaller number precedes the precondition producing the larger number.*

□

7 Conclusion

We have extended the Succinct Solver to allow the use of structured terms in a natural way. We achieve that through two main procedures: preprocessing and solving. In the preprocessing procedure, we transform a clause with structured terms into the clause with only simple terms as the arguments of the predicates through explicit unifications. In the solving procedure we expend the universe dynamically, and once a new ground term is added to the universe we continue the computations which were suspended because of the lack of the full information.

The Succinct Solver (V2.0) has been tested and debugged by the developers with many small scaled examples, e.g. reaching definitions analysis for the *While* language [17], and the control flow analysis for Discretionary Ambients [18] etc., with focus on the functional correctness of extended features. It has been (and is) used by the users with lager scaled applications e.g. security analysis for Demoney etc. The comparison of the Succinct Solver with XSB Prolog [19] with benchmark examples, given by a separate report, shows that essentially the experimental complexity of the Succinct Solver is comparable with XSB Prolog. The user interface has be further extended by an web browser application to run the solver as the server process (c.f. the web page below). The whole sources of the solver together with the user's guide can be downloaded from the Succinct Solver web page:

http://www.imm.dtu.dk/cs/Secure/SuccinctSolver

References

[1] H. Riis Nielson and F. Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. 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. A Succinct Solver for ALFP. *Nordic Journal of Computing*, 9(4), 2002.

[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] F. Baader and K. U. Schulz. Unification Theory – An Introduction. In *Automated Deduction. A basis for application.* W. Bibel and P.H. Schmitt (eds.), Kluwer Academic Publishers, 1998.

[7] H. Comon and Claude Kircher. Constraint Solving on Terms. LNCS 2002, Springer Verlag, 2001.

[8] T. Nipkow, L. C. Paulson and M. Wenzel *Isabelle/HOL A proof Assitant for Hight-Order Logic.* LNCS 2283, Springer Verlag, 2003.

[9] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

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

[11] 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.

[12] J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1984.

[13] M. Ben-Ari. *Mathematical Logic for Computer Science.* C. A. R. Horare Series Editor, Prentics Hall, 1993.

[14] H.Sun, H. Riis Nielson and F. Nielson. Data Structures in the Succinct Solver (V1.0), Technical Report SECSAFE-IMM-005-1.0, 2002.

[15] The Standard ML Basis Library.
http://www.standardml.org/Basis/socket.html.

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

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

[18] H. Riis Nielson, F. Nielson and M. Buchholtz. Security for Mobility. Technical Report IMM-TR-2002-20, 2002.

[19] B. Cui and D. S. Warren. A System for Tabled Constraint Logic Programming. *Computational Logic 2000*, LNAI 1861, Springer Verlag, 2000.