Give Reasoning a Trie

Thomas Prokosch, François Bry

Institute for Informatics, Ludwig-Maximilian University of Munich, Germany

Abstract

A data structure providing an efficient storage and retrieval of logical expressions is a necessary component of every automated reasoning system. Most theorem provers rely on term indexes tuned to unification. Tabled logic programming systems in contrast favour indexes tuned to the retrieval of variants or instances. This article proposes the versatile data structure Instance Trie which supports the unification of queries with stored expressions as well as the retrieval of variants, instances, and generalisations of queries. Instance tries are stable in the sense that their structure is independent of the order in which they are filled or updated. They give rise to an incremental expression retrieval.

1. Introduction

Automated reasoning [1] relies on an efficient storage of logical expressions as well as various forms of expression retrieval. Resolution-based theorem proving systems require first and foremost the retrieval of expressions unifying with queries. If they perform ancestor resolution or make use of lemmas, they also require the retrieval of variants or instances of queries. Meta-programming [2], a form of logic programming, requires unification as well as the retrieval of instances, variants, and generalisations of queries. Tabling [3, 4, 5, 6], a form of memoing used in logic programming, requires the retrieval of variants or generalisations of queries while logic programming's core reasoning requires the retrieval of expressions unifying with queries.

This article proposes instance tries primarily developed for tabled logic programming [3, 4, 5, 6] and meta-programming. Instance tries are versatile in the sense that they support the various retrieval modes needed in automated reasoning: The unification of queries with stored expressions and the retrieval of variants, instances, and generalisations of queries. Instance tries exploit well established techniques: The implementation of variables as pointers of programming languages' run-time systems and the substitution-based tries of the term indexes of resolution-based theorem proving [7, 8, 9, 10]. Instance tries re-use these techniques in a novel manner, though, resulting in a data structure which, in contrast to most of its predecessors, is stable in the sense that it is independent of the order in which it is filled or updated and gives rise to an incremental expression retrieval.

The article is structured as follows. Section 1 is this introduction. Section 2 is devoted to related work. Section 3 briefly recalls the concepts referred to, and introduces the notations used in this article. Section 4 describes the structure of an instance trie. Section 5 addresses various data retrieval modes for instance tries. Section 6 concludes the article.

PAAR'20: 7th Workshop on Practical Aspects of Automated Reasoning, June 29–30, 2020, Paris, France

prokosch@pms.ifi.lmu.de (T. Prokosch); bry@lmu.de (F. Bry)
0000-0003-4367-0832 (T. Prokosch); 0000-0002-0532-6005 (F. Bry)

2. Related work

The book [8] and the book chapter [9] both survey “term indexing”, that is, data structures for the storage and retrieval of first-order logic terms or atoms. They complement each other: [8] covers “out-of-fashion” data structures, [9] covers data structures which did not exist when [8] was written.

Tries have been introduced in [11] for storing strings, the name “trie” in was introduced in [12]. Tries store strings after their lexicographical order. Tries have advantages over hash tables: They are not subject to collisions and are stable in the sense that their structures are independent of the insertion order.

The Path-Indexing Method [13] stores terms in “path lists” themselves stored in tries or hash tables. A Path-Index is not “stable” in the sense that its structure depends on the insertion order what might be beneficial in reasoning with commutative functions [13, p. 15] but makes retrieval less efficient. Path-Indexing is “versatile” in the sense that it supports all query modes variant, instance, generalisation and unification. Path-Indexes disregard variables’ identities [13, p. 6] and therefore return false positives among the answers to retrieval queries.

Dynamic Path Indexing [14] fixes a deficiency of the Path-Indexing Method [13]. For efficiency reasons, a dynamic path index re-orders the terms it stores. Thus, a dynamic path index is not stable. Like a path-index, a dynamic path-index is versatile.

Extended Path Indexing [15] improves the efficiency of Path-Indexing for non-linear queries. While Extended Path Indexing’s algorithms rely on substitutions, its indexes store terms. Like a path-index and a dynamic path-index, a dynamic path-index is not stable but versatile.

Discrimination Trees [16, 17] store terms in their leaves (their inner nodes represent term prefixes). Like path indexes, discrimination trees disregard variables’ identities and therefore return false positives what makes necessary post-retrieval matching or unification tests. Discrimination trees identify and order terms with randomly assigned integers [17, p. 158] what makes them non-stable. They are versatile.

Deterministic Discrimination Trees [18] improve discrimination trees by making a post-retrieval matching test unnecessary for retrieving variant, instances or generalisations of linear queries [18, p. 323]. Deterministic Discrimination Trees are not stable but they are versatile.

Adaptive Discrimination Trees [19] adapt the traversal order on every insertion what makes them non-stable but yields faster retrievals of generalisations, the only query mode supported [19, p. 248]. Adaptive Discrimination Trees are not stable and not versatile.

Abstraction Trees [20] represent terms by substitutions and are structured by anti-unification [21, 22]: Parent nodes carry strict generalisations of their children. The inner nodes of abstraction trees do not store terms. A heuristic decides which terms are anti-unified what makes abstraction trees non-stable. They are versatile.

Substitution Trees [7] combine features of discrimination and abstraction trees and represent the stored terms by substitutions. They have better memory usage and retrieval time than abstraction trees. They are not stable but versatile.

Downward Substitution Trees [23] modify substitution trees to ensure stability and the linearity of deletions.

Code Trees [24] were developed for speeding up resolution. Each query mode requires a specific code tree: Code trees are not versatile but they are stable.

Coded Context Trees [25] like abstraction trees represent terms by substitutions (and currying), are structured by anti-unification and depend on the insertion order: They are not stable. They support only the retrieval of generalisations: They are not versatile.

Tabling [3, 4, 5, 6] consists in sharing, and therefore storing and retrieving, answers to (sub)goals so as to improve both the efficiency and the termination of logic programs.

Dynamic Threaded Sequential Automata (DTSA) [26] have been developed for tabling. They are based on Sequential Factoring Automaton (SFA) which were designed for speeding up resolution. DTSA are substitution-based tries which store data only in their leaves and their parent-child relationship is based on unification. The structure of a DTSA depends on the insertion order what makes them non-stable. They support only the retrieval of instances: They are not versatile.

Time Stamped Tries [4] have also been developed for tabling. They are simpler and require less memory than DTSA. Their retrieval prioritizes the most often instantiated terms. They rely on a hash function based on the derivation time of the terms: They are not stable. They support only the retrieval of variant and instances: They are not versatile.

3. Concepts and notations

Expressions Instance tries are defined in the following referring to *expressions*, a generalisation of first-order terms and first-order atomic formulas.[1] Expressions are defined from symbols as follows.

Finitely many non-variable symbols and infinitely many variables are considered. The non-variable symbols are totally ordered by $\leq_{nv}$ and the variables are totally ordered by $\leq_v$.

A *constructor* c is a pair s/a with s a non-variable symbol and a one of finitely many arities associated with the symbol s.

An *expression* is either a *variable* or a *non-variable expression*. A non-variable expression is either a constructor c of arity 0, or it has the form $c(e_1, \dots, e_n)$ where c is a constructor of arity $n \geq 1$ and $e_1, \dots, e_n$ are expressions.

Two expressions are *variable-disjoint* if none of the variables occurring in the one expression occur in the other.

e_1,..., and e_n are the *direct subexpressions* of $c(e_1, \dots, e_n)$.

Observe that no proper prefix of an expression (in standard or prefix form) is an expression and that there are finitely many constructors.

Orders on expressions and sequences of expressions A total order $\leq_c$ is (lexicographically [27, pp. 18–19] [28, pp. 88–89]) defined as follows on the variables and constructors from the orders $\leq_{nv}$, $\leq_v$, and the order on the natural numbers:

- $v_1 <_c v_2$ if v_1 and v_2 are variables and $v_1 <_v v_2$.
- $v <_c s/a$ if v is a variable and s is a non-variable symbol.

[1]The distinction between terms and atomic formulas is irrelevant to storage and retrieval.

- $s/a_1 <_c s/a_2$ if s is a non-variable symbol and $a_1 < a_2$.
- $s_1/a_1 <_c s_2/a_2$ if s_1 and s_2 are non-variable symbols and $s_1 <_{nv} s_2$.

A total order $\leq_e$ on expressions is lexicographically derived from the total order $\leq_c$ on constructors. Similarly, a total order $\leq_s$ on finite sequences of expressions is lexicographically derived from the total order $\leq_e$ on expressions.

Prefix notation An expression $c(e_1, \dots, e_n)$ is in *standard notation.* This expression can also be written without parentheses in *prefix notation* (also known as *Polish* or *Łukasiewicz notation*) as $c/n\ e_1/a_1\ \dots\ e_n/a_n$ where a_i is the arity of expression e_i $(1 \leq i \leq n)$.

Notations In the following, the lower case letters $a, b, c, \dots, z$ with the exception of v denote the non-variable symbols and $\leq_c$ denotes the standard order on that set; $v_0, v_1, v_2, \dots$ (with subscripts) denote the variables and $\leq_v$ the order on variables: $v_i \leq_v v_j$ if $i \leq j$. The notation $v^1, \dots, v^n$ (with superscripts) denotes n arbitrary variables.

Example 1.

- *$a, a(b)$ and $a(b, c)$ are expressions in standard notation. Their counterparts in prefix notation are $a/0, a/1\ b/0$ and $a/2\ b/0\ c/0$.*
- *v_2 and $p(a, a(v_5))$ are expressions in standard notation. Their counterparts in prefix notations are v_2 and $p/2\ a/0\ a/1\ v_5$.*
- *$a <_e a(b) <_e a(b, c)$ in standard notation and $v_2 <_e p/2\ a/0\ a/1\ v_5$ in prefix notation.*
- *$v_2 \leq_v v_5$.*
- *$[v_6] <_s [v_6, a(b, c), p(a, a(v_5))] <_s [a(b, c), p(a, a(v_5))] <_s [p(a, a(v_5)), a(b, c)]$ where $[e_1, e_2, \dots, e_n]$ denotes the sequence of expressions $e_1, e_2, \dots,$ and e_n.*

Substitutions Let E denote the set of expressions and V the set of variables. A substitution[2] $\sigma = \{v^1 \mapsto e_1, \dots, v^n \mapsto e_n\}$ with $n \geq 0$ denotes a total function $V \to E$ such that

- $v^1, \dots, v^n$ are pairwise distinct variables,
- $\sigma(v^i) = e_i$ for all $i = 1, \dots, n$, and
- $\sigma(v) = v$ if $v \neq v^i$ for all $i = 1, \dots, n$.

The application $e\sigma$ of a substitution $\sigma = \{v^1 \mapsto e_1, \dots, v^n \mapsto e_n\}$ to an expression e is the expression obtained from e by simultaneously replacing for all $i = 1, \dots, n$ every occurrence of each v^i in e by e_i. The application of the *empty substitution* ϵ to an expression leaves it

[2]The binding of a variable v to an expression e is denoted $v \mapsto e$ because it reminds of the implementation of variables as pointers. Another widespread notation for the same, used among others in [29], is e/v.

unchanged: For all expressions e, $e\epsilon = e$. If e is an expression and if σ and τ are substitutions, then $(e\sigma)\tau = e(\sigma\tau)$ where $\sigma\tau$ denotes the composition of σ and τ.

Let $\sigma = \{v^1 \mapsto e_1, \dots, v^n \mapsto e_n\}$ be a substitution. The finite set of variables $\mathtt{dom}(\sigma) = \{v^1, \dots, v^n\}$ is the *domain* of σ. The set of variables modified by σ, that is, variables v such that $v\sigma \neq v$, map to the *range* of σ. The range of a substitution is finite.

A *renaming substitution* is a substitution which is a permutation of the set V of variables.

Substitutions σ_1 and σ_2 are *compatible* if for all $v \in \mathtt{dom}(\sigma_1) \cap \mathtt{dom}(\sigma_2)$ $v\sigma_1 = v\sigma_2$. If σ_1 and σ_2 are compatible substitutions, then $\sigma_1 \cup \sigma_2$ is a well-defined substitution.

Variants, instances, generalisations, and unifiers A *variant* of an expression e is an expression e' such that $e' = e\rho$ for some renaming substitution ρ.

An *instance* of an expression e is an expression e' such that $e' = e\sigma$ for some substitution σ. An instance e' of e is said to *match to e*. A strict instance of an expression e is an instance of e which is not a variant of e.

A *generalisation* of an expression e is an expression g such that e is an instance of g, that is, $e = g\sigma$ for some substitution σ. A strict generalisation of an expression e is a generalisation of e which is not a variant of e.

A *unifier* ν of expressions e_1 and e_2 is a substitution such that $e_1\nu = e_2\nu$.

Matching and unification A *(first-order syntactic) matching problem*, short *matching problem*, of an expression e_1 in an expression e_2 consists in finding substitutions ν such that $e_1 = e_2\nu$. Thus, a matching problem consists in finding substitutions establishing an instance, or symmetrically generalisation, relationship between expressions [30].[3]

A *first-order syntactic unification problem*, short *unification problem*, consists in finding unifiers for two expressions.

If the matching (unification, respectively) problem of e_1 in e_2 (e_1 and e_2, respectively) has solutions, then it has solutions μ *most general matchers*, short *mgm*, of e_1 in e_2 (*most general unifiers,* short *mgu*, of e_1 and e_2), which are [30, 31]

- complete in the sense that for all matchers ν of e_1 in e_2 (all unifiers ν of e_1 and e_2, respectively) there exist a substitution σ such that $\nu = \mu\sigma$,
- equivalent in the sense that two distinct mgms (mgus, respectively) μ_1 and μ_2 of e_1 in e_2 (of e_1 and e_2, respectively) are identical up to a variable renaming (that is, there exists a renaming substitution σ such that $\mu_1\sigma = \mu_2$).

A matching problem is a strengthening of a unification problem: If $e_1 = e_2\tau$, then τ is a unifier of e_1 and e_2. The converse is false: Some solutions to unification problems of e_1 and e_2 are solutions neither of the matching problem of e_1 in e_2, nor of the matching problem of e_2 in e_1. As a consequence, matching problems can be solved with algorithms (called *matching algorithms*) which might be, in some cases, more efficient than algorithms for the unification problem (called *unification algorithms*).

[3]Matching has also been called *filtering*, *one-sided unification*, and *semi-unification.*

4. Instance tries: Structure

An instance trie is a tree T such that:

- The root of T carries, but does not store, a variable.
- Every node of T except the root stores an expression.
- Every arc $N_1 \to N_2$ of T corresponds to non-renaming substitution σ such that if e is the expression stored at node N_1, then the instance $e\sigma$ of e is the expression stored at node N_2.
- The children of a node are ordered (in a manner described below).

Since the substitution corresponding to an arc of an instance trie is not a renaming substitution, the expression stored at a child of a node N is a strict instance of the expression stored at N.

Example 2. *Abstract depiction of an instance trie storing the expressions $p(a, v_1, v_1)$, $p(a, b, b)$, $q(v_5, v_3, v_4, c)$, and $q(v_5, a, b, c)$:*

$$\begin{tikzcd}
& v_0 \arrow[dl, dash] \arrow[dr, dash] & \\
p(a,v_1,v_1) \arrow[d, dash] & & q(v_5,v_3,v_4,c) \arrow[d, dash] \\
p(a,b,b) & & q(v_5,a,b,c)
\end{tikzcd}$$

The root of the instance trie, denoted above v_0, is a variable which is not an expression stored in the instance trie. The children of the root, $p(a, v_1, v_1)$ and $q(v_5, v_3, v_4, c)$ are ordered by $<_e$, the expression order.

In fact, an instance trie does not store expressions but substitutions. Thus, a more faithful though still abstract depiction of the instance trie given above is as follows:

$$\begin{tikzcd}
& v_0 \arrow[dl, dash] \arrow[dr, dash] & \\
\{v_0 \mapsto p(a,v_1,v_1)\} \arrow[d, dash] & & \{v_0 \mapsto q(v_5,v_3,v_4,c)\} \arrow[d, dash] \\
\{v_1 \mapsto b\} & & \{v_3 \mapsto a, v_4 \mapsto b\}
\end{tikzcd}$$

Composing the substitutions along a path from the root to a node yields the expression stored at that node. $q(v_5, a, b, c)$ is for example obtained by $v_0\{v_0 \mapsto q(v_5, v_3, v_4, c)\}\{v_3 \mapsto a, v_4 \mapsto b\}$.

Instance tries do not store expressions in standard notation but instead in prefix notation and furthermore using a low-level representation. The following paragraphs successively address the representations of expressions, substitution application, substitutions, and finally instance tries.

Representation of expressions The representation of an expression in the memory of a run-time system is based on the expression's prefix notation. Assuming that a constructor and a variable are stored in 4 bytes and storage begins at address 0, the representation of $f(a, v_1, b, v_1)$ is:[4]

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19
$f/4$	$a/0$	nil	$b/0$	8

The leftmost, or first, occurrence of the variable v_1 is represented by the value nil which indicates that the variable is unbound. The second occurrence of the variable v_1 is represented by an offset: The address of this second occurrence's representation, 16, minus the offset, 8, is the address of the representation of the variable's first occurrence, 8. Occurrences of (the representation of) a variable like the second occurrence of v_1 in $f(a, v_1, b, v_1)$ and the cell representing such variables like the cell at address 16 in the above representation of $f(a, v_1, b, v_1)$ are called *locally bound variables.*

Three properties of an expression representation are worth stressing:

1. The variables' names are irrelevant to expression representations. If e is an expression and ρ is a variable renaming, then e and $e\rho$ have exactly the same representation.[5] For example, the expression $p(a, v_1, b, v_1)$ is represented exactly like $p(a, v_0, b, v_0)$ and the expression $q(v_5, v_3, v_4, c)$ is represented exactly like $q(v_0, v_1, v_3, c)$. However, this is not the case of the representation of substitution applications: As it is discussed below, $f(a, v_1, v_2, v_3)\{v_2 \mapsto b, v_3 \mapsto v_1\}$ is not represented like $f(a, v_1, b, v_1)$.
2. The representation of an expression is *variable-linear*, short *linear*, in the sense that a variable occurs at most once in an expression. Indeed, a non-linear expression (like $f(a, v_1, b, v_1)$) is represented as the application of a substitution to a linear expression (like $f(a, v_1, b, v_2)\{v_2 \mapsto v_1\}$).
3. In an instance trie, two distinct variables cannot be bound to two distinct expressions in which the same variable occurs. The bindings $v_1 \mapsto f(v_3)$ and $v_2 \mapsto g(v_3)$ for example cannot occur in an instance tree. In an instance trie, such bindings would be expressed for example as $v_1 \mapsto f(v_3)$ and $v_2 \mapsto g(v_4), v_4 \mapsto v_3$.

Thus, expressions can be ordered as if their variables were, in the order of their first occurrence, $v_0, v_1, v_2, \ldots$, that is, as if they were standardized [29].

This representation of $f(a, v_1, b, v_1)$ given above is abstract in the sense that it is a simplification. Its implementation in the heap of a run-time system, that is, the concrete representation, differs from the abstract representation as follows:

- A next-representation address is added at the beginning of a (connected) expression representation so as to ease garbage collecting (but not at the beginning of each subexpression).

[4]Recall that $a, b, c, \ldots, z$ except v denote non-variable symbols and $v_0, v_1, v_2, \ldots$ denote variables.

[5]As a consequence, expressions can be considered standardized [29]. Recall that the (unique) standardized form of $f(v_2, v_5, v_2)$ is $f(v_0, v_1, v_0)$.

- Instead of cells, tokens are considered, each of which consists of one or several consecutive cells.
- A three-valued flag at the beginning of every token indicates the type of its content: Non-variable symbol, non-locally bound variable, or locally bound variable.
- The token for a constructor s/a consists of three successive memory cells, the first of which contains the afore-mentioned type flag, the second the code of the symbol s, and the third the arity a.
- A concrete expression representation might include additional information like variable names or a time stamp.[6]

Assuming that there is no additional information, that p and a are represented by their ASCII codes 112 and 97 respectively, and that nil is represented as 999999[7], the type flag takes the values 0, 1, and 2, and using decimal instead of binary numbers for better readability, the concrete representation of $p(a, v_1, v_1)$ is as follows:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

24	0	112	3	0	97	0	1	999999	2	5

Representation of substitution applications Assuming that the expression $p(a, v_1, v_1)$ is stored at address 0 and the expression $q(b, v_3)$ at address 23, the substitution application $p(a, v_1, v_1)\{v_1 \mapsto q(b, v_3)\}$ is represented as follows:

Representations of $p(a, v_1, v_1)$ and $q(b, v_3)$ (before any substitution application):

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34

$p/3$	$a/0$	nil	4		q/2	b/0	nil

Representations of the same after the application of $p(a, v_1, v_1)\{v_1 \mapsto q(b, v_3)\}$:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34

$p/3$	$a/0$	23	4		q/2	b/0	nil

Thus, binding a variable (like v_1) to an expression (like $q(b, v_3)$) is realised by storing the expression's address (23) at the variable's address (cell 8). Observe that the cell representing the second occurrence of the variable v_1 (cell 12) keeps its offset (4) unchanged. Thus, binding a variable v which occurs in an expression e to an expression e' consists in storing at the leftmost occurrence of v in the representation of e the address of the representation of e', leaving unchanged further occurrences of v in the representation of e. This approach to binding variables ensures that the representation of a substitution application is unique.

[6]Time stamps are needed by logic programming run-time systems for semi-naïve forward chaining and for tabling.

[7]nil can be represented by any other value outside the heap's address space.

Representation of substitutions In an instance trie, a substitution is applied at a node N to the expression representation R stored at the parent node P of N. The substitution's domain consists of all variables with value nil, that is, the non-locally bound variables, of R. Consider for example the arc $P : q(v_5, v_3, v_4, c) \rightarrow N : q(v_5, a, b, c)$ or, in a more faithful depiction, $P : \{v_0 \mapsto q(v_5, v_3, v_4, c)\} \rightarrow N : \{v_3 \mapsto a, v_4 \mapsto b\}$, of Example 3. The substitution $\{v_3 \mapsto a, v_4 \mapsto b\}$ is represented by two lists:

- The sequence of addresses of nil variables (that is, non-locally bound variables) in the expression $q(v_5, v_3, v_4, c)$ at the parent node P (in their order of occurrence): $[v_5, v_3, v_4]$
- The sequence of the values assigned to each of these variables yielding the expression $q(v_5, a, b, c)$ at the child node N: $[nil, a, b]$

More generally, a substitution $\{v^1 \mapsto e_1, \dots, v^n \mapsto e_n\}$ at a node N is represented by two lists:

- The sequence of addresses of nil variables (that is, non-locally bound variables) in the expression representation at the parent node of N.
- The sequence of the expressions or variable addresses[8] assigned to each of these variables at the child node N.

Observe that

- no variables occur in both lists since, as the expressions of second list are created, their variables are represented by so far unused memory cells,
- the first list does not contain any variable which does not occur in the expression at the parent node.

Representation of instance tries The following example illustrates the representation of substitutions in an instance trie.

Example 3. *Abstract depiction of an instance trie storing $q(v_5, v_3, v_4, c)$, $q(v_5, a, b, c)$, $f(a, v_1, v_1, v_2)$, and $f(a, b, b, c)$:*

$$\begin{tikzcd}
 & v_0 \arrow[dl, dash] \arrow[dr, dash] & \\
f(a,v_1,v_1,v_2) \arrow[d, dash] & & q(v_5,v_3,v_4,c) \arrow[d, dash] \\
f(a,b,b,c) & & q(v_5,a,b,c)
\end{tikzcd}$$

Faithful depiction of the same instance trie with a standard representation of substitutions:

$$\begin{tikzcd}
 & v_0 \arrow[dl, dash] \arrow[dr, dash] & \\
\{v_0 \mapsto f(a,v_1,v_1,v_2)\} \arrow[d, dash] & & \{v_0 \mapsto q(v_5,v_3,v_4,c)\} \arrow[d, dash] \\
\{v_1 \mapsto b, v_2 \mapsto c\} & & \{v_3 \mapsto a, v_4 \mapsto b\}
\end{tikzcd}$$

[8] Which turn nil variables in the parent node's expression into locally bound variables in the child node's expression.

Faithful depiction of the same instance trie with an abstract representation of substitutions:

$$\begin{tikzcd}
 & {[]\ [v_0]} \arrow[dl, dash] \arrow[dr, dash] & \\
{[f(a,v_1,v_1,v_2)]\ [v_1,v_2]} \arrow[d, dash] & & {[q(v_5,v_3,v_4,c)]\ [v_5,v_3,v_4]} \arrow[d, dash] \\
{[b,c]\ []} & & {[nil,a,b]\ [v_5]}
\end{tikzcd}$$

A substitution $\{v^1 \mapsto e_1, \dots, v^n \mapsto e_n\}$ at a node N is represented by the expression sequence $[e_1, \dots, e_n]$ for the variables $[v^1, \dots, v^n]$ in the parent node's variable sequence. The children of a node are ordered after their expression sequences. For all expressions e, $nil \leq_e e$, reflecting that as variable precedes every expression. The order on unbound variables is reflected by the positions of nil in expression sequences. The root of an instance trie has an empty expression sequence because it has no parent node. A node with an empty variable sequence [] (like the leftmost leaf in the above example) cannot have any children what reflects that the expression stored at such a node has no strict instance. The node of an instance trie carries two sequences:

- A sequence of expressions or nil representing the bindings of its parent node's variable.
- The sequence of the addresses of the nil variables occurring in the node's expression.

The representations of the two kinds of sequences is as follows:

- The first sequence, the sequence of variable bindings, is a sequence of representations of expressions. Since representations of expressions have variable lengths, the sequence is represented as a linked list.
- The second sequence, the sequence of nil variables, is a sequence of representation of variables, that is, of addresses of memory cells representing variables. Since such cells all have the same length, the sequence can be represented as a memory block beginning with the number of variable addresses stored in the block.

In addition to the afore-mentioned two lists, the representation of the node N of an instance tree includes a data structure storing, and giving a fast access to, the addresses of the children of N. If this number is small, a linked list can be used. Otherwise, a binary tree, or a B-tree is convenient.

The expression stored at a node N of an instance trie is obtained by successively applying to the instance trie's root variable the substitutions along the path from the root to N. Because of this, instance tries are a kind of substitution-based tries or *substitution tree* in [7, 8, 10].[9] Most data structures so far designed for automated reasoning and logic programming are, like instance tries, substitution tries. They differ from each other on how the expression e_N stored at a node N relates to the expression e_P stored at the parent node P of N. In Instance Tries, this relationship is structural: e_N is a strict instance of e_P. For most substitution tries proposed for automated reasoning, this relationship is random: It results from the storing or updating order of expressions.

[9]Substitution-based tries are not expression-based tries because an expression e_C stored at a child C of a node P is not a strict prefix of the expression e_P stored at P. Indeed, no strict prefix of an expression is an expression.

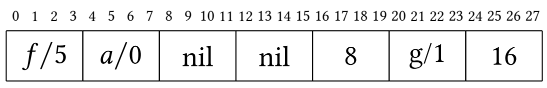
5. Instance tries: Operations

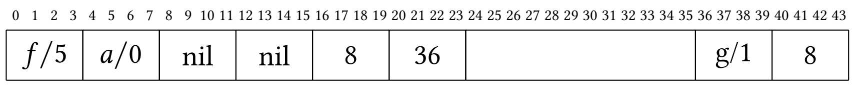
Let q be a query, that is, an expression to be evaluated against the expressions stored in an instance trie T. Four query modes are considered:

1. Variant: Is q a variant of an expression stored in T?
2. Instance: Which expressions stored in T are (strict) instances of q?
3. Generalisation: Which expressions stored in T are (strict) generalisations of q?
4. Unification: With which expressions stored in T does q unify?

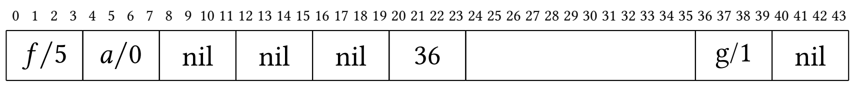
All four query modes are realized by both, a traversal of the instance trie T and a test at each node whether that node's expression is an answer to the query.

Dereferencing The tests are defined below referring to *dereferenced* representations of expressions.[10] Consider the following two representations of $f(a, v_1, v_2, v_1, g(v_1))$:





The first representations of $f(a, v_1, v_2, v_1, g(v_1))$ is dereferenced because, except for the representations of the second and third occurrences of the variable v_1, the variables' values are nil. The second and third occurrences of v_1 cannot be dereferenced like a pointer, because this would result in the following representation of $f(a, v_1, v_2, v_3, g(v_4))$, not of $f(a, v_1, v_2, v_1, g(v_1))$:



A dereferenced representation of an expression e is generated from any representation R_e of e as follows: While traversing R_e from left to right:

- If the cell reached contains a constructor or nil, or the offset of a locally bound variable, then copy its content to a new cell.
- Otherwise (the token reached is a non-locally bound variable v storing the address A of an expression representation), (recursively) dereference the expression representation at address A.

We assume that when an expression representation is dereferenced, the list of the addresses of its nil variables (that is, non-locally bound variables) is constructed. Constructing this list can be done while dereferencing the expression representation, that is, it does not require an additional traversal of the expression representation.

[10]Dereferencing a pointer P means determining the value P points to.

In implementing the algorithm sketch above, care must be given not to trespass expression representations' ends in recursive calls. This is easily cared for in the same traversal of expression representations by using as follows the constructors' arities during a left-to-right traversal of an expression representation E:

Let R denote the number of remaining (sub)expression representations; set $R := 1$ before traversing E, at each constructor s/n perform the update $R := R - 1 + n$ (-1 for the (sub)expression beginning at that constructor and $+n$ for the n subexpression representations now to be traversed), and at each variable perform the update $R := R - 1$. The expression representation's end is reached when $R = 0$.

Since dereferencing an expression copies that expression, dereferencing expressions comes at a cost. This cost is acceptable, though. Indeed, dereferencing speeds up the tests which anyway require to copy the stored expressions they are performed against because they bind variables. Binding variables directly in an instance trie would destroy it.[11]

Representation of a stored expression An expression e stored at a node N of an instance trie retrieved for comparison with a query q has the form $e = e'\sigma$ where e' is the expression stored at the parent node of N and σ is a non-renaming substitution, that is, e is a strict instance of e'. As discussed in the previous paragraph, (the representation of) e' is assumed to be dereferenced. Furthermore, q and e' as well as q and e are variable-disjoint.

The substitution σ is represented as described in Section 4, paragraph "Representation of substitutions" by two lists: A list of variables and a list of expressions or nil. It is also assumed that (the representation of) the expressions in that second list are dereferenced.

Useful information on e', the nature of which depends on the query mode, can be assumed to be known:

- If variants or instances of q are searched for, then e' is a strict generalisation of q (or, equivalently, q is a strict instance of e'). Indeed, otherwise the strict instance $e = e'\sigma$ of e' cannot be a variant or an instance of q and the search through T, which is described below, would not have led to e.

- If generalisations of q are searched for, then e' is already a generalisation of q. Indeed, otherwise, the strict instance $e = e'\sigma$ of which e' is a strict generalisation cannot be a generalisation of q and the search through T, which is described below, woulds not have led to e.

- If expressions unifying with q are searched for, then q and e' unify. Indeed, if q and $e = e'\sigma$ unify, then there exists a substitution μ such that $q\mu = e\mu = e'\sigma\mu$. Since the representations of q and e' are variable disjoint and the domain of the representation of σ contains only variables occurring in e', $q\sigma = q$. Therefore $q\mu = q\sigma\mu = e'\sigma\mu$. That is, $\sigma\mu$ is as unifier of q and e'. Thus, if q and e' do not unify, then q and $e = e'\sigma$ cannot unify, and the search through T, which is described below, would not have led to e.

[11]Copying the representation of the query is not necessary because, by using a log, bindings of query variables can be undone before the query is tested against another stored expression.

Retrieval Both a query q and an expression e stored in an instance tree at a node N tested as a possible answer to q are available as strict instances of the expression e' stored at the parent node of N. Indeed, by definition of an instance trie, e is stored in that form, and during the instance tree traversal q has been recognised as a strict instance of e' for otherwise, q and e would not be compared.

Assume that $q = e'\sigma_q$ and $e = e'\sigma_e$ and recall that each of the representations of σ_q and σ_e is a list consisting of n dereferenced expression representation, where n is the number of nil variables in the representation of e'. The matching problems of q in e and of e in q therefore reduce to matching problems of these two lists:

- If the expressions in the list representing σ_q and σ_e are pairwise variant of each other, then q and e are variant of each other.
- If every expression in the list representing σ_q is an instance of its counterpart in the list representing σ_e, then q is an instance of e, and if furthermore at least one of these expressions is a strict instance of its counterpart, then q is a strict instance of e.
- If every expression in the list representing σ_q is a generalisation of its counterpart in the list representing σ_e, then q is a generalisation of e, and if furthermore at least one of these expressions is a strict generalisation of its counterpart, then q is a strict generalisation of e.
- If the lists representing σ_q and σ_e unify, then q and e unify.

Versatile unification algorithm Instance tries make use of a unification algorithm which determines in a single left-to-right traversal of dereferenced expression representations R_1 and R_2 whether the represented expressions e_1 respectively e_2 are variants of each other, or e_1 is a strict instance of e_2, or e_1 is a strict generalisation of e_2, or none of this hold but e_1 and e_2 are unifiable, or finally e_1 and e_2 are not unifiable. This algorithm is called “versatile” because it is both a matching and a unification algorithm and because it further qualifies how matching expressions do match.

This unification algorithm exploits the afore-mentioned representation of expressions in an instance trie for avoiding unnecessary occurs-checks in some cases:

- on some offset variables under certain conditions
- as long as the left-to-right expression traversal is still in a matching mode (that is, one of variant, strict instance, or strict generalisation)

A description of this unification algorithm is beyond the scope of this paper and is given in [32].

Traversal A left-to-right depth-first traversal of an instance trie is, like for any tree structure, the algorithm of choice, because it has a simple recursive definition and minimizes the space complexity. All query modes but the query mode Unification affect that traversal:

- Variant: The traversal can be interrupted as soon as an answer is found because an instance tree does not contain distinct variant expressions.
- Instance: Subtrees rooted at instances of the query do not have to be traversed because all expressions they store necessarily are answers to the query.
- Generalisation: In a depth-first traversal of an instance tree, the first expression *g* found which generalises *q* and no children of which also generalise *q* determines the answers to *q*: Their set consists of *g* and its ancestors (except the instance trie's root). The subtree rooted at such a node *g* is not traversed.

Insertion and deletion The insertion of an expression *e* in an instance trie *T* is realised by first searching with the matching algorithm mentioned above for a strict generalisation *g* of *e* in *T*, which is always found because the root of an instance trie is a strict generalisation of every expression, second checking whether a child of *g* is a instance of *e*. If a child of *g* is a variant of *e*, then nothing is done. If a child of *g* is a strict instance of *e*, then *e* is inserted between *g* and *e* (what requires no more than updating two pointers). Otherwise, *e* is inserted in the instance trie as an as a new child of *g*.

In order to maintain tree invariants (instances of an expression are always inserted below their first generalization in a tree), children of *g* to the right of *e* need to be searched for instances of *e* and inserted below *e* recursively.

The deletion of an expression *e* from an instance trie *T* is realised by searching with the matching algorithm mentioned above for a variant of *e* in *T*. If such a variant is found at a node *N*, then this node *N* is deleted (what requires no more than updating a pointer) and, after this deletion, each expression stored at children node of *N* is inserted in the subtree rooted at the node *P* which, before the deletion, was the parent node of *N*. Otherwise nothing is done.

6. Conclusion

This article has introduced the data structure Instance Trie salient properties of which are:

- Stability: Instance tries are stable in the sense that their structures are independent of the order in which they are filled or updated.
- Versatility: Instance tries are versatile in the sense that they are well-suited to the retrieval of stored expressions in four query, or retrieval, modes: Variants, instances, generalisations and unification of expressions.
- Incrementality: Instance tries' storage based on the instance relationship gives rise to an incremental expression retrieval.

The companion article [32] gives more details on the versatile unification algorithm mentioned in Section 5. Further work will be devoted to analytical and empirical evaluations of instance tries and to deploying instance tries in a run-time system for tabled logic programming supporting meta-programming.

References

[1] J. A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, volume 1, MIT Press, 2001.

[2] F. Bry, In praise of impredicativity: A contribution to the formalisation of meta-programming, Theory and Practice of Logic Programming 20 (2020) 99–146. URL: http://arxiv.org/abs/1807.06051.

[3] H. Tamaki, T. Sato, OLD resolution with tabulation, in: E. Shapiro (Ed.), Proceedings of the Third International Conference on Logic Programming (ICLP), volume 225 of *LNCS*, Springer, 1986, pp. 84–98.

[4] E. Johnson, C. R. Ramakrishnan, I. V. Ramakrishnan, P. Rao, A space efficient engine for subsumption-based tabled evaluation of logic programs, in: A. M. A., T. Sato (Eds.), Functional and Logic Programming – Proceedings of the International Symposium on Functional and Logic Programming (FLOPS), number 1722 in LNCS, Springer, 1999, pp. 184–299.

[5] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, D. S. Warren, Efficient Tabling Mechanisms for Logic Programs, Technical Report, Department of Computer Science, State University of New York at Stony Brook, 1999.

[6] F. M. F. Cruz, Call Subsumption Mechanism for Tabled Logic Programs, Ph.D. thesis, Departamento de Engenharia Informática, Faculdade de Engenharia da Universidade do Porto, Portugal, 2010.

[7] P. Graf, Substitution tree indexing, in: Proceedings of the International Conference on Rewriting Techniques and Applications (RTA), number 914 in LNCS, Springer, 1995, pp. 117–131.

[8] P. Graf, Term Indexing, volume 1053 of *LNCS*, Springer, 1996.

[9] R. Sekar, I. V. Ramakrishnan, A. Voronkov, Handbook of Automated Reasoning, volume 2, MIT Press, 2001, pp. 1853–1964.

[10] R. Nieuwenhuis, T. Hillenbrand, A. Riazanov, A. Voronkov, On the evaluation of indexing techniques for theorem proving, in: R. Gore, A. Leitsch, T. Nipkow (Eds.), Proceedings of the International Joint Conference on Automated Reasoning (IJCAR), number 2083 in LNCS, Springer, 2001, pp. 257–271.

[11] R. de la Briandais, File searching using variable length keys, in: Proceedings of the Western Joint Computer Conference, 1959, pp. 295–298.

[12] E. Fredkin, Trie memory, Communications of the ACM 3 (1960) 490–499.

[13] M. E. Stickel, The Path-Indexing Method For Indexing Terms, Technical Note 473, SRI International, Menlo Park, California, USA, 1989.

[14] R. Letz, J. Schumann, S. Bayerl, W. Bibel, SETHEO: A high-performance theorem prover, Journal of Automated Reasoning 8 (1992) 183–212.

[15] P. Graf, Extended path-indexing, in: Proceedings of the 2th Conference on Automated Deduction (CADE), number 814 in LNAI, Springer, 1994, pp. 514–528.

[16] W. McCune, An indexing method for finding more general formulas, Association for Automated Reasoning Newsletter 1 (1988) 7–8.

[17] W. McCune, Experiments with discrimination-tree indexing and path-indexing for term retrieval, Journal of Automated Reasoning 9 (1992) 147–167.

[18] A. Gräf, Left-to-right tree pattern matching, in: Proceedings of the 4th International Conference on Rewriting Techniques and Applications (RTA), volume 488 of *LNCS*, Springer, 1991, pp. 323–334.
[19] R. Ramesh, I. Ramakrishnan, R. Sekar, Adaptive discrimination trees, in: Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP), volume 623 of *LNCS*, Springer, 1992, pp. 247–260.
[20] H. J. Ohlbach, Abstraction tree indexing for terms, in: Proceedings of the 9th European Conference on Artificial Intelligence (ECAI), 1990, pp. 479–484.
[21] G. Plotkin, A note on inductive generalization, Machine Intelligence 5 (1970) 153–163.
[22] G. Plotkin, A further note on inductive generalization, Machine Intelligence 6 (1971) 101–124.
[23] K. Hoder, A. Voronkov, Comparing unification algorithms in first-order theorem proving, in: B. Mertsching, M. Hund, M. Z. Aziz (Eds.), Proceedings of KI 2009 – Advances in Artificial Intelligence, 32nd Annual German Conference on AI, number 5803 in LNCS, Springer, 2009, pp. 435–443.
[24] A. Voronkov, The anatomy of Vampire – Implementing bottom-up procedures with code trees, Journal of Automated Reasoning 15 (1995) 237–265.
[25] H. Ganzinger, R. Nieuwenhuis, P. Nivela, Fast term indexing with coded context trees, Journal of Automated Reasoning 32 (2004) 103–120.
[26] P. Rao, C. R. Ramakrishnan, I. V. Ramakrishnan, A thread in time saves tabling time, in: Proceedings of the Joint International Conference and Symposium on Logic Programming (ICSLP), The MIT Press, 1996, pp. 112–126.
[27] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1999.
[28] E. Harzheim, Ordered Sets, Springer, 2006.
[29] J. A. Robinson, A machine-oriented logic based on the Resolution Principle, Journal of the ACM 12 (1965) 23–41.
[30] H.-J. Bürckert, Matching – a special case of unification?, Journal of Symbolic Computation 8 (1989) 523–536.
[31] F. Baader, W. Snyder, Handbook of Automated Reasoning, Elsevier and MIT Press, 2001, pp. 445–532.
[32] T. Prokosch, F. Bry, Unification on the run, in: T. Kutsia, A. M. Marshall (Eds.), The 34th International Workshop on Unification (UNIF'20), number 20-10 in RISC Report Series, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2020, pp. 13:1–13:5. URL: https://www.risc.jku.at/publications/download/risc_6129/proceedings-UNIF2020.pdf.