A Low-Level Index for Distributed Logic Programming

Thomas Prokosch
Institute for Informatics, Ludwig-Maximilian University of Munich, Germany
prokosch@pms.ifi.lmu.de

A distributed logic programming language with support for meta-programming and stream processing offers a variety of interesting research problems, such as: How can a versatile and stable data structure for the indexing of a large number of expressions be implemented with simple low-level data structures? Can low-level programming help to reduce the number of occur checks in Robinson's unification algorithm? This article gives the answers.

1 Introduction and problem description

Logic programming originated in the 1970s as a result on work in artificial intelligence and automated theorem proving [15, 21]. One important concept of logic programming always stood out: The clear separation between the logic component and the control component of a program [22]. In today's computing landscape, where large amounts of (possibly streamed) data and distributed systems with parallel processors are the norm, it becomes increasingly hard to program in an imperative style where logic and control are intermingled.

Therefore, it is worthwhile to investigate how a logic programming language could deal with large amounts of streamed and non-streamed data in a way such that it can adapt itself to changing circumstances such as network outages ("meta-programming"). Creating such a programming language is the primary drive behind the author's line of research.

The main components of a distributed logic programming language are

- a stable indexing data structure to store large amounts of expressions,
- a low-level unification algorithm with almost linear performance, and
- a distributed forward-chaining resolution-based inference engine.

Some of these components have already been investigated; the current status of the research is summarized in this article. The missing parts are outlined in section 6.

2 Logical foundations

This section introduces standard algebraic terminology and is based on [31, 30].

Let $v_0, v_1, v_2, \dots$ denote infinitely many variables, letters $a, b, c, \dots$ (except v) denote finitely many non-variable symbols. v^i (with a superscript) denotes an arbitrary variable.

An *expression* is either a first-order term or a first-order formula. Expressions are defined as follows: A variable is an expression. A nullary expression constructor c consisting of the single non-variable symbol c is an expression. If $e_1, \dots, e_n$ are expressions then $c(e_1, \dots, e_n)$ is an expression with expression constructor c and arity n.

The fusion of the two distinct entities term and formula may seem unusual at first glance. This perspective, however, is convenient for meta-programming: Meta-programming is concerned with the

F. Ricca, A. Russo et al. (Eds.): Proc. 36th International Conference
on Logic Programming (Technical Communications) 2020 (ICLP 2020)
EPTCS 325, 2020, pp. 303–312, doi:10.4204/EPTCS.325.40

generation and/or modification of program code through program code. Thus, meta-programming applied to logic programming may require the modification of formulas through functions which may be difficult to achieve when there is a strict distinction between terms and formulas. Commonly, a so-called *quotation* is used to maintain such a distinction when it is necessary to allow formulas to occur inside of terms. However, it was shown [4, 18, 20] that it is not necessary to preserve such a distinction and that by removing it, the resulting language is a conservative extension of first-order logic [2].

Let E denote the set of expressions and V the set of variables. A substitution σ is a total function $V \to E$ of the form $\sigma = \{v^1 \mapsto e_1, \ldots, v^n \mapsto e_n\}$, $n \geq 0$ such that $v^1, \ldots, v^n$ are pairwise distinct variables, and $\forall i \in \{1, \ldots, n\}$ $\sigma(v^i) = e_i$, and $\sigma(v) = v$ if $v \neq v^i$. A substitution σ is a renaming substitution iff σ is a permutation of variables, that is $\{v^i \mid 1 \leq i \leq n\} = \{e_i \mid 1 \leq i \leq n\}$. σ is a renaming substitution for an expression e iff $\{e_i \mid 1 \leq i \leq n\} \subseteq V$ and for all distinct variables v^j, v^k in e the inequality $\sigma(v^j) \neq \sigma(v^k)$ holds.

The application of a substitution σ to an expression e, written $\sigma(e)$, is defined as the usual function application, i.e. all variables v^i in e are simultaneously substituted with expressions $\sigma(v^i)$. The application of a substitution σ to a substitution τ, written $\sigma\tau$, is defined as $(\sigma\tau)x = \sigma(\tau(x))$.

3 Low-level representations

One of the most important aspects in designing efficient algorithms is finding a good in-memory representation of the key data structures. The in-memory representation of variables, expressions, and substitutions described in this section has already been published in [31, 30] and is based on the prefix notation of expressions. The prefix notation is a flat representation without parentheses; the lack of parentheses makes this representation especially suited for the flat memory address space of common hardware. For example, the prefix notation of the expression $f(a, v_1, g(b), v_2, v_2)$ is $f/5\ a/0\ v_1\ g/1\ b/0\ v_2\ v_2$.

A similar but distinct expression representation is the flatterm representation [5, 6].

3.1 Representation of expressions

An expression representation that is particularly suitable for a run-time system of a logic programming language is as follows: Each expression constructor is stored as a compound of its symbol s and its arity n. Each variable either stores the special value `nil` if the variable is unbound or a pointer if the variable is bound. It is worth stressing that the name of a variable is irrelevant since its memory address is sufficient to uniquely identify a variable. Two distinct expression representations do not share variables.

In order to be able to represent non-linear expressions, i.e. expressions in which a variable occurs more than once, two kinds of variables need to be distinguished: Non-offset variables and offset variables. The first occurrence of a variable is always a non-offset variable, represented as described above. All following occurrences of this variable are offset variables and are represented by a pointer to the memory address of the variable's first occurrence. Care must be taken when setting the value of an offset variable: Not the memory cell of the offset variable is modified but the memory cell of the base variable it refers to.

The type of the memory cell (i.e. expression constructor `cons`, non-offset variable `novar`, or offset variable `ofvar`) is stored as a three-valued flag at the beginning of the memory cell. Assuming that a memory cell has a size of 4 bytes, a faithful representation of the expression $f(a, v_1, g(b), v_2, v_2)$ starting at memory address 0 is:

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
cons	$f/5$	cons	$a/0$	novar	nil	cons	$g/1$	cons	$b/0$	novar	nil	ofvar	4

The offset variable at address 24 contains the value 4 which must be subtracted from its address yielding 20, the address of the base variable the offset variable refers to.

Reading an in-memory expression representation involves traversing the memory cells from left to right while keeping a counter of the number of memory cells still to read. Each read memory cell decreases this counter by one, and the arities of expression constructors are added to the counter. Eventually, the counter will drop to zero which means that the expression has been read in its entirety.

In subsequent examples the expression representation is simplified to not include type flags.

3.2 Representation of substitutions and substitution application

An elementary substitution $\{v^i \mapsto e\}$ is represented as a tuple of two memory addresses, the address of the variable v^i and the address of the first memory cell of the expression representation of e. A substitution is represented as a list of tuples of addresses. Assume the representation of the expression $f(a, v_1, g(b), v_2, v_2)$ starts at address 0 and the representation of the expression $h(a, v_3)$ at address 36, then the substitution $\{v_2 \mapsto h(a, v_3)\}$ is represented as the tuple $(20, 36)$:

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 35	36 37 38 39	40 41 42 43	44 45 46 47
$f/5$	$a/0$	nil	$g/1$	$b/0$	nil	4	$\cdots$	$h/2$	$a/0$	*nil*

Substitution application simply consists of setting the contents of the memory cell of the variable to the address of the expression representation to be substituted. After the substitution application $f(a, v_1, g(b), v_2, v_2)\{v_2 \mapsto h(a, v_3)\}$ memory contents is:

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 35	36 37 38 39	40 41 42 43	44 45 46 47
$f/5$	$a/0$	nil	$g/1$	$b/0$	36	4	$\cdots$	$h/2$	$a/0$	*nil*

Observe that the contents of the offset variable at address 24 keeps its offset 4 unchanged, and that the contents of the non-offset variable at address 20 contains an absolute address.

4 Storage and retrieval

Automated reasoning [34] relies upon the efficient storage and retrieval of expressions. Standard data structures such as lists or hash tables can be used for this task but more specialized data structures, known as term indexing [14, 36] data structures, can significantly improve the retrieval speed of expressions [6, 36, 35]. Depending on the application, certain characteristics of a term indexing data structure are beneficial. For meta-programming [2] the retrieval of expressions unifiable with a query as well as retrieval of instances, generalizations, and variants of a query are desirable. For tabling [39, 32, 19], a form of memoing used in logic programming, the retrieval of variants and generalizations of queries needs to be well supported. In this section, which is based upon already published work [30], *instance tries* are proposed. Instance tries are trees which offer a few conspicuous properties such as:

- Stability. Instance tries are stable in the sense that the order of insertions into and removals from the data structure does not determine its shape.
- Versatility. Instance tries support the retrieval of generalizations, variants, and instances of a query as well as those expressions unifiable with a query.

- Incrementality. Instance tries are based upon the instance relationship which allows for incremental unification during expression retrieval.
- Perfect filtering. Some term indexing data structures require that the results of a query are post-processed [14]. Instance tries do not require such post-processing because querying an instance trie always returns perfect results.

4.1 Review of related work

Term indexing data structures are surveyed in the book [14] and in the book chapter [36] the latter containing some additional data structures which did not exist when the former was written. The latter does not describe dated term indexing data structures.

Tries were invented in 1959 for information re*trie*val [1] while the name itself was coined one year later [9]. Tries exhibit a more conservative memory usage than lists or hash tables due to the fact that common word prefixes are shared and thus stored only once.

Coordinate indexing [16] and path indexing [38] consider positions (or sequences of positions, respectively, so-called *paths*) of symbols in a term with the goal of subdividing the set of terms into subsets. Both coordinate indexing and path indexing disregard variables in order to further lower memory consumption making them non-perfect filters: Subject to these limitations, terms $f(v_0, v_1)$ and $f(v_0, v_0)$ are considered to be equal which means that results returned from a query need to be post-processed to identify the false positives. Several variations of path indexing, such as Dynamic Path Indexing [23] and Extended Path Indexing [12] have been proposed, none of which are stable or perfect filters.

Discrimination trees [25, 26] (with their variants Deterministic Discrimination Trees [11] and Adaptive Discrimination Trees [37]) were proposed as the first tree data structures particularly designed for term storage. However, all of them are non-perfect filters, a shortcoming that Abstraction Trees [27] were able to remedy. Substitution Trees [13] and Downward Substitution Trees [17] further refine the idea of abstraction trees and have been recently extended to also support indexing of higher order terms [29].

While Code Trees [40] and Coded Context Trees [10] are also frequently used in automated theorem provers both data structures are not versatile according to the characterization above.

4.2 Order on expressions

A total order on expressions $\leq_e$ is lexicographically derived from the total order $\leq_c$:

- $v_1 <_c v_2$ for variables v_1, v_2 with an order $\leq_v$ such that $v_1 <_v v_2$,
- $v <_c s/a$ for variable v and non-variable symbol s with arity a,
- $s/a_1 <_c s/a_2$ for non-variable symbol s and $a_1 < a_2$,
- $s_1/a_1 <_c s_2/a_2$ for non-variable symbols s_1, s_2 with an order $\leq_{nv}$ such that $s_1 <_{nv} s_2$.

4.3 Matching and unification modes

A versatile term indexing data structure needs to support the retrieval of expressions that are more general than, a variant of, an instance of, or unifiable with a query. For the construction and querying of instance tries, however, mutually exclusive definitions are required. Expression e_1 is a variant (VR) of expression e_2 iff there exists a renaming substitution ρ for e_1 such that $e_1\rho = e_2$. e_1 is strictly more general (SG) than e_2 and e_2 is a strict instance (SI) of e_1 iff σ is a non-renaming substitution for e_1 such that $e_1\sigma = e_2$. e_1 and e_2 are only unifiable (OU) iff $\exists\sigma\ (e_1\sigma = e_2\sigma)$, σ is most general, and σ is not a renaming substitution for

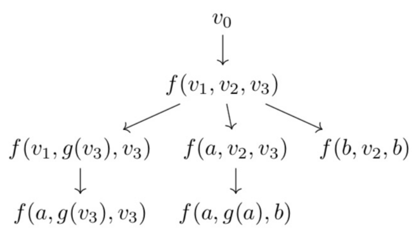

Figure 1: Example of an instance trie

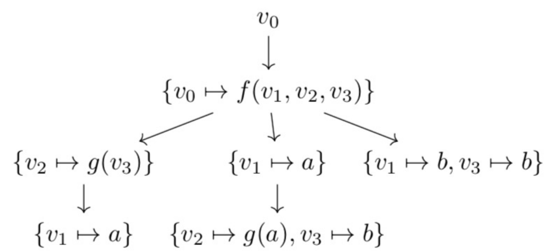

Figure 2: Instance trie using substitutions

both e_1 and e_2. e_1 and e_2 are non-unifiable (NU) iff $\forall\sigma\ (e_1\sigma \neq e_2\sigma)$. A matching-unification algorithm that is able to determine the mode for two expressions is given in Section 5.

4.4 Instance tries

An instance trie is a tree T such that

- Every node of T except the root stores an expression.
- Every child N_c of a node N in T is a strict instance of N.
- Siblings N_s in T are ordered by $<_e$ as described above.

Figure 1 shows an example of an instance trie storing six expressions. Figure 2 shows the same trie using substitutions; this alternate representation is possible due to the strict instance relation between a node and its children. In this representation, repeated application of substitutions to the root variable v_0 along a path yields the corresponding expression. The use of substitutions gives two advantages: First, common symbols are shared further reducing memory consumption. Second, querying an instance trie can make use of incremental unification resulting in faster retrieval.

Retrieving expressions from an instance trie T requires the following steps:

1. Top-down left-to-right traversal of the tree.
2. Expression e of each node N is unified with the query q to determine the matching and unification mode of e and q as outlined above.

All query modes except unification affect the traversal:

- Variant: The traversal can be interrupted as soon as an answer is found.
- Instance: If e is an instance of q then the sub-tree rooted at N does not need to be traversed: All children of N necessarily store instances of q.
- Generalization: The traversal can ignore all child nodes of a node which is not strictly more general than q.

Insertion of an expression e into an instance trie T involves first searching for a node N such that the expression of N is more general than e and the expressions of all children of N are not more general than e. If e is a variant of the expression in N then nothing is done. Otherwise, a new node N' containing expression e is inserted as a child of N (at the correct position among its siblings according to the order $<_e$ defined above) and instances of e (found to the right of N') are inserted below N'.

Deletion of an expression e from an instance trie T requires retrieving the node N containing expression e. If such a node is found then this node N is deleted and, after this deletion, each child node of N is inserted into the node which, before the deletion, was the parent node of N.

5 Low-level unification

Unification, that is determining whether a pair of expressions has a most general unifier (MGU), is an integral part of every automated reasoning system and every logic programming language. Nevertheless, only little attention has been given to potential improvements which develop their full effect at machine level or in an interpreter run-time. This section, based on previously published work [31], outlines a unification algorithm which has been specifically developed for such an environment.

5.1 Review of related work

Since Robinson introduced unification [33], a wealth of research has been carried out on this subject [28, 3, 24, 8]. Nevertheless, only few algorithms are used in practice not only because more sophisticated algorithms are harder to implement but also, unexpectedly, Robinson's unification algorithm is still the most efficient [17]! Consequently, Robinson's unification algorithm has been chosen as a starting point for the following unification algorithm.

5.2 A matching-unification algorithm

The algorithm `unif(e1, e2)` performs a left-to-right traversal of the representation of expressions e_1 and e_2 whose first addresses are `e1` and `e2`, respectively. Let `c`, `c1`, `c2` be addresses of memory cells in the representation of e_1 or e_2. In each step of the algorithm two memory cells are compared based on type and content using the following functions:

- `type(c)`: Returns the type of the value stored at `c`, resulting in `cons`, `novar`, or `ofvar`.
- `value(c)`: Value stored in memory cell `c`.
- `arity(c)`: Arity of the constructor stored in memory cell `c`, or 0.
- `deref(c, S)`: Creates a new expression from the expression representation at `c` and applies substitution `S` to it.
- `occurs-in(c1, c2)`: Checks whether variable at `c1` occurs in expression at `c2`.

The algorithm sets and uses the following variables: `A` (short for "answer") is initialized with `VR` and contains `VR`, `SG`, `SI`, `OU`, or `NU`. Variables `R1` and `R2`, both initialized with 1, contain the number of remaining memory cells to read. The algorithm terminates if `R1 = 0` and `R2 = 0`. `S1` and `S2` contain substitutions for variables in the expression representations of e_1, e_2 and are initialized with the empty lists `S1 := []`, `S2 := []`.

In each step of the algorithm two memory cells `c1` and `c2` are compared (starting with `e1` and `e2`, respectively), with the following four possibilities for each cell, resulting in a total of 16 cases: `type(ei) = cons`, `type(ei) = novar`, `type(ei) = ofvar && deref(ei, Si) != nil`, and `type(ei) = ofvar && deref(ei, Si) = nil`.

Table 1 shows the core of the matching-unification algorithm. For clarity and because the table is symmetric along its principal diagonal only the top-right half of the table contains entries. For space reasons, the table is abbreviated; for the full table refer to [31].

5.3 Illustration of the matching-unification algorithm

An example should illustrate how the matching-unification algorithm works: Expression $f(v_1, v_1)$ at address `e1=0` shall be unified with expression $f(a, a)$ at address `e2=20`:

case	`cons`	`novar`	`ofvar && deref!=nil`	`ofvar && deref=nil`
`cons`	continue or NU	bind, change mode	dereference recursive call	*occurs check* (bind and OU) or NU
`novar`		bind to left	bind, change mode	bind, change mode
`ofvar &&` `deref!=nil`			dereference recursive call	*occurs check* (bind and OU) or NU
`ofvar &&` `deref =nil`				bind

Table 1: Core of the matching-unification algorithm, abbreviated

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
$f/2$	nil	4	$\cdots$	$f/2$	$a/0$	$a/0$

1. Initialization: `A := VR; R1 := 1; R2 := 1; S1 := []; S2 := []`
2. `type(e1) = cons, type(e2) = cons, value(e1) = value(e2)`

   Constructors $f/2$ and $f/2$ match with arity 2: `R1 := R1+2 = 3; R2 := R2+2 = 3`
3. Continue to next cell: Each cell consists of 4 bytes.

   `e1 := e1+4 = 4; e2 := e2+4 = 24; R1 := R1-1 = 2; R2 := R2-1 = 2`
4. `type(e1) = novar, type(e2) = cons`

   First, the non-offset variable at address `e1=4` needs to be bound to the sub-expression starting with the constructor $a/0$ at address `e2=24` by adding the tuple $(4,24)$ to the substitution `S1 := [(4, 24)]`. Note that no occurs check is required when introducing this binding; this is a speed improvement with respect to some other unification algorithms such as Robinson's algorithm [33] or the algorithm from Martelli-Montanari [24].

   Then, change the mode by setting `A:=SG` since the non-offset variable at `e1=4` is strictly more general than the expression constructor $a/0$ at `e2=24`.
5. Continue to next cell: Each cell consists of 4 bytes.

   `e1 := e1+4 = 8 ; e2 := e2+4 = 28; R1 := R1-1 = 1; R2 := R2-1 = 1`
6. `type(e1) = ofvar, type(e2) = cons`

   First, dereference `e1` with `S1` yielding address 24. Dereferencing address `e2=28` yields 28. (The memory cell at address 28 contains a constructor.) Then, call the algorithm recursively with addresses 24 and 28.
7. The recursive call confirms the equality of the expression constructors $a/0$ at address 24 and $a/0$ at address 28, returning to the caller without any changes to variables `A`, `R1`, `R2`.
8. Continue to next cell: Each cell consists of 4 bytes.

   `e1 := e1+4 = 12 ; e2 := e2+4 = 32; R1 := R1-1 = 0; R2 := R2-1 = 0`

   The algorithm terminates because `R1 = 0` and `R2 = 0`. The result `A = SG` and `S1 = [(4, 24)]`, `S2 = []` is returned to the caller. The result is correct since $f(v_1,v_1)$ is strictly more general than $f(a,a)$.

6 Open issues and goals

While some progress towards a distributed logic programming language has already been made, there are still further challenges:

- Instance tries have been fully specified and their implementation is currently ongoing. Upon completion an empirical evaluation of instance tries together with a variety of common term indexing data structures need to verify the expected speed-up of instance tries.
- It is currently being investigated how stream processing can be integrated with logic programming.
- The forward-chaining resolution engine to derive the immediate consequences from a set of expressions in order to perform program evaluation has not been investigated so far.

Follow-up articles will report on each of those aspects of research.

References

[1] René de la Briandais (1959): *File searching using variable length keys*. In: *Proceedings of the Western Joint Computer Conference*, pp. 295–298, doi:10.1145/1457838.1457895.

[2] François Bry (2020): *In Praise of Impredicativity: A Contribution to the Formalization of Meta-Programming*. *Theory and Practice of Logic Programming* 20(1), pp. 99–146, doi:10.1017/S1471068419000024. Available at `https://pms.ifi.lmu.de/publications/PMS-FB/PMS-FB-2018-2/PMS-FB-2018-2-paper-second-revision.pdf`.

[3] Dennis de Champeaux (1986): *About the Paterson-Wegman Linear Unification Algorithm*. *Journal of Computer and System Sciences* 32(1), pp. 79–90, doi:10.1016/0022-0000(86)90003-6.

[4] Weidong Chen, Michael Kifer & David Scott Warren (1993): *HILOG: A Foundation for Higher-Order Logic Programming*. *Journal of Logic Programming* 15(3), pp. 187–230, doi:10.1016/0743-1066(93)90039-J.

[5] Jim Christian (1989): *Fast Knuth-Bendix Completion: Summary*. In Nachum Dershowitz, editor: *Rewriting Techniques and Applications, 3rd International Conference (RTA'89)*, *LNCS* 355, Springer, pp. 551–555, doi:10.1007/3-540-51081-8_136.

[6] Jim Christian (1993): *Flatterms, Discrimination Nets, and Fast Term Rewriting*. *Journal of Automated Reasoning* 10(1), pp. 95–113, doi:10.1007/BF00881866.

[7] Maarten H. van Emden & Robert A. Kowalski (1976): *The Semantics of Predicate Logic as a Programming Language*. *Journal of the ACM* 23(4), pp. 733–742, doi:10.1145/321978.321991.

[8] Gonzalo Escalada-Imaz & Malik Ghallab (1988): *A Practically Efficient and Almost Linear Unification Algorithm*. *Artificial Intelligence* 36(2), pp. 249–263, doi:10.1016/0004-3702(88)90005-7.

[9] Edward Fredkin (1960): *Trie Memory*. *Communications of the ACM* 3(9), pp. 490–499, doi:10.1145/367390.367400.

[10] Harald Ganzinger, Robert Nieuwenhuis & Pilar Nivela (2004): *Fast Term Indexing with Coded Context Trees*. *Journal of Automated Reasoning* 32(2), pp. 103–120, doi:10.1023/B:JARS.0000029963.64213.ac.

[11] Albert Gräf (1991): *Left-to-Right Tree Pattern Matching*. In Ronald V. Book, editor: *Rewriting Techniques and Applications, 4th International Conference (RTA'91)*, *LNCS* 488, Springer, pp. 323–334, doi:10.1007/3-540-53904-2_107.

[12] Peter Graf (1994): *Extended Path-Indexing*. In Alan Bundy, editor: *2nd Conference on Automated Deduction (CADE)*, *LNCS* 814, Springer, pp. 514–528, doi:10.1007/3-540-58156-1_37.

[13] Peter Graf (1995): *Substitution Tree Indexing*. In Jieh Hsiang, editor: *6th International Conference on Rewriting Techniques and Applications (RTA'95)*, *LNCS* 914, Springer, pp. 117–131, doi:10.1007/3-540-59200-8_52.

[14] Peter Graf (1995): *Term Indexing*. *LNCS* 1053, Springer, doi:10.1007/3-540-61040-5.

[15] C. Cordell Green (1969): *Application of Theorem Proving to Problem Solving*. In Donald E. Walker & Lewis M. Norton, editors: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, William Kaufmann, pp. 219–240. Available at `http://ijcai.org/Proceedings/69/Papers/023.pdf`.

[16] Carl Hewitt (1972): *Description and Theoretical Analysis (Using Schemata) of Planner*. Ph.D. thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology. Available at `http://hdl.handle.net/1721.1/6916`.

[17] Kryštof Hoder & Andrei Voronkov (2009): *Comparing Unification Algorithms in First-Order Theorem Proving*. In Bärbel Mertsching, Marcus Hund & Muhammad Zaheer Aziz, editors: *KI 2009: Advances in Artificial Intelligence (KI'09)*, *LNCS* 5803, Paderborn, Germany, pp. 435–443, doi:10.1007/978-3-642-04617-9_55.

[18] Yuejun Jiang (1994): *Ambivalent Logic as the Semantic Basis of Metalogic Programming*. In Pascal Van Hentenryck, editor: *Logic Programming, Proceedings of the Eleventh International Conference*, MIT Press, Santa Marherita Ligure, Italy, pp. 387–401.

[19] Ernie Johnson, C. R. Ramakrishnan, I. V. Ramakrishnan & Prasad Rao (1999): *A Space Efficient Engine for Subsumption-Based Tabled Evaluation of Logic Programs*. In Aart Middeldorp & Taisuke Sato, editors: *Functional and Logic Programming, 4th Fuji International Symposium (FLOPS'99)*, *LNCS* 1722, Springer, Tsukuba, Japan, pp. 284–300, doi:10.1007/10705424_19.

[20] Marianne B. Kalsbeek & Yuejun Jiang (1995): *Meta-Logics and Logic Programming*, chapter A Vademecum of Ambivalent Logic, pp. 27–56. Computation and Complexity Theory, MIT Press. Available at `https://www.illc.uva.nl/Research/Publications/Reports/CT-1995-01.text.pdf`.

[21] Robert A. Kowalski (1974): *Predicate Logic as Programming Language*. In Jack L. Rosenfeld, editor: *Information Processing, Proceedings of the 6th IFIP Congress*, North-Holland, pp. 569–574. Available at `http://www.doc.ic.ac.uk/~rak/papers/IFIP%2074.pdf`.

[22] Robert A. Kowalski (1979): *Algorithm = Logic + Control*. *Communication of the ACM* 22(7), pp. 424–436, doi:10.1145/359131.359136.

[23] Reinhold Letz, Johann Schumann, Stefan Bayerl & Wolfgang Bibel (1992): *SETHEO: A high-performance theorem prover*. *Journal of Automated Reasoning* 8(2), pp. 183–212, doi:10.1007/BF00244282.

[24] Alberto Martelli & Ugo Montanari (1982): *An Efficient Unification Algorithm*. *ACM Transaction on Programming Language Systems (TOPLAS'82)* 4(2), pp. 258–282, doi:10.1145/357162.357169.

[25] William McCune (1988): *An indexing mechanism for finding more general formulas*. *Association for Automated Reasoning Newsletter* 9.

[26] William McCune (1992): *Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval*. *Journal of Automated Reasoning* 9, pp. 147–167, doi:10.1007/BF00245458.

[27] Hans Jürgen Ohlbach (1990): *Abstraction Tree Indexing for Terms*. In: *9th European Conference on Artificial Intelligence (ECAI'90)*, pp. 479–484.

[28] Michael Stewart Paterson & M.N. Wegman (1978): *Linear Unification*. *Journal of Computer and System Sciences* 16(2), pp. 158–167, doi:10.1016/0022-0000(78)90043-0.

[29] Brigitte Pientka (2009): *Higher-Order Term Indexing Using Substitution Trees*. *ACM Transactions on Computational Logic (TOCL)* 11(1), pp. 6:1–6:40, doi:10.1145/1614431.1614437.

[30] Thomas Prokosch & François Bry (2020): *Give Reasoning a Trie*. In: *7th Workshop on Practical Aspects of Automated Reasoning (PAAR'20)*, CEUR Workshop Proceedings, Aachen. To appear.

[31] Thomas Prokosch & François Bry (2020): *Unification on the Run*. In Temur Kutsia & Andrew M. Marshall, editors: *The 34th International Workshop on Unification (UNIF'20)*, *RISC Report Series* 20-10, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, pp. 13:1–13:5. Available at `https://www.risc.jku.at/publications/download/risc_6129/proceedings-UNIF2020.pdf`.

[32] I. V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift & David Scott Warren (1999): *Efficient Access Mechanisms for Tabled Logic Programs*. *Journal of Logic Programming* 38(1), pp. 31–54, doi:10.1016/S0743-1066(98)10013-4.

[33] John Alan Robinson (1965): *A Machine-Oriented Logic Based on the Resolution Principle*. *Journal of the ACM* 12(1), pp. 23–41, doi:10.1145/321250.321253.

[34] John Alan Robinson & Andrei Voronkov, editors (2001): *Handbook of Automated Reasoning*. Elsevier Science Publishers.

[35] Stephan Schulz & Adam Pease (2020): *Teaching Automated Theorem Proving by Example: PyRes 1.2*. In Nicolas Peltier & Viorica Sofronie-Stokkermans, editors: *Automated Reasoning - 10th International Joint Conference (IJCAR'20), Proceedings, Part II*, *LNCS* 12167, Springer, pp. 158–166, doi:10.1007/978-3-030-51054-1_9.

[36] R. Sekar, I. V. Ramakrishnan & Andrei Voronkov (2001): *Handbook of Automated Reasoning*, chapter Term Indexing, pp. 1853–1964. 2, Elsevier Science Publishers, doi:10.1016/B978-044450813-3/50028-X. Available at `http://www.cs.man.ac.uk/~voronkov/papers/handbookar_termindexing.ps`.

[37] R. C. Sekar, R. Ramesh & I. V. Ramakrishnan (1992): *Adaptive Pattern Matching*. In Werner Kuich, editor: *Automata, Languages and Programming, 19th International Colloquium (ICALP'92)*, *LNCS* 623, Springer, pp. 247–260, doi:10.1007/3-540-55719-9_78.

[38] Mark E. Stickel (1989): *The Path-Indexing Method For Indexing Terms*. Technical Note 473, SRI International, Menlo Park, California, USA. Available at `https://www.sri.com/wp-content/uploads/pdf/498.pdf`.

[39] Hisao Tamaki & Taisuke Sato (1986): *OLD Resolution with Tabulation*. In Ehud Shapiro, editor: *Third International Conference on Logic Programming (LP'86)*, *LNCS* 225, Springer, Imperial College of Science and Technology, London, UK, pp. 84–98, doi:10.1007/3-540-16492-8_66.

[40] Andrei Voronkov (1995): *The Anatomy of Vampire*. *Journal of Automated Reasoning* 15(2), pp. 237–265, doi:10.1007/BF00881918.