Efficient Multi-Scale Indexing with Dynamic IntMaps

Albert Eisfeld[1], Stephan Schulz[1]

[1]DHBW Stuttgart, Stuttgart, Germany

Abstract

Term and clause indexing are central technologies for efficient first-order theorem provers and similar reasoning systems. While there is a selection of basic technologies, most of them require an efficient mapping from function symbols (usually encoded as small integers) or other integer values to branches of a tree structure. With today's wide variety of application problems, both the range of possible values and the cardinality of each individual map vary wildly. We present a data structure that is designed to be efficient in time and space at all scales and key distributions. Experiments show very good scalability and a modest improvement over the old, already quite refined data structure in E.

Keywords

term indexing, data structures, theorem proving,

1. Introduction

Most powerful modern theorem provers like E [1], Vampire [2] and iProver [3] are based on saturating calculi, in particular variations of superposition [4] and related calculi. These calculi represent the proof state as a set of clauses. They enrich the proof state by adding logical consequences of existing clauses, with the aim of eventually generating the empty clause as an explicit witness of unsatisfiability (and hence conclude a proof by contradiction). In addition to generating inferences, simplifying inferences like subsumption and rewriting are critical for success. These techniques either remove redundant clauses or replace them with simpler ones, and typically take up most of the time spent by the inference engine.

Successful provers combine efficient execution and intelligent proof search heuristics to achieve their power. Especially for harder problems with large search states, *term indexing* is a critical technology. Indexing of terms (and of clauses) allows a system to quickly identify potential inference partners for generating inferences like paramodulation/superposition or resolution, but in particular also for simplification techniques like rewriting and subsumption.

Most modern term indices are based on (often trie-like) trees. At each node in the tree, branches labelled with function symbols (or variables) lead to sub-trees. A complete branch of the tree represents all terms compatible with a certain query, and the leaves represent sets of terms (or clauses) that are potential answers to this query. In this paper, we are in particular concerned with the efficiency of mapping function (or variable) symbols to tree branches. However, the developed solutions have more general applications, both in indexing and beyond.

2. Background and Motivation

Early contributions to indexing were made by Stickel [5], who introduced *path indexing*, and by McCune, who integrated path indexing and *discrimination tree indexing* into Otter [6], creating the first successful modern saturating prover. Our own prover E uses *perfect discrimination tree indexing* for forward rewriting, *fingerprint indexing* [7] for backwards rewriting and superposition/paramodulation, and *feature vector indexing* [8] for subsumption and related techniques.

Practical Aspects of Automated Reasoning (PAAR) 2026, as part of the Federated Logic Conference (FLoC) 2026, Lisbon, Portugal, July 25, 2026

inf23040@lehre.dhbw-stuttgart.de (A. Eisfeld); schulz@eprover.org (S. Schulz)

0000-0001-6262-8555 (S. Schulz)

When indexing techniques were first developed, most problems for automated theorem provers were handcrafted, and small in number of axioms and signature size. Indexing data structures, mapping (mostly) from function symbols to branches of the index tree, hence usually had a low branching factor and also a very limited range of values to branch on. Implementations of these maps used either (potentially dynamic) arrays with direct mapping, or linear lists with linear search. Arrays are faster, but if the node is not densely populated, they may waste space. Linear lists have a higher per-key cost, both in speed and memory, but handle sparse key spaces efficiently.

In E, our original implementation used two dynamic arrays to handle each node — one for variable-indexed branches, one for function-symbol indexed branches. This worked well for most problems of the time. However, in 2003, Urban was converting the MIZAR library [9] to first-order logic [10, 11] and began experiments running E on the resulting problems. MIZAR problems have many thousands of function symbols, and the resulting large, but sparsely populated arrays led to an explosion of memory consumption by E's perfect discrimination tree indices (at that time the only indexing data structure used). Urban solved this by hacking the prover and replacing the function symbol array with a binary search tree. While this solved his immediate problem, it decreased performance for other classes of problems, and was not the optimal solution for all situations.

However, inspired by Urban's work, we replaced both the original solution and Urban's hack with a dedicated `IntMap` data type. Its abstract interface supports inserting and deleting key/value pairs, querying the value for a given key, and iteration over all or an arbitrary sub-range of key/value pairs stored in a given map.

Before we discuss the details of the original and the new `IntMap`, we present a couple of observations that shape the design. First, the proof state of saturating theorem provers tends to increase in size over time, and so does the number of indexed clauses. There are occasional cases where already indexed clauses become redundant and have to be removed from the index (*backward simplification*), but this is a rare occurrence. Typically, all indices (and all `IntMaps` used in them) tend to grow over time.

The most frequent operation by far is the query for a given key (and its associated value). Insertions are orders of magnitude more frequent than deletions. In-order iteration over key/value pairs is a frequent operation as well — more frequent than insertions, but less frequent than look-ups.

Our normal use cases are term and clause indices. These are typically arranged as trees, with each node of the tree representing a set of terms or clauses, and each branching splitting this set into disjoint subsets. Each branch is typically labelled by a small integer (the *key*), often representing the encoding of a function symbol (i.e. its index in the signature table) or a variable (typically a small negative number), and resulting from tests on the terms in that branch. As a result, the sets (and the branching factors) of the early nodes are often quite large, but as we follow the branches, the subsets and hence the branching factors get smaller and smaller. Earlier nodes are accessed much more often (dominating access times), but there are many more nodes deeper in the tree, which are thus contributing significantly to memory use.

While the branching factor goes down with depth, the range (i.e. the difference between the smallest and the largest key) only goes down statistically, and not to the same degree. We have fewer keys, but they are drawn from the same total range. This range varies a lot between problems. Variables are typically restricted to the low dozens. Function symbols (and predicate symbols) can vary from single operators to tens of thousands for problems from Mizar and SUMO [12], and over a million for example problems from Cyc [13].

These observations lead us to the conclusion that the mapping of keys (integers) to branches in the indices must be efficient in time and memory at many different scales.

3. The Evolution of IntMap

As stated above, our first implementation of branching in perfect discrimination trees relied on two separate dynamic arrays (a generic datatype already implemented in E's libraries), with direct mapping from positive `long int` keys to `void*` values. These supported only 0-based indexing. Urban replaced

the function symbol side of the index by NumTrees, another of E's built-in data types. NumTrees are one of several implementations of Splay trees [14] in E, in particular Splay trees that use long int keys and either long int or void* values. Splay trees are structurally normal binary search trees, and can be used as such. However, most operations that require key lookup rearrange the tree, to move the requested key or one of its neighbours to the root of the tree. As a result, Splay trees provide an amortised logarithmic cost guarantee, and in particular, if only a subset of n keys is frequently accessed, essentially behave like a tree of only these n keys.

We then introduced IntMap as an abstract data type. It supports insertion, deletion, lookup, and ordered iteration via an iterator data type with *allocate*, *next* and *free* operations.

This IntMap datatype had a polymorphic implementation, with 4 distinct cases:

- *Empty*: The initial state on creation. No key/value pair is stored. The iterator for an empty IntMap simply returns an unspecified key and value NULL, the marker for the end of iteration.
- *Single*: A frequent case in indexing are long branches created by a single indexed term. To accommodate this case efficiently, we provide an implementation that stores a single key and associated value. The iterator for this case uses a boolean value to indicate if this one key/value pair has been seen already. If not, it is returned, if so, it returns NULL (with an unspecified key) again.
- *Array*: Dense populations of multiple keys are stored in a PDRangeArray. This is a flexible dynamic array type from E's library. It allows indexing with long int values starting at an arbitrary (potentially negative) offset. This array automatically grows as needed, but never shrinks (remember that deletions are rare events). The iterator for the array case maintains the index of the next unseen array element. The *next* operation searches for the next non-NULL value in the array and returns the corresponding key/value pair, or NULL if it runs out of array elements.
- *Tree*: Sparse key/value populations are stored in a NumTree, as described above. The iterator uses a stack to describe the current state of iteration, and uses a non-destructive in-order (Left-Node-Right) traversal to access all nodes. It returns the stored key/value pair at each node, or NULL if it runs out of nodes.

In both of the multi-key-modes, the IntMap monitors population density. If the density falls below a certain threshold value, and the map is in array mode, it reorganises itself into tree mode. If the density in tree mode increases beyond a second (larger) threshold, it switches from tree to array mode. The difference in the two thresholds provides a hysteresis that avoids overly frequent reorganisations. In our implementation, we switch to array if more than one in 8 spaces are occupied, and we switch to tree if less than one in 4 spaces are occupied. When first switching to multi-key-form with density in the hysteresis range, array is preferred (on the assumption that the density is more likely to increase than to fall).

3.1. Recent Improvements

Observations have shown that the key-value-pairs often lie in the memory as clusters, so that there are some smaller regions which are significantly more densely populated than the overall set of all key-value-pairs considered for a tree. A possible reason for this is that in larger specifications, there are sub-theories mostly dealing with a subset of functions and relations, while also being tied into the common theory with some shared symbols. This is then reflected in indexing data structures for clauses from these sub-theories.

To accommodate this circumstance, the structure of the IntMap datatype was simplified to use only one case instead of the two *Array* and *Tree* for multi-key-modes, combining both of the older cases into one. Key/Value pairs are stored in a tree of smaller arrays. This new *Array-Tree* is based on the NumArrTree datatype, a binary tree that contains arrays with a size of 2^n entries in each node. Keys are effectively split into two parts, one part (the high bits) selecting the array in the tree, the other (the low n bits) the index of the value in the array.

Figure 1 shows the visualisation of an exemplary set of key-value-pairs. The number next to the node represents the key of the tree node and thus the high bits of each key in that array. The index in the array represents the low 2 bits of the key.

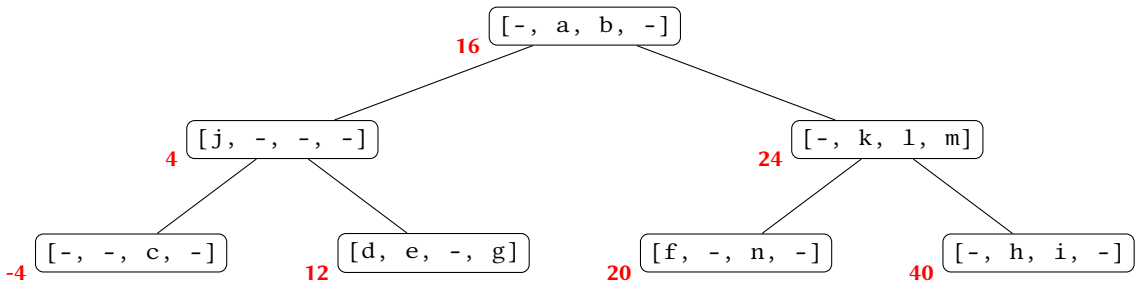

Figure 1: Visualisation of exemplary `NumArrTree` binary tree with an array size of 4. Values in the array that are marked with '-' represent `NULL`-values, i.e. these keys are not associated with any value.

Using the `NumArrTree` binary tree, the new `IntMap` datatype still supports insertion, deletion, lookup and ordered iteration as in the older version.

Similar to the *Tree* case, the ordered iteration uses a non-destructive in-order traversal. A stack is used to keep track of the current array in the tree. Because of the nested structure, iterating over the tree nodes alone would yield only values at index 0 of the array. To iterate over all key-value-pairs, the iterator uses an additional index counter to indicate the next unused entry in the current array. Iteration is hence done at two levels: First, the iterator looks for the next occupied entry in the current array. If it runs out of possibilities, the iterator moves to the next tree node and resets the internal index counter. In terms of memory, the iterator combines both the stack and the array index of the old model, thus using slightly more memory. However, iterators are rare objects, and the (dynamic) stack can typically be smaller, since there are fewer tree nodes.

Using the exemplary tree from Figure 1, an iterator would return the values in the following order: `↑c, ↑j, d, e, g, ↑a, b, ↑f, n, ↑k, l, m, ↑h, i`, where each of the ↑-occurences marks transition to a new tree node. The values have the keys depicted in Table 1.

Table 1
Key-value-pairs of exemplary `NumArrTree` binary tree in Figure 1

Value	c	j	d	e	g	a	b	f	n	k	l	m	h	i
Key	-2	4	12	13	15	17	18	20	22	25	26	27	41	42

The proposed `NumArrTree` binary tree comes with benefits for memory and runtime. Each node now can store multiple key-value-pairs while still needing only two words of memory for the left and right child-nodes to maintain the tree structure. Furthermore, the amount of operations for rearranging the tree is reduced on average, since fewer nodes can hold the same number of values. Additionally, reorganising the `IntMap` when the critical density is reached is now obsolete, since there is only one multi-key-mode.

Since the structure is optimised on the common case, it is expected to perform worse than the original structure in runtime and memory in edge cases for the key/value distribution. If the key-value-pairs are either very densely or very sparsely populated, organising the pairs in an array or a binary tree respectively will be optimal for memory and in the case of the array state, also in runtime. However, on average we expect better behaviour, and in particular better worst-case behaviour.

4. Experimental Results

To examine whether the changed data structure actually improves prover runtime and memory footprint, we ran the prover on various problems from the TPTP library [15], version 9.2.1. In order to provide a sufficient comparison, the theorem prover was built using the original state of the `IntMap` data structure

with two states for organising multiple key-value-pairs, as well as six versions of the improved `IntMap`, with array-sizes of 1, 2, 4, 8, 16 and 32.

All versions of the program were built using the gcc compiler with the optimisation flags `-O03 -fno-common`. The tests were done on a Linux server with Intel Xeon Platinum 8176 CPUs running at 2.10GHz under Ubuntu 24.04.3 LTS. The machine had 1.5TB physical RAM. The source code is available at https://github.com/aallbert/eprover.

To obtain the time used to find the proof for the corresponding problem, the Linux `time` command was used. All problems were run on their own core with a soft runtime limit of 300 seconds, i.e. the prover terminates itself in a controlled manner after the time limit, providing statistics. The command flags used are the following:

```
time taskset -c 32-63 ./eprover -auto -detsort-new -detsort-rw
-soft-cpu-limit=300 -cpu-limit=350 <problem>
```

We picked 158 problems from the First-order Theorems category of the CADE ATP System Competition [16] (years 2019–2025) to give us an interesting selection of problems with a good distribution of difficulties.

From these problems, we excluded all problems where all versions ran shorter than 5 seconds to avoid measurement noise. We also excluded problems where all versions ran longer than 300 seconds, since those problems indicate that the prover did not find a proof in time. If at least one version ran between 5 and 300 seconds, the problem is included in the analysis even if the other versions fall outside that interval.

This leaves 34 problems with valid data, where at least one version run meets the stated criteria. The number of times each version was the fastest in finding a proof and the total runtime for all 34 problems are depicted in Table 2.

Table 2
Number of times each version of the data structure was fastest (left) and total time spent on the 34 problems (right). Note that the *Standard Version* refers to the `IntMap` data structure that uses the binary tree and array to organise the key-value-pairs.

Array Size	Fastest Version
4	11
2	10
Standard Version	9
16	2
1	1
8	1
32	0

Array Size	Total Time spent(s)
8	3158.9
16	3234.8
Standard Version	3314.8
1	3325.2
32	3366.8
4	3390.1
2	3591.8

After looking at the runtime distribution of different functions in the program using a profiler, a maximum improvement of 2% was expected, since all functions related to the `IntMap` take up roughly 2% of the runtime for each problem. Table 2 shows that the differences in total runtime in the right table are unexpectedly larger than 2%. Additionally, in the set of 34 valid problems are some outliers where the time differences between versions are greater than 80%. This effect has occurred before, and indicates that the proof search (which can depend on in-memory allocation of pointers) took a different path. These outliers hence are not suitable to evaluate the quality of the data structures.

The results show that the versions with an array size of 1 and 32 perform worse than the Standard Version in both metrics. Surprisingly, the number of times a version solved a problem the fastest does not correlate to its overall runtime. As demonstrated in Table 2, the versions with an array size of 2 and 4 score high in the number of times they are the fastest, but low in the total time they spent on the problems. Note that versions 2 and 4 have more negative and positive outliers, i.e. they reach the 300 seconds run time limit where other versions complete the problem in under 120 seconds and vice

Table 3
Total time spent on the 34 valid problems where positive and negative outliers are *excluded*. Note that the *Standard Version* refers to the `IntMap` data structure that uses the binary tree and array to organise the key-value-pairs.

Array Size	Total Time spent (s)
2	2144.8
4	2211.3
8	2248.2
Standard Version	2251.8
1	2265.0
16	2283.1
32	2376.6

versa. When excluding positive and negative outliers, versions 2 and 4 perform much better in overall runtime compared to the Standard Version as shown in Table 3.

An array size of 8 provides a stable version that performs best in runtime and outperforms the Standard Version even when excluding the outliers. Figure 2 shows that the difference between both versions increases in favor of the improved version for problems that run longer than 60 seconds.

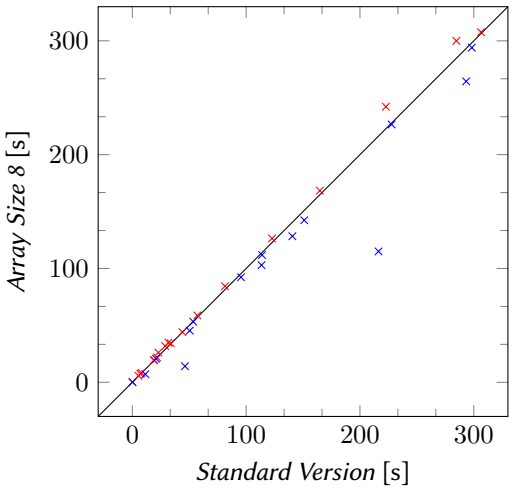

Figure 2: Runtime data of the valid problems for the Standard Version and for the improved version with an array size of 8. Red crosses indicate cases where the Standard Version was faster; blue crosses indicate cases where the improved version with an array size of 8 was faster.

For the 34 valid problems, the memory consumption, i.e. how much memory was allocated (in total, counting all allocations, not just the maximum memory used at any one time) to find a proof, was also monitored. The results are shown in Table 4.

Table 4
Number of times each version of the data structure consumed the least amount of memory (left) and total memory allocated on the 34 valid problems (right). Note that the *Standard Version* refers to the `IntMap` data structure that uses the binary tree and array to organise the key-value-pairs.

Array Size	Least Memory
Standard Version	20
8	9
32	4
16	1
4	0
2	0
1	0

Array Size	Total Memory allocated (GB)
Standard Version	4198.4
8	4360.3
32	4415.7
16	4468.6
4	4470.5
2	4561.6
1	4585.2

None of the improved versions show a smaller overall memory consumption compared to the standard version of `IntMap`. This suggests that the time savings do not directly correlate with the amount of memory consumed and are more dependent on functions that require heavy computation.

5. Future Work

Our current implementation is already performing quite well. There are several options to further improve the data structure, but none that is likely to make a significant impact in overall prover performance. The most attractive idea, if primarily from a simplicity and aesthetics perspective, would be to use dynamically sized array segments in the current array tree structure, i.e. to merge neighbouring arrays if the resulting larger array would be full enough, and likewise to split them if the population factor sinks below a threshold. This would allow an even more smooth transition between all-array and all-tree representations.

Another idea would be to analyse the problems on typical patterns and set the array size according to these patterns. Since some problems show drastic outliers, changing the array size after the prover read all axioms could have a significant impact on the runtime of some cases.

On the practical side, we will merge the current version of the code into the main development branch in the near future.

6. Conclusion

In older work, Nieuwenhuis, Hillenbrand, Riazanov and Voronkov have found that, back in 2001, the exact choice of indexing mechanism for a given retrieval operation was less important than the decision to pick any over none [17], and that details of implementation are not usually critical. Urban's experience shows that this did not hold up for all cases if the landscape of problems addressed by automated theorem provers changes significantly, e.g. with the introduction of much larger real-world problems (which lead, among other things, to the idea of a-priory premise selection and the *Workshop on Empirically Successful Automated Reasoning in Large Theories* (ESARLT) [18], held with CADE-21 in 2007 in Bremen). However, in this work the idea has reasserted itself — the new approach shows some advantages, but at least in our test cases, they seem fairly marginal in runtime. With the improved `IntMap` data structure, a slight but significant gain in speed has been achieved when considering all valid test-problems, although causing an increased memory footprint. Besides its impact on runtime, the improved `IntMap` also delivers a more elegant data structure, where switching the organisation between a binary tree and an array becomes obsolete.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] S. Schulz, S. Cruanes, P. Vukmirović, Faster, higher, stronger: E 2.3, in: P. Fontaine (Ed.), Proc. of the 27th CADE, Natal, Brasil, number 11716 in LNAI, Springer, 2019, pp. 495–507.

[2] F. Bártek, A. Bhayat, R. Coutelier, M. Hajdú, M. Hetzenberger, P. Hozzová, L. Kovács, J. Rath, M. Rawson, G. Reger, M. Suda, J. Schoisswohl, A. Voronkov, The Vampire diary, in: R. Piskac, Z. Rakamaric (Eds.), Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part III, volume 15933 of *Lecture Notes in Computer Science*, Springer, 2025, pp. 57–71. doi:10.1007/978-3-031-98682-6_4.

[3] A. Duarte, K. Korovin, Implementing superposition in iProver (system description), in: N. Peltier, V. Sofronie-Stokkermans (Eds.), Proc. of the 10th IJCAR, Paris (Part II), volume 12167 of *LNAI*, Springer, 2020, pp. 158–166.
[4] L. Bachmair, H. Ganzinger, Rewrite-Based Equational Theorem Proving with Selection and Simplification, Journal of Logic and Computation 3 (1994) 217–247.
[5] M. E. Stickel, The Path-Indexing Method for Indexing Terms, Technical Note 473, Artificial Intelligence Center, SRI International, Menlo Park, California, USA, 1989.
[6] W. McCune, Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval, Journal of Automated Reasoning 9 (1992) 147–167.
[7] S. Schulz, Fingerprint Indexing for Paramodulation and Rewriting, in: B. Gramlich, U. Sattler, D. Miller (Eds.), Proc. of the 6th IJCAR, Manchester, volume 7364 of *LNAI*, Springer, 2012, pp. 477–483.
[8] S. Schulz, Simple and Efficient Clause Subsumption with Feature Vector Indexing, in: M. P. Bonacina, M. E. Stickel (Eds.), Automated Reasoning and Mathematics: Essays in Memory of William W. McCune, volume 7788 of *LNAI*, Springer, 2013, pp. 45–67.
[9] A. Trybulec, H. A. Blair, Computer assisted reasoning with MIZAR, in: Proc. of the 9th IJCAI, Los Angeles, volume 85, Morgan Kaufmann, 1985, pp. 26–28.
[10] J. Urban, Translating Mizar for first order theorem provers, in: A. Asperti, B. Buchberger, J. H. Davenport (Eds.), Mathematical Knowledge Management, Springer, 2003, pp. 203–215.
[11] J. Urban, MPTP - motivation, implementation, first experiments, Journal of Automated Reasoning 33 (2004) 319–339. URL: https://doi.org/10.1007/s10817-004-6245-1. doi:10.1007/S10817-004-6245-1.
[12] I. Niles, A. Pease, Toward a Standard Upper Ontology, in: C. Welty, B. Smith (Eds.), Proc. 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001), 2001, pp. 2–9.
[13] D. Ramachandran, P. Reagan, K. Goolsbey, First-orderized ResearchCyc: Expressiveness and Efficiency in a Common Sense Knowledge Base, in: P. Shvaiko (Ed.), Proc. of the AAAI Workshop on Contexts and Ontologies: Theory, Practice and Applications (C&O-2005), 2005.
[14] D. Sleator, R. Tarjan, Self-Adjusting Binary Search Trees, Journal of the ACM 32 (1985) 652–686.
[15] G. Sutcliffe, The TPTP problem library and associated infrastructure - from CNF to TH0, TPTP v6.4.0, Journal of Automated Reasoning 59 (2017) 483–502. doi:10.1007/s10817-017-9407-7.
[16] G. Sutcliffe, The CADE ATP System Competition – CASC, AI Magazine 37 (2016) 99–101. doi:10.1609/aimag.v37i2.2620.
[17] R. Nieuwenhuis, T. Hillenbrand, A. Riazanov, A. Voronkov, On the Evaluation of Indexing Techniques for Theorem Proving, in: R. Goré, A. Leitsch, T. Nipkow (Eds.), Proc. of the 1st IJCAR, Siena, volume 2083 of *LNAI*, Springer, 2001, pp. 257–271.
[18] G. Sutcliffe, J. Urban, S. Schulz (Eds.), Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories, volume 257 of *CEUR Workshop Proceedings*, 2007.