HimML: Standard ML with Fast Sets and Maps

Jean Goubault

Bull S.A., rue Jean Jaurès, 78 340 Les Clayes-sous-Bois, France Tel: (33 1) 30 80 69 28

Jean.Goubault@frcl.bull.fr

Abstract

We propose to add sets and maps to Standard ML. Our implementation uses hash-tries to code them, yields fast general-purpose set-theoretic operations, and is based on a run-time where all equal objects are shared. We present evidence that this systematic use of hash-consing, and the use of hash-tries to code sets, provide good performance.

1 Introduction

Sets have been an adequate foundation for mathematics for nearly a century, and are also an important conceptual medium in computer science. Modern specification languages like VDM [18] and Z [30] are based on sets. But few programming languages provide general-purpose sets and maps: although they could be adequate for prototyping, it is feared that they would be too slow for real applications.

We have designed and implemented an extension of Standard ML [17], called HimML¹ [12] providing fast general (polymorphic) set-theoretic data-structures, and a comprehensive set of efficient operations on them.

After mentioning related work in set-theoretic languages and justifying our choice of Standard ML as a target language (Section 2), we describe our representation of sets, based on hash-tries (Section 3). We discuss the use of systematic hash-consing to support the kind of allocation needed in this approach, analyze its performance (Section 4), and examine its consequences on the choice of garbage collection algorithms and their performance (Section 5). Section 6 is the conclusion.

2 Related Work

2.1 Other Set Languages

Few languages provide a general-purpose set data structure. SETL [29] is a notable exception: in this imperative language, all objects, even sets, can be put into sets. Manens [21] and its successor S3L [20] are lambda-calculi applied to sets. Manens represents them with arrays and indices, and S3L uses partially unfolded trees of continuations, allowing for infinite sets; the tree structure is used to ensure execution fairness, not for efficiency.

When efficiency is sought (infinite sets won't be considered here), special representations are needed. With a total order on elements, balanced (say, AVL) trees decrease the complexity for access and update from linear (for lists) to logarithmic [24]; other operations like intersection and comparison remain expensive. Hash-tables are better [19], especially with separate chaining. Trabb Pardo [32] shows that destructive intersection is then linear in the lower of the cardinals of the two input sets. Hash tables are provided in Common-Lisp [31] or Icon [16] for example, but operations on them are destructive: this is not suited to functional languages like ML. If non-destructiveness is a need, Trabb Pardo showed that hash-tries [19] are good candidates for access, update and intersection (see also [33, 13]).

This is the approach we took in HimML [12], an extension of Standard ML [17] with sets and maps. But this needs fast polymorphic hash-code and equality functions. This is in particular crucial when sets can be put inside sets: Yellin [35] provides sets with constant-time equality, with a structure reminiscent of based sets à la SETL [28]; but the more sets there are, the slower his scheme is. We claim that using hash-consing [2] to share equal objects, even sets, is a easier and faster solution.

2.2 Why Standard ML?

Hash-tries and hash-consing are very efficient, but they do not mix well with side-effects: modifying an object in general involves changing its hash-code and recomputing all data structures containing it, or risking run-time inconsistencies. Moreover, if all equal objects are shared (hash-consed), then modifying one object that happens to be equal to another one also modifies the latter.

This problem was obviated by Goto [10], among others, by distinguished monocopy (shared) and unshared objects. This solves the problem, although at the price of making the language harder to use.

The nice thing about Standard ML is the fact that objects are either immutable or compared by their addresses. For example, references are mutable, but are compared by their addresses; tuples are compared by their contents, but are immutable. This effectively forbids all problems with side-effects. To our knowledge, Standard ML is the only language with this property (if we except purely functional languages, of course), which made it a target of choice for our techniques, preferably to SETL or Lisp for instance.

¹'Himmel' means 'sky' or 'heaven' in German; HimML is a recursive acronym for '<u>H</u>imML <u>i</u>s a <u>m</u>ap-oriented <u>ML</u>.

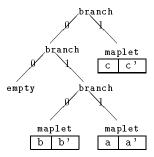


Figure 1: The map $\{a \Rightarrow a', b \Rightarrow b', c \Rightarrow c'\}$, with h(a) = 010110, h(b) = 111010, h(c) = 100101.

3 An Efficient Representation for Sets and Maps

We represent finite sets by hash-tries [19, 32], also known as radix-exchange trees. Sets are too poor a structure, though, because no information is attached to elements. We generalize them to finite (set-theoretic) maps, as in specification languages like VDM [18] or Z [30]. Note that most complex data structures in computer science are maps [22]: databases, tables, graphs, transition diagrams, ... We regard maps as basic: HimML sets are maps from elements to the 0-tuple ().

Hash-tries are built with a hash function h, such that for every object x in the system, h(x) is an integer in $[0,2^m[\ (m\geq 0),\ h(x)]$ must characterize x semantically, that is, x and y are equal (in the sense of ML's = predicate) if and only if h(x)=h(y). In HimML, h(x) is the address of x, and hash-consing is used to ensure that there is only one copy of any given object: more on this in Section 4.

The type of hash-tries is, in ML notation (this is not visible to the HimML programmer):

empty is the empty map; maplet(x,y) is the cardinal 1 map, mapping x to y; branch(g,d) is the disjoint union of g, the left branch, and d, the right branch; its cardinal is the sum of those of g and d.

Write h(x) in binary, as $\sum_{i=0}^{n-1} h_i(x) 2^i$, i.e. $h_i(x)$ is the *i*th bit in the address of x: in a node branch(g,d) at depth i in the trie, g is the submap of the x's such that $h_i(x) = 0$, and d that of the x's such that $h_i(x) = 1$. Moreover, to minimize trie size and ensure canonicity, only maps and submaps with at least two entries use branch.

To illustrate this, look at Figure 1, where a three-element map {a => a', b => b', c => c'} (mapping a to a', b to b', and c to c') is drawn. In this figure, we assume that a lies at address 010110 in binary, b lies at 111010, and c lies at 100101. To find whether a is in the map, for example, we look at its lowest significant (rightmost) bit; this is 0, so go down the left branch of the root node. We are now at a branch node again, so look at the second bit from the right; this is 1, so take the right branch. Again, we are at a branch node, and the next bit is 1 again, so we take the right branch again,

coming to a maplet node with first component a (so a indeed is in the map) and second component a' (so it maps a to a').

Algorithms are straightforward. Applying a map to an element (or testing for set membership), removing an element from the domain of a map (or from a set), adding an entry to a map (or adding an element to a set) are done by going down the trie, deciding at each branch node which edge to follow according to $h_i(x)$, i.e., the *i*th bit from the right [32, 13]. These algorithms are non-destructive. In the case of adding or removing elements, the resulting map is built up when we come back up the trie, building a new branch node at each level. In doing this, we don't forget to keep the invariant, that is, instead of building branch(l,r), we return empty when both l and r are empty, we return l when r is empty and l is a maplet, and we return r when l is empty and r is a maplet. We call this normalizing the map.

Binary operations like intersection, union, symmetric difference and difference of sets, or their analogues for maps (restriction to a set, to the complement of a set, overwriting a map by another, symmetric difference) traverse both maps in parallel. In short, to implement any of these operations \circ , we just use the fact that \circ operates orthogonally to the structure, i.e., branch $(l_1, r_1) \circ$ branch $(l_2, r_2) =$ branch $(l_1 \circ l_2, r_1 \circ r_2)$ (and then we normalize the right-hand side). This property usually does not hold for balanced trees (notably for AVL trees), and this is one reason why the latter are not suited to computing unions or intersections.

Although maps are not balanced, average-case complexity is low [32]. Assuming h(x) uniformly distributed over $[0,2^m[$, and m large enough, a map of cardinal n has average size 2.44n (AVL trees use between 1.5n and 2n), average height $2\log_2 n$ (twice as high as AVL trees). Computing the cardinal of, accessing an element in, adding one to, removing one from a map is in $O(\log n)$; also, for all practical purposes, all the binary operations we mentioned are linear in the lower of the cardinals of the two input maps [13]. (To be precise, call n_1 the lower one, n_2 the higher one, then the complexity is O(1) if $n_1 = 0$, and $O(n_1 + \log(n_2/n_1))$ otherwise.) Finally, note that the efficiency of our set algorithms does not depend on the size of elements, since only their address matters.

4 Systematic Hash Consing

We take h(x) as the address of x in memory, which is safe only if equality-admitting objects have at most one copy. Hash-consing [2] is one solution: it was originally used to save memory in large programs. It was then used to speed up algorithms, notably in computer algebra [11]. Hash-consing, as other algorithms achieving similar goals like list condensing, suffers from two defects: first, it is unsafe to modify a sharable structure; and allegedly, managing sharing is costly both in time and space.

We have already shown in Section 2.2 why Standard ML provided a unique opportunity to disregard the first point. Adding maps coded as hash-tries does not change the picture, since hash-tries are canonical forms for maps: semantic equality always agrees with equality of addresses.

As for the second point, our experience is that a good sharing allocator is inefficient neither in space, nor in allocation time, nor in garbage collection time. Benchmark results can be found in [14].

To implement sharing with hash-consing, we use a global hash-table, with slots containing unsorted lists (sorted lists were actually slower) of colliding entries, to remember previously allocated records, boxed numbers, etc. Its size H is a prime number [19] for a good distribution of data over the slots; we use H=23227. Then, the hash-table proper does not take up too much memory, but the collision lists use up to as many memory as the shared data. In HimML, this is reduced by inlining the link fields of the collision lists in the shared data themselves. We estimate the swelling due to managing sharing to less than 1.5 on average.

4.1 Efficiency of Sharing

The major difficulty is speed. In theory, access to the table is in time O(n/H) (and n can grow much larger than 23227). Most authors therefore try to keep n below H, for example by increasing the table size when it becomes almost full, which yields logarithmic access times, as in the CaML implementation of hash-tables [34]. We found that we didn't need such schemes, provided we maintain so-called optimization bits in the objects: an object a has its bit set if it is referenced by some shared object. Then, to share (a,b) where a or b has a cleared bit, we do not need to sweep through the collision list to find whether it has already been allocated, and we can insert the couple right in front of the list. After sharing, we set the bits of a and b.

This simple trick suffices to speed allocation considerably: preliminary experiments on a first version of the allocator, then not yet integrated inside any language implementation [13], showed that building and sharing an already recorded couple is as fast as constructing an unshared couple from a free list, and building and sharing an entirely new couple is at most 2 (on a MIPS 3000) to 2.9 times (on a 68020; 2.6 on a Sparc) times slower than building an unshared couple, at least for up to 13 collisions per slot on average. This is not to say that hash-consing runs in constant time, but that the ratio of hash-table traversal time to real allocation time remains constant: when more and more tuples are allocated, cache hits happen more often, which slows down allocation, but does not affect accessing the hash-table so much, since it is so small.

More realistic benchmark tests were then conducted [14] to observe the influence of hash-consing on the performance of non-set-theoretic operations; they showed that this slowdown was actually never attained. To do these benchmarks, we built two versions of the HimML evaluator. As it is written in C, it was mainly a matter of using #ifdefs at the right places in the code. The first version shares every number, record, tuple or map. In the second one, allocation is redefined as non-sharing, and equality is rewritten as the usual recursive descent algorithm; it cannot handle sets correctly, but it works on classical Standard ML programs. We then tested both versions on standard benchmarks [6]. Results are reported in Figure 2.

The programs of the benchmark are the following. KB performs Knuth-Bendix completion on group axioms; it is the same version by G.Huet as the one used in A.Appel's

benchmarks [3]. boyer checks a tautology by rewriting. church adds 256 to the elements of the list of the first 10000 reals (in the standard test, they are integers; but at the time these tests were conducted, HimML had only boxed reals, because of early design decisions). div_e does the Euclidean division of 12 by 12, representing integers with 0 and successor. integre integrates x^2 between 0 and 1 with 10000 sample points 10 times. sieve is a naive functional implementation of the sieve of Eratosthenes, finding all primes under 20000 (as boxed reals, here). solitaire solves a solitaire board game, using references and arrays. sumlist builds the list of numbers (reals) up to 10000 and sums them up, 100 times. tak computes the sum of 50 computations of tak(18, 12, 6), where takis Gabriel's function; takE does the same, but using exceptions to return results.

Most figures are not impressive in absolute value, and they should not be, as they show execution times for an interpreter, and moreover all numbers were handled as boxed reals. However, the results are instructive in that they indicate what is changed when we share or don't share data.

The worst experienced slowdown was by 30% for integre (note that we share reals, too); but, on 10 programs, 6 were actually speeded up by systematic hash-consing, including both "real-world" tests of the suite, KB (6% faster) and boyer (82% faster). Surprisingly, we gained more in terms of speed than in terms of space: the sizes of the cores were about the same with and without sharing for most programs.

The tests were done on an interpreted implementation, and interpretation damps the speed ratios, as most of the execution time is spent in the code of the evaluator, which does not depend on sharing. Moreover, garbage collections come up rather frequently, because the evaluator has to keep trace of all its internal structures. It is therefore to be expected that compiled code should give rise both to increased speed ratios (both speed-ups and slowdowns) and to decreased garbage collection times.

These benchmarks don't benefit directly from sharing. KB and boyer benefit indirectly from sharing in that they rather often use the equality function, which is much faster in the sharing version, as it is then a simple comparison of pointers. We could say that sharing memoizes [1] the equality function, i.e. it remembers previous results of equality tests on substructures of the data to compare. (As an aside, notice also that as an implementation technique for Standard ML, sharing justifies that = has polymorphic type 'a * 'a -> bool: indeed, a claimed property of ML-style polymorphism is that functions of polymorphic type never have to peek inside substructures of variable type; this holds in our implementation, as = only compares the addresses of its two arguments, ignoring their structure.)

As the benchmarks do not benefit directly from sharing, they cannot be representative of another class of algorithms, for which our systematic hash-consing technique provides great rewards. One example comes from BDDs: BDDs (Binary Decision Diagrams) were rediscovered by Bryant in 1986 [5, 4] as a tool for handling propositional formulas and quickly testing whether they are satisfiable; BDDs are shared Shannon trees, i.e. shared nested if-then-else formulas, and yield impressively fast algorithms [7] in several domains including formal hardware verification, where they have since become standard

Program	total time	GC time	# GC	GC time/total	core size	init+ compile time
KB	85.0 (90.2) s.	29.1 (30.7) s.	92 (93)	$34.2\% \ (34.0\%)$	2.3 (2.3) Mb	3.3 s.
boyer	167.1 (303.6) s.	53.4 (56.1) s.	80 (80)	$32.0\% \ (18.5\%)$	2.3 (5.9) Mb	7.0 s.
church	573.9 (580.1) s.	81.9 (104.4) s.	174 (291)	14.3%~(18.0%)	3.5 (3.1) Mb	0.9 s.
div_e	184.6 (197.1) s.	85.8 (90.2) s.	645 (655)	$46.5\% \ (45.8\%)$	0.8 (0.8) Mb	1.1 s.
integre	36.8 (28.3) s.	7.7 (6.8) s.	23 (23)	20.9% (24.1%)	2.7 (2.7) Mb	0.9 s.
sieve	155.1 (128.2) s.	17.1 (17.9) s.	21 (26)	$11.0\% \ (13.9\%)$	8.2 (7.0) Mb	0.9 s.
solitaire	562.4 (570.8) s.	105.9 (189.2) s.	691 (1936)	$18.8\% \ (33.1\%)$	0.8 (0.8) Mb	1.3 s.
sumlist	313.0 (244.2) s.	75.7 (71.3) s.	158 (173)	$24.2\% \ (29.2\%)$	2.7 (2.7) Mb	0.9 s.
tak	617.1 (664.1) s.	195.9 (254.7) s.	2319 (3226)	$31.7\% \ (38.4\%)$	1.6 (1.2) Mb	0.9 s.
takE	1228.5 (1298.8) s.	326.0 (375.9) s.	3957 (4630)	$26.5\% \ (28.9\%)$	1.6 (1.6) Mb	0.9 s.

Figure 2: Comparative speed of sharing vs. non-sharing implementations (non-sharing time in parentheses)

tools. Another example is term unification [27], where the naive algorithm takes exponential time and space in the size of the unificands, but becomes quadratic if terms are represented as directed acyclic graphs instead of as trees; smarter methods even achieve a linear bound [25]. Such examples occur surprisingly frequently in the domain of symbolic computation (computer algebra, logic, graph handling, etc.)

4.2 Set Efficiency

The efficiency of set-theoretic operations has also been verified in practice. First, the HimML type-checker and translator (to an interpreted form) run with symbol tables, substitutions from type variables to types, etc., represented as maps, and can process 200-300 source lines per second on a Sparc 2; this was done deliberately to test whether a massively set-theoretic approach to coding was realistic or not. To give a reference point, we estimate [14] that if the Standard ML of New Jersey compiler [3] did as few work as the HimML system (using the table on p.199 of [3], we count only the Parse, Semantics and Translate phases, which account for 7.7% to 14.7% of compilation time), it would process 170-325 lines per second on the same machine. Of course, these figures are mere indications, not definite evidence. But at least, using sets did not seem to have incurred any particular slowdown.

And second, we also implemented a sophisticated 3000-line automated theorem prover in HimML, using quite complicated set-theoretic data structures [15]; not only is the code fast, but knowing that all operations would be fast enabled us to think freely in terms of data structures usually dismissed as too slow, like sets of sets. For example, we generate substitutions σ by unification; such substitutions must instantiate unquantified formulas Φ to $\Phi\sigma$, and we look for one that would make $\Phi\sigma$ propositionally unsatisfiable. We don't want to consider the same substitution twice, so we keep and update the set past of all previous substitutions. Substitutions have type var -m> term (read: map from type var to type term), so past naturally has type (var -m> term) set, where τ set is an abbreviation for τ -m> unit. If there are n substitutions in past, checking whether σ is in past in HimML takes $O(\log n)$ time, independently of the size $|\sigma|$ of σ or the sizes of the elements of past; contrast this with any non-sharing implementation of sets, where for each $\sigma' \in past$, we would have to compare σ with σ' by sweeping over the variables in their domains.

Another example is the connection graph we use in this prover. A connection graph is a graph whose vertices are all atomic formulas (atoms) $P(t_1,\ldots,t_m)$ of Φ , and whose unoriented edges link unifiable atoms. The edges are labeled with the most general unifiers of their two end points. Because the most frequent operations on this graph are to find all labels of edges starting from a given end point, and given a label (a unifier) σ , to find the equivalence relation it induces on atoms ($A \cong A'$ iff σ unifies A and A'), it was easiest to encode this graph as two conjugate maps of respective types atom -m> unif set and unif -m> atom equivalence unif = var -m> term as before, and we implemented equivalence relations as maps from elements to their equivalence classes, i.e. ''a equivalence ''a -m> ''a set.

If the efficiency of our map data structures depended on the structure of their elements, operations on sets of substitutions, or worse, on our connection graphs, would be utterly impractical. With hash-tries and hash-consing, and in spite of its high complexity, our prover manages to prove almost all of Pelletier's test problems [26], usually within seconds.

To give a taste of the way sets and maps can be profitably used in HimML, we have included in the appendix a commented example of a small piece of HimML that implements applicative-style Union-Find structures. This will also give a view of the actual syntax (new functions, new syntactic constructs) of HimML.

5 Garbage Collection

To garbage-collect a shared world, it is necessary to clean up the sharing hash-table. With a mark-and-sweep collector, a second sweep phase is added, where the hash-table is traversed, and entries corresponding to freed objects are deleted. In HimML, basically, objects are Lisp tagged cons-cells; records of more than 2 fields have a pointer to a heap-allocated vector of fields. A special tag is reserved for free objects, i.e, in the free list: asking whether an object has been freed means reading the value of a tag.

Stop-and-copy algorithms are sometimes preferred to mark-and-sweep, since they compact memory, improving locality of reference, and run in time proportional to the cells:

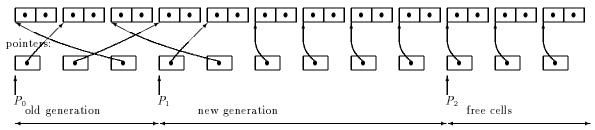


Figure 3: An allocation bucket

quantity of live data only. But as they move objects, they are incompatible with our scheme, which identifies objects with their addresses. A solution is to use markand-sweep for shared objects, and stop-and-copy for others, like run-time environments (stack frames). Mixing both approaches, as pioneered in 1990 by V.Delacour [8], is reported to work well in practice. We have compared a pure mark-and-sweep and a mixed-mode garbage collector [14], and found that, although the system spent a smaller percentage of its time in garbage collection with mixed-mode (8-15% vs. 10-30%), run times were generally comparable but core sizes were much bigger, by 5 to 15 times. This is in accordance to experience in other, non-sharing implementations [36].

The allocation policy that we used for doing all benchmarks and measurements until here was to allocate conscells from a free list, as this is easy to implement and usually fast enough. We recently changed the allocation and garbage collection policy in a way that enabled us to implement a generation scavenging scheme on a markand-sweep-like (non copying) collector. This sped up the garbage collector (GC cost was reduced to 8-10% of computation time, of which 3% are still due to the traversal of the global sharing hash-table), and also marginally sped up the allocator, although this was hard to measure, as other optimizations have been brought to the system at the same time.

The idea is the following. Cells are allocated from buckets, which are arrays of a fixed, high enough number of cells (currently 512 per bucket). Buckets are linked together, and when there are no more free cells in the current bucket, the allocator switches to the next bucket in the row. Each bucket contains an abstraction of a free list, in the guise of an additional array of pointers to cells (see Figure 3). To allocate a new cell, the allocator reads off the cell pointed to by the pointer at address P_2 , and increments P_2 . To garbage-collect the bucket, after marking all cells, we sweep through the new generation part of the array of pointers (from P_1 to P_2), and swap pointers along to compact the zone; afterwards, all live cells are pointed by pointers from P_1 to some P_3 between P_1 and P_2 , and cells pointed to by pointers from P_3 to P_2 are now free: we then set P_2 to P_3 .

This way, as in a copying collector, allocation proceeds in a stack-like fashion. Moreover, although cells are not movable, pointers to cells can be modified to achieve most of the same effect. We have not yet implemented a Cheney-style collector based on this analogy, but this looks promising. This scheme adds an extra indirection to Cheney-style allocators because we have to go through

pointers to allocate new cells, but no indirection penalty is to be paid when following links from one cell to another (cells still link directly to each other). Generations then also become clearly marked in each bucket, so conventional generational collection techniques apply.

The free-list implementation is more economical in terms of space, as the set of free cells is then completely represented through links already present in the cells themselves; in our new scheme, we have to allocate an additional array of pointers. On most machines, cells are 16 bytes long (two components, one link in hash-table slots used for managing sharing, and various tags and bits), and pointers to cells are 4 bytes long. So the space overhead this incurs is roughly 25%, for a reduction of GC cost to 8-10% of computation time.

6 Conclusion

Sets and maps are powerful conceptual constructs, and deserve an efficient general-purpose implementation. Representing them by a combination of hash-tries with systematic sharing by hash-consing has a good theoretical and practical behavior. We were surprised by the measures of efficiency we got: first, our prototype compiler is about as fast as the corresponding parts of a well-crafted one; second, the systematic sharing scheme slows down the system by not much, and actually speeds it up in more than half the cases tested.

It will nonetheless be interesting to specialize the representation of sets to better-suited ones (like bit-vectors for sets of integers, for example), at least for local, non-escaping data inside lexical blocks. Automatic inference of representations, as pioneered in [28] for SETL, or as used in [9], might be put to good use in a HimML compiler.

References

- H. Abelson, G. J. Sussman, and J. Sussman. Structure and Interpretation of Computer Programs. MIT Press, 1985.
- [2] J. Allen. Anatomy of Lisp. McGraw-Hill, 1978.
- [3] A. Appel. Compiling with Continuations. Cambridge University Press, 1992.
- [4] J.-P. Billon. Perfect normal forms for discrete functions. Technical Report 87019, Bull, 1987.

- [5] R. E. Bryant. Graph-based algorithms for boolean functions manipulation. IEEE Trans. Comp., C35(8):677-692, 1986.
- [6] CAML team. Benchmark suite for CAML and Standard ML. available from the author, or from Régis Cridlig (cridlig@dmi.ens.fr).
- [7] E. Clarke. Symbolic model checking: 10¹³⁰ states and beyond. In CADE-11. LNAI 607, 1992.
- [8] V. Delacour. Gestion mémoire automatique pour langages de programmation de haut niveau. PhD thesis, Paris VII, 1991.
- [9] V. Donzeau-Gouge, C. Dubois, and P. Facon. Inférence de types et d'implantations pour les expressions ensemblistes. In JFLA '91, pages 21-32, 1991.
- [10] E. Goto. Monocopy and associative algorithms in an extended Lisp. Technical report, U. Tokyo, 1974.
- [11] E. Goto and Y. Kanada. Recursive hashed data structures with applications to polynomial manipulations. SYMSAC, 1976.
- [12] J. Goubault. The HimML reference manual. available from the author, 1992.
- [13] J. Goubault. Implementing functional languages with fast equality, sets and maps: an exercise in hash consing. Technical report, Bull, 1992.
- [14] J. Goubault. Adding sets to ML: design, implementation and experiments. Technical Report 93039, Bull, 1993.
- [15] J. Goubault. Proving with BDDs and control of information. In CADE-12. Springer, 1994.
- [16] R. E. Griswold and M. T. Griswold. The Implementation of the Icon Programming Language. Princeton University Press, 1986.
- [17] R. Harper, R. Milner, and M. Tofte. The Definition of Standard ML. MIT Press, 1990.
- [18] C. B. Jones. Systematic Software Development Using VDM. Prentice-Hall, 1990.
- [19] D. Knuth. Sorting and Searching, volume 3 of The Art of Computer Programming. Addison-Wesley, 1973.
- [20] J.-J. Lacrampe. S3L à tire d'ailes. to be published, 1992.
- [21] F. Le Berre. Un langage pour manipuler les ensembles: MANENS. PhD thesis, Paris VII, 1982.
- [22] M. Mac an Airchinnigh. Tutorial lecture notes on the Irish school of the VDM. In VDM'91, volume 552 of LNCS, pages 141-237, 1991.
- [23] K. Mehlhorn and A. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 301-341. Elsevier, 1990.
- [24] M. H. Overmars. The Design of Dynamic Data Structures. LNCS 156, 1983.

- [25] M. Paterson and M. Wegman. Linear unification. J.Comp.Sys.Sci., 16:158-167, 1978.
- [26] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. JAR, 2:191-216, 1986. Errata in JAR, 4:235-236, 1988.
- [27] J. Robinson. A machine-oriented logic based on the resolution principle. JACM, 12(1):23-41, 1965.
- [28] E. Schonberg, J. Schwartz, and M. Sharir. An automatic technique for selection of data representations in SetL programs. ToPLaS, 3(2):126-143, 1981.
- [29] J. Schwarz, R. Dewar, E. Dubinski, and E. Schonberg. Programming with Sets: An Introduction to SETL. Springer, 1986.
- [30] J. Spivey. An introduction to Z and formal specifications. Soft. Eng. J., 1989.
- [31] G. L. Steele. Common Lisp. Digital Press, 1984.
- [32] L. Trabb Pardo. Set Representation and Set Intersection. PhD thesis, Stanford U., 1978.
- [33] J. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 9. Elsevier, 1990.
- [34] P. Weis et al. The CAML reference manual. Technical report, INRIA, 1989. v.2.6.
- [35] D. M. Yellin. Representing sets with constant time equality testing. J.Alg., 13:353-373, 1992.
- [36] B. Zorn. Comparing mark-and-sweep and stop-andcopy garbage collection. In LFP, pages 87-98, 1990.

A Applicative Union-Find Structures in HimML: a Taste of Sets and Maps

This appendix is a simple illustration of the use of sets and maps in HimML. The full set of additional syntax constructs and primitives, as well as detailed informal explanations of their semantics can be found in [12].

Union-Find structures [23] are well-known data structures for representing equivalence classes of objects, which are endowed with two natural operations: Find computes a canonical representative of the equivalence class of an object, and Union merges two equivalence classes described by their canonical representatives. Implicitly, Union modifies the underlying equivalence class, and is implemented by physically modifying the corresponding data structure.

Union-Find structures are trees whose nodes (leaves are considered to be nodes as well) are decorated with objects. Edges are viewed as directed links that point upward (toward the roots), so that in a sens these trees are turned upside-down. A root of the tree does not point to any node, and represents a canonical representative of an equivalence class. All other nodes point to higher in the tree, and following the links eventually reaches a root: this is how the Find function works.

To implement Union on two roots r_1 and r_2 , one of them is made to point to the other (we assume that $r_1 \neq$

 r_2). To keep the structure balanced on average, roots are also decorated with the numbers of elements in the class they describe. Then, assuming that r_1 has the fewest elements, r_1 is made to point to r_2 , so that r_2 becomes the new root, and its count of elements is incremented by the count of elements for r_1 .

If x is an element of a class with n elements, Find then takes roughly $O(\log n)$ steps, and Union takes O(1) time. (Notice that, in practice, it means that the time taken by both operations is indistinguishable from constant time: on a 32-bit computer, no more than 2^{32} objects are addressable at the same time, so that $\log n \leq 32$.) This efficiency is usually seen as coming from the fact that Union is allowed to physically modify the underlying data structure, and that non-imperative Union-Find structures would be too slow, because of the required amount of copying.

Maps allow us to side-step the difficulty by encoding links as maplets in a global map representing the structure. We then lose a logarithmic factor on all operations, but experience (see main text) has always shown this to be negligible. (Notice, by the way, that side-effects have not disappeared totally, as they are required to maintain the internal global hash-table that we use to manage sharing.)

So, a Union-Find structure on objects of type ''a is a record with fields link: ''a -m>''a (mapping sources of links to their destinations, going up towards roots) and info: ''a -m> int (mapping roots to the cardinal of the associated equivalence class). Notice the double quotes on the type variable: objects in a Union-Find structure must indeed have an associated equality function, so that their type must admit equality [17]. In general, type τ -m> τ' is only legal when τ admits equality; this needed a slight amendment of the static semantics of type declarations in Standard ML, to allow for partial type functions ('a -m> 'b is then not legal, whereas ''a -m> 'b is, reflecting the domain of definition of the type function -m>).

Because we wanted to have curly braces denote set and map expressions, we also had to change the syntax of record expressions and types to use different delimiters. We chose | [and] | to replace { and } in that case. Making type functions and declarations partial, and changing this bit of syntax were the only two needed changes in the definition of Standard ML, viewed as a subset of HimML. (Other changes were mere additions, to account for set and map operations and various other goodies.)

We can now write the type declaration for applicative Union-Find structures:

```
type ''a ufind =
    |[link : ''a -m> ''a,
    info : ''a -m> int]|
```

We must also provide a new empty Union-Find structure:

```
val empty_equiv = |[link = {}, info = {}]|
    : ''a ufind
```

where $\{\}$ denotes the empty map, and is a special case of the notation of maps by enumeration $\{x_1 \Rightarrow y_1, \ldots, x_n \Rightarrow y_n\}$. In case all y_i 's are equal to (), the map is a set, and we write $\{x_1, \ldots, x_n\}$.

To implement Find, we use the following HimML primitives. If m is a map and x is an object,

x inset m tests whether x is in the domain of m (we have inset: ''a * (''a -m>'b) -> bool), and ?m x returns its image, or raises the exception MapGet if x is not in the domain of m (we have ?: ''a -m>'b -> ''a -> 'b). So we can write a version of Find, parameterized by a given Union-Find structure:

```
fum Find (|[link,...]|: ''a ufind): ''a -> ''a =
   let fum f x =
        if x inset link
        then f (? link x)
        else x
   in
        f
   end
```

It then takes $O(\log n \cdot \log n')$ time to follow the links, where n is the cardinal of the class of \mathbf{x} and n' is the size of the structure. (Remember that logarithmic factors are low; in practice, the difference is usually hardly noticeable.)

To implement Union, we shall need a few other HimML operators, whose origin goes back to the VDM specification language [18]. The first is ++: (''a-m>'b) * (''a-m>'b) -> ''a-m>'b, the infix override operator: f ++ g is the map whose domain is the union of the domains of f and g, and which maps x to ?g x if x inset g, or to ?f x otherwise (g "overrides", or takes precedence over f). When f and g are sets, this is just the union operation; HimML gives the redundant name U to this operation in this restricted case.

Restriction operators are also needed. f < | g returns a map whose domain is the intersection of those of f and g, and which coincides with g on this domain (g "restricted to" the domain of f). f <- | g does the complementary operation, i.e. it returns a maps whose domain is that of g minus that of f, and which otherwise coincides with g (g "restricted by" the domain of f). On sets, these would be the intersection (& in HimML) and set difference (\)) operations. HimML provides a whole slew of other operations, as well as notations for set comprehensions and quantifications over elements and even submaps (subsets) of maps and sets; we refer the interested reader to [12].

These few operators are enough to define the Union operation, which must take a Union-Find structure, two roots, and return a new Union-Find structure:

Then Union takes $O(\log n')$ time.

To conclude, why bother with Union-Find structures when all we want is finite equivalence relations? The favored style in HimML is to use maps whenever they provide a more direct implementation of the mathematical object we wish to code. And a natural representation for equivalence relations are maps from elements to their equivalence classes. So, a preferred implementation of equivalence relations would be:

```
type ''a eqvrel = ''a -m> ''a set
val empty_eqvrel = {} : '', a eqvrel
fun class_of (r: ''a eqvrel') (x : ''a) =
   if x inset r
       then ?r x
   else {x}
fun Find r x = choose (class_of r x)
fun Union (r: ''a eqvrel) (x: ''a, y: ''a)
           : ''a eqvrel =
   let val ca = class_of r x
        and cb = class_of r y
   in
       if ca=cb
          then r
       else let val merged = ca U cb
            in
               r ++ \{x => merged
                    | x in set merged}
            end
   end
```

where ''a set is an abbreviation for ''a -m> unit, choose: (''a -m> 'b) -> ''a chooses an element from the domain of the map in argument (deterministically, i.e. $m=m'\Rightarrow$ choose m= choose m'), and the last significant line shows an example of a map comprehension computing the constant map that sends every element of merged to the set merged itself.

Find is then slightly faster, although Union will be slower. But the main benefit is that this implementation is easily expandable and maintainable: it is as easy to define extra operations, like computing the intersection of equivalence classes (more exactly, the equivalence relation generated by all equalities that hold in both equivalence relations in argument), or restricting equivalence classes to or by sets of objects given as arguments. (The latter, in particular, was needed in our theorem prover for restricting connection graphs to a given set of atoms.) On the other hand, it is hard to tailor Union-Find structures to accommodate such extensions with reasonable efficiency.