Speeding up Lisp-based Symbolic Mathematics

Richard J. Fateman and Mark Hayden*
Computer Science Division, EECS Dep't, University of California at Berkeley

Abstract

Two techniques for speeding up a traditional Lisp-based symbolic manipulation system (Macsyma) were outlined in a previous paper [2]. These were: using unique representations for equivalent "kernel" expressions (where a kernel here is basically anything but a sum), and using hash tables for an unordered representation of sums. New tests in the context of a complete version of Macsyma suggest that a speedup of — in some cases — orders of magnitude can be obtained. In some cases this appears to match or exceed the speed of Mathematica, a recently written system written in a lower-level language (e.g. C).

1 Introduction

How much is the speed of a computer algebra system (CAS) determined by the language in which it is written? We would like to believe the major issues are appropriate choices of data structures and algorithms, and that one is no longer, in this day and age, restricted to low-level languages. Indeed, a credo of programming language designers is that a well-designed high-level language and a good compiler will take nearly full advantage of the underlying hardware; writing in assembly language is not necessary. On the other hand, we continue to hear, anecdotally, that Lisp-based computer algebra systems are necessarily slow because they are written in Lisp. Could it be that Lisp (a) does not support appropriate data structures, or (b) is inappropriate for expressing algorithms, or (c) is it (inevitably?) implemented poorly to not take good advantage of the underlying hardware? One way of putting the lie to all of (a), (b), and (c) is to provide believable benchmarks where a Lisp-based system is faster than an allegedly fast commercial C-based system.

2 Some Evidence

Consider the explicit computation (expansion) of a sum

$$S_N = \sum_{i=1}^{N} a_i$$

for various values of N, and where a_i is an uninterpreted subscripted name. The result is an expression with n terms. How fast does Macsyma, in some sense a "classic" Lisp-based CAS, compute this (considering that it generates the terms one by one and inserts them into a running sum)?

Here is a table of experimental measurements showing N vs. the standard time[1] (in seconds) as well as our improved time, using a version of Macsyma ("Maxima") compiled in Allegro Common Lisp 3.1.15.3. This was timed in 1994 when this article was first written. A commercial Macsyma 419.0 (1995) on a Sun SPARC still shows the same growth rate as the standard column below as well as in the tables to follow.

N	standard	new
5	0.087	0.027
10	0.161	0.040
40	0.722	0.123
100	2.272	0.287
200	6.075	0.590

The new column shows a substantial improvement. Full data illustrates that the standard method is $O(n^2)$ while the new method is roughly $O(n)$. At size 200 there is a factor of 10 speedup. We believe that a major part of the speedup is that the standard version uses an $O(n^2)$ insertion routine to accumulate the elements in the sum.

Now consider the same problem except that a_1 is an encoding of $f(x)$, and for $i > 1$, $a_i = f(a_{i-1})$. That is, the elements in the sums are successively deeper nested expressions. For example, $S_2 = f(x) + f(f(x))$.

N	standard	new
5	0.157	0.027
10	0.197	0.030
30	1.123	0.077
40	2.860	0.100
50	6.103	0.117
80	33.716	0.177
100	72.930	0.217

At size 100 the new code is over 330 times faster than the standard code. The old code is slow because the simplifier, in order to order the terms, must repeatedly scan down into the substructure of each element.

*This work was supported in part by the National Science Foundation Infrastructure Grant number CDA-8722788

[1] Most times were averaged over several runs, and lisp garbage collection time was standardized by performing a collection before each timing run. A generation-scavenging collection on our machine seems to take only about 0.2 seconds in this test environment, and would normally be a small percentage (say 5%) of cpu and real time. Times taken on a MIPS M/120 computer.

How fast is Mathematica on this same test? We made sure to pre-compute and store the a_i values first. Then Mathematica is an (approximately constant) factor of 5-7 faster than our faster code. We conjecture that part of this speed is obtained by using a program with somewhat less generality (fewer fruitless simplifications attempted) and/or tighter coding of ancillary simplification functions. One could also conjecture a C vs. Lisp difference, but evidence from additional timings below suggests this may not be an inherent factor.

Note once again that the timing gap in the table above between the standard and new versions continue to grow for larger problems: the new version runs in time that is apparently immune to the size of the objects in the sum, and the standard version is not.

Now consider the problem of computing the difference between S_N and S_{N-1}. This computation requires that all the terms but one cancel. Let us first consider when the sum has simple subscripted variables a_i. We have included Mathematica times here as well.

N	standard	new	Math'ca
5	0.008	0.008	0.002
50	0.222	0.026	0.018
100	0.820	0.043	0.038

The times for Mathematica to perform this operation are about the same as our new code for large problems. At size 100, each of these is about 20 times faster than the old code. Although Mathematica has a clear edge on small problems, it appears that the gap closes on larger ones.

Now consider our second variation, where $a_i = f(a_{i-1})$, $a_0 = x$.

N	standard	new	Math'ca
5	0.008	0.008	0.004
50	5.044	0.028	0.095
100	69.208	0.043	0.356

At size 100 we observe that our new code provides a speed factor of 1600 over the old Macsyma code, and a factor of 8 over Mathematica. Note also that the new code is immune to the difference between simply a_i and an element that is an i-deep nested function.

3 Changes to the Code

In changing Macsyma, our objective is to maintain exactly the same functionality.

Two ways occurred to us:

(1) All changes to existing code maintain the "contract" between input and output of changed modules. In this case, the work-horse functions of the simplifier (`simplus` and `simptimes`) must return equivalent answers to those before the change. Since virtually no programs in Macsyma call the simplifier except through the main program (`simplifya`), which then calls `simplus` and `simptimes`, this is easy to do. Unfortunately, this severely limits the level of improvements possible since considerable "trivial" data passes back and forth through this "contract".

(2) One (or more) new data types can be introduced into Macsyma, and the "contracts" for other functions must be changed to accept and perhaps return results containing the new data type. This has the major advantage of limiting conversions to and from the new data type and thus saving time and space. It has the major disadvantage that the new data type may not be recognized properly in the large number of unexamined programs that might meet up with it.

This notion of "contract" is entirely informal, and appears nowhere in the original Macsyma code except (rarely) in comments. It is therefore non-trivial to enforce such contracts on new non-conforming data [1]. Our approach was to establish a global flag that is tested to see whether we must return an old-style list data-type (`ret2list = t`) or not. To be compatible with the mass of (as yet) unexamined program modules in Macsyma, we bind the `ret2list` flag to `t` and then invoke such programs. Where we have checked out the programs, namely within the confines of the simplifier, display, and rational function packages, (and, as we find the time and necessity, additional modules) we can use either the traditional Lisp s-expression form used in Macsyma e.g.`((mplus) x y)` or our new representation of a hash-table to represent explicit sums of elements. A small number of patches to each of these packages was necessary to assure that the hash-table representation could be dealt with properly.

3.1 Unique Subexpressions

In order to make this system work we must assure that two occurrences of the same expression, say $\sin(x+3y)$ are not only `equal` in the Lisp sense, but also `eq`. That is, all references to this expression must be pointers to the same location in memory. The basic idea behind the "hashing cons" we use was expressed in 1976 by Goto and Kanada [5]. Each time we compute a Lisp `cons` in constructing an algebraic expression, we check first to see if another `cons` cell exists with the same `car` and `cdr`. This checking is done by installing all such "unique" cons cell in a hash table (see Fateman [2]). This is a substantial additional overhead on a per-cons basis, in the hope of reaping returns in more efficient processing.

This uniqueness means we can, without examining substructure, assure that two expressions, say $3x^2y^3$ and $4x^2y^3$ both have the same (i.e. `eq`) hash key, namely the key based on x^2y^3. In a hash-table representing the sum of a number of terms, we can then expect that the keys for the two terms will and thus we need only add $3+4$ and store this value in the hash table at the location keyed to x^2y^3.

3.2 Fast Structural Comparison

A second major feature we introduced to the changed Macsyma, although somewhat orthogonal to the representation of sums as hash tables, is a technique for fast structural comparison.

Comparing objects is one of the most basic functions of computers. For fixed size objects such as integers or bytes, computers can apply ordering comparisons very quickly. However, comparisons of large compound data objects, what we here call structural comparisons, can be time consuming.

Structural comparisons traverse rooted data structures to apply an ordering on valid structures. Normally, the worst-case comparison occurs when two structures are isomorphic because both structures have to be completely traversed to determine that they are isomorphic. However, the introduction of canonicalization eliminates this worst-case: all isomorphic structures have the same address so a pointer comparison is all that is needed to determine equality. Then, the worst case becomes nearly isomorphic structures: most of the structures must be traversed even with canonicalization. These worst cases are somewhat uncommon so structural comparisons can generally be quick. This is analogous to lexicographic string comparison: most strings have different first letters so most comparisons end after checking the first letters.

In the general case, structural comparisons do not necessarily behave nicely. The ordering that one applies can be arbitrary and can make nearly complete traversals of structures with every comparison, as in comparing $f(f(x))$ with $f(f(f(x)))$. In every comparison, at least one structure must be scanned completely. Macsyma's `great` function exhibits worst-case behavior on this case. This is significant because `great` is used almost universally to order mathematical expressions which are represented as compound data structures, such as the ordering of terms in a sum. The ordering that `great` applies is often based on an expression's apparently most complex variable and not on its function. For example, $\cos(\exp(x^2))$ and $\log(\tan(\exp(y + 3)))$ could be quickly ordered by the differences in the operators (cos vs. log), but for aesthetic reasons, `great` ignores the functions and examines the variables (here x vs. y) also. Though *some* structural comparisons are fast, there is nothing to prevent structural comparisons from having an average behavior close to worst-case.

3.3 A Naive Speedup

An obvious way to speed up structural comparisons is to "memoize" the comparison function [6]. Memoization involves saving the results of all comparisons. Before comparing two structures, a check is made to see if that comparison has already been made. If it has, then the old result can be returned. Memoization can be implemented speed-efficiently with a hash table that takes the two compared structures and hashes into the table storing the result. Making a hash table lookup does not slow the comparison down much if it fails, and if it succeeds the comparison is done. Unfortunately this requires $O(n^2)$ space for n elements, and furthermore does not take advantage of the valuable fact that `great` should be a total ordering and hence is transitive.

3.4 Computing Total-Ordering Numbers

How can we take advantage of this total ordering property? Our scheme is to tag structures with numbers that have the same ordering as the structures they are attached to. Comparing two tagged structures then involves looking up these rankings and quickly comparing numbers instead of the structures. As before, the structures can be efficiently tagged using a hash table, only this time the hash table uses one key (the structure) instead of two. The only detail left is how to pick ordering numbers to assign to structures. However, resolution of this turns out to not be a minor detail.

Rankings can be assigned by using a binary search tree into which structures are inserted as they are compared. For speed and space efficiency, we implemented the ordering numbers as fixnums (a Lisp primitive data type that maps directly to machine integers encoded as "illegal pointers"). Two versions were implemented.

1. Keep inserting and interpolating numbers until the tree becomes too crowded at one location for an additional required insertion. The new node's number must be between the numbers of the next and previous nodes. We average those two numbers and truncate the result to get the new integer. If the new number is not different from an adjacent node, it is time to rebalance. At that time the whole tree is globally rebalanced and renumbered. Hash entries must also be updated so that there is $O(1)$ access to an ordering number, given a structure.

2. Alternatively, keep up a more elaborate tree structure using AVL balancing, and through local links only a local renumbering is tried first. Failure to find a good interpolating average causes a local renumbering to propagate upward.

Because rebalancing the tree and respacing the numbers both take $O(n)$ time for n entries, recurring compaction can cause speed to be seriously degraded. However, compaction should not normally happen too often. Fixnums generally have a range of -2^{29} to 2^{29}. For compaction to occur, the tree must have a chain of about 30 nodes. The easiest (but not only) example that causes such a chain is one in which which ever-"greater" structures are inserted in order. This is highly unlikely in some "random" scenario, but not so unlikely in our deliberately nasty benchmark examples in which (say) elements a_i are generated in sequence.

In general, this approach is helpful if the simplifier or other programs make comparisons repeatedly on elements that have been ranked. A situation in which only "first comparisons" occur makes all this attempt at optimization quite fruitless, wasting both time and space.

Tests (in this section performed on a DECstation 5100 using Allegro Common Lisp) were designed as follows. Lists with varying numbers of Macsyma expressions were constructed, placed in random order, and then sorted (using the library quicksort algorithm in that Common Lisp) once with the regular `great`; then with the upgraded `great` with an empty tree and hash table (no total-ordering numbers precomputed); and then again with the upgraded `great` but with the tree and hash table initialized as a consequence of the previous sort. Times are given in milliseconds. In addition to times, we also counted calls: these specifically refer to the number of calls to the original `great` function, including its recursive calls to itself. No calls are made for the second sorting with the new `great` because all comparisons can be computed from the total-ordering numbers. We therefore omit the column calls (since it would be 0's).

The first test consists of sorting a random ordering of n subscripted uninterpreted elements $\{a_i\}$. The lisp data representation is something like `((a simp array) i)`.

n	old ms	old calls	new ms	new calls	repeat ms
50	17	446	117	534	16
100	34	1132	250	1279	50
200	67	2660	700	2954	67
300	100	4488	950	4764	234
400	133	6142	1334	6711	333
500	184	7732	1734	8677	416
600	233	10072	2366	10859	500
700	283	11770	2333	12803	600
800	317	13864	2683	15041	700
900	367	15550	3117	17306	783
1000	417	17476	4000	19577	850
2000	916	38952	10450	43483	1934
3000	1467	62090	20383	69228	3133

This data is unimpressive. For sorting simple objects like a_{25} and a_{26}, no speedup is obtained; in fact there is about a factor of two slowdown in the "repeated" column from the old method. And for the "first time" sorting, there is a factor of 10 or so slowdown! There are even *more* calls to `great` because it is not only necessary to compare the two elements being sorted, but additional calls are used to insert terms into a tree. Fortunately, this case should not be typical: we are inserting items into a tree in the worst possible way for our program: generating a new item, inserting it, and never really reusing it. If this case were common, we'd have a losing design here. While variations on the programs may be worth examining, let us instead turn to a different test case.

The next test below shows the cost of the sorting but with a_i now the iterated function we've used before. That is, $a_2 = f(f(x))$ etc.

n	old ms	old calls	new ms	new calls	repeat ms
10	17	119	17	48	0
20	67	641	50	191	0
30	183	1555	100	431	17
40	466	3091	283	774	17
50	883	4967	316	1214	17
60	1667	7645	500	1754	33
70	2883	11823	666	2397	34
80	4450	15792	933	3137	50

These tests clearly show that especially after the "great" tree is initialized, there is a drastic improvement in comparison time (A factor of 89!). The earlier benchmarks in section 2 benefitted from this. As we said earlier [2] citing a a 1981 paper [3], we found that in a typical run of Macsyma, some 60% of the execution time is spent executing code in the kernel of Lisp. The four most frequently referenced "code pages" were, in order, those which contained the code for `cons/newint` (programs which allocate cells for lists and numbers), `equal`, garbage collection, and `greaterp/lessp`. In the benchmarking run illustrated in that paper, these 4 pages accumulated 1.8 million references. No other pages had more than 0.15 million references, and most had between 0 and 10 references. At the time we suggested that one could expect an improvement by recoding `equal` for higher efficiency (in assembler, for example) to run Macsyma faster. It is clear now that recoding Macsyma to *avoid* the use of `equal` can have a far more substantial benefit, all other factors remaining the same. This is the approach taken here by replacing `equal` by the primitive function `eq`, (pointer-equality, typically one or two instructions) and in effect precomputing comparisons by the total-ordering tree.

4 Weak Vectors for GC

A disadvantage of our technique for unique storage of Lisp cells is the use of one or more system-wide hash tables for all Lisp cons cells used to construct algebraic data. In particular, there is an entry in this table for each such cons-cell, in effect raising the memory cost for these cons-cells (typically 2 pointers or 64 bits in size) by the cost for the hash-table entry: another 64 bits. We have not figured a technique to avoid this, and can only point out that memory is getting cheaper rapidly. Occasionally we may be saving memory by not storing duplicate copies of the same data. (Not all cons-cell data is incorporated in these hash tables. Cons cells that are used for local computation should not be hashing conses.)

A second problem is that the hash-table entry represents an apparently permanent link to each cons. As a consequence, in the typical Lisp system, the garbage collection mark phase will alway use the system hash table as part of its root collection of accessible data to mark from. Thus, any data that has been canoncalized (that is, all hashed conses) would seem to be permanent, and its data space never reclaimable, even if all user-accessible pointers to the data have been changed.

Fortunately, such situations have been encountered before, and a ready-made solution exists. Allegro Common Lisp has what are called "weak vectors". If a data object is pointed to *only* by an entry in a vector allocated as a weak vector, then it is *not* marked, and will be collected. A pointer from a weak vector to an item that has been collected is changed to a null pointer. Thus, building our hash table using weak vectors solves the garbage problem. Other implementations of Common Lisp should have corresponding features, although perhaps slightly different access to them.

5 C vs. Lisp

We are not persuaded by these results, nor the brief review of tests with Maple (see section 7), another CAS with a kernel written in C, that the choice of programming language (C or Lisp) is the dominating factor in the speed of a CAS. By choosing the test, we can find any of Maple, Mathematica, or Macsyma fastest. We support the incorporation of algorithmic improvements into either of the C-based systems. Should they be able to do so in an economical fashion, it is possible that they would catch up to or beat the Lisp code where it is faster. Lisp's advantage may be, however, that it is "lighter on its feet" and can be changed more easily to incorporate new algorithms or data-structure improvements. This is the kind of competition which leads to improved systems and programming languages.

6 What Next

Not all programs will be sped up by the changes proposed here. Indeed, as we noted, some (somewhat perverse) programs that generate numerous independent variables and do not do much with them, will be considerably slowed down. Furthermore, subsystems that do not use the general algebraic list form for their major computation will not be affected at all. Some of the more efficient code in Macsyma already uses a rather different form, canonical rational expression (CRE) form [1]. The only changes needed for integration of these changes with CRE form have to do with conversion from a hash table into CRE form.

We have provided copies of the changed code to the principal commercial holder of the MIT license (Macsyma Inc.) for their consideration in installing and extending these features. We can provide copies to any recipients of the Department of Energy Macsyma library code as well. At the moment, the code uses a few locutions specific to Allegro Common Lisp (ACL) which may inhibit its adoption by Macsyma Inc. on other systems. ACL is implemented on a variety of architectures, Sun SPARC, MIPS/SGI, Motorola 68000, HP,IBM RS-6000, DEC-Alpha, Intel (Windows) and a variety of other architectures, but is not free. Free Lisp implementations such as GCL or AKCL conforming substantially to the Common Lisp standard will likely require only modest adaptations.

One may reasonably ask why one should consider improving Macsyma, when there are other newer systems: The major advantage of working to improve Macsyma is that one is starting with a system that has a long history of testing. We believe it is preferable to start with a correct program and make it fast, rather than take a fast program and try to make it correct.

Another efficiency possibility we hope to pursue is the development of some system routines that are easily run in parallel. The traditional Lisp representation for lists tends to interfere with parallel programming efforts—divide-and-conquer approaches are slowed if one must count half-way down a list for the "divide" step! The new system routines we have developed are vector, rather than list based, and should be much easier to use for parallel processing [7]. While we can encourage speculation on the use of parallel machines for hash-table based processing as in this paper, we are skeptical at the moment that current operating systems and processors have too high an overhead for process dispatch. For many small to medium sized problems, parallel systems are likely to be overall slower than serial computation. We assume there is a prospect of payoff in parallelism for those problems of truly huge size that are occasionally encountered.

It is possible to build another system, or incorporate into a system now being constructed, these ideas, but it is important that such new systems not forget the lessons of the older ones. We hope that systems of the future will be able to use these algorithms and data structures and yet maintain a perspective that recognizes correctness, speed, and the need to scale up to large problems.

7 Final notes

An earlier version of this paper was submitted to ISSAC 94. This section attempts to respond to the useful comments of both anonymous and self-identified referees.

On the suggestion (nay, insistence!) of M. Monagan, we ran some tests on Maple on similar problems. This is relevant because the hash-coding tricks and unique (canonical) storage ideas are a basic implementation notion built into the Maple system. It makes sense to examine how effectively these ideas are used.

Our initial efforts to time Maple were confounded by a peculiarity of the Maple command-line evaluation scheme. Maple appeared to be rather slow because the large expressions we produced were subjected to an entirely fruitless re-evaluation. This is done on the top-level (but only top-level) expressions in Maple. Anyone considering benchmarking Maple with "large results" returned should be cautious.

Thus it was necessary to enclose the timing tests in a simple Maple procedure to try to isolate the "true" Maple computation costs.

We were then easily able to time the computation of $S(n) := f(x) + f(f(x)) + \cdots$ for expressions for any number n terms. While adding n terms together takes negligible time, even for very large n (reported as 0.0 seconds on a Sparc1+ running Maple V), another problem crops up: the time to *guarantee they are sorted in lexicographic order* is several times slower than our improved Macsyma or Mathematica. How can this be? The explanation lies in Maple's standard of simplification, which is not directly comparable to the other systems. For example, $x^2 + 1 + x^4$ would not be considered simplified by Mathematica or Macsyma. Yet if you typed in an expression first this way in Maple, it would consider this simplified. Thus to make Maple comparable to the other systems, we had to use its `sort` function before returning the results.

If the terms being sorted were of the form a_i, the sorting takes time only slightly slower than either of the other systems.

When Maple is faced with computing $S(n) - S(n-1)$ it clearly shines. It reports negligible times for $n = 100$; at $n = 1000$, a far larger test than we attempted with the other systems, it takes 0.05 secs. Assuming about linear growth for this test, it appears to be more than 10 times faster than either the new Macsyma or Mathematica!

Our conclusion with respect to Maple is therefore: it does indeed have a faster hashing mechanism than the hcons system we put together in our Lisp, and it is used for impressive effect.

Even so, we do have some suggestions for the Maple implementors: In our tests, Maple's sorting of distinct terms was quite slow. A mechanism such as our total-order numbering might provide a massive improvement for Maple on a "simplification with sorted result" benchmark.

Given that this hashing idea is used so well in Maple, why is it interesting to discuss a (somewhat less effective) version retrofitted to Lisp, and used for Macsyma? What is 'publishable' about this result?

In 1994 we were not making an extensive survey of systems, but comparing new vs. old Macsyma. We added in Mathematica as a convenient data point. This system was more recently designed than Maple and was written in C. Its implementers apparently missed out using unique-coding.

The preparation of a complete survey of CAS is not the point of this paper, but we believe that others can easily construct test cases on their own systems.

Remarkably, we could take "standard" Macsyma—a system whose simplification program's basic design goes back 30 years—and could speed up this elaborate program by some critical changes by orders of magnitude on identified benchmarks.

We would also like to make the point that as a design and benchmarking issue, here we have explicitly measured some implementation options not usually examined in quite this way: we have more clearly separated our design (hashing and ordering) from the language and representation issue. By pushing the responsibility for representation (as hash tables) toward the (Lisp) language implementer we have freed the CAS builder from forever writing some bit-twiddling assembler or C-coded kernel for each new computer that appears. We can even make our system faster on existing systems by recompiling with a newer Lisp compiler. We can separate out our unique contributions, the CAS code, from that other body of reusable code and optimizations shared with all the other language users for basic facilities like hash tables.

8 Acknowledgments

We would like to thank M. Monagan for extensive correspondence on this topic. The views expressed here are those of the authors and are not necessarily endorsed by others.

References

[1] R. Fateman. MACSYMA's General Simplifier: Philosophy and Operation. In V. E. Lewis, ed. *Proc. of the 1979 MACSYMA Users' Conference* (MUC-79), Washington, D.C., June 20-22, 1979, MIT Lab. for Computer Science, 563-582.

[2] R. Fateman. "Canonical Representations in Lisp and Applications to Computer Algebra Systems," ISSAC-91.

[3] J. Foderaro and R. Fateman. Characterization of VAX Macsyma. In P. S. Wang, (ed.) *Proc. of 1981 ACM Symp. on Symbolic and Algebraic Comp.* Snowbird, Utah, Aug, 1981 14-19.

[4] Mark Hayden. "Fast structural comparisons in Lisp," term paper for CS282, Fall, 1991. (R. Fateman), EECS Dept. Univ. Calif. Berkeley.

[5] E. Goto and Y. Kanada. Hashing Lemmas on Time Complexities with Applications to Formula Manipulation. In R. D. Jenks, (ed.) *Proc. of 1976 ACM Symp. on Symbolic and Algebraic Comp.* Yorktown Heights, N. Y., August, 1976. 154–158.

[6] Donald Michie. "Memo" Functions and Machine Learning, *Nature, 218*, (1968), 19–22.

[7] Tak W. Yan. "A rational function arithmetic and simplification system in Common Lisp," *SIGSAM Bull. 25*, no. 4 (Oct., 1991), 4–6.