# Sparse Constant Propagation via Memory Classification Analysis

Naftali Schwartz*

March 15, 1999

**Abstract**

This article presents a novel Sparse Constant Propagation technique which provides a heretofore unknown level of practicality. Unlike other techniques which are based on data flow, it is based on the execution-order summarization sweep employed in Memory Classification Analysis (MCA), a technique originally developed for array dependence analysis. This methodology achieves a precise description of memory reference activity within a summary representation that grows only linearly with program size. Because of this, the collected sparse constant information need not be artificially limited to satisfy classical data flow lattice requirements, which constrain other algorithms to discard information in the interests of efficient termination. Sparse Constant Propagation is not only more effective within the MCA framework, but it in fact generalizes the framework. Original MCA provids the means to break only simple induction and reduction types of flow-dependences. The integrated framework provides the means to also break flow-dependences for which array values can be propagated.

## 1 Introduction

The propagation of sparse constants within aggregate data structures has proven significantly more difficult to perform practically than the propagation of scalar constants[WZ91]. In fact, the only work focusing on this problem to date[SK98] suggests that in the interests of efficiency no more than a small number of constant elements be collected for each program aggregate. The fundamental difficulty in generalizing scalar propagation to aggregate propagation has been that the classical data flow-based abstract interpretation, which can provide only a bounded number of constants with respect to the program size in the scalar case, can produce an unbounded number of constant elements within an aggregate.

This basic difficulty is of course not confined merely to the propagation of constant values, but arises whenever analysis of program aggregates is required. It happens that the summarization of the memory reference behavior of aggregates has been an extremely active area of research within the automatic parallelization community, where good aggregate summarization techniques have been held to hold the key to effective interprocedural dependence analysis for parallelization. Since dependence analysis in the scalar case computes a superset of the information required for constant propagation, as constants are propagated precisely across flow dependences, it turns out that the summarization techniques employed for general dependence analysis can be useful in generalized constant propagation. We will also see that the dependence analysis is itself considerably enhanced when combined with this propagation.

This paper is organized as follows. First, Section 2 provides a brief overview of basic dependence analysis for parallelization and reviews aggregate summarization techniques, focusing on the promising Linear Memory

*Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012-1185, nschwart@cs.nyu.edu.

Access Descriptor (LMAD). Section 3 describes the Memory Classification Analysis (MCA) build around the LMAD, and Section 4 describes the extensions to the LMAD and to MCA which would support constant propagation in tandem with dependence analysis. Section 5 then shows how the integration of these techniques leads to an improved MCA. Section 6 considers some advanced examples. The following Section 7 details limitations and possible extensions of our work. Section 8 reviews related work and Section 9 concludes.

# 2 Array Summarization for Dependence Analysis

An important motivation for dependence analysis is program parallelization[AK87], where it is desired to shorten the execution time of a program by splitting its execution across a number of processors. This program division will be incorrect if it does not respect program dependences. There are three important types of dependences: flow, anti and output. A statement $B$ is said to be *flow-dependent* on a statement $A$ if it reads a variable which $A$ has written, *anti-dependent* on $B$ if it writes a variable the $B$ has read, and *output-dependent* on $B$ if it writes a variable which $B$ has written. The latter two types of dependence are memory-related and can be "broken" by transforming the program to perform the write at $B$ (and subsequent reads of the value) to a fresh variable. SSA form[CFR+91] efficiently realizes this transformation.

However, the ineffectiveness of SSA form in the context of program arrays has resulted in the adoption of ad-hoc methods for dependence testing of array references. These methods provided a means for deciding the existence of dependence between any two array references based on their subscript expressions and surrounding loop bounds. This decision algorithm would then be applied to each pair of possibly aliased array references within the program section of interest, typically a candidate loop. While sufficient for small programs, the high asymptotic cost of pairwise comparisons has limited its application to larger codes.

To obtaining pragmatic implementations, various access summarization techniques have been studied, and a number of different paradigms have gained popularity. Linear constraint techniques [TIF86] are based on finding a Fourier-Motzkin [Wil76] solution to a system of linear inequalities which characterize the conditions under which a pair of array accesses can overlap. Although the Fourier-Motzkin procedure is expensive and produces real solutions rather than only the relevant integer ones, the Omega Test[Pug91] has extended it to alleviate these complications. However, precise linear constraints cannot completely describe accesses involving non-affine conditionals or when the overall shape of an access does not form a convex hull. Because the latter can be produced by an intersection operation between two otherwise convex regions, frequent approximation is inevitable.

Triplet notation[HK91], in which each access region is characterized by an upper bound, lower bound and stride in each dimension, has also gained a number of proponents. In Gu, et. al's[GLL97] interpretation of this scheme, every access contains a guard specifying the conditions under which it occurs. Approximation is avoided by maintaining high-level representations of operations which cannot be performed precisely. For example, a *difference list* of regions is associated with a region when not enough is known to perform a required difference operation. However, complications may arise in the presence of Fortran-style array reshapes, for which no general method for relating accesses interprocedurally has been devised. Furthermore, this scheme relies on the dubious assumption that all array access occur within declared bounds.

Recently, Paek, et al.[PHP98] have proposed a representation based on the Linear Memory Access Descriptor (LMAD). The LMAD achieves maximum precision by describing accesses uniformly as a (recursive) sequence of equidistant element references. An arbitrarily complex multidimensional traversal can be uniquely represented (up to LMAD equivalence) with only a base offset and a series of stride/span pairs (access dimensions). The size of each element completes the exact memory map for the access.

LMAD construction for an array access starts with the formation of a linearized access expression which is the sum of the products of each dimension's access expression and the size of each element of that dimension,

```
REAL A(14,*);
DO I=1,2
    DO J=1,2
        DO K=1,10,3
            ...  A(J,K+26*(I-1)) ...
        END DO
    END DO
END DO
```

$$
\begin{array}{l}
(\mathcal{A}^{3,\ 14,\ 26}_{9,\ 14,\ 26} - 26) \\
(\mathcal{A}^{3,\ 14}_{9,\ 14} + 26I - 26) \\
(\mathcal{A}^{3}_{9} + 26I + 14J - 26) \\
(\mathcal{A} + 26I + 14J + K - 26)
\end{array}
$$

Figure 1: Hoeflinger's Example Showing Expansion Through Indices

and in which each index variable has been normalized. This symbolic expression is repeatedly *expanded* through successive index variables, starting with the innermost. As each index variable is substituted out, a stride/span pair is inserted into the LMAD. The stride is set to the increase in base expression due to the increment by one of the normalized index variable, and the span is set to the difference between the base offset expression with the normalized index variable set to its minimum and maximum values. Figure 1 presents a corrected version of Hoeflinger's LMAD construction example[Hoe98] detailing the expansion through each index variable.

Conditional statements are taken into account by associating each LMAD with an execution predicate. Because the LMAD uses array declarations in the same manner they are used in a compiler, namely, to calculate actual element offsets, it avoids the complexity of continually relating subscript expressions to array bounds expressions. By explicitly linearizing array accesses, the LMAD offers a level of accuracy unmatched by the other representations: it directly supports programmer linearization of array references, traditionally the bane of dependence analysis, and offers full Fortran-style array reshape transparency while avoiding the in-bounds assumption.

# 3 Memory Classification Analysis

Memory Classification Analysis is an execution-order sweep through the program, which is assumed to be comprised of a series of nested contexts. Summary sets which describe the memory activity for each of these contexts are generated in a manner in which a single *representative dependence* of a particular type is present between the summary sets of two different contexts exactly when there is an actual dependence between the contexts. There are three summary sets for each context, or grain: Read-Only (RO), Read/Write (RW) and Write-First (WF). Together, they contain the collection of LMADs describing all memory activity. All dependences between two grains $A$ and $B$ are then precisely characterized by following equations:

$$
\begin{array}{ll}
\text{Flow:} & (A_{RO} \cap B_{RW}) \cup (A_{WF} \cap B_{RO}) \cup (A_{WF} \cap B_{RW}) \cup (A_{RW} \cap B_{RO}) \cup (A_{RW} \cap B_{RW}) <> \emptyset \\
\text{Anti/Output:} & (A_{RO} \cap B_{WF}) \cup (A_{WF} \cap B_{WF}) \cup (A_{WR} \cap B_{WF}) <> \emptyset
\end{array}
$$

Flow-dependences, unless of a simple inductive or reductive nature, prevent parallelization across dependence grains, while anti- and output-dependences can be removed using variable privatization.

The Access Region Test[HPP96] uses this classification scheme to check for dependences between loop iterations. First summary sets are formed for a single iteration of the loop. Then these sets are expanded through the loop index variable. One check is made for internal overlap in any of the descriptors contained in one of the two write sets. Another check is made for intersection among any of the three summary sets. The final check for intersection between descriptors within the same write set completes the cross-iteration dependence test.

The MCA framework also incorporates conditional array access, by attaching a (true by default) predicate to each Access Region Descriptor (ARD). The framework is kept efficient by the aggressive *coalescing* of related ARDs. For example, a series of writes to successive array indices under the similar conditions will be incorporated into the appropriate write set as only a single ARD. Aggressive simplification keeps the analysis efficient, as the analysis cost rises with the size of each summary set. To maintain complete precision, however, no coalescing is performed in the case where attached predicates are dissimilar.

# 4 Sparse Constant Propagation Extensions

The essential observation is that the Memory Classification Analysis can perform constant propagation almost for free. This may be accomplished by adding a "current value" field to each LMAD. Of course, this field will only be set for LMADs in the write sets. This field is then used much as any attached predicate as a discriminating factor for future coalescing operations.

In Figure 2 we show several styles of initializations with which our enhanced MCA must deal. The example in Figure 2(a) is taken from the regex library distributed with the GNU FSF version of `grep`[Foub]. Since none of the accessed array sections overlap, no special action over Paek's MCA is taken, other than the integration of the constant initializer value as part of the LMAD.

Figure 2(b), taken from the Huffman code inflater of the GNU FSF version of `gzip`[Fouc], is slightly more complex. In this case, the original MCA would actually coalesce the LMADs produced by the series of assignments together. In our case, we respect the differing constant values by disallowing the coalescing operation unless the assigned values also match.

The third example of Figure 2(c) is taken from the utility library of `zsh`[Tea]. In this case, the Write-First set generated from the first loop is $\{ \texttt{typtab}^1_{256} + 0 \}$, from the second loop $\{ \texttt{typtab}^1_{32} + 0, \texttt{typtab}^1_{32} + 128 \}$, and from the last statement $\{ \texttt{typtab} + 127 \}$. Because the latter two Write-First sets are subsets of the first, ordinary MCA would ignore them and use the first set as the overall program segment summary. However, in using the assigned constant value in the classification, we must split up the monolithic LMAD into its uniform constituent parts. The overall Write-First summary set which respects constant assignments is: $\{ [\texttt{typtab}^1_{32} + 0]_{1<<9}, [\texttt{typtab}^1_{95} + 32]_0, [\texttt{typtab}^1_{33} + 127]_{1<<9}, [\texttt{typtab}^1_{96} + 160]_0 \}$, where each LMAD is shown with its initial constant value as an outside subscript.

```
for ( c = 'a'; c <= 'z'; c++ )
    re_syntax_table[c] = 1;
for ( c = 'A'; c <= 'Z'; c++ )
    re_syntax_table[c] = 1;
for ( c = '0'; c <= '9'; c++ )
    re_syntax_table[c] = 1;
re_syntax_table['_'] = 1;
```

(a) Syntax Table Initialization

```
for ( i = 0; i < 144; i++ )
    l[i] = 8;
for ( ; i < 256; i++ )
    l[i] = 9;
for ( ; i < 280; i++ )
    l[i] = 7;
for ( ; i < 288; i++ )
    l[i] = 8;
```

(b) Huffman Code Initialization

```
for ( t0 = 0; t0 != 256; t0++ )
    typtab[t0] = 0;
for ( t0 = 0; t0 != 32; t0++ )
    typtab[t0] =
        typtab[t0+128] = 1<<9;
typtab[127] = 1<<9;
```

(c) Type Table Initialization

Figure 2: Array Initialization Examples

### 4.1 Expression Initializers

Although the above approach works well for constant initializers, in practice many useful initialization expressions are written in terms of index and other program variables. Propagation of these values requires us to store full expressions as initializers. These expressions can then be evaluated at selected values of the index variable using a sophisticated symbolic evaluation subsystem, such as that described by Gerlek, et al.[GSW95] for classifying induction sequences.

Furthermore, incorporating this capability permits us to retain the original structure of the analysis. Instead of keeping array sections with different initializers as separate LMADs, we can coalesce as usual and maintain precise initialization information using a complex initialization expression.

For example, the preceding summary Write-First set could also be simplified to:

$$\{\ [\mathtt{typtab}^1_{256} + 0]_{0<=i\&\&i<32||127<=i\&\&i<160?(1<<9):0}\ \}$$

where the variable `i` is the offset from the base of the descriptor. Although somewhat more complex, this scheme is preferable because it provides a uniform way of dealing with differing neighbor constant initializers and general expression initializations.

It is also worth noting that significant splintering of the LMADs, which is the inevitable alternative, can impact the performance of the MCA, which depends on maintaining summary sets with as few elements as possible to keep operations between and within sets efficient.

## 5 A Better Memory Classification Analysis

As described in Section 3, ordinary Memory Classification Analysis can only break flow-dependences of an inductive or reductive nature. However, our Sparse Constant Propagation enhancements provide the tools to break certain ordinary flow dependences as well. Figure 3 shows how a typical array dependence taken from the `diff3` program of the GNU FSF `diffutils` distribution[Foua] is broken.

Figure 3(a) shows an array `mapping` which is initialized in the first if-statement grain, and used in the second loop grain. These two grains would be inseparable by Memory Classification Analysis alone, but in tandem with Sparse Constant Propagation, the flow dependence can be broken by transforming the code to that shown in Figure 3(b).

This is performed as follows. The Write-First set for the earlier grain contains the LMAD

$$\{\ [mapping^1_3 + 0]_{i==0?0:strcmp(file[2],"-")==0?(i==1?2:1):(i==1?1:2)}\ \}$$

for `mapping`. When this value is then used in the second grain, we can use instead the closed form expression as a function of the index variable. This transformation removes the LMAD for `mapping` from the Read-Only summary set of the later grain, rendering the two grains parallelizeable.

This transformation is a generalization of induction variable substitution. In induction variable substitution, a closed form expression for an induction variable is generated in terms of the index variable to break a flow dependence. This transformation is more general in that the closed form expression is generated from a sequence of arbitrary array assignments.

## 6 More Complex Examples

In Figure 4(a)-(c) we present more complex cases. In all three cases, the Write-First summary set generated by the first loop is: $\{\ [c^1_5 + 0]_1\ \}$. In 4(a), the second loop contains a Write-First reference to `c` and to `a`.

```
if (strcmp (file[2], "-") == 0)
{
    mapping[0] = 0;
    mapping[1] = 2;
    mapping[2] = 1;
}
else
{
    mapping[0] = 0;
    mapping[1] = 1;
    mapping[2] = 2;
}
for (i = 0; i < 3; i++)
    rev_mapping[mapping[i]] = i;
```

(a) Original Code

```
if (strcmp (file[2], "-") == 0)
{
    mapping[0] = 0;
    mapping[1] = 2;
    mapping[2] = 1;
}
else
{
    mapping[0] = 0;
    mapping[1] = 1;
    mapping[2] = 2;
}
for (i = 0; i < 3; i++)
    rev_mapping[i==0 ?  0 :
      strcmp(file[2],"-")==0 ?
      (i==1?2:1) :  (i==1?1:2)] = i;
```

(b) Flow-Dependence Free Code

Figure 3: Broken Flow Dependence

These start out as $\{\ [c+i]_2,\ [a+i]_2\ \}$, where the initial value for `a` is simply copied from the constant value assigned to `c`. Expansion through the loop variable results in $\{\ [c_5^1+0]_2,\ [a_5^1+0]_2\ \}$. There is no internal overlap within descriptors, no overlap between descriptors of the same set, and no intersection between sets, so no further analysis is needed.

In case 4(b), the value written to each element of `a` comes from array `c`, but until we generate a summary of the behavior of the entire loop we cannot be sure that the value is indeed a constant. Therefore, the Write-First set for the first statement in the second loop starts out as $\{\ [a+i]_{c[i]}\ \}$. The Read-Only set for the statement is $\{\ [c+i]\ \}$. The Write-First set for the second statement of the second loop is $\{\ [c+i]_2\ \}$, which in tandem with the identical Read-Only set gets converted to a Read/Write set. When we expand the LMADs through the `i`, we can also change the `c[i]` initializer of the LMAD for `a` to 1, because it is now known that no other index of `c` was written to during the course of the loop, and every value written to `a` from `c` must have come from outside of the loop.

Case 4(c) is complicated because expansion of the LMADs through the second loop index and subsequent

```
for ( i = 0; i < 5; i++ )
    c[i] = 1;
for ( i = 0; i < 5; i++ )
{
    c[i] = 2;
    a[i] = c[i];
}
```

(a) Write Before Read

```
for ( i = 0; i < 5; i++ )
    c[i] = 1;
for ( i = 0; i < 5; i++ )
{
    a[i] = c[i];
    c[i] = 2;
}
```

(b) Write After Read

```
for ( i = 0; i < 5; i++ )
    c[i] = 1;
for ( i = 0; i < 5; i++ )
{
    a[i] = c[i];
    c[4-i] = 1;
}
```

(c) Writes and Reads Intertwined

Figure 4: More Complex Cases

application of the Access Region Test reveals overlap between the Read-Only and Write-First summary sets which contain overlapping references to array `c`. In this and similar complex cases, a good deal of sophistication will be required to precisely characterize the sequence of values assigned from `c`. Therefore, it seems appropriate to fall back on more general and expensive element-level methods such as [SK98].

# 7 Limitations and Extensions

The Memory Classification analysis described by Hoeflinger[Hoe98] does not support recursive cycles in the call graph. Extension of MCA by iteration over these cycles would make the method more general.

Although our discussion has been couched only in terms of propagation of constants and expressions, this technique also has very important implications for aggregate Shape Analysis. Shape Analysis[SRW96] computes a conservative description of the structure of program heap space produced by a series of pointer operations. What has not been yet been shown is how to handle shape analysis over an array of pointers. Using the present constant summarization technique, and allowing Static Shape Graphs[SRW96] for constant "values", we can obtain a precise and concise description even of arrays of heap objects. This line of research has been pursued in a companion paper[Sch99].

# 8 Related Work

The first practical algorithm for Constant Propagation based on SSA form was by Wegman and Zadeck[WZ91]. Extending this for Sparse Constant Propagation could be accomplished only inefficiently by the treatment of entire aggregates as scalars and the introduction of special Access and Update functions as suggested by Cytron, et. al[CFR+91].

Choi, et. al[CCF94] showed that introducing these functions not only increases the volume of imprecise information in the analysis, but actually obscures the data-flow by replacing all def-def chains with use-def chains even when this is semantically inaccurate. Instead, they advocated maintaining def-def chains, but in a factored form which would keep the analysis manageable.

Steensgaard[Ste95] improved on this factorization through the use of storage fragmentation. He showed that by classifying areas of storage as being the referents of only a limited number of program pointers, further reduction of the SSA into *assignment factored* form was possible. He also demonstrated that finer levels of fragmentation could be obtained by more elaborate pointer analysis.

Knobe and Sarkar's recent introduction of Array SSA form[KS98] has provided a new unified framework for reasoning about program aggregates, offering precise data flow information on a per-element basis. However, this is also its biggest weakness. The Array SSA-based Sparse Constant Propagation[SK98] algorithm they propose cannot deal effectively with arrays of unbounded size. Indeed, as presented in their paper, Array SSA form relies on the allocation of additional space of the same magnitude as the original arrays, something which may not be known until run-time. True, the Array SSA technique lends itself to a run-time implementation, but this is irrelevant for constant propagation, which is a fundamentally static analysis.

The present work can be seen as extending the assignment factored SSA form of Steensgaard to fragment not only program aggregates from one another, but to logically fragment particular aggregates into constituent pieces according to the manner in which they are used in the program. The original MCA merely classifies memory accesses by type of access; this work introduces the finer classification within write accesses of the value written, producing an effective Sparse Constant Propagation algorithm which operates within the framework of MCA.

# 9 Conclusion

Paek, et al.[PHP98] expressed the belief that simplification of array access patterns using the notions of strides and spans would benefit all array-related analyses. This study has shown how this technique may be profitably applied to the long-standing Sparse Constant Propagation problem. A number of other solutions have been proposed, but because they all operate fundamentally at the element level, they cannot deal effectively with the potentially unlimited volume of information which can be generated.

Hoeflinger[Hoe98] proposes an elaborate interprocedural analysis framework built around a powerful Region processor. Benefits of this would be both conceptual and practical. On the one hand, this architecture would neatly sidestep the age-old analysis phase-ordering dilemma. On the other, the entire system would benefit uniformly from incremental improvements to this central processor. This phenomenon has been amply demonstrated in this work, where enhancing the Region processor for Sparse Constant Propagation has produced a more powerful Memory Classification Analysis capable of breaking array-based flow dependences.

# 10 Acknowledgements

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; by the National Science Foundation under grant number CCR-94-11590. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

# References

[AK87] Randy Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[CCF94] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2), 1994.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Foua] GNU Free Software Foundation. diffutils. Available as ftp://prep.ai.mit.edu/pub/gnu/diffutils/diffutils-2.7.tar.gz.

[Foub] GNU Free Software Foundation. grep. Available as ftp://prep.ai.mit.edu/pub/gnu/grep/grep-2.0.tar.gz.

[Fouc] GNU Free Software Foundation. gzip. Available as ftp://prep.ai.mit.edu/pub/gnu/gzip/gzip-1.2.4.tar.

[GLL97] Junjie Gu, Zhiyuan Li, and Gyungho Lee. Experience with efficient array data flow analysis for array privatization. *ACM SIGPLAN Notices*, 32(7):157–??, July 1997.

[GSW95] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[HK91] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[Hoe98] J. Hoeflinger. *Interprocedural Parallelization Using Memory Classificaiton Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 1998.

[HPP96] Jay Hoeflinger, Yunheung Paek, and David Padua. Region-based parallelization using the region test. Technical Report 1514, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Devevelopment, December 1996.

[KS98] K. Knobe and V. Sarkar. Array SSA Form and its use in Parallelization. In *Principles of Programming Languages*, pages 107–120, 1998.

[PHP98] Yunhueng Paek, Jay Hoeflinger, and David Padua. Simplification of array access patterns for compiler optimizations. *ACM SIGPLAN Notices*, 33(5):60–71, May 1998.

[Pug91] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pages 4–13, Albuquerque, N.M., November 1991.

[Sch99] N. D. Schwartz. Memory Classification Analysis for Recursive C Structures. Technical Report TR1999-776, New York University, Department of Computer Science, Feb 1999.

[SK98] V. Sarkar and K. Knobe. Enabling Sparse Constant Propagation of Array Elements via Array SSA Form. In *Static Analysis Symposium*, pages 33–56, 1998.

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg Beach, Florida, 21–24 January 1996.

[Ste95] Bjarne Steensgaard. Sparse functional stores for imperative programs. *ACM SIGPLAN Notices*, 30(3):62–70, March 1995.

[Tea] Zsh Development Team. zsh. Available as ftp://ftp.sterling.com/zsh/zsh-3.1.0.tar.gz.

[TIF86] Rémi Triolet, François Irigoin, and Paul Feautrier. Direct parallelization of Call statements. In *ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 175–185, Palo Alto, Calif., June 1986.

[Wil76] H. P. Williams. Fourier-Motzkin elimination extension to integer programming problems. *Journal of Combinatorial Theory (A)*, 21:118–123, 1976.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.