LISP AND SYMBOLIC COMPUTATION: An International Journal 2:3/4: 179–396, 1989

The Interprocedural Analysis and Automatic Parallelization of Scheme Programs

WILLIAMS LUDWELL HARRISON III* (*harrison@uicsrd.csrd.uiuc.edu*)

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
305 Talbot Laboratory
104 South Wright Street
Urbana, IL 61801

(Received: March 1, 1989)

(Revised: May 10, 1989)

Keywords: Interprocedural Analysis Abstract Interpretation, Automatic Parallelization, Program Transformation, Parallel Processing, Symbolic Computation, Lisp, Scheme

Abstract. Lisp and its descendants are among the most important and widely used of programming languages. At the same time, parallelism in the architecture of computer systems is becoming commonplace. There is a pressing need to extend the technology of automatic parallelization that has become available to Fortran programmers of parallel machines, to the realm of Lisp programs and symbolic computing. In this paper we present a comprehensive approach to the compilation of Scheme programs for shared-memory multiprocessors. Our strategy has two principal components: *interprocedural analysis* and *program restructuring*. We introduce *procedure strings* and *stack configurations* as a framework in which to reason about interprocedural side-effects and object lifetimes, and develop a system of interprocedural analysis, using abstract interpretation, that is used in the dependence analysis and memory management of Scheme programs. We introduce the transformations of *exit-loop translation* and *recursion splitting* to treat the control structures of iteration and recursion that arise commonly in Scheme programs. We propose an alternative representation for s-expressions that facilitates the parallel creation and access of lists. We have implemented these ideas in a parallelizing Scheme compiler and run-time system, and we complement the theory of our work with "snapshots" of programs during the restructuring process, and some preliminary performance results of the execution of object codes produced by the compiler.

*This work was supported in part by the **National Science Foundation** under Grant No. NSF MIP-8410110, the **U.S. Department of Energy** under Grant No. DE-FG02-85ER25001, the **Office of Naval Research** under Grant No. ONR N00014-88-K-0686, the **U.S. Air Force Office of Scientific Research** under Grant No. AFOSR-F49620-86-C-0136, and by a donation from the **IBM Corportation**.

Contents

1 Introduction **185**
1.1 Motivation and Approach 185
1.2 The Input Language: Scheme 187

2 The Interprocedural Analysis of Scheme Programs **189**
2.1 Motivations 189
2.1.1 Side-Effects and Dependence Analysis 190
2.1.2 Object Lifetimes and Memory Management 190
2.1.3 Folding Procedural Constants and Merging Contours 192
2.2 Overview of our Approach 192
2.3 Notational Conventions 194
2.4 Abstract Interpretation 195
2.5 Concrete Semantics 199
2.5.1 The Language $\mathcal{L}$ 199
2.5.2 Procedure Strings 201
2.5.3 A Semantics for $\mathcal{L}$ in Terms of Procedure Strings . . 203
2.5.4 Abstraction in the Face of Reflexivity 207
2.5.5 Modified Domain Definitions for $\mathcal{L}$ 208
2.5.6 A Modified Semantics for $\mathcal{L}$ 210
2.6 Optimal Solutions in Terms of Procedure Strings 217
2.6.1 Side-Effects, in Terms of Procedure Strings 218
2.6.2 Stack Allocation, in Terms of Procedure Strings . . 223
2.6.3 Generalized Hierarchical Allocation and Deallocation 224
2.6.4 Examples of Side-Effects and Object Lifetimes . . . 225
2.6.5 Some Observations 227
2.7 Stack Configurations 228
2.8 The Abstraction of Operations Over Procedure Strings . . . 231
2.9 Abstract Semantics 236
2.10 Approximate Solutions in Terms of Stack Configurations . . 252
2.10.1 Side-Effects, in Terms of Stack Configurations 253
2.10.2 Stack Allocation, in Terms of Stack Configurations . 254
2.10.3 Generalized Hierarchical Allocation and Deallocation 255
2.11 A Shift in Perspective (and in Accuracy) 257

2.11.1 Side-Effects under $\mathcal{E}_4$ . . . 272
2.11.2 Stack-Allocation under $\mathcal{E}_4$ . . . 274
2.11.3 Generalized Hierarchical Storage Management . . . 275
2.12 Adding Flow-Sensitivity to the Analysis . . . 275
2.13 Examples of Analysis under $\mathcal{E}_5$ . . . 289
2.14 Mutable Data and Aliasing . . . 291
2.15 Management of a Hierarchical, Shared Memory . . . 296
2.16 In the Absence of `call/cc` . . . 299
2.17 Environment Pruning . . . 306

3 The Automatic Parallelization of Scheme Programs 308
3.1 The Program Representation . . . 309
3.2 Preparatory Optimizations . . . 314
3.2.1 Contour Merging . . . 315
3.2.2 Tail-Recursion Elimination . . . 317
3.2.3 Common Subexpression Elimination . . . 320
3.2.4 More Contour Merging . . . 322
3.3 Exit-Loop Translation . . . 322
3.3.1 Replacing Exits with Recurrences . . . 325
3.3.2 Variable Expansion . . . 327
3.3.3 Loop Distribution . . . 329
3.3.4 Reordering the Subloops . . . 331
3.3.5 Eliminating Unused Computation . . . 331
3.3.6 The Parallel Computation of the Number of Iterations 334
3.3.7 Marking Doalls and Recurrences . . . 336
3.3.8 Closed-Form Solution for the Number of Iterations . 336
3.3.9 Restructuring the Recurrences . . . 339
3.3.10 Allocating and Initializing Expanded Variables . . . 341
3.3.11 Loop Fusion . . . 341
3.3.12 Cobegin Insertion . . . 341
3.4 Recursion Splitting . . . 343
3.4.1 Overview . . . 345
3.4.2 Forming the Forward and Backward Loops . . . 346
3.4.3 Exit-Loop Translation of the Forward Loop . . . 347
3.4.4 Variable Expansion and the Bottom of Recursion . . 356

3.4.5 Parallelization of the Backward Loop 356
3.5 High-Level (Coarse-Grained) Parallelism 358
3.6 Organization of the Compiler 381
3.7 S-expressions in Parcel 382
3.8 Relation to Previous Work 386
3.9 Preliminary Performance Results 389

4 Preliminary Performance Results **389**

5 Conclusions **390**

6 Acknowledgements **391**

7 Vita **396**

List of Figures

1 A Sample Scheme Program 200
2 The Sample Program Rewritten in $\mathcal{L}$ 201
3 Domain Definitions for $\mathcal{S}_1$ and $\mathcal{E}_1$ 204
4 The Semantic Function $\mathcal{S}_1$ 206
5 The Semantic Function $\mathcal{E}_1$ 206
6 Domain Definitions for $\mathcal{S}_2$ and $\mathcal{E}_2$ 210
7 The Semantic Function $\mathcal{S}_2$ (part 1 of 2) 211
8 The Semantic Function $\mathcal{S}_2$ (part 2 of 2) 212
9 The Semantic Function $\mathcal{E}_2$ 212
10 Example of Stack-Allocated Variables 225
11 Example of Side-Effects and Object Lifetimes 226
12 Abstract Domains for $\mathcal{E}_3$ 237
13 Abstraction Maps 238
14 Partial Orderings 239
15 LUB Operators Over the Abstract Domains 240
16 The Semantic Function $\mathcal{S}_3$ (Part I) 243
17 The Semantic Function $\mathcal{S}_3$ (Part II) 244
18 The Semantic Functions $\mathcal{S}_3^*$ and $\mathcal{E}_3$ 244
19 An Example of the Inaccuracy of $\mathcal{E}_3$ 257

20 An Example of the Inaccuracy of $\mathcal{E}_3$, Rewritten in $\mathcal{L}$ 258
21 Abstraction Maps 260
22 Auxiliary Functions for $\mathcal{E}_4$ 262
23 The Semantic Function $\mathcal{S}_4$ (Part I) 263
24 The Semantic Function $\mathcal{S}_4$ (Part II) 264
25 The Semantic Function $\mathcal{S}_4$ (Part III) 265
26 The Semantic Functions $\mathcal{S}_4^*$ and $\mathcal{E}_4$ 265
27 Example of Overlapping Variable Lifetimes 276
28 Abstract Domains for $\mathcal{E}_5$ 277
29 *RdEnv* and *WrEnv* 279
30 Abstraction Maps 280
31 Partial Orderings 281
32 LUB Operators Over the Abstract Domains 282
33 Movement Functions for $\mathcal{E}_5$ 283
34 The Semantic Function $\mathcal{S}_5$ (Part I) 285
35 The Semantic Function $\mathcal{S}_5$ (Part II) 286
36 The Semantic Function $\mathcal{S}_5$(Part III) 287
37 The Semantic Functions $\mathcal{S}_5^*$ and $\mathcal{E}_5$ 288
38 Example of Side-Effects and Object Lifetimes 290
39 `sum-of-integers`, Rewritten in $\mathcal{L}$ 290
40 `cons` in Terms of Closures 291
41 Example of Side-Effects and Object Lifetimes 292
42 User Structures, in Terms of Closures 293
43 An Abstraction of `cons`, for Analysis 295
44 A Procedure Calling Graph 297
45 The Semantic Function $\mathcal{S}_6$ 300
46 The Semantic Function $\mathcal{E}_6$ 300
47 The Semantic Function $\mathcal{S}_7$ (Part I) 302
48 The Semantic Function $\mathcal{S}_7$(Part II) 303
49 The Semantic Functions $\mathcal{S}_7^*$ and $\mathcal{E}_7$ 304
50 `fact`, Before and After CPS Conversion 306
51 Quicksort 309
52 Quicksort Program, after Macro-Expansion 310
53 The Initial Representation of Quicksort (Part 1) 312
54 The Initial Representation of Quicksort (Part 2) 313

55 $-$-sortby-& is Merged into $-$-sortby 315
56 Tail-Recursion is Eliminated from $-$-splitby 317
57 A Continuation-Passing Version of Factorial 319
58 A Common Subexpression is Eliminated in $-$-sortby . . 320
59 $-$-splitby is Merged into $-$-sortby 323
60 The Quicksort Program, after Preparatory Transformations 324
61 Exit Branches are Eliminated from $-$-sortby 326
62 Variables are Expanded in $-$-sortby 328
63 Loops are Distributed in $-$-sortby 330
64 Distributed Loops are Reordered in $-$-sortby 332
65 Exit Path Computations are Eliminated in $-$-sortby . . 333
66 Recurrences and Parallel Loops are Identified in $-$-sortby 337
67 A Closed-Form Solution for t-59 is Found 338
68 Recurrences are Restructured for Parallel Execution 340
69 The Final, Parallel Version of $-$-sortby 342
70 The Procedure tak 343
71 The Initial Representation of tak 344
72 Forward and Backward Loops are Formed in $-$-tak . . . 346
73 Variables Defined in the Forward Loop are Expanded 348
74 The Forward Loop is Cleaned Up Before Proceeding 350
75 The Forward Loop is Distributed, and the Subloops Reordered351
76 Exit Path Computations are Deleted in $-$-tak 352
77 Parallel Loops (from the Forward Loop) are Recognized . . 353
78 A Closed-Form Solution for t-44 is Found 354
79 The Restructuring of the Forward Loop is Completed . . . 355
80 Variables Defined in the Backward Loop are Expanded . . . 357
81 The Backward Loop is Distributed 359
82 Parallel Loops and Recurrences are Recognized 360
83 Parallel Loops are Coalesced 361
84 After Recursion Splitting 362
85 The Final (Parallel) Version of $-$-tak 363
86 The Procedures rewrite and rewrite-args 364
87 The Procedure rewrite-with-lemmas, and its Subroutines 365
88 The Procedure $-$-rewrite-with-lemmas After Parsing . 366
89 $-$-rewrite-with-lemmas-one-way-unify-lst 367

90 The Procedure `$-$-rewrite-with-lemmas-one-way-unify` After Parsing . . . 368
91 The Procedure `$-$-rewrite-args` After Parsing . . . 369
92 The Procedure `$-$-rewrite` After Parsing . . . 369
93 The Forward and Backward Loops are Formed . . . 370
94 Variables Defined in the Forward Loop are Expanded . . . 371
95 The Forward Loop is Cleaned up Before Proceeding . . . 371
96 The Forward Loop is Distributed . . . 372
97 Subloops of the Forward Loop are Reordered . . . 373
98 Exit-Path Computations are Eliminated . . . 374
99 Doall Loops and Recurrences are Recognized . . . 374
100 A Closed-Form Solution is found for `t-204` . . . 375
101 Subloops of the Forward Loop are Fused . . . 375
102 `allocate` and `restore` Forms are Introduced . . . 376
103 Recurrences from the Forward Loop are Translated . . . 377
104 Variables Defined in the Backward Loop are Expanded . . . 378
105 Doalls and Recurrences from the Backward Loop are Recognized . . . 379
106 Recurrences from the Backward Loop are Translated . . . 380
107 The Final, Parallel Version of `$-$-rewrite-args` . . . 380
108 The Organization of the Parcel Compiler . . . 382
109 Two S-expressions Using Parcel's Representation . . . 383
110 The Result of appending to `y` . . . 385
111 An Unusual Case of Sublist Sharing . . . 387
112 Preliminary Performance Figures for Parcel - CPU+GC Seconds . . . 390

1 Introduction

1.1 Motivation and Approach

Lisp figures prominently among programming languages, in part because it is, by the standards of our discipline, an old language, and therefore enjoys what little respect time affords the creations of engineering; in part because it is fundamentally elegant and powerful, as is Church's lambda calculus [16], upon which it is loosely based; and in part because of the singular flexibility of its central data structure, the list. The drawbacks

of Lisp programming, in particular the problems of dynamic binding of variables and of the proliferation of dialects, have been addressed, and seem in large part to have been alleviated by promulgation of the Scheme [41] and Common Lisp [6] standards. Its expressive strength, and its wide portability insure that Lisp will remain a popular programming language, or, at the very least, will exert a potent influence over future language designs.

At the same time, parallelism in the architecture of computer systems has become commonplace. There is little doubt that it offers the most direct route to very high rates of computation. Machines such as the Alliant FX/8 [2], the BBN Butterfly [1], and the Thinking Machines Connection Machine [27], have made parallel processing commercially viable, and few ideas in science have more enduring impact than those that make their originators wealthy.

The collision of the forces of software engineering and those of parallel architecture has brought forth any number of alternative solutions to the problem of programming these new machines. At the risk of drawing artificial boundaries, we may divide these solutions into those which would do the work of parallelization automatically, and those which would leave such work to the programmer. This criterion is artificial in that there are probably no systems of parallel programming which leave every detail of parallel execution to the programmer, and likewise none (or few) which require no effort beyond that needed to develop the same program for a sequential machine. We may likewise characterize an approach by the class of machines to which it is applicable. Again, there is probably no approach which is uniformly effective, across all parallel architectures, nor one which has nothing to offer beyond its applicability to a single machine.

Nevertheless, to proceed as though these distinctions were hard and fast, the solution proposed in this paper is fully automatic, and applicable to shared-memory multiprocessors, such as IBM's RP3 [40], Alliant's FX/8 [2], or the Cedar machine of the University of Illinois [30]. In short, we propose the design of an optimizing compiler to produce an object code for a shared-memory multiprocessor from a sequential Scheme program. Except where stated explicitly in the text, no restrictions are placed upon the program.

Our compilation strategy will have two large components: *interprocedural analysis* and *program restructuring*. The goal of interprocedural analysis will be to collect information concerning *interprocedurally visible side-effects* and the *lifetimes of dynamically instantiated objects*. We introduce *procedure strings* as a framework in which to reason about side-effects and object lifetimes at run-time, and *stack configurations*, an abstraction of procedure strings, as a framework in which to reason about side-effects and object life-

times at compile-time. In terms of these structures we specify a system of interprocedural analysis as an *abstract interpretation* of Scheme programs, prove its correctness, and show that it is a powerful basis for the *dependence analysis* and *memory management* of Scheme programs.

Our discussion then takes something of a sharp turn, and we introduce the system of program transformation used to parallelize Scheme programs automatically in Parcel [4], the compiler and run-time system in which the ideas of this paper have been implemented. Having collected interprocedural information, and having used it to assess the interprocedural dependence structure of the computation at hand, and the lifetimes of the objects created during the computation, the Parcel compiler turns to each procedure of the program being compiled, and restructures it into two versions: one parallel, one sequential. The run-time system we have constructed to complement the compiler makes use of these two versions to achieve good utilization of processors without excessive overhead. We introduce the transformations of *exit-loop translation* and *recursion splitting* to treat the iterative and recursive control structures found commonly in Lisp and Scheme programs. We illustrate these transformations with "snapshots" of programs taken during the compilation process. The Parcel run-time system makes use of an unusual representation for s-expressions that facilitates the parallel creation and access of lists, and allows the fast solution of recurrence relations over list data. We show how the Parcel compiler extracts and recognizes such recurrence relations, and illustrate the representation and its properties. We present some preliminary performance results of object codes produced by the Parcel compiler and executed on an Alliant FX/8, under the Parcel run-time system. Finally, we compare this work with that of researchers in related areas.

1.2 The Input Language: Scheme

The language accepted by the Parcel compiler is Scheme, as defined in [41]. What makes Scheme appropriate as the input language to our compiler? First, it is a small language, with semantics that are clear and simple. This is valuable when writing a conventional compiler; it is utterly invaluable when writing a compiler that performs detailed analysis and radical transformation of an input program. Each transformation is an opportunity to violate the semantics of a program; if those semantics are simple, then a proof of its correctness will be more manageable and believable.

Second, it is a language of powerful and general constructs. We can be sure that techniques that are effective when applied to Scheme will find applications, often in a restricted or specialized sense, to other languages. (We will point such applications out from time to time.)

Third, much useful computation can be performed in Scheme without the use of side-effects. We will see that side-effects are largely a matter of perspective, and that there is no reason to throw up our hands simply because a function modifies free variables or compound data. Nonetheless, code that is laden with side-effects will certainly be more difficult to parallelize than code that is not. For this reason a language that encourages programming without side-effects is appropriate as input to a parallelizing compiler.[1]

The language accepted by most restructuring compilers is Fortran [39]. What advantages does Scheme offer over Fortran? It might be thought that the flexibility of dynamically allocated storage and pointers would impede the dependence analysis, and thus the automatic parallelization of a program. In fact, we might argue that the contrary is the case, for a somewhat subtle reason. A Fortran program begins running with all of the storage upon which it will operate declared statically.[2] This means that as execution proceeds deeper into the calling tree, data that is being read and written is ever more likely to have been previously written, and to be subsequently read. There is no mechanism for allocating storage whose lifetime is restricted to a subtree of the calling tree.[3] By contrast, a Scheme program may (and typically does) allocate storage at all points of a computation, and by static analysis we may discern that a dynamically allocated object is limited, in lifetime, to a particular subcomputation. Such restricted lifetime is the stuff of parallelism, as well as the stuff of efficient memory management, as we will see in section 2.

Finally, the reader who is familiar with conventional implementations of Scheme, particularly those based upon *continuation-passing style conversion* (CPS conversion) should put aside assumptions concerning the implementation of Scheme's features, particularly concerning the environment, procedure calling and returning, and first-class continuations. For example, we will speak of the objects created by invocation of `call/cc` (continua-

[1]It is interesting that while the Scheme definition leaves the order of evaluation of the arguments to a procedure application unspecified, this is no help in parallelization, as the simple example ⟦`(f (set! x (1+ x)) (set! x (1+ x)))`⟧ shows. The arguments to `f` may be evaluated in any order (with the same outcome) but not simultaneously, because of race conditions. Since such effects may occur remotely, non-trivial automatic parallelization seems to require interprocedural information.

[2]This is true of most implementations of Fortran, although it is not mandated by the standard.

[3]This is a bit simple-minded. In fact, parallelizing compilers for Fortran expand scalar variables into vectors or arrays as a matter of course, and for precisely this reason. However, the more complex a data structure becomes (scalar variables are the limiting case of simplicity of structure) the more difficult it becomes to expand it; we would argue that the penalty for failing to do so is lower in Lisp than in Fortran, since much dynamic allocation of storage occurs in the course of a Lisp program.

tions) as distinct from those created by evaluation of `lambda` expressions (closures). In some implementations of Scheme, there is no distinction between these objects, because first-class continuations are implemented as lexical closures; nevertheless, we will find it useful to distinguish these two types.

It will be important for us to speak precisely about identifiers, variables, and bindings. Here, *identifier* will be used to mean a textual object, such as `x` or `car`, which is written by a programmer. A *variable* (which has an identifier), may be a formal parameter or local variable of a lambda expression, or global (defined outside all user lambda expressions). In the expression `(lambda (x y) (lambda (x) (set! y z)))`, there are three formal parameters and one free variable; two of the formal parameters have the same identifier (`x`). The free variable (whose identifier is `z`) may be global, or may be a formal parameter or local variable of a lambda expression which surrounds this expression. By a variable *binding* we will mean the association of a variable with a location in memory. In terms of our example, if the outer lambda expression is applied to two values, two memory locations will be set aside for the instances of its formal parameters. We say that the variables are *bound* to these locations. These locations will initially contain the values to which the function was applied; if the inner lambda expression is subsequently applied, the value of `y` will be altered (it will be assigned but not rebound). Identifiers and variables exist at compile-time, whereas bindings exist at run-time.

2 The Interprocedural Analysis of Scheme Programs

2.1 Motivations

Abstract interpretation [17, 18] and dataflow analysis [26, 14] share the goal of deriving information from a program text that is at once specific enough to permit the efficient implementation of the computation it expresses, and general enough to be valid in every state in which the program may be executed. The advantage of the former is in viewing this process as an abstraction of a denotational definition of the program: exact properties which hold for an instance of the program over a particular input data set, are reflected in the abstract domain as less exact properties which hold over many input data sets. The advantage of the latter is its operational nature: the conditions for optimization frequently depend upon mechanical or structural qualities of the computation which are most easily gleaned from, for example, the program's control flow graph. In this section we will borrow ideas from both. Our goal is the formulation of an interprocedural dataflow analysis framework for Scheme programs, but this framework will

be derived through a series of alternate semantics for the language (some concrete, some abstract) whose formal properties will give us confidence that the analysis provides sensible information.

2.1.1 Side-Effects and Dependence Analysis

Before creating a program analysis framework, it is well to have questions in mind whose answers justify the expense of analysis. In our case, the first objective is the automatic parallelization of Scheme programs by a restructuring compiler, and in particular, the automatic extraction of *high-level* (or *coarse-grained*) *parallelism* from programs: parallelism which results in the concurrent execution of lengthy, interprocedurally involved subcomputations. Part of this compilation process is the *dependence analysis* [12, 13, 45] of the program: in order for restructuring and parallelization to proceed, the precedence constraints of the original computation must be discovered. Specifically, we wish to construct for each procedure application in the program, a set which identifies the mutable objects (variables, cons cells, vectors, etc.) that may be modified (written) during the subcomputation that is initiated by the application and terminated by its return. Likewise, we wish to construct a set which identifies the objects that may be used (read) during the subcomputation. We will refer to these as *def* and *use* sets, respectively. Ideally, these sets would include only those objects that are relevant from the caller's point of view. We require that they have a reasonable and useful interpretation in light of the irregular, non-local control flow made possible by `call/cc`.

Our use for these sets will, as mentioned above, be in inferring the dependence structure of a computation, in order that high-level parallelism may be extracted from it.[4] We will intersect the *use* and *def* sets of two procedure invocations to discover any dependence constraints between them. We must therefore err, in our estimation of side-effects, in favor of adding too many objects to a *use* or *def* set. That is, our program transformations will be legal only in the absence of certain dependences. We must therefore arrange that, if such dependences might exist at run-time, they are represented in our *def* and *use* sets at compile-time.

2.1.2 Object Lifetimes and Memory Management

The second major application of our program analysis framework is to the *memory management* of Scheme programs, whether sequential or parallelized. For instance, much of the effort of implementing Scheme effi-

[4] If our only concern was the identification of fine-grained parallelism, we could restrict our attention to "innermost" computations which do not cross procedure boundaries; such is the approach of most vectorizing compilers, which therefore do not depend so heavily upon interprocedural analysis.

ciently is spent in the careful handling of first-class closures and continuations. Their generality means that, in the absence of information proving otherwise, a variable that may be captured by lexical closure must be heap-allocated, to provide for the event in which it is referenced after the procedure by which it is bound has terminated.[5] This difficulty, known as the *upward funarg* problem [8], pales by contrast to the implications of `call/cc`. A continuation, when applied, reactivates all procedure instances that were active ("on the stack") at its creation. This application may occur after these procedures have been exited normally, with the consequence that any variables needed for their resumption must be given space in the heap (and subsequently reclaimed). Yet, as when they are captured by closures in continuation passing style [5, 29] or by continuations used as non-local exits ("throws"), it is frequently the case that variables need not be heap-allocated; the difficulty is simply in anticipating the lifetime of the closure or continuation, in comparison to the lifetimes of the variable bindings it captures.[6] Because the notions of object lifetime and dependence are so nearly related, we may as easily turn our analysis of side-effects to the problem of allocating variables on a stack, when their lifetimes permit. This leads to a *simple and efficient implementation of closures and continuations* at run-time.

A closely related problem depends, as well, upon anticipating the lifetimes of objects: that of their *placement within a hierarchical memory.* Suppose that the machine for which we are compiling has not a single shared memory visible to all processors, but instead a hierarchy of shared memory, with the property that locations which are lower (nearer to the processors) in the hierarchy are visible to fewer processors, but less costly to access, than locations which are higher in the hierarchy. During the execution of the code produced by our compiler, objects will be allocated simultaneously by many processors. Each such object must be allocated as low in the hierarchy as possible for the sake of access time, but high enough to ensure its visibility to all processors making use of it.[7] This may be seen

[5]This is too strongly stated. It need only be the case that such a variable *appear* to have been heap-allocated, that provision be made for reference to it subsequent to the termination of the procedure that binds it. There is no end of run-time devices to effect this appearance.

[6]In the case of continuation passing style, closures representing continuations are passed downward, as parameters. As long as these closures are only applied, captured in other downward closures, or passed as parameters to further procedure applications, they cannot outlive the procedure instances that bind the free variables they contain. `call/cc`, of course, is a mechanism by which the user can gain access to these continuation objects, and having done so he will straightaway store one in a global variable and ruin everything.

[7]Clearly, such placement depends as much upon the processor allocation and scheduling discipline used in executing a parallel program, as upon the anticipated lifetimes of the objects being allocated. We will make our assumptions explicit when we define the

as a variation on the heap versus stack problem above, in which instead of two choices of an area from which to allocate, there is a spectrum from least expense / shortest lifetime to greatest expense / longest lifetime. We will consider the problem in this latter form until, in subsection 2.15, we treat the problem which interests us more directly.

2.1.3 Folding Procedural Constants and Merging Contours

Another significant optimization made possible by the analysis described below is the *folding of procedural constants.* For instance, we may expand applications of intrinsic procedures in-line (sometimes called *open coding*), when it is determined that a variable in the operator position of an application has as its only value an intrinsic procedure. This is, in general, made impossible by Scheme's semantics, which allow that the global variable `car`, for example, may be assigned the value of `cdr` during a procedure invocation, with the result of changing the behavior of all users of the variable from that point. By couching this as a constant propagation problem [46], we are able to detect both those applications of an intrinsic function that are made from the top-level variable by the same name, as well as those that occur as the result of parameter passing or assignment to a different variable; and we are able to do so without alteration to the semantics of the language.[8]

As we will see in section 3, the analysis described below can also be applied to the problem of *contour merging*, or more generally, of expanding user procedures in-line. This has the effect of eliminating needless procedure calls, and of making the computation performed by a program more visible to the compiler.

2.2 Overview of our Approach

We will develop solutions to the above problems in several steps. First, we propose an alternate semantics for Scheme, nearly related to the standard semantics, which introduces the *procedure string*, a device for recording the interprocedural behavior of a running program. The straightforward abstraction of this semantics leads to abstract domains containing higher-order objects (functions) over reflexive domains, whereas our purpose requires a more concrete compile-time representation of the values assumed by variables. We therefore modify the semantics such that its abstraction results in domains which are both finite and non-reflexive.

problem more sharply, at the end of this section.

[8]This optimization is less dependent upon the specifics of our analysis than are the other optimizations mentioned above. It is a by-product of any method of constant propagation over procedural domains that retains enough information to determine when intrinsic functions are being applied.

Second, we propose an optimal solution to each of the problems described above (identification of side-effects, stack-allocation of variables, and placement of data within a hierarchical memory) in terms of procedure strings, similar in spirit to the MIN algorithm for page replacement: a solution that is unobtainable, because it requires foresight and pertains only to a single instance of the program (that is, to the particular input data set used to build the procedure strings).

Procedure strings are an infinite set, and are exact in a way that makes them unsuitable for use in static analysis. Our third step, then, is to abstract procedure strings into *stack configurations*, a finite set each member of which represents an infinite set of procedure strings, yet contains enough information to be useful in static analysis. We formulate conservative solutions to each of the above problems in terms of stack configurations.

Fourth, we present an abstract semantics based on stack configurations. We show that the abstraction preserves the meaning of the program and the procedure strings it describes. The beauty of a carefully chosen abstract domain is that operations upon its members, while preserving the meaning of analogous operations upon members of the concrete domain, occur *within the abstract domain.* When the abstract domains and the operations upon them are sufficiently simple, the abstract semantics give rise to a practical dataflow analysis algorithm. We show how the abstraction we have constructed may be adapted for both flow-sensitive and flow-insensitive dataflow analyses.

Fifth, we note that our construction of stack configurations in the abstract semantics, while correct, causes unnecessary information loss. We show that this is corrected by a simple shift of perspective.

For the sake of simplicity, the presentation to this point assumes a subset of Scheme that includes no mutable compound data objects (lists, vectors, etc.) In concluding our discussion of interprocedural analysis, we extend the technique to accommodate such data.

In the end, we are left with a framework for program analysis that allows us to evaluate the lifetimes of, and side-effects upon all forms of dynamically allocated objects provided in Scheme, from variables captured by lexical closures and continuations, to mutable cons cells and user structures. This has a most significant consequence for automatic parallelization: it permits the extraction of parallelism from procedures that are invoked at all levels of the calling tree of a program, from the lowest (innermost) computations to the highest (outermost), for the reason that a side-effect upon an object `x` that occurs (directly) within a procedure `f` need not "pollute" all procedures that call `f` (directly or indirectly), but is limited in visibility according to the lifetime of `x`. Also, the framework permits us to speak, at compile time, of distinct instances of dynamically allocated objects *that arise from a single*

lexical construct. That is, we may distinguish several instances of cons cells that result from a single piece of program text, or of several instances of a particular bound variable. We will see that this gives a sharpness to the analysis that is absent from conventional techniques for alias analysis. We will likewise see that discovering restrictions upon the lifetimes of objects is useful in placing them within a memory system, whether on the stack of a sequential Scheme evaluator, or in the hierarchical shared memory of a multiprocessor executing automatically parallelized programs.

As will become clear as the discussion progresses, the techniques described in this section may be applied in a straightforward manner to other procedural languages; they are particularly appropriate for languages such as C and Pascal, which make use of recursion and manipulate dynamically allocated storage. In fact, because most programming languages in wide use lack such radically general features as Scheme's first-class procedures and continuations, the methodology illustrated in this section may be applied to such languages, in large part as a specialization of the concrete and abstract semantics given below. As an extension of the work described in this paper, we are implementing a compiler which accepts a variety of source languages

2.3 Notational Conventions

In this subsection we review the notation of lambda calculus and the terminology of domains that is used in the discussion which follows. Our notation is consistent with that in [44, 11], and the reader who is unfamiliar with the concepts reviewed below will find a thorough introduction in those texts.

We will make heavy use of the lambda calculus [16] in the discussion below. There are, in essence, just two kinds of expressions in the lambda calculus: *abstractions* and *applications*. An expression of the form $\lambda a.e$ is called an abstraction, and denotes a function of a single argument a, whose body is e. An expression of the form $e_1 e_2$ denotes the application of the function e_1 to an argument e_2. e_1 has type $A \to B$, e_2 has type A, and $e_1 e_2$ has type B for some domains A and B (we will characterize these domains shortly). Parentheses may be used to group subexpressions, so that $(e_1 e_2)$, $(e_1)e_2$ and $e_1(e_2)$ are equivalent ways of writing the application of e_1 to e_2. All of our functions will be curried (that is, will have exactly one argument). Function application is left associative, so that *abcd* should be read as $(((ab)c)d)$.

By a domain D we mean a set upon which is imposed a chain-complete partial order, denoted by $\sqsubseteq_D$. That is, D has a distinguished least element $\bot_D$, and every non-decreasing chain $x_1 \sqsubseteq_D x_2 \sqsubseteq_D \cdots$ with $x_i \in D$

has a least upper bound (LUB) in D, written $x_1 \sqcup_D x_2 \sqcup_D \cdots \sqcup_D x_n$ or $\sqcup_D\{x_1, x_2, \ldots, x_n\}$. D may have a distinguished greatest element $\top_D$ as well. We write $x \sqsubseteq_D y$ to denote ordering among members of D. $D_1 + \cdots + D_n$ denotes the *separated sum* of D_1 through D_n; that is, the bottom element $\bot_{D_1+\cdots+D_n}$ of $D_1 + \cdots + D_n$ is is less than each of $\bot_{D_1}$ through $\bot_{D_n}$. While every member of this sum (other than its bottom element) has the form $\langle d_i, i\rangle$ where $d_i \in D_i$, we omit coercions between $D_1 + \cdots + D_n$ and its subdomains (that is, treat $d_i \in D_i$ as a member of $D_1 + \cdots + D_n$), where they are clear from context. The partial order within $D_1 + \cdots + D_n$ is implied by the partial orders $\sqsubseteq_{D_1}, \ldots, \sqsubseteq_{D_n}$. If $d_i \sqsubseteq_{D_1+\cdots+D_n} d_j$ where $d_i, d_j \in D_1 + \cdots + D_n$ then either $d_i = \bot_{D_1+\cdots+D_n}$ or $d_i, d_j \in D_k$, for $1 \leq k \leq n$, and $d_i \sqsubseteq_{D_k} d_j$. $D_1 \times \cdots \times D_n$ denotes the *non-strict* product of D_1 through D_n. The bottom element $\bot_{D_1\times\cdots\times D_n} = \langle \bot_{D_1}, \ldots, \bot_{D_n}\rangle$, and is distinct from $\langle d_1, ..., d_n\rangle$ where $\bot_{D_i} \sqsubset_{D_i} d_i$ for some $1 \leq i \leq n$. As when summing domains, the partial order within $D_1 \times \cdots \times D_n$ is implied by $\sqsubseteq_{D_1}, \ldots, \sqsubseteq_{D_n}$. If $\langle d_1, ..., d_n\rangle \sqsubseteq_{D_1\times\cdots\times D_n} \langle e_1, ..., e_n\rangle$, then $d_i \sqsubseteq_{D_i} e_i$ for all $1 \leq i \leq n$. $D \to E$ denotes the domain of continuous functions from D to E. The bottom element of $D \to E$ is $\lambda d.\bot_E$, and $f \sqsubseteq_{D\to E} g$ if $(fx) \sqsubseteq_E (gx)$ for all $x \in D$. The notation $D* \to E$ represents the sum $E + (D \to E) + (D \to D \to E) \cdots$. Occasionally we will have need to enumerate a function (that is, to represent it directly as a subset of the Cartesian product $A \times B$). In such a case we will write $\{a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots, a_n \mapsto b_n\}$ where $a_1, a_2, \ldots, a_n \in A$, $b_1, b_2, \ldots, b_n \in B$, and $fa_i = b_i$ for $1 \leq i \leq n$.

We write $f[y/x]$ to represent the function that is everywhere identical to f, except at x, where its value is y. We write $f[y//x]$ to denote $f[(fx) \sqcup y/x]$, the function everywhere identical to f, except at x, where its value is consistent with, but possibly greater than fx. (Whenever we use this notation, the least upper bound $(fx) \sqcup y$ will exist.) It follows, of course, that $f \sqsubseteq f[y//x]$ regardless of the value of y. Syntactic objects will be surrounded in double brackets, as in $[\![$`(set! x y)`$]\!]$

2.4 Abstract Interpretation

In this subsection, we review as much of the theory of abstract interpretation as is needed to follow the main lines of our construction, with the goal of defining terms and giving the reader an intuitive feel for our approach. See [17, 18, 28] for a more complete introduction to the topic.

How is abstract interpretation relevant to the writing of a compiler? We have, at the outset of our task, a host of techniques for implementing the various features of the source language. Some of these techniques are of such generality that they may be legally employed under all circumstances.

Others are more efficient than the most general techniques, but may be employed only under special circumstances. Practical considerations aside, the obvious method of compilation would be to execute a program over all possible input data sets, and note which of the special conditions needed to trigger an optimization are satisfied by every instance of the program. We would then compile the optimization, where permitted, into the object code, in full confidence that our translation would be a legal one. Since this is impossible, we settle instead for an approximation to the above process. Abstract interpretation provides such an approximation.

We begin, then, with an concrete semantics for our language, a definition that gives precise, mathematical meaning to programs in the language. The abstraction of such a semantics normally proceeds in several steps. First, we select domains for abstraction. These may be domains that are visible within the programming language (such as integers or symbols), or domains that are used only within the language definition (domains over which the semantic functions, or their auxiliary functions, are defined). For each such domain we create a corresponding abstract domain, and define an *abstraction map* which carries members of the concrete domain into members of the corresponding abstract domain. In this paper, such a map is written as Abs_D, where D is the concrete domain being abstracted, and has type $D \to \hat{D}$, where $\hat{D}$ is the corresponding abstract domain.

Because abstract domains may be smaller than their concrete counterparts, such maps need not be one-one. That is, many members of an concrete domain may be mapped onto a single element of the corresponding abstract domain. We think of each member $\hat{a}$ of an abstract domain $\hat{D}$ as equivalent to (or representative of) a subset of the corresponding concrete domain D. In order greatly to simplify the mathematics of the abstraction, we require that every $\hat{a} \in \hat{D}$ signify an *ideal* of D. If an ideal includes an element $a \in D$, then it includes every element $b \in D$ such that $b \sqsubseteq_D a$. Furthermore, if it contains a chain $a_1 \sqsubseteq_D a_2 \sqsubseteq_D \cdots$, then it includes the least upper bound of the chain. In the jargon of power domain theory, the members of our abstract domains are *downwardly closed* and *upwardly complete*.[9] This construction is known as the Hoare power domain [43], and allows us to use the natural ordering of subsets (according to inclusion) as the partial ordering in the abstract domains; we will return to this momentarily. A thorough discussion of Hoare power domains and their properties

[9]Because each ideal includes the bottom element of the concrete domain, when viewed as a set of possibilities the ideal suggests that the bottom element is a possibility. This means, for example, that when our concrete domain is a function space whose bottom element represents non-termination or undefinedness, we cannot exclude non-termination or undefinedness as a possibility. We can, however, discern the case in which undefinedness is the *only* possibility.

is found in [43, 15].

A function that maps each member of an abstract domain onto the ideal it represents (of the corresponding concrete domain), is called a *concretization map*. In this paper, such a map is denoted by $Conc_{\hat{D}}$, where $\hat{D}$ is an abstract domain, and will have type $\hat{D} \to P(D)$, where $P(D)$ is the power domain of D (in our case, the set of ideals of D). Given an abstraction map Abs_D, we may define the corresponding concretization map as

$$Conc_{\hat{D}} \equiv \lambda \hat{d}.\{d \mid Abs_D d \sqsubseteq_{\hat{D}} \hat{d}\}.$$

The definition says that $\hat{d}$ represents the set of all elements of D whose abstractions are consistent with $\hat{d}$. Likewise, given a concretization map $Conc_{\hat{D}}$, we may define the corresponding abstraction map as

$$Abs_D \equiv \lambda d. \sqcap_{\hat{D}} \{\hat{d} \mid d \in Conc_{\hat{D}} \hat{d}\}.$$

This definition says that the abstraction of d is the least ideal represented in $\hat{D}$ that contains d. Since the members of a Hoare power domain are partially ordered by inclusion, when we write $\hat{a} \sqsubseteq_{\hat{D}} \hat{b}$, we will mean that $Conc_{\hat{D}} \hat{a} \subseteq Conc_{\hat{D}} \hat{b}$; and if we define the partial ordering $\hat{a} \sqsubseteq_{\hat{D}} \hat{b}$ differently, we must prove that our definition is equivalent to $Conc_{\hat{D}} \hat{a} \subseteq Conc_{\hat{D}} \hat{b}$.

Having abstracted the primitive domains of our semantics, and having made precise what these abstractions represent, we abstract functions over the primitive types. The functions we abstract are those used to give meaning to the programming language. Our abstractions of these functions must preserve the meaning of the corresponding concrete functions. There are many senses in which meaning can be preserved. In this case, we mean that the result of projecting operands onto abstract domains, and applying an abstract function, must be a member of an abstract domain (that is, an ideal of the concrete domain) that contains the result of applying the concrete function, and projecting the concrete result onto the abstract domain. Perhaps it is helpful to think of this is as follows. The members of our abstract domains represent sets of values, for example, the values that may be assumed by a user variable, or the values that may be returned by an auxiliary function applied within our semantic functions. We wish for an abstract interpretation of the program to inform us of all possible values that may be assumed by the user variable, or all possible values that may be returned by the auxiliary function. We must arrange to err in favor of overestimation of such sets. In light of this choice, the statement $\hat{x} \sqsubseteq_{\hat{D}} \hat{y}$ may be read as saying that x is consistent with, but is more precise than $\hat{y}$. That is, all possibilities suggested by $\hat{x}$ are suggested by $\hat{y}$ as well, but

$\hat{y}$ may suggest possibilities not suggested by $\hat{x}$. In terms of subsets of the concrete domain D, $Conc_{\hat{D}}\ \hat{x} \subseteq Conc_{\hat{D}}\ \hat{y}$.

We need to pause to make as clear as possible the distinction between the partial ordering among members of an abstract domain, and the partial ordering among members of the corresponding concrete domain. In each of our concrete domains there is a distinguished element $\perp$ (bottom) which is less than every other element in the domain, according to the partial order of the domain. For instance, the bottom element in the domain of values that may be computed by a program is undefinedness (non-termination or error). In information content, this value is consistent with any better defined value (such as the integer 5), but it less informative. This much should be familiar to the reader, from introductory semantics.

Here's the rub. If $\hat{D}$ is an abstract domain as we have defined it above, and if D is the corresponding concrete domain, then every member of $\hat{D}$ contains the bottom element of D, $\perp_D$ (since ever member of $\hat{D}$ is an ideal of D). But we have said that if $\hat{x} \sqsubseteq_{\hat{D}} \hat{y}$, then $\hat{x}$ is at least as informative as $\hat{y}$. Clearly the smallest ideal that can be produced by Abs_D is $\{\perp_D\}$, which by our pronouncements is at once the most informative member of $\hat{D}$,[10] and the set containing only the least informative member of D. This is a paradox in appearance only. To know that a program never terminates is a special case of knowing that some of its statements (here, the final statement among them) never execute. This is enormously informative, since we may give whatever translation we like to code that goes unused. In the lattice of functions ordered by information content, the function that never terminates is the least informative, and is consistent with every other function (it returns the same value as every other function, when it terminates). In the lattice of conditions that enable optimization, that a piece of code never executes is as strong and specific a condition as possible. Our conclusions are the same when $\perp$ represents error. We assume that the program being compiled is correct, since we will certainly be unable to give a meaningful translation to a meaningless program. Therefore, when error appears as a possible outcome of the execution of a statement, the consistent assumption is that the possibility is apparent only, that one of the other outcomes occurs in reality. When error appears to the be only outcome, the consistent assumption is that the statement is never executed. Therefore it is of no consequence that the bottom element is overloaded with the meanings non-termination, error, etc. In all cases, the assumption of a correct input program leads us to treat the undefined value consistently

[10] In [43], larger elements of the Hoare powerdomain $\hat{D}$ are regarded as more informative, for the reason that they represent better defined elements of D. From our perspective, however, a larger element represents more possibilities; in the extreme, every member of D is a possibility, and this gives us no leverage in optimization.

(as non-execution).

Given an abstract semantics for the language, it remains only to create (via the abstraction maps) a representation of the initial states from which execution of the program may proceed, and to evaluate the program in this abstract state, using the abstract semantics. By observation of this evaluation, we may answer the questions, albeit with reduced accuracy, that we wished to ask about the execution of the program over all possible sets of input data, using the concrete semantics. The formal correspondence between the concrete and the abstract guarantees that the answers we obtain, if interpreted properly, will be consistent with the best (most informative) answers possible, to these questions.

2.5 Concrete Semantics

2.5.1 The Language $\mathcal{L}$

We will begin by treating a simple variant of Scheme (call it $\mathcal{L}$), which is described by the following grammar.

$$
\begin{array}{lll}
\mathcal{L} & ::= & \mathit{Stmt}^{+} \\
\mathit{Stmt} & ::= & [\![\texttt{(set!}\ V\ \texttt{(}\ V^{+}\ \texttt{))}\ ]\!] \\
 & | & [\![\texttt{(set!}\ V\ \texttt{(lambda (}\ V^{*}\ \texttt{) <}\ V^{*}\ \texttt{>}\ \mathit{Stmt}^{+}\ \texttt{))}\ ]\!] \\
 & | & [\![\texttt{(set!}\ V\ \texttt{(call/cc}\ V\ \texttt{))}\ ]\!] \\
 & | & [\![\texttt{(if}\ V\ \texttt{(goto}\ N\ \texttt{) (goto}\ N\ \texttt{))}\ ]\!] \\
 & | & [\![\texttt{(return}\ V\ \texttt{)}\ ]\!] \\
 & | & [\![\texttt{(end)}\ ]\!] \\
V & ::= & \text{identifier} \\
N & ::= & \text{statement index}
\end{array}
$$

A program in $\mathcal{L}$ consists of a sequence of statements, the last of which is an `end` form. Each procedure in $\mathcal{L}$ has, in addition to its parameter list, a list (surrounded by angle brackets) of local variables. The local variables are bound to locations when the procedure is applied, but have undefined values until they are assigned. We will assume that any literal data needed by a program are held in global variables in the initial state. Similarly, intrinsic procedures are held in global variables in the initial state. These include an identify function for effecting assignment from one variable to another. A procedure application in $\mathcal{L}$ is "flat" (the operator and arguments must be variables), and its return value is stored into a variable (the value may go unused). Not accidentally, this resembles the traditional *quadruple* representation used in optimizing compilers [10]. Likewise, an `if` form in $\mathcal{L}$ functions only as a branch, and an explicit `return` form is used for normal exit from a procedure. Finally, `call/cc` is treated as a special form, not

```
(define sum-of-squares (lambda_α (m n k)
   (if (= m n)
       (* m m)
       (sum-of-squares (1+ m)
                       n
                       (lambda_β (x) (+ x (* m m)))))))
```

Figure 1: A Sample Scheme Program

as a variable.[11] It is a simple matter to rewrite a Scheme program in a form that resembles $\mathcal{L}$. Parcel effects such a transformation while parsing its input.

We assume that as part of the process of translation from Scheme to $\mathcal{L}$, variables are renamed, so that distinct variables have distinct identifiers. With each `lambda` expression (`lambda`$_\alpha$ (`x`$_1 \cdots$ `x`$_m$) <`x`$_{m+1} \cdots$ `x`$_n$> $S_{i_1} \cdots S_{i_n}$) of the program is associated a distinct index $\alpha \in \Lambda$. We write λ_α for the α^{th} `lambda` expression of the program. Likewise, with each statement S_i of the program is associated the distinct index $i \in N$, and the successor function $Succ : N \rightarrow N$ defines the flow of control between statements, in the absence of explicit branches. (The structure of $\mathcal{L}$, including its numbered, flat expressions and the addition of explicit `end` and `return` forms, is intended to make more natural the adaptation of its semantics for dataflow analysis.)

An example of a Scheme program, and the corresponding program in $\mathcal{L}$, is given in Figures 1 and 2. In the latter, every statement is subscripted by its statement index. In this case, the successor function is defined as follows: $Succ\ 1 = \perp_N$, $Succ\ 2 = 10$, $Succ\ 10 = \perp_N$, $Succ\ 3 = 5$, $Succ\ 5 = 6$, $Succ\ 6 = 7$, $Succ\ 4 = 7$, $Succ\ 7 = \perp_N$, $Succ\ 8 = 9$, $Succ\ 9 = 11$, and $Succ\ 11 = \perp_N$. λ_α (when rewritten in $\mathcal{L}$) has five local variables, and λ_β has two. The successor function is not defined for statement 10 (an `if` form) because the statement that follows it in execution order is determined by the branch that is taken (either to statement 3 or statement 4), and not by the successor function. Nor is the successor function defined for either of the `return` forms in the example. The variable `sum-of-squares` is assumed to be bound at the global level.

[11] Given `call/cc` as a special form, one may write `(set! my-call/cc (lambda (f) (call/cc f)))`, to achieve the effect of the *variable* `call/cc` as it is defined in Scheme. The value of this variable (`my-call/cc`) can be passed as an argument to a procedure, for example, while the "value" of a special form cannot be.

```
(set! sum-of-squares (lambda_α (m n k) <t1 t2 t3 t4 t5>
   (set! t1 (= m n))_2
   (if t1 (go 3) (go 4))_10
   (set! t3 (1+ m))_3
   (set! t4 (lambda_β (x) <t10 t11>
      (set! t10 (* m m))_8
      (set! t11 (+ x t10))_9
      (return t11)_11) )_5
   (set! t5 (sum-of-squares t3 n t4))_6
   (set! t5 (* m m))_4
   (return t5)_7 )_1
```

Figure 2: The Sample Program Rewritten in $\mathcal{L}$

2.5.2 Procedure Strings

Consider a program A consisting of λ_α, λ_β and λ_γ. We will associate a procedure string p_i with every state q_i of an execution of A. Let the procedure string p_0 that corresponds to the initial state q_0 of A be ϵ (empty). We will append the term α^d, β^d or γ^d to the current procedure string whenever λ_α, λ_β or λ_γ is applied, respectively, and we will append the term α^u, β^u or γ^u to the current procedure string whenever control returns from λ_α, λ_β or λ_γ respectively.

Suppose the first procedure application which occurs during execution is of λ_α, and let q_1 be the state which results. Then $p_1 = \alpha^d$. Suppose next that λ_β is applied, and that q_2 results. Then $p_2 = \alpha^d\beta^d$. The superscripts in $\alpha^d\beta^d$ indicate that control has moved *downward* into λ_α, and subsequently downward into λ_β. At this point, let control return to λ_α, resulting in q_3. We indicate by $p_3 = \alpha^d\beta^d\beta^u$ that control has moved *upward* from λ_β. Now let consecutive applications of λ_α, λ_γ and λ_α result in q_6 in which $p_6 = \alpha^d\beta^d\beta^u\alpha^d\gamma^d\alpha^d$. We may read the history of A's interprocedural behavior to this point from p_6: there have been three applications of λ_α, and one each of λ_β and λ_γ. Apparently λ_β is no longer active, whereas the other procedure instances are still active.

Now suppose that λ_α has bound variables `x` and `y`. Between states q_0 and q_6 there were three instances of λ_α (and thus three instances of `x` and `y`), corresponding to the procedure strings $p_1 = \alpha^d$, $p_4 = \alpha^d\beta^d\beta^u\alpha^d$ and $p_6 = \alpha^d\beta^d\beta^u\alpha^d\gamma^d\alpha^d$ of the states that follow each application of λ_α. We will use the procedure string of the state at its point of creation to distinguish one instance of `x` from another, and likewise for instances of `y`. We will not confuse instances of `x` with instances of `y`, since the *aliasing* [14] of variables

is impossible in Scheme.[12] Thus, every state q will contain a function that maps each variable that is lexically visible in q to the procedure string of the state in which the variable was bound. We will call this procedure string the *birth date* of the variable. The environment of q will be a map from a variable and its birth date, to the current value of that instance of the variable.

Before formalizing these observations in a definition of $\mathcal{L}$, we must decide how to describe the effect of continuations on procedure strings. We were able to discern, above, by inspection of a procedure string, what interprocedural movements had led to the state corresponding to the string, and what procedures were active in that state.[13] We must formalize this analysis, for we require that the same be true of a procedure string that results from the application of a continuation. Let us define a function $Net : P \to P$ which deletes every pair of the form $\alpha^d\alpha^u$ from its argument, until no further such deletions are possible. For example,

$$Net\ \alpha^d\beta^d\gamma^d\gamma^u\beta^u\alpha^d\gamma^d\alpha^d = Net\ \alpha^d\beta^d\beta^u\alpha^d\gamma^d\alpha^d = \alpha^d\alpha^d\gamma^d\alpha^d.$$

Intuitively, $Net\ p$ represents all "unmatched" procedure activations and deactivations in p. We say that a procedure string p is *balanced* if $Net\, p = \epsilon$ (and thus Net is a function that deletes all balanced substrings from its argument). Now we may make our observation about active procedures precise. The active procedures in a state q may be read directly from $Net\ p$, where p is the procedure string corresponding to q. For instance, $Net\ p_6 = \alpha^d\alpha^d\gamma^d\alpha^d$, and we observed above that in state q_6 there were three active instances of λ_α and one of λ_γ.

Suppose that we are given procedure string p, and are asked to form another procedure string q such that $p+q$ is balanced (where $+$ represents concatenation). By the definition of Net, we seek q such that $Net(p+q) = \epsilon$. In other words, q must contain a "match" for every unmatched term of $Net\ p$. Clearly there are many strings q which satisfy this requirement, because adding a balanced substring to any such q produces another. Let us imagine that there is a function $Inv : P \to P$ so that $Inv\ p$ is the shortest procedure string such that $p + (Inv\, p)$ is balanced. If, for example, $p = \alpha^d\beta^d\beta^u\gamma^d$, then $Inv\, p = \gamma^u\alpha^u$. This definition of Inv is sensible only if p is a procedure string accumulated from the initial state of the program. If, for example, p is an arbitrary substring of the procedure string of a

[12] Aliasing may arise in Scheme, however, from operations upon cons cells and user-defined structures; and an effect very much like aliasing may arise by the use of closures, and assignments to the free variables they capture. Such aliasing is accommodated neatly within our framework for side-effect analysis, as we will see in subsection 2.14.

[13] Operationally, these correspond to the procedure instances that are "on the stack" at the point in question.

state during execution, it will not (in general) be "invertible". Consider $p = \alpha^u\beta^d\gamma^d$; there is no procedure string which, when appended to p, will result in a balanced procedure string, because the leading α^u term in p can be balanced only by a corresponding α^d term to its left. The action of *Inv* upon an arbitrary p will be to reverse *Net* p, and "invert" each of the d's and u's in the result. If $p = \alpha^u\beta^d\gamma^d$, then $Inv\ p = \gamma^u\beta^u\alpha^d$. We will prove below that when p has an "inverse", that these two definitions of $Inv\ p$ are equivalent.

Now consider a continuation k which is formed in a state q to which the corresponding procedure string is p, and applied in a state q' to which the corresponding procedure string is p'. Let q'' be the state that results after the application of k, and let p'' be the procedure string that corresponds to q''. What should be the value of p''? Applying k has the effect of exiting any procedures that were not active at q but are at q', and re-activating any procedures that were active at q but are not at q'. The string $p' - p$ (where $x - y$ is the suffix of x not contained in y) describes all activity which occurred between q and q'. It follows that $Inv(p' - p)$ is the (shortest) string which "undoes" the net effect of that activity. Therefore $p'' = p' + Inv(p' - p)$ seems to be the procedure string we want. As a record of interprocedural activity, it indicates that we progressed to state q', at which point k was applied, causing the net effect of all movements made between k's formation in q and its application in q' to be undone. We will prove that this is the desired value of p'', below.

2.5.3 A Semantics for $\mathcal{L}$ in Terms of Procedure Strings

As a first step toward our program analysis framework, we will construct a semantics for $\mathcal{L}$ in terms of procedure strings. It is our immediate goal to create a definition of the language which will allow us to formulate optimal solutions to the problems, defined in subsection 2.1, which motivate us in this section.

The domain definitions for our first semantics for $\mathcal{L}$ are presented in Figure 3.

N is the domain of statement indices, over which the function *Succ* is defined. The primitive domains P, N, Λ, *Int*, and *Bool* are flat domains with distinguished least elements $\perp_P$, $\perp_N$ $\perp_\Lambda$, $\perp_{Int}$, and $\perp_{Bool}$ respectively, whose non-bottom members are incomparable. It is important to remark that the definition of P in Figure 3 does not delimit the set of procedure strings that arise from program executions, nor even the substrings of such procedure strings. The definitions implies, for example, that $\alpha^d\beta^u \in P$, whereas such a (sub)string could not arise from a program execution. The subset of P with which we are concerned will be defined via some theorems in the discussion below.

$$
\begin{array}{ll}
P = (\Lambda^d \mid \Lambda^u)^* & \text{(procedure strings)} \\
B = V \to P & \text{(birth date maps)} \\
E = V \times P \to D & \text{(environments)} \\
C = Q \to D^* \to Q & \text{(closures)} \\
K = Q \to D \to Q & \text{(continuations)} \\
D = C + K + PrimOp + Int + Bool & \text{(values)} \\
Q = N \times P \times B \times E \times K & \text{(states)}
\end{array}
$$

Figure 3: Domain Definitions for $\mathcal{S}_1$ and $\mathcal{E}_1$

The structure of the compound domains B, E, C, K, D and Q are as described in subsection 2.3. A state $q \in Q$ is a 5-tuple of a statement index, a procedure string, a map from lexically visible variables to their birth dates, an environment, and a continuation. An environment is a map from a variable and its birth date to the current value of the variable. Closures and continuations are similar to one another in structure and effect. A closure is a function from a state (the state in which the closure is applied) and a set of values (its actual parameters) to a new state (the state from which execution proceeds within the body of the applied lambda expression). A continuation is a function from a state (the state in which it is applied) and a value (the argument to the continuation, which becomes the value of the expression which created the continuation), to a new state (the state from which execution proceeds following return from the initiating expression).

Continuations do not play quite so pervasive a role here as in most formal definitions of Scheme. We make use of the continuation component of a state only when crossing procedure boundaries (i.e., when entering or leaving a procedure, or when applying a continuation created with `call/cc`). The sequencing of control within a procedure (for example, when evaluating an `if` form) depends upon statement indices, and does not involve continuations directly. This is because we wish to collect information concerning interprocedural flow of control and data, and it serves us to this end to isolate such information within the semantics. Put another way, procedure strings are unaffected by intraprocedural movements of control.

Int and *Bool* represent integers and booleans, and *PrimOp* is the domain of primitive operators over these types. We take the meaning of the members of these domains for granted. The semantic functions defined in Figures 4 and 5 are $\mathcal{S}_1$ (for *step*) and $\mathcal{E}_1$ (for *eval*). $\mathcal{S}_1$ maps each state onto its successor; it describes a single step of evaluation. $\mathcal{E}_1$ simply composes applications of $\mathcal{S}_1$. The state q_f that results from executing a

program in an initial state q_0 is the least fixed point[14] of $\mathcal{E}_1$ that satisfies $q_f = \mathcal{E}_1 q_0 = \mathcal{E}_1 q_f$. An expression of the form

$$\begin{array}{ll} & expr_0 \\ \text{where} & x_1 = expr_1 \\ & x_2 = expr_2 \\ & \cdots \\ \text{and} & x_n = expr_n \end{array}$$

should be understood as meaning roughly the same as

$$\begin{array}{l} \texttt{(let*}\ ((x_1\ \ expr_1) \\ \qquad (x_2\ \ expr_2) \\ \qquad \cdots \\ \qquad (x_n\ \ expr_n)) \\ \quad expr_0) \end{array}$$

in Scheme, so that x_1 may appear in $expr_2 \ldots expr_n$, x_2 may appear in $expr_3 \ldots expr_n$, etc., and any of x_1 through x_n may appear in $expr_0$.

Consider the definition of $\mathcal{S}_1$ in the case that S_i is an `if` expression. The definition says that the state $\mathcal{S}_1 q$ which results from a single step of evaluation in state q, is the state whose procedure string is p (the same as that of q), whose statement index is either m or n, depending upon the value of the variable x, and whose b (map from variables to birth dates), e (environment) and k (continuation) components are the same as those of q. The other cases within the definition are read similarly. Before looking at this definition in more detail, let's consider the function $\mathcal{E}_1$.

$\mathcal{E}_1$ is the only recursive definition in this semantics. We could write it as the least fixed point of a functional, and show that the fixed point exists, but such reasoning is more relevant to a discussion of denotational semantics per se.[15] See [43, 44, 11]. Here we simply accept the recursive definition, and resort to theorem-making when it is necessary to validate something novel to our approach.

Theorem 1 *$\mathcal{E}_1$ preserves the standard semantics of $\mathcal{L}$.*

[14]The reason for defining evaluation in this way will be clear when we make these semantics the basis of an iterative data flow algorithm which converges to a fixed point.

[15]Besides, the purist will find much to object to in our definition of $\mathcal{L}$. For instance, the function *Succ* is dependent entirely upon the structure of a particular program, and yet it occurs free within the definition of $\mathcal{S}_1$. The point is that we have, from the outset, made concessions to our intended use for this semantics, as a stepping stone toward a practical framework of program analysis.

Let $q = \langle i, p, b, e, k \rangle \in Q$. Then $\mathcal{S}_1 : Q \to Q$ is defined, according to the form of statement S_i, as follows.

$S_i = [\![\texttt{(set! x (f } \texttt{y}_1 \cdots \texttt{y}_m\texttt{))}]\!] \Rightarrow$
$\mathcal{S}_1 q = e\langle [\![\texttt{f}]\!], b[\![\texttt{f}]\!]\rangle q(e\langle [\![\texttt{y}_1]\!], b[\![\texttt{y}_1]\!]\rangle) \cdots (e\langle [\![\texttt{y}_m]\!], b[\![\texttt{y}_m]\!]\rangle)$

$S_i = [\![\texttt{(set! x (call/cc f))}]\!] \Rightarrow$
$\mathcal{S}_1 q = e\langle [\![\texttt{f}]\!], b[\![\texttt{f}]\!]\rangle q \lambda s \lambda d. \langle Succ\, i, p' + Inv(p' - p), b, e'[d/\langle [\![\texttt{x}]\!], b[\![\texttt{x}]\!]\rangle], k\rangle$
where $s = \langle i', p', b', e', k'\rangle$ (state at application of the continuation)

$S_i = [\![\texttt{(set! f (lambda}_\alpha\texttt{ (}\texttt{x}_1 \cdots \texttt{x}_m\texttt{) <}\texttt{x}_{m+1} \cdots \texttt{x}_n\texttt{> } S_j \cdots \texttt{))}]\!] \Rightarrow$
$\mathcal{S}_1 q = \langle Succ\, i, p, b, e[c/\langle [\![\texttt{f}]\!], b[\![\texttt{f}]\!]\rangle], k\rangle$
where $c = \lambda r \lambda d_1 \cdots \lambda d_m.$
$\langle\ j,$
$p' + \alpha^d,$
$b[p' + \alpha^d/[\![\texttt{x}_1]\!]] \cdots [p' + \alpha^d/[\![\texttt{x}_n]\!]],$
$e'[d_1/\langle [\![\texttt{x}_1]\!], p' + \alpha^d\rangle] \cdots [d_m/\langle [\![\texttt{x}_m]\!], p' + \alpha^d\rangle],$
$\lambda s \lambda d. \langle Succ\, i', p'' + Inv(p'' - p'), b', e''[d/\langle [\![\texttt{y}]\!], b[\![\texttt{y}]\!]\rangle], k'\rangle\rangle$
where $r = \langle i', p', b', e', k'\rangle$ (state at application of the closure)
$s = \langle i'', p'', b'', e'', k''\rangle$ (state at `return` from the application)
and $S_{i'} = [\![\texttt{(set! y } \cdots \texttt{)}]\!]$

$S_i = [\![\texttt{(if x (goto m) (goto n))}]\!] \Rightarrow$
$\mathcal{S}_1 q = \langle$ if $e\langle x, bx\rangle = true$ then m else $n, p, b, e, k\rangle$

$S_i = [\![\texttt{(return x)}]\!] \Rightarrow$
$\mathcal{S}_1 q = kq(e\langle [\![\texttt{x}]\!], b[\![\texttt{x}]\!]\rangle)$

$S_i = [\![\texttt{(end)}]\!] \Rightarrow$
$\mathcal{S}_1 q = q$

Figure 4: The Semantic Function $\mathcal{S}_1$

$$\mathcal{E}_1 : Q \to Q \equiv \lambda q.\ \text{Let } q' = \mathcal{S}_1 q \quad \text{in} \quad \text{if } q' = q \text{ then } q \text{ else } \mathcal{E}_1 q'$$

Figure 5: The Semantic Function $\mathcal{E}_1$

Sketch of proof: $\mathcal{E}_1$ differs materially from a standard semantics [41] only in its representation of the environment (store). We may define a conventional store by the following correspondence. Let E (the domain of environments) correspond to the standard domain of stores, and let the product $V \times P$ (variables and their birth dates) correspond to locations within stores. We must show that every instance of every bound variable is assigned a unique location in the store. Since a location is a pair $\langle v, p \rangle$ of a variable and its birth date, it is impossible for $\mathcal{E}_1$ to assign a single location to instances of distinct variables. We must therefore show that separate instances of a single bound variable are assigned distinct locations in the store. To do so it is sufficient to show that every state in which the variable is bound has a distinct procedure string. Let x be a bound variable of λ_α, and let q_j and q_k be distinct states in which λ_α is applied. Without loss of generality, assume $j < k$. By the definition of $\mathcal{S}_1$ in the case of closure formation, the birth dates of the instances of x corresponding to q_j and q_k are $p_j + \alpha^d$ and $p_k + \alpha^d$. Since the procedure string of each state is a prefix of the procedure string of its successor, $p_j + \alpha^d$ is a prefix of p_k. Thus $p_j + \alpha^d$ and $p_k + \alpha^d$ are distinct. □

For the purpose of showing preservation of a standard semantics, we need only prove that our method of identifying variable instances by their birth dates is equivalent to the "*NewLoc*" function used in a standard semantics to generate unique locations within a store. There is, however, far more information in the birth date of a variable instance than is needed to distinguish it from other instances of the same variable. We will characterize that information in a series of theorems, shortly. We must first decide if the semantics we have proposed is a suitable basis upon which to construct a framework of static program analysis.

2.5.4 Abstraction in the Face of Reflexivity

We have now a concrete definition of $\mathcal{L}$, that constructs procedure strings as it evaluates a program. We suspect (and will show it to be so shortly) that these procedure strings are ideal for answering our questions about side-effects and object lifetimes. Recalling our outline of abstract interpretation from subsection 2.4, the next step is the abstraction of this semantics, with the hope of observing the (abstract counterparts of the) procedure strings accumulated during abstract evaluation.

We turn to the domain definitions of Figure 3, looking for primitive (first-order) domains to abstract. The choices are P (procedure strings), N (statement numbers), *Int* and *Bool* (integers and booleans). If we were to abstract each of these domains completely away (that is, map each to an abstract domain of a single element), we would be left with a domain of values (the abstraction of D) which would contain higher-order objects,

namely primitive operators, closures, and continuations. We will suppose, for the moment, that the problem of primitive operators in the domain of values could be overcome with little difficulty. In the case of continuations and closures, however, we would be left to ponder the functions (from states and values to new states) in our abstract domain of values.

If an analysis of a program is performed, and the result is a function, then in a sense the analysis has not been completed: there is a measure of uncertainty left, embodied in the parameters of the function. This uncertainty is resolved by applying the function to values (that is, by eliminating the degree of freedom represented by the parameters). There seem to be two choices here. We could attempt, at compile time, to enumerate the function (that is, construct a representation of the function as a subset of the Cartestian product of its domain and its range). This would entail the application of the function to every value in its domain. But the functions representing closures and continuations are defined over reflexive domains; this process would result, in general, in yet further functions in the same domains. In short, there need be no finite enumeration of such a function, in terms of primitive domains, even when the primitive domains are themselves finite. Another choice is to suspend the resolution of uncertainty until run time, by making of the function a test to be compiled into the object code. To draw upon a problem to which abstract interpretation has traditionally been applied, if we analyze the strictness of a function f, and our analysis returns to us a function, which expresses the strictness of f in terms of the strictness of its parameters, we might compile this decision-making function into the object code, and use it at run-time to select between alternative means of evaluating an application of f. This approach is suggested in [28].

We choose instead to return to the concrete semantics upon which the abstract are based, and seek a representation for the domain of values that leads to abstractions which are more amenable to compile-time examination. Such representations are bit-vectors of reaching definitions, sets of aliased variables, upper and lower bounds upon the values of integer variables, etc. When we have made such an attempt, and find that still we lack sufficient information to produce an acceptably efficient translation of the program, then we may consider such devices as compiling multiple versions of the program, and spending additional running time deciding between versions.

2.5.5 Modified Domain Definitions for $\mathcal{L}$

The difficulty we encountered in our first attempt at abstraction resulted from the *reflexivity* of the domain D of values, and this reflexivity was introduced by our representation of closures and continuations. Let us look

closely at the way they are used, to see how they might be represented differently. There are several states which are relevant to the formation and application of a closure: the state q in which it is formed, the state r in which it is applied, the state s from which the application returns, and the state t from which execution proceeds, following the return. By examination of the rule within $\mathcal{S}_1$ for the formation of a closure, we see that the only information imparted to the closure from state q is the variable birth date map b. This accords with intuition: to form a lexical closure we need only know which lambda expression is the object of closure, and the bindings of its free variables at the point of closure. Any additional information needed to apply the closure, or to return from its application, may be (indeed, must be) garnered at the points of application and return. This suggests that we represent a closure as a member of the product $\Lambda \times B$, of lambda expression indices and variable birth date maps. The most serious difficulties this creates are in the restoration of the birth date map b, the statement index i, and the continuation k following the return from a closure application. All but the continuation k will be bundled into the continuation of the state which immediately follows the application. (If we were to put k into this continuation, then we would not have rid ourselves of the reflexivity of D.) The restoration of k will be effected by a function r, which is passed through the sequence of states, but is not in D, of type $P \to K$. It will be the birth date of a procedure instance[16] that is used to restore its continuation, whenever control returns to the instance. The continuation of every procedure instance will contain the birth date of its caller; at the point of return, this birth date will be passed to r, which will return the continuation of the caller. To effect this linking of continuations, we will make the birth date of the current procedure instance a component of every state. (The members of $R = P \to K$ will be called *restoration functions*.)

We turn now to the rule, within the definition of $\mathcal{S}_1$, for the formation of a continuation (the `call/cc` rule). The continuation which is passed to the argument of `call/cc` is the function

$$\lambda s \lambda d.\langle Succ\, i, p' + Inv(p' - p), b, e'[d/\langle [\![\mathtt{x}]\!], b[\![\mathtt{x}]\!]\rangle], k\rangle.$$

The information which is imparted to this continuation from the state q in which it is created, is the procedure string p, the statement number i, the birth date map b, and the continuation k of the current procedure instance. All of these components of the continuation are used to construct the state which results from application of the continuation. The procedure string p and statement index i are first-order values, and so are unlikely

[16]Recall that the birth date of a procedure instance is the procedure string of the first state in which the instance is active.

$$
\begin{array}{ll}
P = (\Lambda^d \mid \Lambda^u)^* & \text{(procedure strings)} \\
B = V \rightarrow P & \text{(birth date maps)} \\
E = V \times P \rightarrow D & \text{(environments)} \\
C = \Lambda \times B & \text{(closures)} \\
K = N \times B \times P \times P & \text{(continuations)} \\
D = C + K + PrimOp + Int + Bool & \text{(values)} \\
R = P \rightarrow K & \text{(restoration functions)} \\
Q = N \times P \times B \times E \times K \times P \times R & \text{(states)}
\end{array}
$$

Figure 6: Domain Definitions for $\mathcal{S}_2$ and $\mathcal{E}_2$

to cause any real difficulty. Although b is a function, its type is simply $V \rightarrow P$, so that its inclusion in the continuation causes no reflexivity in D. We described above the mechanism by which continuations will be linked. The birth date o of the procedure instance in which the `call/cc` expression is evaluated is included in the continuation it creates. At the same time, a restoration function is constructed (that is, accumulated as the state sequence progresses) which returns k when applied to o. This function will be used to restore k in the state which follows application of the continuation. A continuation will therefore be a member of $N \times B \times P \times P$ of the form $\langle i, b, p, o \rangle$.

The new domain definitions, based upon these observations, are given in Figure 6. A member $\langle i, p, b, e, k, o, r \rangle$ of the domain Q of states is now a 7-tuple of a statement index i; a procedure string p; a birth date map b; an environment e; a continuation k; a procedure instance birth date o; and a restoration function r. The important change from Figure 3 is that the domains are no longer reflexive.

2.5.6 A Modified Semantics for $\mathcal{L}$

A modified semantics for $\mathcal{L}$ is presented in Figures 7 8, and 9. The auxiliary function $Container : N \rightarrow \Lambda$ is defined such that $Container\, i = \alpha$, where λ_α is the lambda expression (immediately) containing statement S_i. The semantic functions $\mathcal{S}_2$ and $\mathcal{E}_2$ are exactly analogous to $\mathcal{S}_1$ and $\mathcal{E}_1$. The bulk of the activity, in this definition of $\mathcal{L}$, is in the application of closures and continuations, rather than in their formation. This reflects the fact that they are no longer represented by functions which contain all of the actions to be taken at the points of closure, application, and return. Instead, those actions have migrated to the appropriate points within the semantic functions. Examination of the definition of $\mathcal{S}_2$ reveals that this is

Let $q = \langle i, p, b, e, k, o, r \rangle \in Q$. Then $\mathcal{S}_2 : Q \to Q$ is defined as follows:

$S_i = [\![\texttt{(set! x (f } \texttt{y}_1 \cdots \texttt{y}_m\texttt{))}]\!]$ or $S_i = [\![\texttt{(set! x (call/cc f))}]\!] \Rightarrow$
  if $e\langle [\![\texttt{f}]\!], b[\![\texttt{f}]\!] \rangle = \langle \alpha, b' \rangle \in C$
  then $\mathcal{S}_2 q = \langle\ j,$
    $p + \alpha^d,$
    $b'[p + \alpha^d / [\![\texttt{z}_1]\!]] \cdots [p + \alpha^d / [\![\texttt{z}_n]\!]],$
    $e',$
    $\langle i, b, p, o \rangle,$
    $p + \alpha^d,$
    $r[k/o] \rangle$
    where $\lambda_\alpha = [\![\texttt{(lambda (z}_1 \cdots \texttt{z}_m\texttt{) <z}_{m+1} \cdots \texttt{z}_n\texttt{> } S_j \cdots\texttt{)}]\!]$
    and $e' =$ if $S_i = [\![\texttt{(set! x (f } \texttt{y}_1 \cdots \texttt{y}_m\texttt{))}]\!]$
      then $e\ [e\langle [\![\texttt{y}_1]\!], b[\![\texttt{y}_1]\!] \rangle / \langle [\![\texttt{z}_1]\!], p + \alpha^d \rangle] \cdots$
        $[e\langle [\![\texttt{y}_m]\!], b[\![\texttt{y}_m]\!] \rangle / \langle [\![\texttt{z}_m]\!], p + \alpha^d \rangle]$
      else $e\ [\langle i, b, p, o \rangle / \langle [\![\texttt{z}_1]\!], p + \alpha^d \rangle]$
  else if $e\langle [\![\texttt{f}]\!], b[\![\texttt{f}]\!] \rangle = \langle j, b', p', o' \rangle \in K$
  then $\mathcal{S}_2 q = \langle\ Succ\ j, p + Inv(p - p'), b', e', ro', o', r[k/o] \rangle$
    where $S_j = [\![\texttt{(set! z (call/cc g))}]\!]$
    and $e' =$ if $S_i = [\![\texttt{(set! x (f } \texttt{y}_1\texttt{))}]\!]$
      then $e[e\langle [\![\texttt{y}_1]\!], b[\![\texttt{y}_1]\!] \rangle / \langle [\![\texttt{z}]\!], b'[\![\texttt{z}]\!] \rangle]$
      else $e[\langle i, b, p, o \rangle / \langle [\![\texttt{z}]\!], b'[\![\texttt{z}]\!] \rangle]$

Figure 7: The Semantic Function $\mathcal{S}_2$ (part 1 of 2)

more a cosmetic than a substantive change.

Consider the case within $\mathcal{S}_2$ of evaluation of a `return` expression. In both $\mathcal{S}_1$ and $\mathcal{S}_2$ the actions to be taken in this case are embodied in the continuation k of a procedure instance; but while in $\mathcal{S}_1$, the continuation was (textually) part of the closure whose application initiated the procedure instance, in $\mathcal{S}_2$ it is a 4-tuple created at the point of application. The birth date of the procedure instance to which control is returning is the fourth component of the continuation of the current state. As promised, this birth date is used to retrieve the continuation in effect upon return, via the restoration function r. The construction of procedure strings within $\mathcal{S}_2$ is exactly as in $\mathcal{S}_1$.

Before proving the equivalence of the definitions of $\mathcal{E}_1$ and $\mathcal{E}_2$, let us write some theorems which characterize the procedure strings constructed during evaluation. It is intended that each of these theorems have a simple, intuitive interpretation in terms of the interprocedural behavior of a pro-

$S_i = [\![\texttt{(set! f (lambda}_\alpha\ \texttt{(x}_1 \cdots \texttt{x}_m\texttt{) <x}_{m+1} \cdots \texttt{x}_n\texttt{> } \cdots\ \texttt{))}]\!] \Rightarrow$
$\mathcal{S}_2 q = \langle Succ\, i, p, b, e[\langle \alpha, b\rangle / \langle [\![\texttt{f}]\!], b[\![\texttt{f}]\!]\rangle], k, o, r\rangle$

$S_i = [\![\texttt{(if x (goto m) (goto n))}]\!] \Rightarrow$
$\mathcal{S}_2 q = \langle \text{ if } e\langle [\![\texttt{x}]\!], b[\![\texttt{x}]\!]\rangle = true \text{ then } m \text{ else } n, p, b, e, k, o, r\rangle$

$S_i = [\![\texttt{(return x)}]\!] \Rightarrow$
$\mathcal{S}_2 q = \langle Succ j, p + Inv(p - p'), b', e[e\langle [\![\texttt{x}]\!], b[\![\texttt{x}]\!]\rangle / \langle [\![\texttt{y}]\!], b'[\![\texttt{y}]\!]\rangle], ro', o', r\rangle$
where $S_j = [\![\texttt{(set! y } \cdots\texttt{)}]\!]$
and $k = \langle j, b', p', o'\rangle$

$S_i = [\![\texttt{(end)}]\!] \Rightarrow$
$\mathcal{S}_2 q = q$

Figure 8: The Semantic Function $\mathcal{S}_2$ (part 2 of 2)

$\mathcal{E}_2 : Q \to Q \equiv \lambda q.$ Let $q' = \mathcal{S}_2 q$
in if $q' = q$ then q else $\mathcal{E}_2 q'$

Figure 9: The Semantic Function $\mathcal{E}_2$

gram, and that it illuminate a salient characteristic of procedure strings. While it will be proved using the semantics of $\mathcal{E}_2$, each holds in the case of $\mathcal{E}_1$, and the analogous proof is nearly identical to that for $\mathcal{E}_2$.

Theorem 2 *Let $q = \langle i, p, b, e, k, o, r\rangle$ be a state during the evaluation of a program by $\mathcal{E}_2$. Then there exists a procedure string u such that*

$$Net(p + u) = \epsilon.$$

Proof: The theorem says that every procedure string which corresponds to a state during evaluation may be extended to a balanced procedure string. Its proof is by induction on the number of states in the sequence described by the evaluation. In the initial state, $p = \epsilon$, and $Net(p + \epsilon) = \epsilon$ trivially. Assume that the theorem is true for sequences of n or fewer states, let q be the n^{th} state in the sequence of an evaluation, and let $q' = \mathcal{S}_2 q = \langle i', p', b', e', k', o', r'\rangle$. If $p' = p$, then the theorem holds trivially by induction. If $p' = p + \alpha^d$, then p' is extensible to a balanced procedure string by induction, since $Net(p' + \alpha^u) = Net\ p$ and p is extensible to a balanced procedure string. Else, $p' = p + Inv(p - p'')$, where p'' is a prefix of p. Suppose that p' cannot be extended to a balanced procedure string. Then

$$Net(p + Inv(p - p'')) = \cdots \alpha^u \cdots \text{ for some } \alpha \in \Lambda.$$

This α^u must be within $Inv(p - p'')$, since if $Net\, p = \cdots \alpha^u \cdots$ then p is not extensible to a balanced string (since the matching α^d must be found to the *left* of α^u), contradicting the induction hypothesis. Therefore $Inv(p - p'')$ has the form $X^{-1} + \alpha^u + Y^{-1}$, and $Net(p - p'') = Y + \alpha^d + X$ (where $X^{-1} \equiv Inv\ X$). Since

$$Net(p + Inv(p - p'')) = Net(p'' + Net(p - p'') + Inv(p - p''))$$

and since the α^d within $Net(p - p'')$ does not annihilate the α^u within $Inv(p - p'')$, X must have the form $R + \beta^u + S$. (The pair $\beta^u\beta^d$ within

$$Net(Net(p - p'') + Inv(p - p''))$$

prevents the annihilation of the matched pair $\alpha^d\alpha^u$.) Then $Net(p - p'') = Y + \alpha^d + R + \beta^u + S$. Since α^d can be annihilated only by an α^u to the right of S,

$$Net\, p = Net(p'' + (p - p'')) = \cdots \alpha^d + R + \beta^u + S.$$

Since β^u can only be annihilated by a β^d to the left of α^d, p is not extensible to a balanced procedure string, a contradiction of the induction hypothesis. Therefore p' is extensible to a balanced procedure string. □

We say that a procedure string p is *d-monotonic* if $p \in (\Lambda^d)^*$, and that p is *ud-bitonic* if $p \in (\Lambda^u)^*(\Lambda^d)^*$. We define *u-monotonic* and *du-bitonic* similarly. Of course, every d-monotonic or u-monotonic string is trivially ud-bitonic as well. If p is ud-bitonic, then $Inv\, p$ is also ud-bitonic. (Why?)

Corollary 1 *Let $q = \langle i, p, b, e, k, o, r\rangle$ be a state during the evaluation of a program by $\mathcal{E}_2$. Then $Net\, p$ is d-monotonic.*

Proof: By Theorem 2, p is extensible to a balanced procedure string. Suppose that $Net\, p$ is not d-monotonic. Then $Net\, p = \cdots \alpha^u \cdots$, and p is not extensible to a balanced procedure string, since a matching α^d must be found to the left of α^u, to be annihilated by *Net*, a contradiction of Theorem 2. □

Corollary 2 *Let $q = \langle i, p, b, e, k, o, r\rangle$ and $q' = \langle i', p', b', e', k', o', r'\rangle$ be states during the evaluation of a program by $\mathcal{E}_2$, such that q' precedes q. Then $Net(p - p')$ is ud-bitonic.*

Proof: By Theorem 2, p' and p are extensible to balanced procedure strings. Suppose that $Net(p - p')$ is not ud-bitonic. Then

$$Net(p - p') = \cdots \alpha^d \beta^u \cdots .$$

Since α^d can be annihilated only by an α^u to the right of β^u,

$$Net\, p = \cdots \alpha^d \beta^u \cdots .$$

Since β^u can be annihilated only by a β^d to the left of α^d, p is not extensible to a balanced procedure string, a contradiction. □

Given a ud-bitonic procedure string p, we will sometimes wish to refer to its u-monotonic prefix, or its d-monotonic suffix. We will denote these as $UpRun\, p$ and $DownRun\, p$ respectively; they satisfy

$$p = UpRun\, p + DownRun\, p$$

for any ud-bitonic p. Corollary 2 then says that the net effect of the inter-procedural flow of control between any two states during evaluation may be summarized as a sequence of procedure deactivations (a u-monotonic prefix), followed by a sequence of procedure (re)activations (a d-monotonic suffix).

Theorem 3 *Let $q = \langle i, p, b, e, k, o, r\rangle$ be a state during the evaluation of a program by $\mathcal{E}_2$, and let p' be a prefix of p. Then*

$$Net(p + Inv(p - p')) = Net\, p'.$$

Proof: By Corollaries 1 and 2, $Net\,p$ and $Net\,p'$ are d-monotonic, while $Net(p-p')$ and $Inv(p-p')$ are ud-bitonic. Since $p = p' + (p-p')$ and $Net\,p$ is d-bitonic,

$$Net\,p' = p'' + DownRun(Inv(p-p')) \text{ for some } p'' \in P$$

(that is, the suffix of $Net\,p'$ must annihilate $UpRun(Net(p-p'))$.) We have

$$\begin{aligned} Net(p + Inv(p-p')) &= Net(p' + (p-p') + Inv(p-p')) \\ &= Net(p' + Net(p-p') + Inv(p-p')) \\ &= Net(p'' + DownRun(Inv(p-p')) \\ &\qquad + UpRun(Net(p-p')) \\ &\qquad + DownRun(Net(p-p')) \\ &\qquad + UpRun(Inv(p-p')) \\ &\qquad + DownRun(Inv(p-p'))) \\ &= Net(p'' + DownRun(Inv(p-p'))) \\ &= Net\,p' \end{aligned}$$

□

Theorem 4 *Let $q = \langle i, p, b, e, k, o, r\rangle$ be a state during the evaluation of a program by $\mathcal{E}_2$. Then*

$$Net\,p = Net\,o$$

Proof: By induction on the number n of steps in the evaluation. In the initial state $p = o = \epsilon$, and the theorem is satisfied trivially. Assume that it holds for evaluations of n or fewer states, let q be the n^{th} state of evaluation, and let $q' = \langle i', p', b', e', k', o', r'\rangle = \mathcal{S}_2 q$. There are two cases in which $p \neq p'$.

- $p' = p + \alpha^d$. In this case (the application of a closure) $o' = p'$, and the theorem is satisfied.
- $p' = p + Inv(p - p'')$. Here p' is the procedure string of the state which results from a continuation application or a procedure return. In either case, the continuation $k = \langle i'', b'', p'', o''\rangle$ being applied[17] satisfies $Net p'' = Net o''$ by induction. By the definition of $\mathcal{S}_2$, $o' = o''$, since o' is the birth date of the procedure instance to which control is returning, and that is the procedure instance in which k was formed. By Theorem 3, $Net(p + Inv(p - p'')) = Net\,p''$, and we have that $Net\,p' = Net\,p'' = Net\,o'' = Net\,o'$.

[17] In the case of a `return` form, k is the 5th component of the state from which control is returning.

□

Theorems 3 and 4 are the justification for our interpretation of *Net p* as a listing of the procedures that are active ("on the stack") in the state to which p is the corresponding procedure string. They conjoin to demonstrate that whenever control returns to a procedure instance, *Net* applied to the current procedure string will reveal that the same procedures are active as when the procedure instance was born (that is, the same as *Net* applied to the birth date of the procedure instance).

To show equivalence between $\mathcal{E}_1$ and $\mathcal{E}_2$, we will show that there is a straightforward, component-wise correspondence between the states that occur during an evaluation under $\mathcal{E}_1$, and the states that occur during the same evaluation under $\mathcal{E}_2$.

Theorem 5 *$\mathcal{E}_1$ and $\mathcal{E}_2$ are equivalent definitions of $\mathcal{L}$.*

Sketch of proof: By induction upon the length of a sequence of states. We will show an example of the reasoning, in the case of the rule for application of a continuation. Assume that the program is evaluated from equivalent initial states, and that the theorem holds for sequences of no more than n states. We will show equivalence in the first five components of analogous states, since the final components (birth date and continuation restoration function) of the states of $\mathcal{E}_2$ are material only their effect upon the first five. Let $q_1 = \langle i_1, p_1, b_1, e_1, k_1 \rangle$ be the n^{th} state during evaluation under $\mathcal{E}_1$, and let $q_2 = \langle i_2, p_2, b_2, e_2, k_2, o, r \rangle$ be the n^{th} state during evaluation under $\mathcal{E}_2$. Let ${q_1}' = \langle {i_1}', {p_1}', {b_1}', {e_1}', {k_1}' \rangle$ be the state which satisfies ${q_1}' = \mathcal{E}_1\, q_1$, and let ${q_2}' = \langle {i_2}', {p_2}', {b_2}', {e_2}', {k_2}', o', r' \rangle$ be the state which satisfies ${q_2}' = \mathcal{E}_2\, q_2$.

By the rules within $\mathcal{S}_1$ for the formation and application of a continuation, we have that ${p_1}' = p_1 + \mathit{Inv}(p_1 - p'')$, where p'' is the procedure string of the state in which the continuation was formed. (The procedure string p'' is captured within the lambda expression which represents the continuation, in the state of its formation.) Similarly, ${i_1}' = \mathit{Succ}\, i''$, where i'' is the statement index of the state in which the continuation was formed. (Again, this statement index is captured by the lambda expression which represents the continuation.) ${b_1}' = b''$, where b'' is the variable birth date map of the state in which the continuation was formed. (This map, too, is captured at the point of the continuation's creation.) ${e_1}' = e_1[d/\langle [\![\mathtt{x}]\!], b[\![\mathtt{x}]\!] \rangle]$, where d is the value passed as the argument to the continuation, and `x` is the variable which receives the value of the originating `call/cc` expression. Finally, ${k_1}' = k''$, the continuation of the state in which the `call/cc` expression was evaluated.

By the rules within $\mathcal{S}_2$ for the formation and application of a continua-

tion, we have that $p_2' = p_2 + Inv(p_2 - p'') = p_1'$, where p'' is the procedure string of the state in which the continuation was formed. (The procedure string p'' is the third component of the continuation.) $i_2' = Succ\, i'' = i_1'$, where i'' is the statement index of the state in which the continuation was formed. (i'' is the first component of the continuation.) $b_2' = b'' = b_1'$, the b component of the state in which the continuation was formed (the second component of the continuation). $e_2' = e_2[d/\langle [\![\mathtt{x}]\!], b''[\![\mathtt{x}]\!]\rangle]$, where d is the value passed as the argument to the continuation, and `x` is the variable which receives the value of the originating `call/cc` expression. The birth date map used to update the receiving environment is equal to b_1', and it follows that the same location within each environment will be modified following return of the `call/cc` expression, and $e_2' = e_1'$. Finally, $k_2' = ro''$, where o'', the fourth component of the continuation, is the birth date of the procedure instance in which the `call/cc` expression was evaluated. By the definition of $\mathcal{S}_2$, r was updated to return the continuation of the state in which the `call/cc` expression was evaluated, the analogue of k_1'. □

2.6 Optimal Solutions in Terms of Procedure Strings

At this point, we have a definition of $\mathcal{L}$ (over non-reflexive domains) that constructs a sequence of procedure strings as it evaluates a program. Our goal is to build approximations to these procedure strings at compile time, and to use these approximations to guide the optimizer. In this subsection we show that procedure strings are an ideal form of information concerning side-effects and object lifetimes. Having done so, we will return to the semantics of the last subsection, and form from them an abstraction based upon an approximation to procedure strings.

Before proceeding, let us recall the basics of dependence analysis. The traditional types of dependence are *flow-*, *anti-*, and *output-dependence*. A dependence arises when two subcomputations S_1 and S_2 (where S_1 precedes S_2 in time) each access a single location in memory, and at least one of them modifies the location. A flow-dependence arises when S_1 writes and S_2 reads the location. An anti-dependence arises when S_1 reads and S_2 writes the location. An output-dependence arises when S_1 writes, and S_2 overwrites the location. No constraint upon execution order is implied if S_1 reads, and S_2 also reads the location. See [47, 12, 13] for dependence testing between statements within a single procedure, in array-based languages; see [45] for dependence testing in the presence of subroutine calls, in array-based languages.

2.6.1 Side-Effects, in Terms of Procedure Strings

Suppose we are asked to determine what side-effects a subcomputation S has, and that we have at our disposal all of the states of the program, before, during and after S. First we must determine what is meant by a side-effect. We will adopt a somewhat unusual perspective, summarized in the following definition.

Definition 1 *A subcomputation S has a* side-effect *upon a* mutable *object X if X exists prior to S, and S makes a reference to (use or modification of) X.*

There is much to explain in this definition. Why do we include uses (and not merely modifications) in our definition of side-effects? What, in light of the non-local control flow made possible by `call/cc`, is the duration of S? That is, when does S begin and end, given that procedures may be arbitrarily exited and re-entered by the use of continuations? Why do we distinguish objects which predate S from those created during S? We will argue that this definition, while somewhat unfamiliar, describes the essence of side-effects, and is the appropriate definition for our purpose.

First, by our definition of flow-, anti-, and output-dependences above, we see that side-effects give rise to dependence only when at least one of the side-effects is a modification. Nevertheless, because every dependence involves two references, one of which may be a use (and not a modification), to be made the basis of an interprocedural dependence test, our definition of side-effects must regard quantities that are read, as well as those that are written, during each subcomputation.

The construction of procedure strings in $\mathcal{E}_1$ and $\mathcal{E}_2$ gives a concrete meaning to the duration of a subcomputation. Where no continuations are involved, we mean by the duration of a subcomputation S, the time between the procedure application which initiates S, and the return from that application. Suppose that S is initiated normally, by application of a closure, but that during S a continuation is applied which was created prior to S, and has therefore the effect of exiting S entirely. Let p be the procedure string of the state in which the continuation is applied, and let p' be the procedure string of the state in which the continuation was formed. By the definition of $\mathcal{S}_2$, the procedure string of the state following application of the continuation is $p + Inv(p - p')$. Recall from the discussion preceding Theorem 3 that the suffix $Inv(p - p')$ describes (first) the exit of any procedures which are active at application of the continuation, but not at the point of the its creation, and (second) the re-activation of any procedures which were active at the continuation's creation, but not at the point of its application. This is the natural interpretation of a ud-bitonic string, such

as $Inv(p{-}p')$: a sequence of procedure deactivations, followed by a sequence of procedure (re)activations. In short, we have taken great pains to insure that interpreting the sequence of procedure strings of an evaluation just as though continuations were not present (that is, only in terms of normal procedure entrance and exit) is sensible and intuitive. When a procedure string contains a term of the form α^u, we know that an instance of λ_α has been exited, whether by continuation or normal return; and when it contains α^d, we know that an instance of λ_α has been (re)activated, whether by application of a closure of λ_α, or by a continuation which was formed when an instance of λ_α was active, and applied after that instance of λ_α terminated. We will define the duration of a subcomputation then, by the balanced procedure string which is delimited by its initiation and termination. To repeat, its initiation may correspond to a fresh closure application, or to the reactivation of a previously exited procedure instance, and its termination may correspond to a normal return or to a non-local exit effected by application of a continuation. The distinction is made unimportant by the construction of procedure strings. (We include the procedure application that initiates it, if any, in a subcomputation. We likewise include the assignment to the variable which captures its return value, if any, in a subcomputation.)

Why does Definition 1 distinguish objects which are referenced during S, and existed prior to S, from those which are referenced during S, but were created during S? The creation of a new object X during S implies, of itself, no dependence to S from the surrounding computation, or vice versa. Furthermore, any modifications that occur to X during S are invisible from without S.

This explanation may fail to put the matter to rest. Let X be created (and possibly modified) during S, and used after the conclusion of S. Suppose we grant that there are no visible side-effects to X during S; but if this estimation of side-effects becomes the basis for our dependence testing, are we not obliged to include a modification of X among the side-effects of S, in order to recognize the dependence from S to the use of X? The point is that for such a dependence to exist, X must be accessed following S, and such an access must begin with a location which is known both to S and the subsequent computation, such as the variable which receives the return value of S, or another variable which serves as a point of communication between S and the surrounding computation. By identifying such points of communication, we find the "roots" of all dependences which originate from S. We will prove that all such points of communication are locations which exist prior to S. We assume, in the proof below, that we may distinguish the object X, in which we are interested, from all other values in the environment. That is, we will not be concerned with the trivial objection

that X may be "communicated" from within S by simply recomputing its value, or by arranging that X be a constant whose value is known outside of S, etc: such devices do not give rise to dependence. To communicate X from within S, we assume that a chain of memory accesses must occur from the point of its computation to the point of its use. We will write q_i to mean the i^{th} state in the sequence described by the evaluation of a program under $\mathcal{E}_2$.

Theorem 6 *Let S be a subcomputation, defined by a balanced procedure string s, during the evaluation of a program by $\mathcal{E}_2$, let $X \in D$ be an object computed during S, and let q_i be a state, subsequent to the termination of S, in which a variable x is accessed, such that x has the value X. Then there is a state q_j, $j \leq i$, also subsequent to the termination of S, in which a variable y is accessed, such that y is bound prior to S, y is modified during S, and either $i = j$ and $x = y$, or there is a dependence from the access of y in q_j to the access of x in q_i.*

Proof: By induction on the number n of states between the termination of S and q_i. Let $n = 0$. Then q_i is the state which follows the termination of S immediately. In this case, no procedures are applied (and thus no variables are bound) between the termination of S and q_i. Suppose that x is bound during S, and let λ_α be the binding lambda expression. Since s is balanced, the binding instance of λ_α terminates during S. But x is in the lexical environment of q_i. Therefore x is captured by a closure or continuation c during S, which is applied between the termination of S and q_i. This is impossible, since q_i is the first state following the termination of S. Therefore x is bound prior to S, and since X is computed after the binding of x, x is assigned the value of X during S. Letting $x = y$ and $i = j$, the theorem is satisfied for $n = 0$.

Now assume the theorem holds when there are n or fewer states between the termination of S and q_i, $n \geq 1$. Let q_i, as defined in the theorem, be the n^{th} state following the termination of S. Let q_b be the state in which x is bound, and let λ_α be the binding lambda expression. There are three cases.

1. x is bound prior to S. Since q_b precedes S, and since X is computed during S, x must be assigned after being bound. If this assignment occurs during S, then the theorem follows at once, by letting $x = y$ and $i = j$. Otherwise, the assignment occurs in a state q_k between the termination of S and q_i. However, this assignment involves an access to a variable z whose value is X, since by the definition of $\mathcal{S}_2$, the value of every expression is either passed from a variable as the argument to a continuation, or to a `return` form. By induction,

there exists a state q_j, $j \leq k < i$, subsequent to the termination of S, in which a variable y is accessed such that y is bound prior to S, y is modified during S, and either $j = k$ and $y = z$ or there is a dependence from the access of y in q_j to the access of z in q_k. There is a dependence from the the access of z in state q_k to the assignment to x in state q_i, and therefore by the transitivity of dependence, there is a dependence from the access of y in q_j to the access of x in q_i.

2. x is bound during S. Because s is balanced, the binding instance of λ_α terminates during S. But x is in the lexical environment of q_i. It is therefore captured by a closure or continuation $c \in D$ during S, which is applied between the termination of S and q_i. Let q_k be the state in which this application occurs, and let z be the variable in the operator position of the application. By induction, there is a state q_j, $j \leq k < i$, subsequent to the termination of S, in which a variable y is accessed, such that y was modified during S, and either $j = k$ and $y = z$ or there is a dependence from the access of y in q_j to the access of z in q_k. There is a dependence from the application of c in q_k to the access of x in q_i, and therefore by the transitivity of dependence, there is a dependence from the access of y in q_j to the access of x in q_i.

3. x is bound after the termination of S. Since x has the value X in q_i, x is either assigned this value after its binding, or bound with X as its initial value. In either case, the assignment or binding procedure application necessitates an access to a variable whose value is X, and the theorem holds by the argument of transitivity made in cases 1 and 2 above.

□

All dependences of a computation must be honored[18] by our restructuring compiler. Given a subcomputation S, there are several ways in which dependences may arise due to S. Let R be the entire computation preceding S, and T the entire computation which follows S. Any dependence from R to S obviously involves an object that exists prior to S. A dependence from S to T may involve an object that exists prior to S, or an object created during S. The remarkable fact proved in Theorem 6 is that any dependence from S to T that involves an object created during S results (by transitivity) from a dependence that involves (only) an object that exists prior to S! By enforcing each "primary" dependence by which a "secondary" dependence is transitively induced, we guarantee enforcement of the secondary

[18]More precisely, they must appear to be honored. We will see, when we consider the restructuring phase of compilation, that the distinction is sometimes useful.

dependence. (See [36] for a thorough discussion of dependence enforcement via synchronization.) Thus we need only regard dependences to and from S that involve objects that exist prior to S. Definition 1, then, accords with the requirements of dependence analysis.

Having arrived at a satisfactory definition of a side-effect, let us cast the definition in terms of procedure strings.

Theorem 7 *Let $\dot{\mathbf{x}}$ be an instance of the variable* $\mathbf{x}$*, let the procedure string p_b be the date of its birth (in state q_b), and let p_r be the procedure string of a state q_r in which a reference to $\dot{\mathbf{x}}$ takes place. Then $Net(p_r - p_b)$ contains a term of the form α^d if and only if the instance $\dot{\lambda}_\alpha$ of λ_α corresponding to this term has a side-effect upon $\dot{x}$.*

Proof:

- *If.* Suppose that $\dot{\lambda}_\alpha$ has a side-effect upon $\dot{\mathbf{x}}$. Then by Definition 1, $\dot{\lambda}_\alpha$ is active at q_r, and therefore $Net\, p_r = \ldots \alpha^d \cdots$ where α^d corresponds to the activation (or reactivation, by application of a continuation) of $\dot{\lambda}_\alpha$. Further, since the corresponding α^u must be found to the right of this α^d, we have that $Net(p_r - p_b) = \cdots \alpha^d \cdots$.

- *Only if.* Suppose that $Net(p_r - p_b)$ contains a term of the form α^d, and let $\dot{\lambda}_\alpha$ be the instance of λ_α that is applied in state q_a that corresponds to the term. Then $\dot{\mathbf{x}}$ was bound prior to q_a, and $\dot{\lambda}_\alpha$ was active at q_r (that is, the matching α^u term which denotes the deactivation of λ_α is absent from p_r). By Definition 1, $\dot{\lambda}_\alpha$ has a side-effect upon $\dot{\mathbf{x}}$.

□

The string $p_r - p_b$ is a record of the interprocedural movements between the point at which $\dot{\mathbf{x}}$ is bound and a point at which it is referenced. According to the discussion above, if the net effect of that movement has been *downward* into an instance $\dot{\lambda}_\alpha$ of λ_α, then $\dot{\lambda}_\alpha$ has a side-effect upon $\dot{\mathbf{x}}$, since $\dot{\mathbf{x}}$ existed prior to $\dot{\lambda}_\alpha$'s activation (or reactivation), and was referenced while $\dot{\lambda}_\alpha$ was still active. The side-effect that Theorem 7 attributes to the procedure instance $\dot{\lambda}_\alpha$ is visible to the procedure which invokes $\dot{\lambda}_\alpha$. Let the invoking procedure instance be $\dot{\lambda}_\delta$. If $\dot{\lambda}_\delta$, too, has a side-effect as a result of the reference to $\dot{\mathbf{x}}$ at p_r, then it, too, will be represented in $Net(p_r - p_b)$ as a term δ^d. (This depends entirely upon the movements that $\dot{\mathbf{x}}$ describes with respect to $\dot{\lambda}_\delta$.) This gives us a perfect test for side-effects, in the sense that we may state exactly which procedure instances have side-effects as a result of each variable reference that occurs during execution.

As a special case of Theorem 20, we may have that $\alpha = \delta$ (that is, the routine within which x is referenced directly may have a side-effect as a result of the reference), and even that $\alpha = \beta$. In order to have $\dot{\lambda}_\alpha = \dot{\lambda}_\beta$ where $\dot{\lambda}_\beta$ is the instance of λ_β that binds $\dot{\mathbf{x}}$, it must be that $\dot{\lambda}_\beta$ is deactivated, and reactivated by application of a continuation created while it was active, *and* that $\dot{\mathbf{x}}$ is captured by a closure or continuation while $\dot{\lambda}_\beta$ is active (so that $\dot{\mathbf{x}}$ describes a movement whose net shape, with respect to $\dot{\lambda}_\beta$, is first upward, then downward).

2.6.2 Stack Allocation, in Terms of Procedure Strings

Now let us turn to the problem of allocating variable instances on a stack.

Theorem 8 *Let λ_β be a procedure which binds a variable* $\mathbf{x}$*, let p_b be the birth date of an instance $\dot{\lambda}_\beta$ of λ_β, let $\dot{\mathbf{x}}$ be the instance of* $\mathbf{x}$ *bound by $\dot{\lambda}_\beta$, and let p_r be the procedure string of a state q_r in which a reference is made to $\dot{\mathbf{x}}$. Then $Net(p_r - p_b)$ contains a term β^u, if and only if $\dot{\lambda}_\beta$ is deactivated before $\dot{\mathbf{x}}$ is referenced in q_r.*

Proof:

- *If.* Suppose that $\dot{\lambda}_\beta$ is deactivated before $\dot{x}$ is referenced in the state whose procedure string is p_r. Then $p_r = p_b + (p_r - p_b)$, where $p_b = \cdots \beta^d$, and $p_r - p_b = \cdots \beta^u \cdots$ where this matching $\beta^d \beta^u$ pair corresponds to the activation and deactivation of $\dot{\lambda}_\beta$. Therefore

$$Net(p_r - p_b) = \cdots \beta^u \cdots$$

  as desired.

- *Only if.* Recall that the active procedures in q_r are read from the string $Net\, p_r$. The theorem says, then, that if $Net(p_r - p_b)$ contains the term β^u, then p_r contains the balanced substring, which begins with β^d and ends with β^u, that corresponds to the subcomputation initiated by the binding instance of λ_β. (This balanced substring is deleted from $Net\, p_r$.) By the rule within $\mathcal{E}_2$ for closure application, p_b ends in β^d. Thus, if $Net(p_r - p_b)$ contains β^u, then it must begin with β^u (to balance the procedure string of the entire computation). Therefore if $Net(p_r - p_b)$ contains β^u, then in p_r the β^d in which p_b ends will be matched by the first unmatched β^u in $p_r - p_b$. By our interpretation of $Net\, p_r$, this means that the instance of λ_β that binds $\dot{\mathbf{x}}$ bas been deactivated before q_r.

□

Assuming the stack frame associated with a procedure instance is overwritten when the instance is exited, any variables it binds that are referenced following its exit must be allocated in the heap. If, however, we examine all references to its bound variables, and find that all occur prior to deactivation of the procedure instance, then the variables may be bound to locations on the stack. Actually, to emphasize the impracticality of this test in its current form, we should write, "then the variables *could have been* bound to locations on the stack." This test is similar to the optimal MIN algorithm for page replacement in virtual memory management: it requires foresight. Nevertheless, we can put it to very practical use, since a data flow analysis based upon it will have a sort of "blurry" foresight.

Suppose that a procedure instance $\dot{\lambda}_\beta$ is deactivated, and reactivated by application of a continuation, and that all references to the variable instances it binds occur while the procedure instance is active (that is, following its reactivation, but prior to any further deactivations). In this case, its bound variables must be heap-allocated, since they are referenced after the procedure instance is deactivated. The reader may verify that the initial deactivation of $\dot{\lambda}_\beta$ is revealed by Theorem 8, and therefore that the need to heap-allocate activation records of procedure instances which are re-activated by application of first-class continuations is recognized by the theorem.

2.6.3 Generalized Hierarchical Allocation and Deallocation

We may easily extend the result of the last subsection to accommodate a richer selection of areas from which to allocate than the two-fold distinction between stack and heap. In the extreme, we are led to the following tactic for storage management. With each procedure instance we associate a list of objects to be deallocated upon its exit. When allocating an object, we add it to the "to be deallocated" list of the nearest procedure instance which will outlive all references to the object. (In the worst case, this will be the topmost procedure instance, the root of the stack.) For each object, then, we must find the maximum m over all references to the object, of the number of procedure instances, of those active at the point of the object's creation, that are exited before the point of reference. We place the object on the deallocation list found m procedure instances "above" the procedure instance in which it is created.

Let $\dot{\mathtt{x}}$ be an instance of the variable $\mathtt{x}$ that is bound in state q_b, and let q_r be a state in which $\dot{\mathtt{x}}$ is referenced. Let p_b and p_r be the procedure strings corresponding to q_b and q_r, respectively. As in the case of stack allocation, we will consider the procedure string $Net(p_r - p_b)$. Every term in $Net(p_r - p_b)$ of the form α^u denotes the exit of an instance of λ_α, which

```
(define fact ( lambda_α (n k)
   (if (= n 0)
       (k 1)
       (fact (1- n)
             ( lambda_β (m) (k (* n m)))))))
(fact 10 ( lambda_γ (x) x))
```

Figure 10: Example of Stack-Allocated Variables

instance was active in q_b. (Since the balancing α^d is not found in $p_r - p_b$, it must be in p_b.) The number of such terms (summing over all $\alpha \in \Lambda$) is the count of procedure instances, of those active at q_b, that are outlived by $\dot{\mathtt{x}}$ between q_b and q_r inclusive. Let the maximum of this count, over all references to $\dot{\mathtt{x}}$ (that is, over all states q_r in which $\dot{\mathtt{x}}$ is referenced) be m. We may place $\dot{\mathtt{x}}$ on the deallocation list associated with the m^{th} procedure instances above the procedure instance active in q_b, knowing that the deallocation list is associated with a procedure instance that is not outlived by $\dot{\mathtt{x}}$.

Actually, we are not proposing this seriously as a storage management strategy; it is instead a simple motivation for the very nearly related problem of placing dynamically allocated objects within a hierarchical shared memory. We will return to the problem in which we have genuine interest in subsection 2.15.

2.6.4 Examples of Side-Effects and Object Lifetimes

Consider first the example of Figure 10. The factorial function is shown, written in *continuation passing style* (we will have more to say about this style below). The local variables of λ_α (`n` and `k`) are captured by the closure of λ_β, which is clearly a downward funarg. Suppose, as per the example, that the expression `(fact 10 (lambda`$_\gamma$ `(x) x))` is evaluated, and let p_b be the birth date of one of the instances of `n` during the evaluation, and p_r be the procedure string of the state in which this same instance of `n` is referenced, within λ_β. Then we have that

$$p_b = \alpha^d \cdots \alpha^d$$

(one or more terms),

$$p_r = \alpha^d \cdots \alpha^d \beta^d \cdots \beta^d,$$

and

$$Net(p_r - p_b) = \alpha^d \cdots \alpha^d \beta^d \cdots \beta^d.$$

Since this holds for all choices of p_b and p_r, the instances of `n` may be stack-allocated by Theorem 8. The same is true of the instances of `k`.

```
(define accum-fn ( lambda_α (x)
   ( lambda_β (y) (set! x (+ x y)) x)))
(define apply-to-range ( lambda_γ (lo hi fn)
   (if (= lo hi)
       (fn lo)
       (begin (fn lo)
              (apply-to-range (1+ lo) hi fn)))))
(define sum-of-integers ( lambda_σ (m n)
   (apply-to-range m n (accum-fn 0))))
(define list-of-sums ( lambda_ϵ (l1 l2)
   (if (null? l1)
       #f
       (cons (sum-of-integers (car l1) (car l2))
             (list-of-sums (cdr l1) (cdr l2))))))
```

Figure 11: Example of Side-Effects and Object Lifetimes

Now consider the example of Figure 11. The procedure `accum-fn` returns a procedure (an instance of λ_β) that captures a state variable (an instance of `x`). When λ_β is applied, its argument is added to `x`, and the accumulated sum is returned. `apply-to-range` applies a procedure (its third argument) to every integer between `lo` and `hi`, inclusive. `sum-of-integers` first creates an accumulating function with a call to `accum-fn`, and then invokes `apply-to-range` to sum the integers in the range of `m` to `n`, inclusive. Finally, `list-of-sums` takes two lists of integers, applies `sum-of-integers` to the corresponding members of the lists, and forms a list of these sums.

Let us begin by considering an invocation of λ_α (that is, of `accum-fn`). An instance $\dot{\mathtt{x}}$ of `x` is born by this invocation; let its birth date be p_b. Suppose that, by subsequent application of the return value of λ_α, $\dot{\mathtt{x}}$ is referenced in a state whose procedure string is p_r. We will have that

$$Net\,(p_r - p_b) = \cdots \alpha^u \cdots,$$

which by Theorem 8 implies that $\dot{\mathtt{x}}$ cannot be stack-allocated. λ_α has no side-effects, for the reason that it is not active when $\dot{\mathtt{x}}$ is referenced. That is, $Net(p_r - p_b)$ will never contain a term of the form α^d.

Now consider an application of `sum-of-integers`. The first action taken is to invoke `accum-fn`; as above, let the instance of `x` that is created be $\dot{\mathtt{x}}$, and its birth date be p_b. The instance $\dot{\lambda}_\beta$ of λ_β that is returned by `accum-fn` is passed to λ_γ (`apply-to-range`), where it is applied repeatedly, causing references to $\dot{\mathtt{x}}$. Let p_r be the procedure string of the state in which one such reference occurs. We have that

$$Net(p_r - p_b) = \cdots \gamma^d \cdots,$$

and therefore that the active instance of λ_γ has a side-effect upon $\dot{\mathtt{x}}$. The active instance of λ_σ (`sum-of-integers`) has no such side-effect, because $Net(p_r - p_b)$ contains no term of the form σ^d (no term involving λ_σ at all, in fact).

Finally, consider an application of λ_ϵ. We have seen that `sum-of-integers` has no side-effects, and therefore each recursive instance of λ_ϵ is independent of the others (aside from the formation of the list of results). In short, the state variables (instances of `x`) that are created to form each sum within `list-of-sums`, are invisible to the caller of `sum-of-integers`. There is therefore a potential for high-level parallelism in this computation; we would hope to construct a framework of program analysis that would reveal this parallelism.

We saw above that an instance $\dot{\mathtt{x}}$ of `x` (the variable bound by λ_α) cannot be stack-allocated, for the reason that references are made to it after the termination of the binding instance of λ_α. This was reflected as a term α^u in the string $Net(p_r - p_b)$ that summarizes the activity between the binding of $\dot{\mathtt{x}}$ and a reference to it. However, it is easy to see that this is the only term denoting upward movement, within $Net(p_r - p_b)$, and therefore that $\dot{\mathtt{x}}$ may be deallocated upon exit of the instance of λ_σ which creates it (by an invocation of λ_α).

2.6.5 Some Observations

It is interesting to juxtapose the results of this subsection concerning side-effects with those concerning heap-allocation. Theorems 7 and 8 lead us to the conclusion that *downward* movements give rise to side-effects, while *upward* movements give rise to heap-allocation. This is interesting because upward movements (as manifest in the *upward funarg problem*) are perhaps the central issue in the sequential implementation of a language with first-class procedures (that is, upward movements prevent the evaluation of such a language by a simple stack mechanism); but Theorem 7 suggests that downward movements (of mutable objects) may be among the central issues in the parallel implementation of such languages, for (by Theorem 6) all interprocedural dependences arise from such downward movements.

To digress, there is a pronounced shortcoming of Fortran, as a language for parallel processing, that is set in sharp relief by these theorems. In Fortran, all storage is allocated, effectively, at the global level; in terms of procedure strings, we would say that every such object has a birth date of ϵ (the empty procedure string). Therefore, every reference to such an object (assuming it is a mutable object, such as a scalar variable or array element) will, by Definition 1, induce a side-effect in every procedure that is active when the reference is made. Now, we could sharpen our definition of dependence, so that each re-definition of the object is viewed, in effect,

as a separate instantiation of the object; this approach is taken in [19]. Put another way, by examination of the definitions and uses of a scalar variable, we may discover that the single name may be replaced by several variables, whose lifetimes are mutually disjoint. Having done so, we may discover that the side-effects upon the resultant (newly introduced) variables have more restricted visibility than those upon the original. Indeed, any dependence test may be sharpened by giving it a measure of flow-sensitivity. The point being made here is that the visibility of a side-effect upon an object is circumscribed by the lifetime of the object; in the case of the statically allocated storage of Fortran, all objects have the maximum possible lifetime. This is reflected in Theorem 7, by the fact that a statically allocated object describes a downward movement through every procedure that is active when it is referenced.

2.7 Stack Configurations

The construction of a useful abstraction is a practical matter, constrained by opposing requirements: to restrict information content so that the abstract domain may be represented and manipulated efficiently by a computer, on the one hand, and to preserve information content so that when applied to real programs, the resulting analysis is sufficiently powerful to yield appreciable performance improvements, on the other. The first step in the abstraction of procedure strings, then, is to separate the information they contain into the essential and the inessential; our abstraction should dispose, as much as possible, only of the latter. Turning to the examples of dependence analysis, stack-allocation, and hierarchical storage management, we see, of the solutions we have proposed to these problems, that

- each makes use of a difference of two strings,
- each makes use of strings which have been reduced by the *Net* operator, and
- none depends upon the order of elements within the strings (once reduced by the *Net* operator).

The structure of our abstraction will take advantage of the second and third of these points, whereas to take advantage of the first point (indeed, to render our abstraction satisfactory in accuracy) will require an alteration to the way procedure strings are constructed in the semantics. We will return to this point.

We need two auxiliary functions, $Trace : P \to \Lambda \to P$ and $Dir : P \to \Lambda \to \Delta$ before we can define the abstraction map itself. Let $p \in P$ and $\alpha \in \Lambda$ ($p \neq \bot_P$ and $\alpha \neq \bot_\Lambda$). $Trace\ p\alpha$ is the result of deleting all terms

from p other than α^d or α^u. Let $\Delta = \{\epsilon, \mathbf{d}, \mathbf{dd}^+, \mathbf{u}, \mathbf{uu}^+, \mathbf{u}^+\mathbf{d}^+\}$. The six members of Δ are tokens representing the regular expressions ϵ, d, dd^+, u, uu^+ and u^+d^+ respectively. The function *Dir*, when applied to a procedure string p and a lambda expression index α, returns a member of Δ, according to the structure of $Net(Trace\, p\alpha)$. It is defined as

$$\begin{aligned} Dir \equiv \lambda p \lambda \alpha.\ \text{Let } & Net(Trace\, p\alpha) = \alpha^{x_1}\alpha^{x_2}\cdots\alpha^{x_n} \\ \text{in } & \text{case } x_1x_2\cdots x_n \in \epsilon : \quad \epsilon \\ & \qquad\qquad\quad d : \quad \mathbf{d} \\ & \qquad\qquad\quad dd^+ : \quad \mathbf{dd}^+ \\ & \qquad\qquad\quad u : \quad \mathbf{u} \\ & \qquad\qquad\quad uu^+ : \quad \mathbf{uu}^+ \\ & \qquad\qquad\quad u^+d^+ : \quad \mathbf{u}^+\mathbf{d}^+ \end{aligned}$$

Intuitively, the function *Trace* extracts from a procedure string p all of the information concerning a single procedure λ_α; the function *Net* when applied to the result, produces a string which summarizes the net movements described by p, *with respect to* λ_α. The function *Dir* (for *direction*) then characterizes this movement as one of six types. By Corollary 2, the procedure strings to which we will apply *Trace* and *Dir* are ud-bitonic, and it is easy to see that the disjoint union of the six regular languages represented by the members of Δ is the language u^*d^*.

We want to summarize the net movements made with respect to each lambda expression, in the abstraction of a procedure string; this means isolating all those terms within the string that pertain to a single lambda expression, and applying the *Net* operator to the result, as in the definition of *Dir*. It is possible to proceed in the opposite order, by first applying *Net* and afterwards isolating all terms pertaining to a single lambda expression. It will be useful to be sure that the outcome is independent of the order in which we proceed.

Theorem 9 *Let p_1 and p_2 be the procedure strings of two states during the evaluation of a program under $\mathcal{E}_2$, where p_1 is a prefix of p_2. Then*

$$Net(Trace(p_2 - p_1)\alpha) = Trace(Net(p_2 - p_1))\alpha.$$

Proof: Suppose the theorem is false. Then it must be that

$$Net(Trace(Net(p_2 - p_1)\alpha)) \neq Trace(Net(p_2 - p_1))\alpha;$$

that is, that further annihilation of matching $\alpha^d\alpha^u$ pairs is possible within $Net(p_2 - p_1)$, once the terms within it that are unrelated to α are deleted. Therefore p_1 must be of the form $\ldots\alpha^d + X + \alpha^u\ldots$ where X is an unbalanced string. But then p_1 cannot be extended to a balanced procedure string, contradicting Theorem 2. □

As a special case of this theorem, we may let $p_1 = \epsilon$, so that the result applies to all procedure strings that correspond to states during evaluation. Intuitively, the theorem says that the net movements described by a procedure string with respect to one lambda expression are independent of the net movements it describes, with respect to other lambda expressions.

A *stack configuration* is a member of the set $\hat{P} = \Lambda \to 2^{\Delta}$ of maps from lambda expression indices to subsets of Δ. The abstraction map Abs_P is defined as

$$Abs_P \equiv \lambda p.\lambda\alpha.\text{if } p = \bot_P \text{ or } \alpha = \bot_\Lambda \text{ then } \{\} \text{ else } \{Dir\ p\alpha\}.$$

Notice that if a lambda expression λ_β is not represented in a procedure string $p \neq \bot_P$, then p's image in $\hat{P}$ will map β to $\{\epsilon\}$.

The image in $\hat{P}$ of a (fully defined) procedure string maps each (fully defined) lambda expression index onto a singleton subset of Δ; if projection from P to $\hat{P}$ were the only means of constructing stack configurations, then they would be better defined as members of $\Lambda \to \Delta$ (with Δ extended to include a bottom element). We will more often, however, arrive at stack configurations via functions that operate upon other stack configurations, and these will give rise to stack configurations with less information than those that result from projection from the concrete domain of procedure strings. The least upper bound operator $\sqcup_{\hat{P}}$ is defined simply as

$$\sqcup_{\hat{P}} \equiv \lambda\hat{p}\lambda\hat{q}.\lambda\alpha.(\hat{p}\ \alpha) \cup (\hat{q}\ \alpha)$$

and the partial order $\sqsubseteq_{\hat{P}}$ among members of $\hat{P}$ is defined as

$$\sqsubseteq_{\hat{P}} \equiv \lambda\hat{p}\lambda\hat{q}.(\hat{p}\ \alpha) \subseteq (\hat{q}\ \alpha)\ \ \forall\alpha \in \Lambda.$$

($\sqcup_{\hat{P}}$ and $\sqsubseteq_{\hat{P}}$ have been defined as functions of type $\hat{P} \to \hat{P} \to \hat{P}$ and $\hat{P} \to \hat{P} \to Bool$ respectively, although we will write them as infix operators, as is traditional.)

We should be certain that our handling of the bottom element $\bot_P$ of P is sensible. It is abstracted to the element $\lambda\alpha.\{\}$ in $\hat{P}$. By the discussion of concretization maps in subsection 2.4,

$$Conc_P(Abs_P \bot_P) = \{p \mid Abs_P p \sqsubseteq_{\hat{P}} \lambda\alpha.\{\}\} = \{\bot_P\}$$

since any procedure string $p \neq \bot_P$ is abstracted to a stack configuration that maps every fully defined lambda expression index to a non-empty subset of 2^{Δ}. Therefore the bottom element of P is abstracted to the bottom element of $\hat{P}$, as expected. Since the domain of procedure strings is flat, it follows that Abs_P is (trivially) monotonic and continuous.

We can be quite precise about the information loss that occurs during this abstraction. First, if two procedure strings have the same *Net* value, they are indistinguishable after abstraction. Second, if a procedure string that has been reduced by the *Net* operator, is a permutation of another that is likewise reduced by the *Net* operator, then they will be indistinguishable after abstraction. Finally, if two procedure strings are equivalent according to the above criteria, and in addition, the regular expressions that describe the contributions of each lambda expression to one of the strings, are equivalent to the corresponding regular expressions for the other (or, more exactly, the corresponding regular expressions are equivalent modulo the six classes defined by Δ), then the two strings are indistinguishable after abstraction.

2.8 The Abstraction of Operations Over Procedure Strings

There are several operations upon procedure strings which must be abstracted to stack configurations, in a way that preserves the meaning of procedure strings. To be precise, if $f : P \to P \to \cdots \to P$ is an n-ary function from procedure strings to procedure strings, then an abstraction $\hat{f} : \hat{P} \to \hat{P} \to \cdots \to \hat{P}$ of f must satisfy

$$Abs_P(fp_1 \ldots p_n) \sqsubseteq_{\hat{P}} \hat{f}(Abs_P p_1) \ldots (Abs_P p_n) \tag{1}$$

or, equivalently

$$fp_1 \ldots p_n \in Conc_{\hat{P}}(\hat{f}(Abs_P p_1) \ldots (Abs_P p_n)). \tag{2}$$

In words, the projection onto $\hat{P}$ of the result of applying f to arguments, should be represented by (contained in the concretization of) the result of first projecting the arguments onto $\hat{P}$, and then applying $\hat{f}$. The direction of inclusion is important: we will begin abstract interpretation by projecting some initial values onto abstract domains; afterwards we will operate entirely within the abstract domains, by applying the abstractions of functions. To be meaning-preserving, the (concretization of the) result must contain all possible outcomes in the concrete domain.

Let's begin with the abstraction of *Net* and *Inv*. *Net* may be abstracted to the identify function

$$\hat{Net} \equiv \lambda \hat{p}.\hat{p}$$

since, as we observed above, $Abs_P(Net\ p) = Abs_P p$. The abstraction of *Inv* is also very simple. There is a symmetry among the members of Δ, which is induced by the *Inv* operator. For example, if $Dir\ p\alpha = \mathbf{u}$, then $Dir(Inv\ p)\alpha = \mathbf{d}$; if $Dir\ p\alpha = \mathbf{dd}^+$, then $Dir(Inv\ p)\alpha = \mathbf{uu}^+$. The converse

of each of these equations is true as well: if $Dir\, p\alpha = \mathbf{d}$, then $Dir(Inv\, p)\alpha = \mathbf{u}$, and so on. By way of example, suppose that $p = \beta^u\alpha^d\beta^d$. Then $\hat{p}\alpha = \{\mathbf{d}\}$ and $\hat{p}\beta = \{\mathbf{u}^+\mathbf{d}^+\}$ where $\hat{p} = Abs_P p$. On the other hand, if $\hat{p'} = Abs_P(Inv p)$, then $\hat{p'}\alpha = \{\mathbf{u}\}$ and $\hat{p'}\beta = \{\mathbf{u}^+\mathbf{d}^+\}$. Let us write $\delta_1 \leftrightarrow \delta_2$ when

$$(Dir\, p\alpha = \delta_1) \Rightarrow (Dir(Inv\, p)\alpha = \delta_2)$$

and

$$(Dir\, q\alpha = \delta_2) \Rightarrow (Dir(Inv\, q)\alpha = \delta_1)$$

for all $p, q \in P$, where $\delta_1, \delta_2 \in \Delta$. It is easy to see that $\mathbf{d} \leftrightarrow \mathbf{u}$, $\mathbf{dd}^+ \leftrightarrow \mathbf{uu}^+$, $\mathbf{u}^+\mathbf{d}^+ \leftrightarrow \mathbf{u}^+\mathbf{d}^+$, and $\epsilon \leftrightarrow \epsilon$. We may therefore abstract the function Inv to the function

$$\hat{Inv} \equiv \lambda\hat{p}.\lambda\alpha.\{\delta_1 \mid \delta_1 \leftrightarrow \delta_2, \delta_2 \in \hat{p}\alpha\}.$$

It follows, by explicit construction, that $Abs_P(Inv\, p) = \hat{Inv}(Abs_P p)$. The abstractions of *Net* and *Inv* are *maximal* in that they satisfy the requirement posed by Equation 1 as strongly as possible (by equality instead of mere inclusion).

There are two more operations over procedure strings that must be abstracted to stack configurations: $+$ and $-$. We begin with the former; the abstraction of the latter is then easily derived. Let us look at concatenation in terms of members of Δ. If $\hat{p}$ is a stack configuration such that $\hat{p}\alpha = \{\mathbf{d}\}$, then it represents procedure strings whose *Net* values contain one term (α^d) that pertains to λ_α. Likewise, if $\hat{q}$ is a stack configuration such that $\hat{q}\alpha = \{\mathbf{u}\}$, then it represents procedure strings whose *Net* values contain only the term α^u that pertains to λ_α. Therefore the *Net* value of the concatenation of two procedure strings $p \in Conc_{\hat{P}}\hat{p}$ and $q \in Conc_{\hat{P}}\hat{q}$, will contain no terms of the form α^d or α^u (because the matching $\alpha^d\alpha^u$ pair within the concatenation will be annihilated by *Net*). In terms of stack configurations, $(\hat{p} \oplus \hat{q})\alpha = \{\epsilon\}$, where $\oplus$ is the abstraction of concatenation we have yet to define.

There is an implicit assumption here that p describes a legitimate evaluation sequence, and that q describes a legitimate sequel to that evaluation sequence. This is the sensible assumption, since the only concatenation we are interested in abstracting is that by which a procedure string is lengthened during computation. Alternatively, we may cast this assumption into the terms of Theorem 9: if we extract the terms pertinent to λ_α from $p+q$, we have $\alpha^d\alpha^u$, which is certainly annihilated by *Net*. Theorem 9 then tells us that when $p + q$ describes an evaluation sequence, that $Net(p + q)$ will contain no α terms, since $Net(Trace(p + q)\alpha)$ is empty. Therefore, of all procedure strings represented by $\hat{p}$ and $\hat{q}$, in the abstraction of concatenation we regard only those pairs p, q such that $p + q$ is extensible to a balanced string.

Let us define the function $Cat : \Delta \to \Delta \to 2^{\Delta}$ according to the following table:

Cat	ϵ	d	dd^+	u	uu^+	u^+d^+
ϵ	$\{\epsilon\}$	$\{d\}$	$\{dd^+\}$	$\{u\}$	$\{uu^+\}$	$\{u^+d^+\}$
d	$\{d\}$	$\{dd^+\}$	$\{dd^+\}$	$\{\epsilon\}$	$\{u, uu^+\}$	$\{d, dd^+, u^+d^+\}$
dd^+	$\{dd^+\}$	$\{dd^+\}$	$\{dd^+\}$	$\{d, dd^+\}$	$\{\epsilon, d, dd^+, u, uu^+\}$	$\{d, dd^+, u^+d^+\}$
u	$\{u\}$	$\{u^+d^+\}$	$\{u^+d^+\}$	$\{uu^+\}$	$\{uu^+\}$	$\{u^+d^+\}$
uu^+	$\{uu^+\}$	$\{u^+d^+\}$	$\{u^+d^+\}$	$\{uu^+\}$	$\{uu^+\}$	$\{u^+d^+\}$
u^+d^+	$\{u^+d^+\}$	$\{u^+d^+\}$	$\{u^+d^+\}$	$\{u, uu^+, u^+d^+\}$	$\{u, uu^+, u^+d^+\}$	$\{u^+d^+\}$

The table should be read as follows. The value of $Cat\delta_1\delta_2$ is found at the intersubsection of the row that is headed by δ_1 and the column that is headed by δ_2. The following theorem describes the meaning of Cat.

Theorem 10 *Let* $\delta_1 = Dir\, p_1\alpha$ *and* $\delta_2 = Dir\, p_2\alpha$, *for* $p_1, p_2 \in P$, $p_1, p_2 \neq \bot_P$. *Then* $Dir(p_1 + p_2)\alpha \in Cat\, \delta_1\delta_2$.

Proof: By enumeration of the possible forms of $Net(Trace\, p_1\alpha)$ and $Net(Trace\, p_2\alpha)$. We give an example of the reasoning, in the case that $Dir\, p_1\alpha = \mathbf{dd}^+$ and $Dir\, p_2\alpha = \mathbf{u}$. In this case, $Net(Trace\, p_1\alpha)$ has the form $\alpha^d\alpha^d\ldots$ (two or more α^d's), and $Net(Trace\, p_2\alpha)$ has the form α^u. Therefore, $p_1 + p_2$ will end in a matching $\alpha^d\alpha^u$ pair; this pair is annihilated by Net, leaving one or more α^d terms in the result. The possible values of $Dir(p_1 + p_2)\alpha$ are therefore summarized as $\{\mathbf{d}, \mathbf{dd}^+\}$. This set is exactly the value of $Cat\, \mathbf{dd}^+\, \mathbf{u}$.

The other values of the function Cat are verified by similar reasoning. □

Given Cat, a natural abstraction $\oplus$ of concatenation of procedure strings is defined by

$$\hat{p_1} \oplus \hat{p_2} = \lambda\alpha. \bigcup \{Cat\, \delta_1\delta_2 \mid \delta_1 \in \hat{p_1}\alpha, \delta_2 \in \hat{p_2}\alpha\}.$$

We must show that this abstraction preserves the meaning of concatenation; that is, that it satisfies Equation 1.

Theorem 11

$$Abs_P(p_1 + p_2) \sqsubseteq_{\hat{P}} (Abs_P p_1) \oplus (Abs_P p_2),$$

for all $p_1, p_2 \in P$.

Proof: Let us first consider the case where at least one of p_1 or p_2 is $\bot_P$. Without loss of generality, assume that $p_1 = \bot_P$. Then $p_1 + p_2 = \bot_P$,

and $Abs_P(p_1 + p_2) = \bot_{\hat{P}}$, for all $\alpha \in \Lambda$. Now suppose that neither of p_1, p_2 is undefined. We showed in Theorem 10 that for every $p_1, p_2 \in P$,

$$Dir(p_1 + p_2)\alpha \in Cat(Dir\ p_1\alpha)(Dir\ p_2\alpha)$$

for all $\alpha \in \Lambda$. Therefore

$$\{Dir(p_1 + p_2)\alpha\} \subseteq Cat(Dir\ p_1\alpha)(Dir\ p_2\alpha) \text{ for all } \alpha \in \Lambda.$$

By the definitions of Abs_P, $\sqsubseteq_{\hat{P}}$ and $\oplus$, this means that

$$Abs_P(p_1 + p_2) \sqsubseteq_{\hat{P}} (Abs_P p_1) \oplus (Abs_P p_2),$$

as desired. □

We may write this result in the form of Equation 2, as

$$p_1 + p_2 \in Conc_{\hat{P}}((Abs_P p_1) \oplus (Abs_P p_2)).$$

This, in turn, implies that $p_1 + p_2 \in Conc_{\hat{P}}(\hat{p_1} \oplus \hat{p_2})$ for all $p_1 \in Conc_{\hat{P}}\hat{p_1}$, $p_2 \in Conc_{\hat{P}}\hat{p_2}$. That is, the concatenation of two procedure strings p_1 and p_2 is contained in (the concretization of) the abstract concatenation (via $\oplus$) of any two stack configurations $\hat{p_1}$ and $\hat{p_2}$ whose concretizations contain p_1 and p_2, respectively (since $Abs_P\, p_1$ and $Abs_P\, p_2$ are the least such $\hat{p_1}$ and $\hat{p_2}$).

We need an abstraction of $-$ to complement the abstraction we have created for $+$. A natural abstraction $\ominus$ is given by

$$\hat{p_1} \ominus \hat{p_2} = \lambda\alpha.\{\delta \mid Cat\ \gamma\delta \cap (\hat{p_1}\alpha) \neq \{\} \text{ for some } \gamma \in (\hat{p_2}\alpha)\}.$$

As an example, if $\hat{p_1}\alpha = \{\mathbf{dd}^+\}$ and $\hat{p_2}\alpha = \{\mathbf{d}\}$ then $(\hat{p_1} \ominus \hat{p_2})\alpha = \{\mathbf{d}, \mathbf{dd}^+\}$. In order to appreciate the loss of information entailed by this abstraction, consider that if $\hat{p_1}\alpha = \{\mathbf{u}^+\mathbf{d}^+\}$, then $(\hat{p_1} \ominus \hat{p_1})\alpha = \Delta$. Written otherwise, we have that $((Abs_P p) \ominus (Abs_P p))\alpha = \Delta$ whereas $Trace(p - p)\alpha = \epsilon$, whenever $(Abs_P p)\alpha = \{\mathbf{u}^+\mathbf{d}^+\}$. The result is the same if $(Abs_P p)\alpha = \{\mathbf{dd}^+\}$. Because the operation $-$ is central to the solutions we detailed in subsection 2.6, this loss of information would be devastating to the effectiveness of the program analysis framework we are constructing, were we to make direct use of $\ominus$. Fortunately, by a simple shift in perspective, we can arrange to extract the information we need without making use of $\ominus$ at all. We will return to this in subsection 2.12.

The following result verifies that $\ominus$ is a sensible abstraction of $-$.

Theorem 12

$$Abs_P(p_1 - p_2) \sqsubseteq_{\hat{P}} (Abs_P p_1) \ominus (Abs_P p_2),$$

for all $p_1, p_2 \in P$.

Proof: In case either p_1 or p_2 is $\perp_P$, both sides of the equation are $\perp_{\hat{P}}$, and the theorem is satisfied trivially. When p_1 and p_2 are fully defined, we have that

$$((Abs_P p_1) \ominus (Abs_P p_2))\alpha$$
$$= \{\delta \mid Cat\gamma\delta \cap (Abs_P p_1)\alpha \neq \{\} \text{ for some } \gamma \in (Abs_P p_2)\alpha\}.$$

By Theorem 10,

$$Dirp_1\alpha = Dir(p_2 + (p_1 - p_2))\alpha \in Cat(Dirp_2\alpha)(Dir(p_1 - p_2)\alpha).$$

This means that

$$(Abs_P p_1)\alpha \subseteq Cat(Dirp_2\alpha)(Dir(p_1 - p_2)\alpha) \text{ for all } \alpha \in \Lambda,$$

$$(Abs_P p_1)\alpha \subseteq Cat((Abs_P p_2)\alpha)((Abs_P(p_1 - p_2))\alpha) \text{ for all } \alpha \in \Lambda,$$

and therefore

$$(Abs_P p_1)\alpha \subseteq \{\delta \mid Cat\gamma\delta \cap ((Abs_P p_1)\alpha) \neq \{\} \text{ for some } \gamma \in (Abs_P p_2)\alpha\}$$

which implies, by the definition of $\ominus$, that

$$Abs_P(p_1 - p_2) \sqsubseteq_{\hat{P}} (Abs_P p_1) \ominus (Abs_P p_2).$$

□

The following result shows that stack configurations capture well the notion of "net" procedure strings.

Theorem 13

$$Abs_P(Net(p_1 + p_2)) \sqsubseteq_{\hat{P}} (Abs_P p_1) \oplus (Abs_P p_2)$$

and

$$Abs_P(Net(p_1 - p_2)) \sqsubseteq_{\hat{P}} (Abs_P p_1) \ominus (Abs_P p_2),$$

for all $p_1, p_2 \in P$.

Proof:

- **I.** By Theorem 11,

  $$Abs_P(p_1 + p_2) \sqsubseteq_{\hat{P}} (Abs_P p_1) \oplus (Abs_P p_2),$$

  and by the definition of Abs_P,

  $$Abs_P(p_1 + p_2) = Abs_P(Net(p_1 + p_2)).$$

  Therefore

  $$Abs_P(Net(p_1 + p_2)) \sqsubseteq_{\hat{P}} (Abs_P p_1) \oplus (Abs_P p_2).$$

- **II.** By Theorem 12,

$$Abs_P(p_1 - p_2) \sqsubseteq_{\hat{P}} (Abs_P p_1) \ominus (Abs_P p_2),$$

and by the definition of Abs_P,

$$Abs_P(p_1 - p_2) = Abs_P(Net(p_1 - p_2)).$$

Therefore

$$Abs_P(Net(p_1 - p_2)) \sqsubseteq_{\hat{P}} (Abs_P p_1) \ominus (Abs_P p_2).$$

□

Theorem 13 is a most useful result, because from it follows immediately that for all $p_1 \in Conc_{\hat{P}} \hat{p_1}$ and $p_2 \in Conc_{\hat{P}} \hat{p_2}$, $Net(p_1 - p_2) \in Conc_{\hat{P}}(\hat{p_1} \ominus \hat{p_2})$ (by the fact that $Abs_P p_1$ and $Abs_P p_2$ are the least $\hat{p_1}$ and $\hat{p_2}$ such that $p_1 \in Conc_{\hat{P}} \hat{p_1}$ and $p_2 \in Conc_{\hat{P}} \hat{p_2}$). The same is true of $Net(p_1 + p_2)$. There are many abstractions of procedure strings for which this result does not hold, since $Net(p_1 + p_2) \neq (Net\, p_1) + (Net\, p_2)$, and $(Net\, p_1) - (Net\, p_2)$ is meaningless, in general.

2.9 Abstract Semantics

We are now ready to abstract the meaning of $\mathcal{L}$, using $\mathcal{E}_2$ as a basis. It must be said at the outset that many abstractions are possible, and are much the easier for our having rid the semantic domains of reflexivity. The conflicting goals of an abstraction are, as always, to minimize information content for efficiency in program analysis, on the one hand, and to maximize information content for efficiency in program execution on the other. To appreciate its practicality, we must see both the dataflow framework an abstraction suggests (to estimate our investment in compile time), and examples of its behavior when applied to real Scheme programs (to estimate the return on our investment, at run time). Furthermore, since interprocedural analysis is but one phase of parallelizing compilation, we cannot judge its effectiveness in isolation from the "active" phases of the compiler, which make use of its results to trigger restructuring and optimization. The abstraction we develop below is but one point on a continuum of choices, and no claims are made for it now, other than that it is correct, representative of the possibilities, and avoids the ridiculous extremes of complete information loss or retention.

The equations of the abstract semantic domains are presented in Figure 12; their abstraction maps are presented in Figure 13; the partial orderings within the abstract domains are defined in 14; and the LUB operators

$$\hat{\Lambda} = 2^{\Lambda}$$
$$\hat{N} = 2^{N}$$
$$\hat{B} = V \to \hat{P}$$
$$\hat{C} = \hat{\Lambda} \times \hat{B}$$
$$\hat{K} = \hat{N} \times \hat{B} \times \hat{P}$$
$$\hat{D} = \hat{C} \times \hat{K} \times \hat{PrimOp} \times \hat{Int} \times \hat{Bool}$$
$$\hat{E} = V \to \hat{D}$$
$$\hat{R} = \Lambda \to \hat{K}$$
$$\hat{T} = \hat{P} \times \hat{B} \times \hat{E} \times \hat{K} \times \hat{R}$$
$$\hat{Q} = \Lambda \to \hat{T}$$

Figure 12: Abstract Domains for $\mathcal{E}_3$

within the abstract domains are defined in 15. In the case of the compound domains $\hat{B}$, $\hat{C}$, $\hat{K}$, $\hat{D}$, $\hat{E}$, $\hat{R}$, $\hat{T}$ and $\hat{Q}$ these functions are straightforward compositions of the corresponding functions over the primitive domains, as described in subsection 2.3. The partial orderings and LUB operators are defined as curried functions, although we will write them, in the conventional way, as infix operators. The notation $a \wedge b$ denotes the logical conjunction of a and b.

A member $\hat{\alpha} \in \hat{\Lambda}$ is simply a subset of Λ. It is important to make clear the distinction between the *representation* (in this case a subset of Λ) from the thing that is *represented* (in this case an ideal of Λ). Here, the relationship is simple, and is given by

$$Conc_{\hat{\Lambda}} \equiv \lambda\hat{\alpha}.\hat{\alpha} \cup \{\bot_{\Lambda}\}.$$

This abstraction is *maximal* in that every ideal of Λ is represented in $\hat{\Lambda}$. Most of our abstractions are more abstract than this one.

A member $\hat{b} \in \hat{B}$ maps a variable to an abstraction of its birth date. There is no information loss, during abstraction, in the domain[19] of the function: when considered as a subset of the product $V \times \hat{P}$, the cardinality of the map is the same, before and after abstraction. However, there is information loss in the range of the function, as the abstraction of procedure strings involves a loss of information, as described in subsection 2.7.

An abstract environment $\hat{e} \in \hat{E}$ maps a variable to an abstraction of

[19]Here we are using *domain* as the space from which a function takes its input values, and not the space containing the function.

$$
\begin{aligned}
&Abs_{\Lambda} \equiv \lambda\alpha.\ \text{if } \alpha = \bot_{\Lambda} \text{ then } \{\} \text{ else } \{\alpha\}\\
&Abs_{N} \equiv \lambda i.\ \text{if } i = \bot_{N} \text{ then } \{\} \text{ else } \{i\}\\
&Abs_{B} \equiv \lambda b.\lambda v.Abs_{P}(bv)\\
&Abs_{C} \equiv \lambda\langle\alpha, b\rangle.\langle Abs_{\Lambda}\alpha, Abs_{B}b\rangle\\
&Abs_{K} \equiv \lambda\langle i, b, p, o\rangle.\langle Abs_{N}i, Abs_{B}b, Abs_{P}p\rangle\\
&Abs_{D} \equiv \lambda x.\ \text{if } x = \bot_{D} \text{ then } \langle \bot_{\hat{C}}, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in C \text{ then } \langle Abs_{C}x, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in K \text{ then } \langle \bot_{\hat{C}}, Abs_{K}x, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in PrimOp \text{ then } \langle \bot_{\hat{C}}, \bot_{\hat{K}}, Abs_{PrimOp}x, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in Int \text{ then } \langle \bot_{\hat{C}}, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, Abs_{Int}x, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in Bool \text{ then } \langle \bot_{\hat{C}}, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, Abs_{Bool}x\rangle\\
&Abs_{E} \equiv \lambda e.\lambda v.\ \sqcup_{\hat{D}} \{Abs_{D}(e\langle v, p\rangle) \mid p \in P\}\\
&Abs_{R} \equiv \lambda r.\lambda\alpha.\ \sqcup_{\hat{K}} \{Abs_{K}(r(p + \alpha^{d})) \mid p \in P\}\\
&Abs_{Q} \equiv \lambda\langle i, b, p, e, k, o, r\rangle.\lambda\alpha.\ \text{if } \alpha \neq Container\ i\\
&\qquad\qquad \text{then } \langle \bot_{\hat{P}}, \bot_{\hat{B}}, \bot_{\hat{E}}, \bot_{\hat{K}}, \bot_{\hat{R}}\rangle\\
&\qquad\qquad \text{else } \langle Abs_{P}p, Abs_{B}b, Abs_{E}e,\\
&\qquad\qquad\qquad Abs_{K}k, Abs_{R}r\rangle
\end{aligned}
$$

Figure 13: Abstraction Maps

$$\sqsubseteq_{\hat{N}} \equiv \lambda\hat{i}.\lambda\hat{j}.\hat{i} \subseteq \hat{j}$$
$$\sqsubseteq_{\hat{\Lambda}} \equiv \lambda\hat{\alpha}.\lambda\hat{\beta}.\hat{\alpha} \subseteq \hat{\beta}$$
$$\sqsubseteq_{\hat{B}} \equiv \lambda\hat{b_1}.\lambda\hat{b_2}.(\hat{b_1}v) \subseteq (\hat{b_2}v)\ \forall v \in V$$
$$\sqsubseteq_{\hat{C}} \equiv \lambda\langle\hat{\alpha_1},\hat{b_1}\rangle.\lambda\langle\hat{\alpha_2},\hat{b_2}\rangle.(\hat{\alpha_1} \sqsubseteq_{\hat{\Lambda}} \hat{\alpha_2}) \wedge (\hat{b_1} \sqsubseteq_{\hat{B}} \hat{b_2})$$
$$\begin{aligned}\sqsubseteq_{\hat{K}} \equiv \lambda\langle\hat{i_1},\hat{b_1},\hat{p_1}\rangle.\lambda\langle\hat{i_2},\hat{b_2},\hat{p_2}\rangle.\ & (\hat{i_1} \sqsubseteq_{\hat{N}} \hat{i_2}) \\ & \wedge (\hat{b_1} \sqsubseteq_{\hat{B}} \hat{b_2}) \\ & \wedge (\hat{p_1} \sqsubseteq_{\hat{P}} \hat{p_2})\end{aligned}$$
$$\begin{aligned}\sqsubseteq_{\hat{D}} \equiv \lambda\langle\hat{c_1},\hat{k_1},\hat{f_1},\hat{z_1},\hat{x_1}\rangle.\lambda\langle\hat{c_2},\hat{k_2},\hat{f_2},\hat{z_2},\hat{x_2}\rangle.\ & (\hat{c_1} \sqsubseteq_{\hat{C}} \hat{c_2}) \\ & \wedge (\hat{k_1} \sqsubseteq_{\hat{K}} \hat{k_2}) \\ & \wedge (\hat{f_1} \sqsubseteq_{\hat{PrimOp}} \hat{f_2}) \\ & \wedge (\hat{z_1} \sqsubseteq_{\hat{Int}} \hat{z_2}) \\ & \wedge (\hat{x_1} \sqsubseteq_{\hat{Bool}} \hat{x_2})\end{aligned}$$
$$\sqsubseteq_{\hat{E}} \equiv \lambda\hat{e_1}.\lambda\hat{e_2}.(\hat{e_1}v) \sqsubseteq_{\hat{D}} (\hat{e_2}v)\ \forall v \in V$$
$$\sqsubseteq_{\hat{R}} \equiv \lambda\hat{r_1}.\lambda\hat{r_2}.(\hat{r_1}\alpha) \sqsubseteq_{\hat{K}} (\hat{r_2}\alpha)\ \forall \alpha \in \Lambda$$
$$\begin{aligned}\sqsubseteq_{\hat{T}} \equiv \lambda\langle\hat{b_1},\hat{p_1},\hat{e_1},\hat{k_1},\hat{r_1}\rangle.\lambda\langle\hat{b_2},\hat{p_2},\hat{e_2},\hat{k_2},\hat{r_2}\rangle.\ & (\hat{b_1} \sqsubseteq_{\hat{B}} \hat{b_2}) \\ & \wedge (\hat{p_1} \sqsubseteq_{\hat{B}} \hat{b_2}) \\ & \wedge (\hat{e_1} \sqsubseteq_{\hat{E}} \hat{e_2}) \\ & \wedge (\hat{k_1} \sqsubseteq_{\hat{K}} \hat{k_2}) \\ & \wedge (\hat{r_1} \sqsubseteq_{\hat{R}} \hat{r_2})\end{aligned}$$
$$\sqsubseteq_{\hat{Q}} \equiv \lambda\hat{q_1}.\lambda\hat{q_2}.\lambda\alpha.(\hat{q_1}\alpha) \sqsubseteq_{\hat{T}} (\hat{q_2}\alpha)\ \forall \alpha \in \Lambda$$

Figure 14: Partial Orderings

$$\sqcup_{\hat{N}} \equiv \lambda \hat{n_1}.\lambda \hat{n_2}.\hat{n_1} \cup \hat{n_2}$$
$$\sqcup_{\hat{\Lambda}} \equiv \lambda \hat{\alpha}.\lambda \hat{\beta}.\hat{\alpha} \cup \hat{\beta}$$
$$\sqcup_{\hat{B}} \equiv \lambda \hat{b_1}.\lambda \hat{b_2}.\lambda v.(\hat{b_1} v) \sqcup_{\hat{P}} (\hat{b_2} v)$$
$$\sqcup_{\hat{C}} \equiv \lambda \langle \hat{\alpha_1}, \hat{b_1} \rangle.\lambda \langle \hat{\alpha_2}, \hat{b_2} \rangle.\langle \hat{\alpha_1} \sqcup_{\hat{\Lambda}} \hat{\alpha_2}, \hat{b_1} \sqcup_{\hat{B}} \hat{b_2} \rangle$$
$$\sqcup_{\hat{K}} \equiv \lambda \langle \hat{i_1}, \hat{b_1}, \hat{p_1} \rangle.\lambda \langle \hat{i_2}, \hat{b_2}, \hat{p_2} \rangle.\langle\ \hat{i_1} \sqcup_{\hat{N}} \hat{i_2},\ \hat{b_1} \sqcup_{\hat{B}} \hat{b_2},\ \hat{p_1} \sqcup_{\hat{P}} \hat{p_2} \rangle$$
$$\sqcup_{\hat{D}} \equiv \lambda \langle \hat{c_1}, \hat{k_1}, \hat{f_1}, \hat{z_1}, \hat{x_1} \rangle.\lambda \langle \hat{c_2}, \hat{k_2}, \hat{f_2}, \hat{z_2}, \hat{x_2} \rangle.\langle\ \hat{c_1} \sqcup_{\hat{C}} \hat{c_2},\ \hat{k_1} \sqcup_{\hat{B}} \hat{k_2},\ \hat{f_1} \sqcup_{\hat{B}} \hat{f_2},\ \hat{z_1} \sqcup_{\hat{B}} \hat{z_2},\ \hat{x_1} \sqcup_{\hat{B}} \hat{x_2} \rangle$$
$$\sqcup_{\hat{E}} \equiv \lambda \hat{e_1}.\lambda \hat{e_2}.\lambda v.(\hat{e_1} v) \sqcup_{\hat{D}} (\hat{e_2} v)$$
$$\sqcup_{\hat{R}} \equiv \lambda \hat{r_1}.\lambda \hat{r_2}.\lambda \alpha.(\hat{r_1} \alpha) \sqcup_{\hat{K}} (\hat{r_2} \alpha)$$
$$\sqcup_{\hat{T}} \equiv \lambda \langle \hat{p_1}, \hat{b_1}, \hat{e_1}, \hat{k_1}, \hat{r_1} \rangle.\lambda \langle \hat{p_2}, \hat{b_2}, \hat{e_2}, \hat{k_2}, \hat{r_2} \rangle.\langle\ \hat{p_1} \sqcup_{\hat{C}} \hat{p_2},\ \hat{b_1} \sqcup_{\hat{B}} \hat{b_2},\ \hat{e_1} \sqcup_{\hat{B}} \hat{e_2},\ \hat{k_1} \sqcup_{\hat{B}} \hat{k_2},\ \hat{r_1} \sqcup_{\hat{B}} \hat{r_2} \rangle$$
$$\sqcup_{\hat{Q}} \equiv \lambda \hat{q_1}.\lambda \hat{q_2}.\lambda \alpha.(\hat{q_1} \alpha) \sqcup_{\hat{T}} (\hat{q_2} \alpha)$$

Figure 15: LUB Operators Over the Abstract Domains

the values that may be assumed by the instances of that variable. Here, there is considerable loss of information during abstraction, in the domain of the function; all instances of a particular variable v, represented in the concrete semantics as pairs $\langle v, p \rangle$ where $p \in P$, are collapsed onto the single member v of the domain of the abstract function. Whatever loss of information occurs in abstracting members of D is compounded by the fact that the values of all instances of a variable, over all executions of the program, are coalesced into a single abstract value in the range of an abstract environment. The values assumed by a variable are differentiated only in that there is a separate environment for each lambda expression of the program, and the variable may assume different (abstract) values in each environment.

The product domains $\hat{C}$ and $\hat{K}$ are straightforward, component-wise abstractions of the corresponding concrete domains. (We explain below why members of $\hat{K}$ have only one component in $\hat{P}$, in contrast to those in K, which have two components in P.)

The form of a member $\hat{d} \in \hat{D}$ may be surprising. It it essentially the same as the representation given to a sum of domains in [43]. The idea is simple: $\hat{d}$ represents a set of values which may be drawn from the domains C, K, *PrimOp*, *Int*, and *Bool*. Each component of $\hat{d}$ represents a set of values from one of these domains. If $\hat{d}$ represents, for example, no integer values (or only the undefined integer), then its fourth component will be $\perp_{\hat{Int}}$. (See the discussion in subsection 2.4 for our interpretation of bottom elements.)

An (abstract) restoration map $\hat{r} \in \hat{R}$ is a function from a lambda expression index to an (abstract) continuation. The domain of this function looks odd, since the corresponding concrete maps have type $P \to K$. In abstracting $r \in R$ to form $\hat{r} \in \hat{R}$, every procedure string $p = \cdots \alpha^d$ in the domain of the function has been collapsed to the lambda expression index α. All procedure strings over which r is defined have the form $s + \alpha^d$ for some $\alpha \in \Lambda$ and $s \in P$, since r maps procedure instance birth dates to their continuations, and by the definition of $\mathcal{S}_2$ all such birth dates have the form $\cdots \alpha^d$ (where λ_α is the procedure being applied). The advantage of this abstraction is that we gather, into a single value in the range of $\hat{r}$, the continuations of all instances of a single procedure. Since each continuation contains (an abstraction of) the statement index of the procedure application or `call/cc` expression which creates it, we will be able, after the analysis is complete, to construct an approximation to the calling graph of the program, using members of $\hat{R}$. That is, the abstract continuation of each lambda expression will point us to the locations within the program at which the lambda expression may be applied.

An abstract state $\hat{q} \in \hat{Q}$ is a map from a lambda expression index, to a tuple of information, a member of the abstract domain $\hat{T}$. We should understand $\hat{q}$ as gathering all states in which a lambda expression λ_α is active, into a single value in the range of $\hat{q}$ (the value of $\hat{q}$ at α). There are several dimensions of information loss in this abstraction. First, the statement index of each of the gathered states is lost, so that one cannot distinguish the state after statement i from that after statement j, if S_i and S_j belong to the same lambda expression. This makes our abstraction *flow insensitive*, in the language of data flow analysis, since the structure of control flow within each lambda expression is ignored (that is, the control flow is approximated by assuming that the statements of the lambda expression can occur in any order whatsoever). There is a second dimension of information loss, in the abstraction of states, in that all *instances* of a single statement will be collapsed into a single value in the range of $\hat{q}$: during execution of a program, there may be thousands of states in which a particular statement is active; we will summarize these states (along with all states in which other statements within the same lambda expression are executed) in a single value in the range of $\hat{q}$.

We remarked in subsection 2.7 that under our abstraction of procedure strings, p and o are equivalent if they have the same net value. By Theorem 4, the procedure string of a state, and the birth date of the procedure instance that is active in that state have the same net value. It is for this reason that members of $\hat{T}$ contain only one component in $\hat{P}$ (that is, if they had two such components, the two would be identical).

An abstract semantics for $\mathcal{L}$, based upon these domain equations, abstraction maps, partial orderings, and LUB operators is given in Figures 16, 17 and 18. Recall from subsection 2.3 that $f[x//y]$ denotes the function $f[x \sqcup (fy)/y]$.

In broad outline, $\mathcal{E}_3$ works as follows. We begin with an abstract state $\hat{q}$ (initially, an approximation to the set of states from which execution may commence). $\hat{q}$ is a map from lambda expression indices to members of $\hat{T}$. The tuple $\hat{q}\alpha$ in $\hat{T}$ approximates all states in which α is the active procedure. For each statement $i \in N$ of the program, we apply $\mathcal{S}_3$ to i and to $\hat{q}\alpha$, where λ_α is the procedure that contains statement i. The least upper bound of the set of abstract states that results from these applications is joined with $\hat{q}$, and the result becomes the next abstract state in the sequence that is described by abstract interpretation of the program. (The function $\mathcal{S}_3^*$ collects and joins the applications of $\mathcal{S}_3$). The process ends when a fixed point is reached; that is, when further applications of $\mathcal{E}_3$ result in no change in the abstract state.

Let's visit the definition of $\mathcal{S}_3$ in more detail. It maps a statement number

Let $\hat{t} = \langle \hat{p}, \hat{b}, \hat{e}, \hat{k}, \hat{r} \rangle, i \in N$. Then $\mathcal{S}_3 : N \to \hat{T} \to \hat{Q}$ is defined, according to the form of S_i, as follows.

$$
\begin{array}{l}
S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_m\texttt{))}]\!] \text{ or } S_i = [\![\texttt{(set! x (call/cc f))}]\!] \Rightarrow \\
\quad \mathcal{S}_3 i \hat{t} = \hat{q}_c \sqcup_{\hat{Q}} \hat{q}_k \\
\quad \text{where } \hat{q}_c = \sqcup_{\hat{Q}} \Big\{ \lambda\beta.\ \text{if } \beta \neq \alpha \\
\qquad\qquad\qquad \text{then } \perp_{\hat{T}} \\
\qquad\qquad\qquad \text{else } \langle \hat{p}', \hat{b}'[\hat{p}'//[\![\texttt{z}_1]\!]] \cdots [\hat{p}'//[\![\texttt{z}_n]\!]], \hat{e}', \\
\qquad\qquad\qquad\qquad \langle \{i\}, \hat{b}, \hat{p} \rangle, \hat{r}[\hat{k}//\mathit{Container}\ i] \rangle \\
\qquad\qquad \text{where } \lambda_\alpha = [\![\texttt{(lambda (z}_1 \cdots \texttt{z}_m\texttt{) <z}_{m+1} \cdots \texttt{z}_n\texttt{> } S_j \cdots\texttt{)}]\!], \\
\qquad\qquad\qquad \hat{p}' = \hat{p} \oplus \mathit{Abs}_P(\alpha^d), \\
\qquad\qquad \text{and} \quad \hat{e}' = \text{if } S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_m\texttt{))}]\!] \\
\qquad\qquad\qquad\qquad \text{then } \hat{e}[\hat{e}[\![\texttt{y}_1]\!]//[\![\texttt{z}_1]\!]] \cdots [\hat{e}[\![\texttt{y}_m]\!]//[\![\texttt{z}_m]\!]] \\
\qquad\qquad\qquad\qquad \text{else } \hat{e}[\langle \perp_{\hat{C}}, \langle \{i\}, \hat{b}, \hat{p} \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad \perp_{\hat{PrimOp}}, \perp_{\hat{Int}}, \perp_{\hat{Bool}} \rangle // [\![\texttt{z}_1]\!]] \\
\qquad\qquad\qquad \Big| \ \alpha \in \hat{\alpha} \Big\} \\
\qquad \text{where } \hat{e}[\![\texttt{f}]\!] = \langle \hat{c}', \hat{k}', \ldots \rangle \\
\qquad \text{and} \quad \hat{c}' = \langle \hat{\alpha}, \hat{b}' \rangle \\
\quad \text{and} \quad \hat{q}_k = \sqcup_{\hat{Q}} \Big\{ \lambda\beta.\ \text{if } \beta \neq \mathit{Container}\ j \\
\qquad\qquad\qquad \text{then } \perp_{\hat{T}} \\
\qquad\qquad\qquad \text{else } \langle \hat{p}', \hat{b}', \hat{e}', \hat{r}\beta, \hat{r}[\hat{k}//\mathit{Container}\ i] \rangle \\
\qquad\qquad\qquad\qquad \text{where } \hat{e}' = \text{if } S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_m\texttt{))}]\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{then } \hat{e}[\hat{e}[\![\texttt{y}_1]\!]//[\![\texttt{z}]\!]] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{else } \hat{e}[\langle \perp_{\hat{C}}, \langle \{i\}, \hat{b}, \hat{p} \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \perp_{\hat{PrimOp}}, \perp_{\hat{Int}}, \perp_{\hat{Bool}} \rangle // [\![\texttt{z}]\!]] \\
\qquad\qquad\qquad \text{where } S_j = [\![\texttt{(set! z (call/cc g))}]\!] \\
\qquad\qquad\qquad \Big| \ j \in \hat{j} \Big\} \\
\qquad \text{where } \hat{e}[\![\texttt{f}]\!] = \langle \hat{c}', \hat{k}', \ldots \rangle \\
\qquad \text{and} \quad \hat{k}' = \langle \hat{j}, \hat{b}', \hat{p}' \rangle
\end{array}
$$

Figure 16: The Semantic Function $\mathcal{S}_3$ (Part I)

$$
\begin{aligned}
S_i = [\![&\texttt{(set! f (lambda}_\alpha\ \texttt{(x}_1 \ldots \texttt{x}_m\texttt{)}\ \texttt{<x}_{m+1} \cdots \texttt{x}_n\texttt{>}\ \cdots\texttt{))}]\!] \Rightarrow \\
&\mathcal{S}_3 i\hat{t} = \lambda\beta.\ \text{if } \mathit{Container}\ i \neq \beta \\
&\qquad\qquad \text{then } \bot_{\hat{T}} \\
&\qquad\qquad \text{else } \langle \hat{p}, \\
&\qquad\qquad\qquad \hat{b}, \\
&\qquad\qquad\qquad \hat{e}[\langle\langle\{\alpha\}, \hat{b}\rangle, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle // [\![\texttt{f}]\!]], \\
&\qquad\qquad\qquad \hat{k}, \\
&\qquad\qquad\qquad \hat{r}\rangle
\end{aligned}
$$

$$
\begin{aligned}
S_i = [\![&\texttt{(if x (goto m) (goto n))}]\!] \Rightarrow \\
&\mathcal{S}_3 i\hat{t} = \lambda\beta.\ \text{if } \mathit{Container}\ i \neq \beta \\
&\qquad\qquad \text{then } \bot_{\hat{T}} \\
&\qquad\qquad \text{else } \hat{t}
\end{aligned}
$$

$$
\begin{aligned}
S_i = [\![&\texttt{(return x)}]\!] \Rightarrow \\
&\mathcal{S}_3 i\hat{t} = \sqcup_{\hat{Q}} \Big\{ \lambda\beta.\ \text{if } \mathit{Container}\ j \neq \beta \\
&\qquad\qquad\qquad \text{then } \bot_{\hat{T}} \\
&\qquad\qquad\qquad \text{else } \langle \hat{p'}, \hat{b'}, \hat{e}[\hat{e}[\![\texttt{x}]\!] // [\![\texttt{y}]\!]], \hat{r}(\mathit{Container}\ j), \hat{r}\rangle \\
&\qquad\qquad \text{where } S_j = [\![\texttt{(set! y } \cdots\texttt{)}]\!] \\
&\qquad\qquad \Big|\ j \in \hat{j} \Big\} \\
&\text{where } \hat{k} = \langle \hat{j}, \hat{b'}, \hat{p'} \rangle
\end{aligned}
$$

Figure 17: The Semantic Function $\mathcal{S}_3$ (Part II)

$$\mathcal{S}_3^* : \hat{Q} \to \hat{Q} \equiv \lambda\hat{q}.\ \sqcup_{\hat{Q}} \{\mathcal{S}_3 i(\hat{q}(\mathit{Container}\ i)) \mid i \in N\}$$

$$
\begin{aligned}
\mathcal{E}_3 : Q \to Q \equiv \lambda\hat{q}.\ &\text{Let } \hat{q'} = \mathcal{S}_3^* \hat{q} \\
&\text{in} \quad \text{if } \hat{q'} \sqsubseteq_{\hat{Q}} \hat{q} \text{ then } \hat{q} \text{ else } \mathcal{E}_3(\hat{q} \sqcup_{\hat{Q}} \hat{q'})
\end{aligned}
$$

Figure 18: The Semantic Functions $\mathcal{S}_3^*$ and $\mathcal{E}_3$

i and a tuple $\hat{t} = \langle \hat{p}, \hat{b}, \hat{e}, \hat{k}, \hat{r} \rangle \in \hat{T}$ to an abstract state (a member of $\hat{Q}$). The most interesting case is when S_i (statement i) is a procedure application or an invocation of `call/cc`. The definition of $\mathcal{S}_3$ in this case is shown in Figure 16. The definition looks complex, but it is really quite simply derived from $\mathcal{S}_2$. The variable `f` contains (an abstraction of) the object to be applied (in the case that S_i is a `call/cc` expression, to be applied to the current continuation); this abstract object is retrieved by writing $\hat{e}[\![\texttt{f}]\!]$. Now, $\hat{e}[\![\texttt{f}]\!]$ represents a set of values (an ideal in D), but there are only two kinds of objects that can be applied by a correct program: closures and continuations.[20] Therefore the final state that results from this single step of evaluation (that is, the state $\mathcal{S}_3 i \hat{t}$) is formed by joining the abstract state that results from applying all of the closures represented by $\hat{e}[\![\texttt{f}]\!]$, with the abstract state that results from applying all of the continuations represented by $\hat{e}[\![\texttt{f}]\!]$. These abstract states are called $\hat{q}_c$ and $\hat{q}_k$ respectively. Let's consider their values in turn.

The abstraction of the closures represented by (contained in the concretization of) $\hat{e}[\![\texttt{f}]\!]$ is $\hat{c} = \langle \hat{\alpha}, \hat{b'} \rangle$. The value $\hat{\alpha}$ is the set of indices of those lambda expressions, closures of which are possible values of `f`. The abstract state $\hat{q}_c$ is therefore the LUB of the abstract states that result from the application of (closures of) each of these lambda expressions. Likewise, the abstraction of the continuations represented by $\hat{e}[\![\texttt{f}]\!]$ is $\hat{k} = \langle \hat{j}, \hat{b'}, \hat{p'} \rangle$. The value $\hat{j}$ is the set of those statement indices (of `call/cc` expressions) continuations of which are possible values of `f`. The abstract state $\hat{q}_k$ is therefore the LUB of the abstract states that result from the application of each of these continuations.

The reader may have noticed that there seems to be nothing analogous to the expression $p + Inv(p - p')$ that appeared throughout the definition of $\mathcal{S}_2$. We would expect to find (an abstraction of) such an expression in the cases of continuation application and procedure return, within $\mathcal{S}_2$. Instead, the stack configuration of the abstract state that results from applying an abstract continuation is simply $\hat{p}$, where $\hat{p}$ is the abstraction of the birth date of the procedure instance in which the continuation was formed. That is, the abstract birth date of the current procedure instance in a state that results from application of a continuation, and the abstract procedure string of that state, are the same. Again, the reason is Theorem 4, which shows that $Net\, p = Net\, o$ whenever p is a procedure string of a state, and o is the birth date of the procedure instance that is active in that state. By that

[20] See the discussion of bottom values in subsection 2.4. Even if we were to simulate the application of, say, an integer, we would simply be joining the bottom element of the domain $\hat{Q}$ of abstract states, with the abstract state that results from the application of legitimate (applicable) values. This would have no effect on the outcome, since $\perp \sqcup x = x$.

theorem,

$$Abs_P o' = Abs_P p' = Abs_P(p + Inv(p - p')),$$

where p' is the procedure string of the state in which the continuation is formed, o' is the birth date of the procedure instance within which it is formed, and p is the procedure string of the state in which it is applied.

Two things are to be shown, concerning this abstract semantics: first, that all evaluations terminate; second, that the meaning of the concrete semantics ($\mathcal{E}_2$) is preserved, such that the final state of abstract evaluation approximates not merely the last state(s) of concrete evaluation, but *every* state that occurs during execution. The first of these results is needed if we are to write a compiler that is guaranteed to terminate when analyzing a (possibly erroneous) program; the second allows us to regard the result of abstract interpretation as representative of run-time behavior, and suggests the derivation of a data flow analysis framework from the semantics.

Theorem 14 *$\mathcal{S}_3 i$ is monotonic for all $i \in N$.*

Proof: The following facts are obvious:

1. if $a_1 \sqsubseteq b_1, a_2 \sqsubseteq b_2, \cdots, a_n \sqsubseteq b_n$, then $\sqcup\{a_i\} \sqsubseteq \sqcup\{b_i\}$.
2. if $a_1 \sqsubseteq b_1, a_2 \sqsubseteq b_2, \cdots, a_n \sqsubseteq b_n$, then $\langle a_1, a_2, \ldots, a_n\rangle \sqsubseteq \langle b_1, b_2, \ldots, b_n\rangle$.
3. if $f \sqsubseteq g$ and $x \sqsubseteq y$ then $f[x//z] \sqsubseteq g[y//z]$.
4. if $a_1 \sqsubseteq b_1, a_2 \sqsubseteq b_2, \cdots, a_n \sqsubseteq b_n$, then $\{c_1 \mapsto a_1, c_2 \mapsto a_2, \ldots, c_n \mapsto a_n\} \sqsubseteq \{c_1 \mapsto b_1, c_2 \mapsto b_2, \ldots, c_n \mapsto b_n\}$

The theorem follows by decomposition of the definition of $\mathcal{S}_3$ into monotonic functions over primitive types, according to these four facts. For example, suppose that

$$\hat{t_1} = \langle \hat{p_1}, \hat{b_1}, \hat{e_1}, \hat{k_1}, \hat{r_1}\rangle,$$

$$\hat{t_2} = \langle \hat{p_2}, \hat{b_2}, \hat{e_2}, \hat{k_2}, \hat{r_2}\rangle,$$

$\hat{t_1} \sqsubseteq \hat{t_2}$, and that

$$S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_n\texttt{))}]\!] \text{ or } [\![\texttt{(set! x (call/cc f))}]\!].$$

Let $\mathcal{S}_3 i\hat{t_1} = \hat{q_c'} \sqcup_{\hat{Q}} \hat{q_k'}$, and $\mathcal{S}_3 i\hat{t_2} = \hat{q_c''} \sqcup_{\hat{Q}} \hat{q_k''}$. Then, by Fact 1 above, to show that $\mathcal{S}_3 i$ is monotonic it is sufficient to show that $\hat{q_c'} \sqsubseteq_{\hat{Q}} \hat{q_c''}$ and $\hat{q_k'} \sqsubseteq_{\hat{Q}} \hat{q_k''}$. (We have performed one step of decomposition.) Let us first show that

$\hat{q}'_c \sqsubseteq_{\hat{Q}} \hat{q}''_c$. Following the definition of $\mathcal{S}_3$ in this case, let $\hat{q}'_c = \sqcup_{\hat{Q}} F$ and $\hat{q}''_c = \sqcup_{\hat{Q}} G$. (The members of F and G are functions in $\hat{Q}$.) Again, by Fact 1 above it suffices to show that

$$\forall g \in G, \exists f \in F \mid f \sqsubseteq_{\hat{Q}} g.$$

(This application of Fact 1 is a second step of decomposition.) By the definition of $\mathcal{S}_3$, there is one $f \in F$ for each $\alpha_1 \in \hat{\alpha_1}$, where $\hat{e_1}[\![\mathtt{f}]\!] = \langle\langle \hat{\alpha_1}, \hat{b'_1}\rangle \ldots\rangle$, and there is one $g \in G$ for each $\alpha_2 \in \hat{\alpha_2}$, where $\hat{e_2}[\![\mathtt{f}]\!] = \langle\langle \hat{\alpha_2}, \hat{b'_2}\rangle \ldots\rangle$. Since $\hat{e_1} \sqsubseteq_{\hat{E}} \hat{e_2}$, it follows from the definition of $\sqsubseteq_{\hat{\Lambda}}$ that $\hat{\alpha_1} \subseteq \hat{\alpha_2}$. Thus, for each $g \in G$ there is a function $f \in F$ such that if $g\alpha \neq \perp_{\hat{T}}$, then $f\alpha \neq \perp_{\hat{T}}$ (and there is at most one α such that $g\alpha \neq \perp_{\hat{T}}$ or $f\alpha \neq \perp_{\hat{T}}$). We apply Fact 4 to f and g, which requires that we show that $f\alpha \sqsubseteq_{\hat{T}} g\alpha$ (and only this, since α is the only point at which f and g may differ in value). We continue decomposing the tuples $f\alpha$ and $g\alpha$ in this way, applying Facts 2 and 3 at the next steps. The decomposition terminates in operations over primitive types, because the domains over which $\mathcal{E}_3$ is defined are not reflexive. It is easily verified that these operations (such as $\oplus$ over stack configurations) are monotonic.

The same reasoning applies to the values $\hat{q'_k}$ and $\hat{q''_k}$, and to the other forms of S_i as well. □

Theorem 15 *$\mathcal{S}_3^*$ is monotonic.*

Proof: Let $\hat{q} \sqsubseteq_{\hat{Q}} \hat{r}$. Then $\forall i \in N$,

$$\mathcal{S}_3 i(\hat{q}(\mathit{Container}\ i)) \sqsubseteq_{\hat{Q}} \mathcal{S}_3 i(\hat{r}(\mathit{Container}\ i))$$

by Theorem 14. Therefore

$$\sqcup_{\hat{Q}}\{\mathcal{S}_3 i(\hat{q}(\mathit{Container}\ i)) \mid i \in N\} \sqsubseteq_{\hat{Q}} \sqcup_{\hat{Q}}\{\mathcal{S}_3 i(\hat{r}(\mathit{Container}\ i)) \mid i \in N\},$$

and

$$\mathcal{S}_3^* \hat{q} \sqsubseteq_{\hat{Q}} \mathcal{S}_3^* \hat{r}.$$

□

Theorem 16 *$\mathcal{E}_3 \hat{q}$ terminates, for all $\hat{q} \in \hat{Q}$.*

Proof: Suppose not. $\mathcal{E}_3$ is the only recursively defined function in the abstract semantics. Therefore $\mathcal{E}_3 \hat{q}$ describes an infinite sequence of abstract states (the arguments to $\mathcal{E}_3$ in successive recursive calls), call it $\hat{q_0}, \hat{q_1}, \ldots$. By the definition of $\mathcal{E}_3$, $q_0 \sqsubseteq_{\hat{Q}} q_1 \sqsubseteq_{\hat{Q}} \cdots$, and since evaluation terminates if $q_{i+1} \sqsubseteq_{\hat{Q}} \hat{q_i}$, we have $\hat{q_0} \sqsubset_{\hat{Q}} \hat{q_1} \sqsubset_{\hat{Q}} \hat{q_2} \cdots$. But all ascending chains in $\hat{Q}$ have finite length, a contradiction. □

Theorem 17 *$\mathcal{E}_3$ is monotonic.*

Proof: Let $\hat{r_0} \sqsubseteq_{\hat{Q}} \hat{q_0}$, and let $\hat{r_0} \sqsubseteq_{\hat{Q}} \hat{r_1} \sqsubseteq_{\hat{Q}} \ldots$ and $\hat{q_0} \sqsubseteq_{\hat{Q}} \hat{q_1} \sqsubseteq_{\hat{Q}} \ldots$ be the evaluation sequences described by $\mathcal{E}_3\hat{r_0}$ and $\mathcal{E}_3\hat{q_0}$ respectively. By Theorem 16 each of these sequences is finite. Let the shorter be extended, by replication of its final term, so that both have length n. By Theorem 15 and the definition of $\mathcal{E}_3$,

$$\hat{r_i} \sqsubseteq_{\hat{Q}} \hat{q_i}$$

for all $0 \leq i \leq n$. Therefore

$$\hat{r_n} = \mathcal{E}_3\hat{r_0} \sqsubseteq_{\hat{Q}} \mathcal{E}_3\hat{q_0} = \hat{q_n}.$$

□

Evaluation under $\mathcal{E}_3$ describes an ascending chain of abstract states, and since in each of our abstract domains ascending chains have finite length, $\mathcal{E}_3$ terminates, even when a corresponding evaluation under $\mathcal{S}_2$ will not. This is an essential property of an abstract semantics, if it is to become the basis of an algorithm for static analysis.

It is well that $\mathcal{E}_3$ terminates, but we would feel better knowing what it returns, when it does so. The following two theorems define the sense in which $\mathcal{E}_3$ preserves the meaning of evaluation under $\mathcal{E}_2$.

Theorem 18 *If $q \in Conc_{\hat{Q}}\hat{q}$ then*

$$\mathcal{S}_2 q \in Conc_{\hat{Q}}(\mathcal{S}_3 i(\hat{q}(Container\ i)))$$

where $q = \langle i, p, b, e, k, o, r\rangle$.

Proof: Let $\hat{q}(Container\ i) = \langle \hat{p}, \hat{b}, \hat{e}, \hat{k}, \hat{r}\rangle$, where $q \in Conc_{\hat{Q}}\hat{q}$. By assumption, $p \in Conc_{\hat{P}}\hat{p}$, $b \in Conc_{\hat{B}}\hat{b}$, $e \in Conc_{\hat{E}}\hat{e}$, $k \in Conc_{\hat{K}}\hat{k}$, $o \in Conc_{\hat{P}}\hat{p}$, and $r \in Conc_{\hat{R}}\hat{r}$. We proceed, as in the definitions of $\mathcal{S}_2$ and $\mathcal{S}_3$, based upon the form of S_i. Suppose that

$$S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_m\texttt{))}]\!] \text{ or } [\![\texttt{(set! x (call/cc g))}]\!],$$

and that

$$e\langle [\![\texttt{f}]\!], b[\![\texttt{f}]\!]\rangle = \langle \alpha, b'\rangle \in C.$$

Since $e \in Conc_{\hat{E}}\hat{e}$, we may write (following the form of the definition of $\mathcal{S}_3$ in Figure 16)

$$\hat{e}[\![\texttt{f}]\!] = \langle \hat{c'}, \hat{k'}, \ldots\rangle \in \hat{D}$$

where $\hat{c} = \langle \hat{\alpha}, \hat{b'} \rangle$, $\alpha \in Conc_{\Lambda} \hat{\alpha}$ and $b' \in Conc_{\hat{B}} \hat{b'}$. We consider the components of $\mathcal{S}_2 q$ (see the definition of $\mathcal{S}_2$ in Figure 7). We have that

$$\langle i'', p'', b'', e'', k'', o'', r'' \rangle \in Conc_{\hat{Q}} \hat{q}$$

when

$$\hat{q}(Container\ i'') = \langle \hat{p''}, \hat{b''}, \hat{e''}, \hat{k''}, \hat{r''} \rangle$$

where $p'' \in Conc_{\hat{P}} \hat{p''}$, $b'' \in Conc_{\hat{B}} \hat{b''}$, $e'' \in Conc_{\hat{E}} \hat{e''}$, $k'' \in Conc_{\hat{K}} \hat{k''}$, $o'' \in Conc_{\hat{P}} \hat{p''}$, and $r'' \in Conc_{\hat{R}} \hat{r''}$. Letting

$$q' = \mathcal{S}_2 q = \langle i'', p'', b'', e'', k'', o'', r'' \rangle$$

and

$$\hat{t'} = \mathcal{S}_3 i(\hat{q}(Container\ i))(Container\ i'') = \langle \hat{p''}, \hat{b''}, \hat{e''}, \hat{k''}, \hat{r''} \rangle,$$

we must therefore show that the concretization of each component of $\hat{t'}$ contains the corresponding component of q'.

By the definitions of $\mathcal{S}_2$ and $\mathcal{S}_3$, $p'' = p + \alpha^d$, and $\hat{p''} = \hat{p} \oplus Abs_P(\alpha^d)$. By assumption, $p \in Conc_{\hat{P}} \hat{p}$. By Theorem 11, we have that

$$p + \alpha^d \in Conc_{\hat{P}}(\hat{p} \oplus Abs_P(\alpha^d)).$$

By the definitions of $\mathcal{S}_2$ and $\mathcal{S}_3$,

$$b'' = b'[p + \alpha^d / [\![\mathtt{z_1}]\!]] \cdots [p + \alpha^d / [\![\mathtt{z_n}]\!]],$$

and

$$\hat{b''} = \hat{b'}[\hat{p} \oplus Abs_P(\alpha^d) // [\![\mathtt{z_1}]\!]] \cdots [\hat{p} \oplus Abs_P(\alpha^d) // [\![\mathtt{z_n}]\!]].$$

We know that $b' \in Conc_{\hat{B}} \hat{b'}$, and therefore by the step above, and the definition of $f[x//y]$, $b'' \in Conc_{\hat{B}} \hat{b''}$.

Assume for the moment that

$$S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_m\texttt{))}]\!]$$

By the definitions of $\mathcal{S}_2$ and $\mathcal{S}_3$,

$$e'' = e[e\langle [\![\mathtt{y_1}]\!], b[\![\mathtt{y_1}]\!] \rangle / \langle [\![\mathtt{z_1}]\!], p + \alpha^d \rangle] \cdots [e\langle [\![\mathtt{y}_m]\!], b[\![\mathtt{y}_m]\!] \rangle / \langle [\![\mathtt{z}_m]\!], p + \alpha^d \rangle],$$

and

$$\hat{e''} = \hat{e}[\hat{e}[\![\mathtt{y_1}]\!] // [\![\mathtt{z_1}]\!]] \cdots [\hat{e}[\![\mathtt{y}_m]\!] // [\![\mathtt{z}_m]\!]].$$

By assumption, $e'' \in Conc_{\hat{E}} \hat{e''}$, and therefore

$$e\langle [\![\mathtt{y}_i]\!], b[\![\mathtt{y}_i]\!] \rangle \in Conc_{\hat{D}} \hat{e}[\![\mathtt{y}_i]\!]$$

for $1 \leq i \leq m$. Therefore, by the definition of the notation $f[x//y]$, $e'' \in Conc_{\hat{E}} \hat{e''}$. Similar reasoning holds in the case that

$$S_i = [\![\texttt{(set! x (call/cc f))}]\!].$$

By the definitions of $\mathcal{S}_2$ and $\mathcal{S}_3$, $k'' = \langle i, b, p, o\rangle$, and $\hat{k''} = \langle\{i\}, \hat{b}, \hat{p}\rangle$. Each member of k'' is contained in the concretization of the corresponding member of $\hat{k''}$. (We take $\hat{p}$ as the abstraction of both p and o, since by Theorem 4, $Net\, p = Net\, o$, and therefore $Abs_P p = Abs_P o$.) Therefore $k'' \in Conc_{\hat{K}} \hat{k''}$.

By the definition of $\mathcal{S}_2$ $o'' = p''$. As above, we take $\hat{p''}$ as the abstraction of both the procedure string of the state that results from this application of λ_α, and of the birth date of the new instance of λ_α. Again, this is justified by the fact that by Theorem 4, $Net\, p'' = Net\, o''$, and therefore $Abs_P p'' = Abs_P o''$.

Finally, by the definitions of $\mathcal{S}_2$ and $\mathcal{S}_3$,

$$r'' = r[k/o],$$

and

$$\hat{r''} = \hat{r}[\hat{k}//Container\ i].$$

By the definition of $\mathcal{S}_2$, o must end in the term β^d, where $\beta = Container\, i$, and therefore by the definition of Abs_R, and our assumption that $k \in Conc_{\hat{K}} \hat{k}$,

$$r[k/o] \in Conc_{\hat{R}} \hat{r}[\hat{k}//Container\ i].$$

We can repeat this reasoning in the case that

$$e\langle[\![\texttt{f}]\!], b[\![\texttt{f}]\!]\rangle = \langle j, b', o', p'\rangle \in K.$$

The only interesting argument arises, in that case, when proving that

$$p + Inv(p - p') \in Conc_{\hat{P}} \hat{o'},$$

where $\hat{o'}$ is the abstraction of the birth date of the procedure instance in which the continuation $\langle j, b', o', p'\rangle$ is formed. By Theorem 3,

$$Net(p + Inv(p - p')) = Net\, p';$$

by Theorem 4,

$$Net\, p' = Net\, o',$$

and therefore by the definition of Abs_P,

$$p + Inv(p - p') \in Conc_{\hat{P}} \hat{o'}.$$

Similar reasoning applies to the other forms of S_i. □

Theorem 19 *Let $q_0, q_1, \ldots$ be the sequence of states described by the evaluation of $\mathcal{E}_2 q_0$. Then $q_i \in Conc_{\hat{Q}}(\mathcal{E}_3 \hat{q_0})$ $\forall i$, where $q_0 \in Conc_{\hat{Q}} \hat{q_0}$.*

Proof: By induction on i. Let n be the number of applications of $\mathcal{E}_3$ that occur in the evaluation of $\mathcal{E}_3 \hat{q_0}$, and let $\hat{q_0}, \hat{q_1}, \ldots, \hat{q_n}$ be the sequence of abstract states described by that evaluation, as guaranteed by Theorem 16. As a basis, we have

$$\begin{aligned}\mathcal{E}_3(\hat{q_0}) &= \mathcal{E}_3(\hat{q_0} \sqcup_{\hat{Q}} \hat{q_1}) \\ &= \mathcal{E}_3(\hat{q_0} \sqcup_{\hat{Q}} \hat{q_1} \sqcup_{\hat{Q}} \hat{q_2}) \\ &= \cdots \\ &= \mathcal{E}_3(\hat{q_0} \sqcup_{\hat{Q}} \hat{q_1} \sqcup_{\hat{Q}} \cdots \sqcup_{\hat{Q}} \hat{q_n}) \\ &= \hat{q_0} \sqcup_{\hat{Q}} \hat{q_1} \sqcup_{\hat{Q}} \cdots \sqcup_{\hat{Q}} \hat{q_n}\end{aligned}$$

and therefore $q_0 \in Conc_{\hat{Q}}(\mathcal{E}_3 \hat{q_0})$.

Now assume the theorem holds for q_i. Then

$$q_i \in Conc_{\hat{Q}}(\mathcal{E}_3 \hat{q_0}) = Conc_{\hat{Q}} \hat{q_n}.$$

By Theorem 18,

$$q_{i+1} \in Conc_{\hat{Q}}(\mathcal{S}_3 j(\hat{q_n}(Container\ j)))$$

where $q_i = \langle j, p, b, e, k, o, r \rangle$, and therefore

$$q_{i+1} \in Conc_{\hat{Q}}(\mathcal{S}_3^* \hat{q_n}).$$

By the definition of $\mathcal{E}_3$,

$$\mathcal{S}_3^* \hat{q_n} \sqsubseteq_{\hat{Q}} \hat{q_n}.$$

Therefore

$$q_{i+1} \in Conc_{\hat{Q}} \hat{q_n}$$

and

$$q_{i+1} \in Conc_{\hat{Q}}(\mathcal{E}_3 \hat{q_0}).$$

□

Theorem 19 shows that the abstract state that results from evaluation under $\mathcal{E}_3$ approximates not merely the final state of evaluation (if there is such a final state) under $\mathcal{E}_2$, but *every* state that occurs during evaluation under $\mathcal{E}_2$. In the terminology of [28], we have created a *collecting interpretation* of the program, so called because it collects information from every state that occurs during execution. This collecting interpretation differs

significantly from that presented in [28], however. There, the domains used to collect values are separated from the domains over which evaluation occurs, in order that a power set, and not a power domain, may be used to represent the collected values. Here, we have made no such distinction. The more significant difference is that we have eliminated reflexivity from the domains over which the semantic functions are defined. This allows us to give a concrete representation to closures and continuations, using which we can reason easily about certain operational properties of these higher-order objects. It has also allowed us to write very simple proofs of correctness, which do not involve infinite fixpoints.

Notice that Theorem 19 does not stipulate that evaluation under $\mathcal{E}_2$ must terminate, in order that the result of the corresponding abstract interpretation under $\mathcal{E}_3$ be meaningful; we understand the abstract state $\mathcal{E}_3(Abs_Q q)$ to represent every state that occurs during $\mathcal{E}_2 q$, even if $\mathcal{E}_2 q$ does not terminate. This is precisely what we would hope for, in compiling a program that might (intentionally) not terminate: we wish to know what states may arise during the computation, in order that we correctly compile that portion of the code that *is* used. (Recall from subsection 2.4 that we regard a program that does not terminate as a special case of a program that has unused code; in the case of a non-terminating program, at least its final statement is unused.)

2.10 Approximate Solutions in Terms of Stack Configurations

At this point, we know how stack configurations model procedure strings, and how they can be computed by abstract interpretation, in a way that preserves the meaning of procedure strings. In subsection 2.6 we formulated simple, and optimal solutions to a number of flow analysis problems, in terms of procedure strings. In this subsection we recast these into approximate solutions to the same problems, in terms of stack configurations. By the nature of abstraction, we need *not* derive a new solution to each problem; rather, we "project" the old solution onto the space of stack configurations, in a way that preserves its meaning. In each case, we begin with a statement of the form "X is true of the program, if and only if C is true of the procedure strings described by its execution." We derive the statement "X is true of the program, only if $\hat{C}$ is true of the stack configurations described by its abstract interpretation," where if $\hat{C}$ is true of a stack configuration $\hat{p}$, then C is true of all p such that $p \in Conc_{\hat{P}} \hat{p}$. The loss of information that is suffered in this translation is reflected in the change from "if and only if," within the original statement, to "only if," in the derived statement: we cannot say with certainty that X is true of the program, given that $\hat{C}$ is true of the stack configurations it describes; only

that X is certainly not true of the program if $\hat{C}$ is not true of the stack configurations it describes. We must therefore arrange that transformations and techniques that are not universally applicable are invoked only when the circumstances under which they are illegal have been excluded.

2.10.1 Side-Effects, in Terms of Stack Configurations

It is by abstraction of Theorem 7 that we may turn our abstract interpretation to the analysis of side-effects. The theorem states that if $\dot{\mathbf{x}}$ is a variable instance, the birth date of which is p_b, and which is referenced in a state whose procedure string is p_r, then a procedure instance $\dot{\lambda}_\alpha$ has a side-effect as a result of this reference if and only if $Net\,(p_r - p_b)$ contains a term of the form α^d corresponding to $\dot{\lambda}_\alpha$. We may cast this result into the realm of stack configurations by way of the following theorem.

Theorem 20 *Let $\hat{q_n} = \mathcal{E}_3\hat{q_0}$ where $q_0 \in Conc_{\hat{Q}}\hat{q_0}$. If during the evaluation of $\mathcal{E}_2 q_0$ there is an instance $\dot{\lambda}_\alpha$ of λ_α that has a side-effect upon an instance $\dot{\mathbf{x}}$ of x, then there exists a $\gamma \in \Lambda$ such that*

$$(\hat{p_r} \ominus \hat{p_b})\alpha \cap \{\mathbf{d}, \mathbf{dd}^+, \mathbf{u}^+\mathbf{d}^+\} \neq \{\},$$

where $\hat{q_n}\beta = \langle \hat{p_b}, \ldots \rangle$ and $\hat{q_n}\gamma = \langle \hat{p_r}, \ldots \rangle$, where λ_β is the binder of x, and x is referenced (directly) within λ_γ.

Proof: Let $\dot{\lambda}_\alpha$ be an instance of λ_α that has a side-effect upon an instance $\dot{\mathbf{x}}$ of x, let this side-effect arise from a (direct) reference to $\dot{x}$ within λ_γ, and let λ_β be the binder of x. By Theorem 7, it must be that there exists p_b and p_r such that $Net(p_r - p_b)$ contains a term of the form α^d, where p_b is the procedure string of the state in which $\dot{\mathbf{x}}$ is bound and p_r is the procedure string of a state in which $\dot{\mathbf{x}}$ is referenced within λ_γ (which reference gives rise to the side-effect attributed to λ_α). Let q_b be the state whose procedure string (that is, whose first component) is p_b and q_r be the state whose procedure string is p_r. By Theorem 19, $q_r, q_b \in Conc_{\hat{Q}}\hat{q_n}$, where $\hat{q_n}$ is as defined by the current theorem. Let $\hat{q_n}\beta = \langle \hat{p_b}, \ldots \rangle$, and $\hat{q_n}\gamma = \langle \hat{p_r}, \ldots \rangle$. By the definition of $\sqsubseteq_{\hat{Q}}$ and $\sqsubseteq_{\hat{T}}$, $p_r \in Conc_{\hat{Q}}\hat{p_r}$ and $p_b \in Conc_{\hat{Q}}\hat{p_b}$. By Theorem 13,

$$Net(p_r - p_b) \in Conc_{\hat{P}}(\hat{p_r} \ominus \hat{p_b}).$$

By the form of $Net(p_r - p_b)$,

$$Trace(Net(p_r - p_b))\alpha = \alpha^{a_1} \cdots \alpha^{a_k},$$

where $a_i = d$ for some $1 \leq i \leq k$. By the definition of $Conc_{\hat{P}}$, this implies that

$$(\hat{p_r} \ominus \hat{p_b})\alpha \cap \{\mathbf{d}, \mathbf{dd}^+, \mathbf{u}^+\mathbf{d}^+\} \neq \{\}.$$

□

Theorem 20 should be read as follows. Suppose we are given an initial state q_0 and an abstract state $\hat{q_0}$, the concretization of which contains q_0. We evaluate q_0 and $\hat{q_0}$ under $\mathcal{E}_2$ and $\mathcal{E}_3$ respectively. (Let $\mathcal{E}_3\hat{q_0} = \hat{q_n}$.) If during the evaluation of $\mathcal{E}_2 q_0$ we encounter an instance of λ_α that, by Definition 1, has a side-effect upon an instance of the variable $\mathbf{x}$, then it must be that abstract interpretation (the history of which is collected into $\hat{q_n}$) will reveal that side-effect, in the following way. If λ_γ is the procedure within which $\mathbf{x}$ is accessed directly (to produce the side-effect), and λ_β is the procedure which binds $\mathbf{x}$, then $(\hat{p_r} \ominus \hat{p_b})\alpha$ will contain one of $\mathbf{d}$, $\mathbf{dd}^+$ or $\mathbf{u}^+\mathbf{d}^+$, where $\hat{p_b}$ is the first component of $\hat{q_n}\beta$, and $\hat{p_r}$ is the first component of $\hat{q_n}\gamma$. $\hat{p_b}$ could alternatively (and equivalently) have been defined as $\hat{b}[\![\mathbf{x}]\!]$, where $\hat{q_n}\gamma = \langle \hat{p_r}, \hat{b}, \hat{e}, \hat{k}, \hat{r} \rangle$, since $\hat{b}$ maps the free variables in λ_γ to (abstractions of) their birth dates. We will make use of this fact below.

2.10.2 Stack Allocation, in Terms of Stack Configurations

We may make a similar projection of Theorem 8, which characterizes the conditions under which a variable instance must be heap-allocated in terms of procedure strings, to an analogous theorem over stack configurations. Theorem 8 holds that if $\mathbf{x}$ is a variable bound by λ_β, and p_b is the procedure string that identifies an instance $\dot{\mathbf{x}}$ of $\mathbf{x}$, and p_r is the procedure string of a state in which $\dot{\mathbf{x}}$ is referenced, then this reference takes place after the instance of λ_β that binds $\dot{\mathbf{x}}$ has been exited, if and only if $Net(p_r - p_b)$ has a term of the form β^u. We cast this into the language of stack configurations as follows.

Theorem 21 *Let $\hat{q_n} = \mathcal{E}_3\hat{q_0}$, where $q_0 \in Conc_{\hat{Q}}\hat{q_0}$. Let $\mathbf{x}$ be a variable bound by λ_β. If during the evaluation of $\mathcal{E}_2 q_0$ there is an instance $\dot{\mathbf{x}}$ of $\mathbf{x}$ such that $\dot{\mathbf{x}}$ is referenced following the deactivation of the instance $\dot{\lambda_\beta}$ of λ_β that binds it, then there exists a $\gamma \in \Lambda$ such that*

$$(\hat{p_r} \ominus \hat{p_b})\beta \cap \{\mathbf{u}, \mathbf{uu}^+, \mathbf{u}^+\mathbf{d}^+\} \neq \{\},$$

where $\hat{q_n}\beta = \langle \hat{p_b}, \ldots \rangle$ and $\hat{q_n}\gamma = \langle \hat{p_r}, \ldots \rangle$, and $\mathbf{x}$ is referenced (directly) within λ_γ.

Proof: Let $\dot{\lambda_\beta}$ be the instance of λ_β that binds an instance $\dot{\mathbf{x}}$ of $\mathbf{x}$, such that $\dot{\mathbf{x}}$ is referenced in a state q_r whose procedure string is p_r, following the deactivation of $\dot{\lambda_\beta}$. Let p_b be the birth date of $\dot{\mathbf{x}}$. By Theorem 8, $Net(p_r - p_b) = \cdots \beta^u \cdots$, where β^u corresponds to the deactivation of $\dot{\lambda_\beta}$. Let λ_γ be the procedure within which $\dot{\mathbf{x}}$ is referenced directly in q_r. Let

$\hat{q_n}\beta = \langle p_b, \ldots \rangle$, and let $\hat{q_n}\gamma = \langle p_r, \ldots \rangle$. Then by Theorem 19, and the definition of $\sqsubseteq_{\hat{Q}}$, $p_r \in Conc_{\hat{P}}\hat{p_r}$, and $p_b \in Conc_{\hat{P}}\hat{p_b}$. By the definition of $Conc_{\hat{P}}$ and Theorem 11,

$$Net(p_r - p_b) \in Conc_{\hat{P}}(\hat{p_r} \ominus \hat{p_b}),$$

and since

$$Trace(Net(p_r - p_b))\beta = \beta^{a_1}\beta^{a_2}\cdots\beta^{a_k}$$

where $a_i = u$ for some $1 \leq i \leq k$, we have that

$$(\hat{p_r} \ominus \hat{p_b})\beta \cap \{\mathbf{u}, \mathbf{uu}^+, \mathbf{u}^+\mathbf{d}^+\} \neq \{\}.$$

□

Theorem 21 has the following intuitive interpretation. As before (in the case of Theorem 20), we are given a state q_0, and an abstract state $\hat{q_0}$, the concretization of which contains q_0. We evaluate $\mathcal{E}_2 q_0$ and $\mathcal{E}_3 \hat{q_0}$, and call the latter $\hat{q_n}$. If during the evaluation of $\mathcal{E}_2 q_0$ we encounter an instance $\dot{\mathbf{x}}$ of a variable $\mathbf{x}$, such that $\dot{\mathbf{x}}$ is referenced after the instance $\dot{\lambda_\beta}$ of λ_β that binds it has been deactivated, then abstract interpretation will reveal this fact, in the following way. If λ_γ is the procedure within which the offensive reference to $\mathbf{x}$ takes place, then

$$(\hat{p_r} \ominus \hat{p_b})\beta \cap \{\mathbf{u}, \mathbf{uu}^+, \mathbf{u}^+\mathbf{d}^+\} \neq \{\}$$

where $\hat{p_b}$ is the first component of $\hat{q_n}\beta$, and $\hat{p_r}$ is the first component of $\hat{q_n}\gamma$. As before, $\hat{p_b}$ could equivalently have been defined as $\hat{b}[\![\mathbf{x}]\!]$ where $\hat{q_n}\gamma = \langle \hat{p_r}, \hat{b}, \hat{e}, \hat{k}, \hat{r} \rangle$, since $\hat{b}$ maps the free variables in λ_γ to (abstractions of) their birth dates.

Theorems 20 and 21 may be applied at compile-time, by choosing $\hat{q_0}$ such that $q_0 \in Conc_{\hat{Q}}\hat{q_0}$ for all possible initial states q_0. In that case, Theorem 20 may be invoked to discover, for every procedure application of the program, what side-effects may occur as a result of the application. Likewise, Theorem 21 may be invoked to discover, for every bound variable, whether that variable must be heap-allocated. (In Parcel, we have taken the approach that if *any* variable bound by a lambda expression must be heap-allocated, then the entire activation record for that lambda expression will be placed in the heap, at every application of the lambda expression.)

2.10.3 Generalized Hierarchical Allocation and Deallocation

Finally, we revisit the discussion of subsection 2.6.3, and recast our observations in terms of stack configurations. As before, let $\mathcal{E}_2 q_0$ describe a evaluation sequence from initial state q_0, let $\dot{\mathbf{x}}$ be an instance of a variable

$\mathbf{x}$ that is bound in state q_b, and let q_r be the state in which $\mathbf{x}$ is referenced. Recall that we wished to place $\dot{\mathbf{x}}$ on a list of objects to be deallocated upon exit from a procedure instance, such that the instance outlives $\dot{\mathbf{x}}$. Our observation was that if p_b is the birth date of $\dot{\mathbf{x}}$, and p_r is the procedure string of a state in which it is referenced, then if $Net(p_r - p_b)$ has k terms of the form α^u (summing over all $\alpha \in \Lambda$), then $\dot{\mathbf{x}}$ must be placed on the deallocation list of the k^{th} procedure instance above the one by which $\dot{\mathbf{x}}$ is bound. The maximum m of k, over all p_r that denote states in which $\dot{\mathbf{x}}$ is referenced, points us to the innermost procedure instance that outlives $\dot{\mathbf{x}}$ (and thus to the shortest-lived deallocation list on which it is safe to place $\dot{\mathbf{x}}$).

We may approximate m by use of stack configurations, as follows. Let $\hat{q_n} = \mathcal{E}_3\hat{q_0}$ where $q_0 \in Conc_{\hat{Q}}\hat{q_0}$, and let λ_α be the binder of $\mathbf{x}$. Assigning a weight of 1 to $\mathbf{u}$, a weight of ∞ to $\mathbf{uu}^+$ and $\mathbf{u}^+\mathbf{d}^+$, and a weight of 0 to the other members of Δ, we compute the maximum, over all stack configurations $\hat{p_r}$ such that $\hat{q_n}\beta = \langle \hat{p_r}, \ldots \rangle$ where $\mathbf{x}$ is referenced directly within λ_β, of the weights of the subsets in the range of $\hat{p_r} \ominus \hat{p_b}$. Let $\hat{m}$ be this maximum; it is easy to prove that $\hat{m} \geq m$, for any instance $\dot{\mathbf{x}}$ of $\mathbf{x}$. We interpret $\hat{m}$ exactly as we did m: every instance $\dot{\mathbf{x}}$ of $\mathbf{x}$ is placed on the deallocation list of the $\hat{m}^{\text{th}}$ procedure instance *above* that by which $\dot{\mathbf{x}}$ is bound. If $\hat{m}$ is greater than the number of active procedure instances at the point of $\dot{\mathbf{x}}$'s creation, then $\dot{\mathbf{x}}$ is placed on the deallocation list of the top level (which is to say, $\dot{\mathbf{x}}$ will never be released through this mechanism).

The trouble with this approach is that a single $\mathbf{uu}^+$ or $\mathbf{u}^+\mathbf{d}^+$ within $(\hat{p_r} \ominus \hat{p_b})\beta$ for some $\beta \in \Lambda$, means that every instance of $\dot{\mathbf{x}}$ will be associated with the top level. In order, therefore, to use this as the basis of a practical system of storage reclamation, it would have to be augmented with garbage collection. We could, for example, allocate objects from a free list, and deallocate them (return them to the free list) according to the above scheme, invoking garbage collection when the free list is exhausted. When we had precise information in the form of procedure strings, we had the luxury of placing $\dot{\mathbf{x}}$ on the deallocation list of the innermost procedure instance which would outlive $\dot{\mathbf{x}}$, and we represented this instance as an integer m. The abstraction of this approach to stack configurations causes too great a loss of information, and m is too often approximated by ∞.

We can improve this strategy dramatically by the following observation. All that is required, in the placement of $\dot{\mathbf{x}}$ on a deallocation list, is that we find a procedure instance that outlives $\dot{\mathbf{x}}$ (that is, we need not identify the innermost such instance). This suggests several strategies for placing the instances of $\mathbf{x}$ on deallocation lists. We could, for example, construct the set

$$X = \{\beta \mid (\hat{p_r} \ominus \hat{p_b})\beta \cap \{\mathbf{u}, \mathbf{uu}^+, \mathbf{u}^+\mathbf{d}^+\} = \{\}\}$$

```
(set! f (lambda_α (x)
   (if (null? x)
       #f
      (cons ((lambda_γ () (car x)))
            (f (cdr x))))))
(f '(a b c d))
```

Figure 19: An Example of the Inaccuracy of $\mathcal{E}_3$

at compile time. When x is instantiated, we traverse the active procedure instances from innermost to outermost, until we find a member of X, and place the instance of x on the deallocation list of that instance. We know with certainty that there is an instance of λ_β at every point at which x is instantiated, if $\epsilon \notin \hat{p_b}\beta$, since for every $p_b \in Conc_{\hat{P}}\hat{p_b}$, p_b is d-bitonic by Theorem 1, and if $Trace(Net\ p_b)\beta \neq \epsilon$ then p_b must take the (non-empty) form $\beta^d \cdots \beta^d$, indicating that at least one instance of λ_β is active.

As mentioned in subsection 2.6, we are not proposing this seriously as a storage management strategy; in Parcel, a simple distinction is made between activation records that can be stack-allocated, and those that must be heap-allocated. Rather, this hierarchical strategy is a guise for a problem that is difficult to motivate until we have seen the results of automatic parallelization, namely, the placement of dynamically allocated objects in a hierarchical shared memory. We will address a simplified version of the problem of storage management in a parallel, shared memory setting in subsection 2.15.

2.11 A Shift in Perspective (and in Accuracy)

In subsection 2.7 we remarked that our abstraction $\ominus$ of the difference of two procedure strings entails so great a loss of information as to be practically useless. For example, consider Theorem 21. If $\hat{p_r}$ and $\hat{p_b}$ satisfy $\mathbf{dd}^+ \in \hat{p_r}\alpha$ and $\mathbf{dd}^+ \in \hat{p_b}\alpha$, then $(\hat{p_r} \ominus \hat{p_b})\alpha = \Delta$, and it appears that all instances of x must be heap-allocated. To appreciate just how devastating this is to the accuracy of our analysis, consider Figure 19. (For clarity, the example is presented in Scheme; it is rewritten in a language nearer to $\mathcal{L}$ as in Figure 20.)

Suppose we perform the abstract interpretation $\hat{q_n} = \mathcal{E}_3\hat{q_0}$ of the above program, where $\hat{q_0}$ is initial abstract state in which cons, car, and cdr are defined. (For the moment, ignore the question of how cons is defined, or think of it as defined entirely in terms of closures with free variables [9].)

```
(set! f (lambda_α (x) <t1 t2 t3 t5 t6 t7>
   (set! t1 (null? x))_1
   (if t1 (goto 3) (goto 5))_2
   (set! t2 #f)_3
   (goto 10)_4
   (set! t3 (lambda_γ () <t4>
      (set! t4 (car x))
      (return t4)))_5
   (set! t5 (t3))_6
   (set! t6 (cdr x))_7
   (set! t7 (f t6))_8
   (set! t2 (cons t5 t7))_9
   (return t2)_10)
(f '(a b c d))
```

Figure 20: An Example of the Inaccuracy of $\mathcal{E}_3$, Rewritten in $\mathcal{L}$

During evaluation of this program (under $\mathcal{E}_2$) there may be more than one instance of λ_α active at a time. By Theorem 19, this means that $\mathbf{dd}^+ \in \hat{p_b}\alpha$, where $\hat{q_n}\alpha = \langle \hat{p_b}, \ldots \rangle$. Since λ_γ is applied directly by the instance of λ_α within which it is closed, we also have $\mathbf{dd}^+ \in \hat{p_r}\alpha$ where $\hat{q_n}\gamma = \langle \hat{p_r}, \ldots \rangle$. By the definition of $\ominus$, we have that $(\hat{p_r} \ominus \hat{p_b})\alpha = \Delta$, and by Theorem 21, this implies that all instances of x must be heap-allocated (which is obviously unnecessary in this simple example).

Nevertheless, we are on the right track. We're interested in finding a stack configuration which approximates $p_r - p_b$ for all possible values of p_r and p_b, and we are doing so by finding an approximation $\hat{p_r}$ to all values of p_r, an approximation $\hat{p_b}$ to all values of p_b, and approximating their differences directly with $\ominus$. The trouble is that $\hat{p_r}$ and $\hat{p_b}$ record far more of the history of the computation than interests us. A stack configuration contains only finite information; if the procedure strings p_b that are represented by $\hat{p_b}$ are much longer, or more complex in structure, than the procedure strings $p_r - p_b$ that we are trying to compute, then most of the information content of $\hat{p_r}$ will be consumed in representing the prefixes p_b (since each string p_r represented by $\hat{p_r}$ takes the form $p_b + (p_r - p_b)$). Then our situation will be not unlike that faced when subtracting two floating point numbers of nearly equal value: the result will be dominated by the error inherent in the representation.

The solution to this is a shift in perspective. Rather than "stamping" a variable instance with a fixed birth date, we will associate with each vari-

able instance a stack configuration, initialized to the value $\lambda\alpha.\{\epsilon\}$, which will be carried along with the variable instance as it makes interprocedural movements, and will be updated to reflect those movements. When a variable instance $\dot{\mathtt{x}}$ is referenced directly by a procedure λ_γ, the stack configuration associated with $\dot{\mathtt{x}}$ will have recorded the (net) movements that occurred between the binding of $\dot{\mathtt{x}}$ and the reference to it by λ_γ. In terms of the discussion above, the stack configuration that is accumulated as $\dot{\mathtt{x}}$ undergoes interprocedural movements (or, more precisely, as a closure which captures $\dot{\mathtt{x}}$ undergoes interprocedural movements) is an approximation to the strings $p_r - p_b$ in terms of which the solutions to our data flow problems have been expressed. We will see that this method of computing the solutions has far greater accuracy than the naive approach embodied in $\mathcal{E}_3$.

We are not altering the structure of the domains defined in Figure 12, and thus there is no need to redefine the partial order and LUB operators over the abstract domains. We are, however, changing the meaning of the members of the abstract domains, with respect to their concrete counterparts, and this meaning is defined by the abstraction and concretization maps that carry us between the abstract and concrete realms. The new abstraction maps are presented in Figure 21. Most have changed in type, from their definitions under $\mathcal{E}_3$. We will return to this shortly.

If $\langle\hat{\alpha},\hat{b}\rangle \in \hat{C}$ is an abstract closure that captures a free variable $\mathtt{v}$, then under $\mathcal{E}_4$ (the abstract semantics we are deriving by modification of $\mathcal{E}_3$), $\hat{b}[\![\mathtt{v}]\!]$ is a record of the interprocedural movements that are described by $\mathtt{v}$, from the point at which it is bound to the current state. (We will continue to refer to $\hat{b}$ as a birth date map, even though its new meaning warrants a slightly different name.) Likewise, a member $\langle\hat{\imath},\hat{b},\hat{p}\rangle \in \hat{K}$ has changed in meaning; $\hat{p}$ is now a record of the interprocedural movements described by the abstract continuation, from the point of its formation to the current state, and $\hat{b}$ likewise records the interprocedural movements described by the variables visible in the state in which the continuation was formed, from the points at which they are bound, to the current state. Where all movements, encoded as procedure strings and later as stack configurations, were *absolute* under $\mathcal{E}_2$ and $\mathcal{E}_3$ (being accumulated from the beginning of evaluation), under $\mathcal{E}_4$ they have become *relative* (being accumulated from the points at which procedures are activated during evaluation). This has the effect of reducing the amount of information that need be approximated by a stack configuration (since it records a shorter piece of the history of evaluation), and of simplifying the solutions to our data flow problems (since their solutions are computed directly by evaluation), but of complicating the relationship between the concrete domains of $\mathcal{E}_2$ and the abstract domains of $\mathcal{E}_4$.

$$
\begin{aligned}
&Abs_\Lambda \equiv \lambda\alpha.\ \text{if } \alpha = \bot_\Lambda \text{ then } \{\} \text{ else } \{\alpha\}\\
&Abs_N \equiv \lambda i.\ \text{if } i = \bot_N \text{ then } \{\} \text{ else } \{i\}\\
&Abs_B \equiv \lambda b.\lambda p.\lambda v.Abs_P(p - (bv))\\
&Abs_C \equiv \lambda\langle\alpha, b\rangle.\lambda p.\langle Abs_\Lambda\alpha, Abs_B bp\rangle\\
&Abs_K \equiv \lambda\langle i, b, p, o\rangle.\langle Abs_N i, Abs_B bp, Abs_P p\rangle\\
&Abs_D \equiv \lambda x.\quad \text{if } x = \bot_D \text{ then} \qquad \langle \bot_{\hat{C}}, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in C \text{ then} \qquad \langle Abs_C xp, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in K \text{ then} \qquad \langle \bot_{\hat{C}}, Abs_K xp, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in PrimOp \text{ then } \langle \bot_{\hat{C}}, \bot_{\hat{K}}, Abs_{PrimOp} x, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in Int \text{ then} \qquad \langle \bot_{\hat{C}}, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, Abs_{Int} x, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in Bool \text{ then} \qquad \langle \bot_{\hat{C}}, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, Abs_{Bool} x\rangle\\
&Abs_E \equiv \lambda e.\lambda p.\lambda v.\ \sqcup_{\hat{D}} \{Abs_D(e\langle v, p'\rangle)p \mid p' \in P\}\\
&Abs_R \equiv \lambda r.\lambda p.\lambda\alpha.\ \sqcup_{\hat{K}} \{Abs_K(r(p' + \alpha^d))p \mid p' \in P\}\\
&Abs_Q \equiv \lambda\langle i, b, p, e, k, o, r\rangle.\lambda\alpha.\ \text{if } \alpha \neq Container\ i\\
&\qquad\qquad\qquad\qquad \text{then } \langle \bot_{\hat{P}}, \bot_{\hat{B}}, \bot_{\hat{E}}, \bot_{\hat{K}}, \bot_{\hat{R}}\rangle\\
&\qquad\qquad\qquad\qquad \text{else } \langle Abs_P p, Abs_B bp, Abs_E ep,\\
&\qquad\qquad\qquad\qquad\qquad Abs_K kp, Abs_R rp\rangle
\end{aligned}
$$

Figure 21: Abstraction Maps

Consider Abs_B. It now has type $B \to P \to \hat{B}$. The additional parameter p is the procedure string of the state which contains the value $b \in B$ being projected. That is, p is the first component of a state q, such that b is the second component of q, or such that the environment of q contains a closure or continuation of which b is a component. Under $\mathcal{E}_3$, we knew what members of B were represented by a $\hat{b} \in \hat{B}$, by simple examination of the structure of $\hat{b}$. Under $\mathcal{E}_4$, however, the subset of B represented by $\hat{b}$ is as much a function of the state of which it is a part, as of its structure. This is because $\hat{b}$ now maps a variable v to an abstraction of $p_a - p_b$, where p_b is the birth date of an instance of v, and p_a is the procedure string of the current state. In order to recover v's birth date (which is necessary if we are to concretize $\hat{b}$), we must have a value for p_a (or rather, we must have an approximation to the value of p_a). Said otherwise, the stack configuration of the current state may be seen as the sum of the (abstraction of the) birth date of v, and the movements described by v from the point of its instantiation to the current state (for any v in the lexical scope of the current state). Given the stack configuration of the current state, and the stack configuration that represents these movements, we may reconstruct the birth date of v. We therefore define $Conc_{\hat{B}}$ as

$$\lambda\hat{b}.\lambda\hat{p}.\{b \mid Abs_B bp \sqsubseteq_{\hat{B}} \hat{b}, \text{ for some } p \in Conc_{\hat{P}}\hat{p}\}.$$

The concretization map that corresponds to each abstraction map whose type has been changed by addition of a "context" parameter p, is defined in exactly this way, by addition of a parameter $\hat{p}$, that represents the values that p may assume, during abstraction. For example, $Conc_{\hat{E}}$ is defined as

$$\lambda\hat{e}.\lambda\hat{p}.\{e \mid Abs_E ep \sqsubseteq_{\hat{E}} \hat{e}, \text{ for some } p \in Conc_{\hat{P}}\hat{p}\}.$$

The usual relationship between an abstraction map Abs_X and the corresponding concretization map $Conc_{\hat{X}}$ may be written as $x \in Conc_{\hat{X}}(Abs_X x)$. The corresponding relation for the abstraction maps under $\mathcal{E}_4$ is given by the following theorem. The result is stated in terms of Abs_B and $Conc_{\hat{B}}$, but applies directly to the other abstraction and concretization maps.

Theorem 22

$$b \in Conc_{\hat{B}}(Abs_B bp)\hat{p} \text{ for all } p \in Conc_{\hat{P}}\hat{p}.$$

Proof: Let $p \in Conc_{\hat{P}}\hat{p}$. By the definition of Abs_B and $Conc_{\hat{B}}$,

$$Conc_{\hat{B}}(Abs_B bp)\hat{p} = \{b' \mid Abs_B b'p' \sqsubseteq_{\hat{B}} Abs_B bp \text{ for some } p' \in Conc_{\hat{P}}\hat{p}\},$$

$$
\begin{aligned}
&Move_{\hat{B}} : \hat{B} \to \hat{P} \to \hat{B} \equiv \lambda\hat{b}.\lambda\hat{p}.\lambda v.(\hat{b}v) \oplus \hat{p} \\
&Move_{\hat{C}} : \hat{C} \to \hat{P} \to \hat{C} \equiv \lambda\langle\hat{\alpha}, \hat{b}\rangle.\lambda\hat{p}.\langle\hat{\alpha}, Move_{\hat{B}}\hat{b}\hat{p}\rangle \\
&Move_{\hat{K}} : \hat{K} \to \hat{P} \to \hat{K} \equiv \lambda\langle\hat{i}, \hat{b}, \hat{p}\rangle.\lambda\hat{p'}.\langle\hat{i}, Move_{\hat{B}}\hat{b}\hat{p'}, \hat{p} \oplus \hat{p'}\rangle \\
&Move_{\hat{D}} : \hat{D} \to \hat{P} \to \hat{D} \equiv \lambda\langle\hat{c}, \hat{k}, \hat{f}, \hat{z}, \hat{x}\rangle.\lambda\hat{p}.\langle Move_{\hat{C}}\hat{c}\hat{p}, Move_{\hat{K}}\hat{k}\hat{p}, \hat{f}, \hat{z}, \hat{x}\rangle \\
&Move_{\hat{E}} : \hat{E} \to \hat{P} \to \hat{E} \equiv \lambda\hat{e}.\lambda\hat{p}.\lambda v.Move_{\hat{D}}(\hat{e}v)\hat{p} \\
&Move_{\hat{R}} : \hat{R} \to \hat{P} \to \hat{R} \equiv \lambda\hat{r}.\lambda\hat{p}.\lambda\alpha.Move_{\hat{K}}(\hat{r}\alpha)\hat{p} \\
&Move_{\hat{T}} : \hat{T} \to \hat{P} \to \hat{T} \equiv \lambda\langle\hat{p}, \hat{b}, \hat{e}, \hat{k}, \hat{r}\rangle.\lambda\hat{p'}.\langle\hat{p} \oplus \hat{p'}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Move_{\hat{B}}\hat{b}\hat{p'}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Move_{\hat{E}}\hat{e}\hat{p'}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Move_{\hat{K}}\hat{k}\hat{p'}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Move_{\hat{R}}\hat{r}\hat{p'}\rangle
\end{aligned}
$$

Figure 22: Auxiliary Functions for $\mathcal{E}_4$

and this certainly contains b itself, since $p \in Conc_{\hat{P}}\hat{p}$ by choice, and $Abs_B bp \sqsubseteq_{\hat{B}} Abs_B bp$ trivially. □

The definitions of $\mathcal{S}_4$, $\mathcal{S}_4^*$ and $\mathcal{E}_4$ are presented in Figures 23, 24, and 26. In order to localize the changes to $\mathcal{E}_3$, we make use of the auxiliary functions $Move_{\hat{B}}$, $Move_{\hat{C}}$, $Move_{\hat{K}}$, $Move_{\hat{D}}$, $Move_{\hat{E}}$, $Move_{\hat{R}}$, and $Move_{\hat{T}}$. These are defined in Figure 22. $Move_{\hat{C}}$ maps an abstract closure $\hat{c}$ and a stack configuration $\hat{p}$ to the abstract closure that results when $\hat{c}$ makes the interprocedural movements described by $\hat{p}$. The first component of $\hat{c}$ (the set of indices of the lambda expressions whose closures are represented in $\hat{c}$) is not affected by these movements; the second component (call it $\hat{b}$), which under $\mathcal{E}_4$ maps the free variables in the abstract closure to the movements they have described, following their instantiation, is updated to reflect the movements described by $\hat{p}$. $Move_{\hat{K}}$ is defined similarly. $Move_{\hat{E}}$ computes a new abstract environment, in which every object that records interprocedural movements (closure and continuation) is updated, according to the movements implied by its second argument.

Theorem 23 *If* $q \in Conc_{\hat{Q}}\hat{q}$ *then* $\mathcal{S}_2 q \in Conc_{\hat{Q}}(\mathcal{S}_4 i(\hat{q}(Containeri)))$ *where* $q = \langle i, p, b, e, k, o, r\rangle$.

Proof: Let

$$\hat{q}(Containeri) = \langle\hat{p}, \hat{b}, \hat{e}, \hat{k}, \hat{r}\rangle,$$

where $Abs_Q qp \sqsubseteq_{\hat{Q}} \hat{q}$. By this assumption, $Abs_P p \sqsubseteq_{\hat{P}} \hat{p}$, $Abs_B bp \sqsubseteq_{\hat{B}} \hat{b}$,

Let $\hat{t} = \langle \hat{p}, \hat{b}, \hat{e}, \hat{k}, \hat{r} \rangle, i \in N$. Then $\mathcal{S}_4 : N \to \hat{T} \to \hat{Q}$ is defined, according to the form of S_i, as follows.

$$
\begin{array}{l}
S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_m\texttt{))}]\!] \text{ or } S_i = [\![\texttt{(set! x (call/cc f))}]\!] \Rightarrow \\
\quad \mathcal{S}_4 i \hat{t} = \hat{q}_c \sqcup_{\hat{Q}} \hat{q}_k \\
\quad \text{where } \hat{q}_c = \sqcup_{\hat{Q}} \Big\{ \lambda\beta.\ \text{if } \beta \neq \alpha \\
\qquad\qquad\qquad\qquad \text{then } \perp_{\hat{T}} \\
\qquad\qquad\qquad\qquad \text{else } \langle \hat{p} \oplus \hat{p}', \\
\qquad\qquad\qquad\qquad\quad (\mathit{Move}_{\hat{B}} \hat{b}' \hat{p}')[\lambda\alpha.\{\epsilon\}/[\![\texttt{z}_1]\!]] \cdots [\lambda\alpha.\{\epsilon\}/[\![\texttt{z}_n]\!]], \\
\qquad\qquad\qquad\qquad\quad \mathit{Move}_{\hat{E}} \hat{e}' \hat{p}', \\
\qquad\qquad\qquad\qquad\quad \mathit{Move}_{\hat{K}} \langle \{i\}, \hat{b}, \lambda\alpha.\{\epsilon\} \rangle \hat{p}', \\
\qquad\qquad\qquad\qquad\quad \mathit{Move}_{\hat{R}} (\hat{r}[\hat{k} // \mathit{Container}\ i]) \hat{p}' \rangle \\
\qquad\qquad\qquad \text{where } \lambda_\alpha = [\![\texttt{(lambda (z}_1 \cdots \texttt{z}_m\texttt{) <z}_{m+1} \cdots \texttt{z}_n\texttt{>} \\
\qquad\qquad\qquad\qquad\qquad\qquad S_j \cdots \texttt{)}]\!], \quad \hat{p}' = \mathit{Abs}_P(\alpha^d), \\
\qquad\qquad\qquad \text{and} \quad \hat{e}' = \text{if } S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_m\texttt{))}]\!] \\
\qquad\qquad\qquad\qquad\qquad\quad \text{then } \hat{e}[\hat{e}[\![\texttt{y}_1]\!] // [\![\texttt{z}_1]\!]] \cdots [\hat{e}[\![\texttt{y}_m]\!] // [\![\texttt{z}_m]\!]] \\
\qquad\qquad\qquad\qquad\qquad\quad \text{else } \hat{e}[\langle \perp_{\hat{C}}, \langle \{i\}, \hat{b}, \lambda\alpha.\{\epsilon\} \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \perp_{\hat{Prim}Op}, \perp_{\hat{Int}}, \perp_{\hat{Bool}} \rangle // [\![\texttt{z}_1]\!]] \\
\qquad\qquad\quad \Big| \ \alpha \in \hat{\alpha} \Big\} \\
\qquad \text{where } \hat{e}[\![\texttt{f}]\!] = \langle \hat{c}', \hat{k}', \ldots \rangle \\
\qquad \text{and} \quad \hat{c}' = \langle \hat{\alpha}, \hat{b}' \rangle \\
\quad \vdots
\end{array}
$$

Figure 23: The Semantic Function $\mathcal{S}_4$ (Part I)

$$
\begin{array}{l}
\vdots \\
\text{and} \quad \hat{q_k} = \sqcup_{\hat{Q}} \Big\{ \lambda\beta.\ \text{if } \beta \neq \mathit{Container}\ j \\
\qquad\qquad\qquad\qquad \text{then } \bot_{\hat{T}} \\
\qquad\qquad\qquad\qquad \text{else } \mathit{Move}_{\hat{T}} \langle \hat{p}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \hat{b}', \\
\qquad\qquad\qquad\qquad\qquad\qquad \hat{e}', \\
\qquad\qquad\qquad\qquad\qquad\qquad \hat{r}\beta, \\
\qquad\qquad\qquad\qquad\qquad\qquad \hat{r}[\hat{k} // \mathit{Container}\ i] \rangle (\hat{\mathit{Inv}}\ \hat{p}') \\
\qquad\qquad\qquad\qquad\quad \text{where } \hat{e}' = \text{if } S_i = [\![\texttt{(set! x (f } \mathtt{y}_1 \cdots \mathtt{y}_m \texttt{))}]\!] \\
\qquad\qquad\qquad\qquad\qquad\quad \text{then } \hat{e}[\hat{e}[\![\mathtt{y}_1]\!] // [\![\mathtt{z}]\!]] \\
\qquad\qquad\qquad\qquad\qquad\quad \text{else } \hat{e}[\langle \bot_{\hat{C}}, \langle \{i\}, \hat{b}, \lambda\alpha.\{\epsilon\} \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad \bot_{\hat{Prim}Op}, \bot_{\hat{Int}}, \bot_{\hat{Bool}} \rangle // [\![\mathtt{z}]\!]] \\
\qquad\qquad\quad \text{where } S_j = [\![\texttt{(set! z (call/cc g))}]\!] \\
\qquad\qquad\quad \Big| \ j \in \hat{j} \Big\} \\
\qquad \text{where } \hat{e}[\![\mathtt{f}]\!] = \langle \hat{c}', \hat{k}', \ldots \rangle \\
\qquad \text{and } \ \hat{k}' = \langle \hat{j}, \hat{b}', \hat{p}' \rangle
\end{array}
$$

Figure 24: The Semantic Function $\mathcal{S}_4$ (Part II)

$$
\begin{aligned}
S_i = [\![&\texttt{(set! f (lambda (x}_1 \ldots \texttt{x}_m\texttt{) <x}_{m+1} \cdots \texttt{x}_n\texttt{> } \cdots\texttt{))}]\!] \Rightarrow \\
&\mathcal{S}_4 i \hat{t} = \lambda\beta.\ \text{if } \mathit{Container}\ i \neq \beta \\
&\qquad \text{then } \bot_{\hat{T}} \\
&\qquad \text{else } \langle \hat{p}, \\
&\qquad\quad \hat{b}, \\
&\qquad\quad \hat{e}[\langle \langle \{\alpha\}, \hat{b} \rangle, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}} \rangle // [\![\texttt{f}]\!]], \\
&\qquad\quad \hat{k}, \\
&\qquad\quad \hat{r} \rangle
\end{aligned}
$$

$$
\begin{aligned}
S_i = [\![&\texttt{(if x (goto m) (goto n))}]\!] \Rightarrow \\
&\mathcal{S}_4 i \hat{t} = \lambda\beta.\ \text{if } \mathit{Container}\ i \neq \beta \\
&\qquad \text{then } \bot_{\hat{T}} \\
&\qquad \text{else } \hat{t}
\end{aligned}
$$

$$
\begin{aligned}
S_i = [\![&\texttt{(return x)}]\!] \Rightarrow \\
&\mathcal{S}_4 i \hat{t} = \sqcup_{\hat{Q}} \Big\{ \lambda\beta.\ \text{if } \mathit{Container}\ j \neq \beta \\
&\qquad\qquad \text{then } \bot_{\hat{T}} \\
&\qquad\qquad \text{else } \mathit{Move}_{\hat{T}} \langle \hat{p} \\
&\qquad\qquad\qquad\qquad \hat{b}', \\
&\qquad\qquad\qquad\qquad \hat{e}[\hat{e}[\![\texttt{x}]\!] // [\![\texttt{y}]\!]], \\
&\qquad\qquad\qquad\qquad \hat{r}(\mathit{Container}\ j), \\
&\qquad\qquad\qquad\qquad \hat{r} \rangle (\hat{Inv}\ \hat{p}') \\
&\qquad\quad \text{where } S_j = [\![\texttt{(set! y } \cdots\texttt{)}]\!] \\
&\qquad\quad \Big| \ j \in \hat{j} \Big\} \\
&\text{where } \hat{k} = \langle \hat{j}, \hat{b}', \hat{p}' \rangle
\end{aligned}
$$

Figure 25: The Semantic Function $\mathcal{S}_4$ (Part III)

$$\mathcal{S}_4^* : \hat{Q} \to \hat{Q} \equiv \lambda\hat{q}.\ \sqcup_{\hat{Q}} \{\mathcal{S}_4 i(\hat{q}(\mathit{Container}\ i)) \mid i \in N\}$$

$$
\begin{aligned}
\mathcal{E}_4 : Q \to Q \equiv \lambda\hat{q}.\ &\text{Let } \hat{q}' = \mathcal{S}_4^* \hat{q} \\
&\text{in} \quad \text{if } \hat{q}' \sqsubseteq_{\hat{Q}} \hat{q} \text{ then } \hat{q} \text{ else } \mathcal{E}_4(\hat{q} \sqcup_{\hat{Q}} \hat{q}')
\end{aligned}
$$

Figure 26: The Semantic Functions $\mathcal{S}_4^*$ and $\mathcal{E}_4$

$Abs_E ep \sqsubseteq_{\hat{E}} \hat{e}$, $Abs_K kp \sqsubseteq_{\hat{K}} \hat{k}$, and $Abs_R rp \sqsubseteq_{\hat{R}} \hat{r}$. We proceed, as in the definitions of $\mathcal{S}_2$ and $\mathcal{S}_4$, based upon the structure of S_i. Suppose that

$$S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_n\texttt{))}]\!] \text{ or } [\![\texttt{(set! x (call/cc f))}]\!],$$

and that

$$e\langle [\![\texttt{f}]\!], b[\![\texttt{f}]\!]\rangle = \langle \alpha, b'\rangle \in C.$$

Since $Abs_E ep \sqsubseteq_{\hat{E}} \hat{e}$, we may write (following the form of $\mathcal{S}_4$)

$$\hat{e}[\![\texttt{f}]\!] = \langle \hat{c'}, \hat{k'}, \ldots\rangle \in \hat{C},$$

where $\hat{c'} = \langle \hat{\alpha}, \hat{b'}\rangle$, $\alpha \in Conc_{\hat{A}} \hat{\alpha}$, and $Abs_B b'p \sqsubseteq_{\hat{B}} \hat{b'}$. We consider the components of $\mathcal{S}_2 q$ (see the definition of $\mathcal{S}_2$ in Figures 7 and 8). We have that

$$\langle i'', p'', b'', e'', k'', o'', r''\rangle \in Conc_{\hat{Q}} \hat{q}$$

when

$$\hat{q}(Container\ i) = \langle \hat{p''}, \hat{b''}, \hat{e''}, \hat{k''}, \hat{o''}, \hat{r''}\rangle$$

where $Abs_P p'' \sqsubseteq_{\hat{P}} \hat{p''}$, $Abs_P b''p'' \sqsubseteq_{\hat{B}} \hat{b''}$, $Abs_E e''p'' \in \hat{e''}$, $Abs_K k''p'' \sqsubseteq_{\hat{K}} \hat{k''}$, and $Abs_R r''p'' \sqsubseteq_{\hat{R}} \hat{r''}$. Letting

$$q' = \mathcal{S}_2 q = \langle i'', p'', b'', e'', k'', o'', r''\rangle$$

and

$$\hat{t'} = \mathcal{S}_4 \hat{q} i(\hat{q}(Container\ i))(Container\ i'') = \langle \hat{p''}, \hat{b''}, \hat{e''}, \hat{k''}, \hat{r''}\rangle,$$

we must show that the concretization of each component of $\hat{t'}$ contains the corresponding component of q'.

By the definitions of $\mathcal{S}_2$ and $\mathcal{S}_4$, $p'' = p + \alpha^d$, and $\hat{p''} = \hat{p} \oplus Abs_P(\alpha^d)$, and by Theorem 11, we have that $p'' \in Conc_{\hat{P}} \hat{p''}$.

By the definition of $\mathcal{S}_2$,

$$b'' = b'[p + \alpha^d / [\![\texttt{z}_1]\!]] \cdots [p + \alpha^d / [\![\texttt{z}_n]\!]],$$

and by the definition of $\mathcal{S}_4$,

$$\hat{b''} = (Move_{\hat{B}} \hat{b} \hat{p'})[\lambda\alpha.\{\epsilon\} / [\![\texttt{z}_1]\!]] \cdots [\lambda\alpha.\{\epsilon\} / [\![\texttt{z}_n]\!]].$$

where $\hat{p'} = Abs_P(\alpha^d)$. Then by the definition of Abs_B,

$$\begin{aligned}(Abs_B b''(p+\alpha^d))[\![\mathtt{z}_i]\!] &= Abs_P((p+\alpha^d) - b''[\![\mathtt{z}_i]\!]) \\ &= Abs_P((p+\alpha^d) - (p+\alpha^d)) \\ &= Abs_P(\epsilon) \\ &= \lambda\alpha.\{\epsilon\}\end{aligned}$$

and $\lambda\alpha.\{\epsilon\} \sqsubseteq_{\hat{P}} \hat{b''}[\![\mathtt{z}_i]\!]$, for $1 \leq i \leq n$.

Now let $\mathtt{w} \in V$, $\mathtt{w} \neq \mathtt{z}_i$, $1 \leq i \leq n$. By the fact that $Abs_B b'p \sqsubseteq_{\hat{P}} \hat{b'}$, we have that $(Abs_B b'p)[\![\mathtt{w}]\!] = Abs_P(p - b'[\![\mathtt{w}]\!]) \sqsubseteq_{\hat{P}} \hat{b'}[\![\mathtt{w}]\!]$. By the definition of Abs_B,

$$\begin{aligned}(Abs_B b''(p+\alpha^d))[\![\mathtt{w}]\!] &= Abs_P((p+\alpha^d) - b''[\![\mathtt{w}]\!]) \\ &= Abs_P((p+\alpha^d) - b'[\![\mathtt{w}]\!]).\end{aligned}$$

By the definition of $\mathcal{S}_4$,

$$\begin{aligned}\hat{b''}[\![\mathtt{w}]\!] &= (Move_{\hat{B}}\hat{b'}(Abs_P(\alpha^d)))[\![\mathtt{w}]\!] \\ &= (\hat{b'}[\![\mathtt{w}]\!]) \oplus (Abs_P(\alpha^d)).\end{aligned}$$

By the definition of $+$ and $-$,

$$(p+\alpha^d) - b'[\![\mathtt{w}]\!] = (p - b'[\![\mathtt{w}]\!]) + \alpha^d$$

and therefore

$$\begin{aligned}&(Abs_B b''(p+\alpha^d))[\![\mathtt{w}]\!] \quad = \\ &Abs_P((p+\alpha^d) - b'[\![\mathtt{w}]\!]) \sqsubseteq_{\hat{P}} \hat{b'}[\![\mathtt{w}]\!] \oplus Abs_P(\alpha^d) \\ &\qquad\qquad = \hat{b''}[\![\mathtt{w}]\!]\end{aligned}$$

for all $\mathtt{w} \in V$, $\mathtt{w} \neq \mathtt{z}_i$, $1 \leq i \leq n$, and therefore

$$b'' \in Conc_{\hat{B}}\hat{b''}\hat{p''}.$$

Assume for the moment that $S_i = [\![\texttt{(set! x (f } \mathtt{y}_1 \cdots \mathtt{y}_m\texttt{))}]\!]$, and let

$$\hat{e'} = \hat{e}[[\![\mathtt{y}_1]\!]//[\![\mathtt{z}_1]\!]] \cdots [[\![\mathtt{y}_m]\!]//[\![\mathtt{z}_m]\!]].$$

Then, by the definition of $\mathcal{S}_2$,

$$\begin{aligned}e'' = e[&e\langle[\![\mathtt{y}_1]\!], b[\![\mathtt{y}_1]\!]\rangle/\langle[\![\mathtt{z}_1]\!], p+\alpha^d\rangle] \cdots \\ &[e\langle[\![\mathtt{y}_m]\!], b[\![\mathtt{y}_m]\!]\rangle/\langle[\![\mathtt{z}_m]\!], p+\alpha^d\rangle],\end{aligned}$$

and

$$\hat{e''} = Move_{\hat{E}}\hat{e'}(Abs_P(\alpha^d)).$$

(We must show that $Abs_E e''(p + \alpha^d) \sqsubseteq_{\hat{E}} \hat{e''}$.) We have that $e \in Conc_{\hat{E}}\hat{e}\hat{p}$, and by the definition of the notation $f[x//y]$, we have that $e'' \in Conc_{\hat{E}}\hat{e'}\hat{p}$ or $(Abs_E e''p) \sqsubseteq_{\hat{E}} \hat{e'}$, since $p \in Conc_{\hat{P}}\hat{p}$ by assumption.

$$\hat{e''}[\![\mathtt{w}]\!] = Move_{\hat{D}}(\hat{e'}[\![\mathtt{w}]\!])(Abs_P(\alpha^d))$$

by the definition of $Move_{\hat{E}}$.

$$Abs_D(e''\langle[\![\mathtt{w}]\!], s\rangle)p \sqsubseteq_{\hat{D}} \hat{e'}[\![\mathtt{w}]\!]$$

for all s, by the fact that $(Abs_E e''p) \sqsubseteq_{\hat{E}} \hat{e'}$. We consider the case in which $e''\langle[\![\mathtt{w}]\!], s\rangle$ is a closure, and in which it is a continuation.

I. $e''\langle[\![\mathtt{w}]\!], s\rangle = \langle\alpha, b\rangle \in C$ for some $s \in P$, $w \in V$. We have that $Abs_D\langle\alpha, b\rangle p \sqsubseteq_{\hat{D}} \hat{e'}[\![\mathtt{w}]\!]$. Let $\hat{e}[\![\mathtt{w}]\!] = \langle\hat{c}, \ldots\rangle \in \hat{D}$ where $\hat{c} = \langle\hat{\alpha}, \hat{b}\rangle$. Then

$$Abs_C\langle\alpha, b\rangle p = \langle\{\alpha\}, \lambda v.Abs_P(p - bv)\rangle \sqsubseteq_{\hat{C}} \langle\hat{\alpha}, \hat{b}\rangle.$$

This means that

$$Abs_P(p - bv) \sqsubseteq_{\hat{P}} \hat{b}v, \text{ for all } v \in V$$

$$Abs_P(p - bv) \oplus Abs_P(\alpha^d) \sqsubseteq_{\hat{P}} \hat{b}v \oplus Abs_P(\alpha^d) \text{ for all } v \in V$$

$$Abs_P((p - bv) + \alpha^d) \sqsubseteq_{\hat{P}} \hat{b}v \oplus Abs_P(\alpha^d) \text{ for all } v \in V$$

$$Abs_P((p + \alpha^d) - bv) \sqsubseteq_{\hat{P}} \hat{b}v \oplus Abs_P(\alpha^d) \text{ for all } v \in V$$

$$\lambda v.Abs_P((p + \alpha^d) - bv) \sqsubseteq_{\hat{B}} \lambda v.(\hat{b}v \oplus Abs_P(\alpha^d))$$

$$Abs_B b(p + \alpha^d) \sqsubseteq_{\hat{B}} Move_{\hat{B}}\hat{b}(Abs_P(\alpha^d))$$

and therefore

$$b \in Conc_{\hat{B}}\hat{b}\hat{p''}.$$

This implies that

$$Abs_C\langle\alpha, b\rangle(p + \alpha^d) \sqsubseteq_{\hat{C}} Move_{\hat{C}}\langle\hat{\alpha}, \hat{b}\rangle(Abs_P(\alpha^d))$$

and therefore that

$$Abs_D\langle\alpha, b\rangle(p + \alpha^d) \sqsubseteq_{\hat{D}} Move_{\hat{D}}(\hat{e'}[\![\mathtt{w}]\!])(Abs_P(\alpha^d)).$$

II. $e''\langle[\![\mathtt{w}]\!], s\rangle = \langle j, b, p, o\rangle \in K$. We have that $Abs_D\langle j, b', p', o'\rangle p \sqsubseteq_{\hat{D}} \hat{e'}[\![\mathtt{w}]\!]$. Let $\hat{e}[\![\mathtt{w}]\!] = \langle \hat{c}, \hat{k}, \ldots\rangle$ where $\hat{k} = \langle \hat{j}, \hat{b'}, \hat{p'}\rangle$. Then

$$Abs_K\langle j, b', p', o'\rangle p = \langle \{j\}, \lambda v.Abs_P(p - b'v), Abs_P(p - p')\rangle \sqsubseteq_{\hat{K}} \langle \hat{j}, \hat{b'}, \hat{p'}\rangle.$$

This means that

$$Abs_P(p - b'v) \sqsubseteq_{\hat{P}} \hat{b'}v, \text{ for all } v \in V$$

$$Abs_P(p - b'v) \oplus Abs_P(\alpha^d) \sqsubseteq_{\hat{P}} \hat{b'}v \oplus Abs_P(\alpha^d) \text{ for all } v \in V$$

$$Abs_P((p - b'v) + \alpha^d) \sqsubseteq_{\hat{P}} \hat{b'}v \oplus Abs_P(\alpha^d) \text{ for all } v \in V$$

$$Abs_P((p + \alpha^d) - b'v) \sqsubseteq_{\hat{P}} \hat{b'}v \oplus Abs_P(\alpha^d) \text{ for all } v \in V$$

$$\lambda v.Abs_P((p + \alpha^d) - b'v) \sqsubseteq_{\hat{B}} \lambda v.(\hat{b'}v \oplus Abs_P(\alpha^d))$$

$$Abs_B b'(p + \alpha^d) \sqsubseteq_{\hat{B}} Move_{\hat{B}}\hat{b'}(Abs_P(\alpha^d))$$

and therefore

$$b' \in Conc_{\hat{B}}\hat{b'}\hat{p''}.$$

Likewise, this means that

$$Abs_P(p - p') \sqsubseteq_{\hat{P}} \hat{p'}$$

$$Abs_P(p - p') \oplus Abs_P(\alpha^d) \sqsubseteq_{\hat{P}} \hat{p'} \oplus Abs_P(\alpha^d)$$

$$Abs_P((p - p') + \alpha^d) \sqsubseteq_{\hat{P}} \hat{p'} \oplus Abs_P(\alpha^d)$$

$$Abs_P((p + \alpha^d) - p') \sqsubseteq_{\hat{P}} \hat{p'} \oplus Abs_P(\alpha^d).$$

This implies that

$$Abs_K\langle j, b', p', o'\rangle(p + \alpha^d) \sqsubseteq_{\hat{K}} Move_{\hat{K}}\langle j, \hat{b'}, \hat{p'}\rangle(Abs_P(\alpha^d))$$

and therefore that

$$Abs_D\langle j, b', p', o'\rangle(p + \alpha^d) \sqsubseteq_{\hat{D}} Move_{\hat{D}}(\hat{e'}[\![\mathtt{w}]\!])(Abs_P(\alpha^d)).$$

Since we have that

$$Abs_D(e''\langle[\![\mathtt{w}]\!], s\rangle)(p + \alpha^d) \sqsubseteq_{\hat{D}} Move_{\hat{D}}(\hat{e'}[\![\mathtt{w}]\!])(Abs_P(\alpha^d))$$

for any choice of `w` and `s`, we have that

$$Abs_E e''(p + \alpha^d) \sqsubseteq_{\hat{E}} Move_{\hat{E}}\hat{e'}(Abs_P(\alpha^d)) = \hat{e''}$$

and

$$e'' \in Conc_{\hat{E}}\hat{e''}\hat{p''}$$

as desired.

Next we consider the case that $S_i = [\![\texttt{(set! x (call/cc f))}]\!]$; but this is covered by case 2 above, by letting $[\![\mathtt{w}]\!] = [\![\mathtt{z}]\!]$.

By the definition of $\mathcal{S}_2$, $k'' = \langle i, b, p, o\rangle$, and by the definition of $\mathcal{S}_4$,

$$\hat{k''} = Move_{\hat{K}}\langle\{i\}, \hat{b}, \lambda\alpha.\{\epsilon\}\rangle(Abs_P(\alpha^d)).$$

It is obvious that $i \in Conc_{\hat{\Lambda}}\{i\}$, and we showed above that

$$b \in Conc_{\hat{B}}(Move_{\hat{B}}\hat{b}(Abs_P(\alpha^d)))\hat{p''}.$$

Certainly $p - p \in Conc_{\hat{P}}(\lambda\alpha.\{\epsilon\})$ and therefore

$$(p + \alpha^d) - p = \alpha^d \in Conc_{\hat{P}}((\lambda\alpha.\{\epsilon\}) \oplus (Abs_P(\alpha^d))),$$

since $(\lambda\alpha.\{\epsilon\}) \oplus (Abs_P(\alpha^d))$ maps β to $\{\epsilon\}$ for all $\beta \neq \alpha$, and maps α to $\{\mathbf{d}\}$, by the definition of $\oplus$. Therefore

$$k'' \in Conc_{\hat{K}}\hat{k''}\hat{p''}.$$

Finally, by the definition of $\mathcal{S}_2$, $r'' = r[k/o]$, and

$$\hat{r''} = Move_{\hat{R}}\hat{r'}(Abs_P(\alpha^d)),$$

where $\hat{r'} = \hat{r}[\hat{k}//(Container\ i)]$. We are given that $r \in Conc_{\hat{R}}\hat{r}\hat{p}$, and since $k \in Conc_{\hat{K}}\hat{k}\hat{p}$ by assumption,

$$r'' \in Conc_R(\hat{r}[\hat{k}//(Container\ i)])\hat{p}.$$

Let

$$k' = \langle i', b', p', o'\rangle = r''s$$

for some $s = \cdots \beta^d \in P$, and let

$$\hat{k'} = \langle \hat{i'}, \hat{b'}, \hat{p'} \rangle = \hat{r'}\beta.$$

By the fact that $r'' \in Conc_{\hat{R}} \hat{r'} \hat{p}$, we have that $i' \in Conc_{\hat{\Lambda}} \hat{i'}$, and by arguments we have made above,

$$b' \in Conc_{\hat{B}}(Move_{\hat{B}} \hat{b'}(Abs_P(\alpha^d)))\hat{p''}$$

and

$$(p + \alpha^d) - p' \in Conc_{\hat{P}}(\hat{p'} \oplus (Abs_P(\alpha^d))).$$

Therefore

$$k'' \in Conc_{\hat{K}}(Move_{\hat{K}} \hat{k'}(Abs_P(\alpha^d)))\hat{p''}$$

for all choices of s, and

$$r'' \in Conc_{\hat{R}}(Move_{\hat{R}} \hat{r'}(Abs_P(\alpha^d))) = Conc_{\hat{R}} \hat{r''} \hat{p''}.$$

We therefore have, in the case that $e\langle [\![\mathtt{f}]\!], b[\![\mathtt{f}]\!] \rangle \in C$, the result that

$$\mathcal{S}_2 q \in Conc_{\hat{Q}}(\mathcal{S}_4 i(\hat{q}(Container\ i))).$$

Similar arguments prove the theorem in the case where $e\langle [\![\mathtt{f}]\!], b[\![\mathtt{f}]\!] \rangle \in K$, and for other forms of S_i. □

Theorem 23 is the analogue of Theorem 18, and shows that a single step of abstract interpretation under $\mathcal{S}_4$, preserves the corresponding concrete evaluation step under $\mathcal{S}_2$. To complete the proof of correctness of $\mathcal{E}_4$, we observe simply that Theorem 19 applies directly to $\mathcal{E}_4$, since $\mathcal{E}_3$ and $\mathcal{E}_4$ are identical (modulo their respective invocations of $\mathcal{S}_3^*$ and $\mathcal{S}_4^*$). Likewise, the result of Theorem 14 applies to $\mathcal{S}_4$, because our alterations to $\mathcal{S}_3$ have obviously not affected its monotonicity. We may therefore rewrite Theorems 15 and 16 in terms of $\mathcal{S}_4^*$ and $\mathcal{E}_4$, and thereby show that evaluation under $\mathcal{E}_4$ always terminates. (We will henceforth invoke Theorems 19, 14, 15 and 16 with the understanding that they apply directly to $\mathcal{E}_4$ and its auxiliary functions.)

We're getting closer; $\mathcal{E}_4$ is not complete (we will improve its accuracy by one further modification, shortly), but it captures the essentials of program analysis based upon stack configurations. Let us now revisit each of our data flow problems, and see how their solutions are computed by $\mathcal{E}_4$.

2.11.1 Side-Effects under $\mathcal{E}_4$

Theorem 6 characterizes side-effects in terms of the procedure strings constructed during evaluation under $\mathcal{E}_2$. It holds that if $\dot{\mathtt{x}}$ is an instance of the variable $\mathtt{x}$ the birth date of which is p_b, and if p_r is the procedure string of a state in which $\dot{\mathtt{x}}$ is referenced, then an instance $\dot{\lambda}_\alpha$ of λ_α has a side-effect as a result of the reference, if and only if $Net(p_r - p_b)$ contains a term α^d corresponding to $\dot{\lambda}_\alpha$.

In Theorem 20, we cast this result into the realm of stack configurations constructed by evaluation under $\mathcal{E}_3$. We must now do the same for the stack configurations constructed by $\mathcal{E}_4$. The following theorem is the key.

Theorem 24 *Let $q_o, q_1, \ldots$ be the state sequence described by $\mathcal{E}_2 q_0$, let $\hat{q_n} = \mathcal{E}_4\ \hat{q_0}$ where $q_0 \in Conc_{\hat{Q}} \hat{q_0}$, let $\dot{\mathtt{x}}$ be an instance of the variable $\mathtt{x}$ during $\mathcal{E}_2 q_0$ whose birth date is p_b, and let $\dot{\mathtt{x}}$ be referenced directly within λ_γ in a state $q_r = \langle i, p_r, b, e, k, r, o\rangle$. Then*

$$Net(p_r - p_b) \in Conc_{\hat{P}}(\hat{b}[\![\mathtt{x}]\!])$$

where $\hat{q_n}\gamma = \langle \hat{p_r}, \hat{b}, \ldots\rangle \in \hat{T}$.

Proof: By Theorem 19,

$$p_r \in Conc_{\hat{P}} \hat{p_r}$$

and therefore by Theorem 22,

$$Abs_B b p_r \sqsubseteq_{\hat{B}} \hat{b}.$$

By the definition of Abs_B,

$$(Abs_B b p_r) = \lambda v. Abs_P(p_r - bv)$$

and therefore

$$Abs_P(p_r - b[\![\mathtt{x}]\!]) \in Conc_{\hat{P}}(\hat{b}[\![\mathtt{x}]\!])$$

and

$$Abs_P(p_r - p_b) \in Conc_{\hat{P}}(\hat{b}[\![\mathtt{x}]\!])$$

by choice of $p_b = b[\![\mathtt{x}]\!]$. □

By this result, we may write the following simple theorem that characterizes side-effects under $\mathcal{E}_4$.

Theorem 25 *Let* $\hat{q_n} = \mathcal{E}_4\hat{q_0}$ *where* $q_0 \in Conc_{\hat{Q}}\hat{q_0}$. *If during the evaluation of* $\mathcal{E}_2 q_0$ *there is an instance* $\dot{\lambda_\alpha}$ *of* λ_α *that has a side-effect upon an instance* $\dot{x}$ *of* x, *then there exists a* $\gamma \in \Lambda$ *such that*

$$(\hat{b}[\![\mathbf{x}]\!])\alpha \cap \{\mathbf{d}, \mathbf{dd}^+, \mathbf{u}^+\mathbf{d}^+\} \neq \{\}$$

where $\hat{q_n}\gamma = \langle \hat{p_r}, \hat{b}, \ldots \rangle \in \hat{T}$, *and* $\mathbf{x}$ *is referenced directly within* λ_γ.

Proof: Let $\dot{\lambda_\alpha}$ be an instance of λ_α that has a side-effect upon an instance $\dot{\mathbf{x}}$ of $\mathbf{x}$, let this side-effect arise from a reference to $\dot{\mathbf{x}}$ (directly) within λ_γ, and let p_b be the birth date of $\dot{\mathbf{x}}$. By Theorem 6, there must be a state q_r whose procedure string is p_r, in which $\dot{\mathbf{x}}$ is referenced directly by λ_γ (which reference gives rise to the side-effect attributed to λ_α), such that $Net(p_r - p_b)$ contains a term α^d corresponding to $\dot{\lambda_\alpha}$. By Theorem 20, $q_r \in Conc_{\hat{Q}}\hat{q_n}$, where $\hat{q_n}$ is as defined by the current theorem. Let $\hat{q_n}\gamma = \langle \hat{p_r}, \hat{b} \ldots \rangle$. Then, by Theorem 24,

$$Net(p_r - p_b) \in Conc_{\hat{P}}(\hat{b}[\![\mathbf{x}]\!]).$$

By the form of $Net(p_r - p_b)$,

$$Trace(Net(p_r - p_b))\alpha = \alpha^{a_1} \cdots \alpha^{a_k},$$

where $a_i = d$ for some $1 \leq i \leq k$. By the definition of $Conc_{\hat{P}}$, this implies that

$$(\hat{b}[\![\mathbf{x}]\!])\alpha \cap \{\mathbf{d}, \mathbf{dd}^+, \mathbf{u}^+\mathbf{d}^+\} \neq \{\}.$$

□

What could be easier? At compile-time, we compute $\hat{q_n} = \mathcal{E}_4\hat{q_0}$, where $\hat{q_0}$ represents every initial state from which the program might execute. Afterwards, we notice that λ_γ makes a reference to a mutable variable $\mathbf{x}$, and we wonder what procedures in the program might have a side-effect (by Definition 1) as a result of the reference. The answer is contained in $\hat{b}[\![\mathbf{x}]\!]$, where $\hat{q_n}\gamma = \langle \hat{p_r}, \hat{b} \ldots \rangle \in \hat{T}$. If $(\hat{b}[\![\mathbf{x}]\!])\alpha$ contains none of $\mathbf{d}$, $\mathbf{dd}^+$, or $\mathbf{u}^+\mathbf{d}^+$, then no instance of λ_α has a side-effect as a result of this reference. If $(\hat{b}[\![\mathbf{x}]\!])\alpha$ contains one of $\mathbf{d}$, $\mathbf{dd}^+$, or $\mathbf{u}^+\mathbf{d}^+$, there may be an instance of λ_α that has a side-effect as a result of this reference. This uncertainty is the cost of abstraction. We have chosen to err in favor of over-estimation of side-effects, for the reason that parallelizing transformations are inhibited by side-effects, and it is always safe (correct) to inhibit a transformation.

2.11.2 Stack-Allocation under $\mathcal{E}_4$

We may likewise translate our reasoning about the stack-allocation of variables into the terms of $\mathcal{E}_4$. The exact conditions under which a variable instance must be heap-allocated are given in Theorem 8. It holds that if $\dot{\mathbf{x}}$ is an instance of $\mathbf{x}$ bound by an instance $\dot{\lambda_\beta}$ of λ_β, where p_b is the birth date of $\dot{\mathbf{x}}$, p_r is the procedure string of a state in which reference is made to $\dot{\mathbf{x}}$, and $Net(p_r - p_b) = \ldots \beta^u \ldots$, then $\dot{\lambda_\beta}$ is deactivated before this reference takes place (and therefore $\dot{\mathbf{x}}$ must be allocated in the heap, assuming that heap and stack are the only alternatives).

Theorem 21 is the abstraction of this result to the stack configurations constructed by evaluation under $\mathcal{E}_3$, and we repeat the exercise now, for the case of $\mathcal{E}_4$.

Theorem 26 *Let $\hat{q_n} = \mathcal{E}_4 \hat{q_0}$, where $q_0 \in Conc_{\hat{Q}} \hat{q_0}$. If during the evaluation of $\mathcal{E}_2 q_0$ there is an instance $\dot{\mathbf{x}}$ of the variable $\mathbf{x}$ such that $\dot{\mathbf{x}}$ is referenced following the deactivation of the instance of the procedure λ_β that binds it, then there exists a $\gamma \in \Lambda$ such that*

$$(\hat{b}[\![\mathbf{x}]\!])\beta \cap \{\mathbf{u}, \mathbf{uu}^+, \mathbf{u}^+\mathbf{d}^+\} \neq \{\},$$

where $\hat{q_n}\gamma = \langle \hat{p_r}, \hat{b}, \ldots \rangle \in \hat{T}$, and $[\![\mathbf{x}]\!]$ is referenced (directly) within λ_γ.

Proof: Let $\dot{\lambda_\beta}$ be the instance of λ_β that binds an instance $\dot{\mathbf{x}}$ of $\mathbf{x}$, such that $\dot{x}$ is referenced in a state q_r, following the deactivation of $\dot{\lambda_\beta}$. Let p_r be the procedure string of q_r, and let p_b be the birth date of $\dot{\mathbf{x}}$. By Theorem 8, $Net(p_r - p_b) = \cdots \beta^u \cdots$, where β^u corresponds to the deactivation of $\dot{\lambda_\beta}$. Let λ_γ be the procedure within which $\dot{\mathbf{x}}$ is referenced directly in q_r, and let $\hat{q_n}\gamma = \langle \hat{p_r}, \hat{b}, \ldots \rangle$. Then, by Theorem 24,

$$p_r - p_b \in Conc_{\hat{P}}(\hat{b}[\![\mathbf{x}]\!]).$$

By the form of $Net(p_r - p_b)$,

$$Trace(Net(p_r - p_b))\beta = \beta^{a_1} \cdots \beta^{a_k},$$

where $a_i = u$ for some $1 \leq i \leq k$. By the definition of $Conc_{\hat{P}}$, this implies that

$$(\hat{b}[\![\mathbf{x}]\!])\beta \cap \{\mathbf{u}, \mathbf{uu}^+, \mathbf{u}^+\mathbf{d}^+\} \neq \{\}.$$

□

This suggests a simple compile-time procedure for partitioning variables into those that may be instantiated on the stack, and those that must be

instantiated in the heap. We perform the abstract interpretation $\hat{q_n} = \mathcal{E}_4\hat{q_0}$, where $\hat{q_0}$ is representative of the starting states of the program being compiled. We then notice that λ_γ makes a reference to $\mathbf{x}$, and we wish to know whether $\mathbf{x}$ need be heap-allocated as a result of (an instance of) this reference. Again, the answer is contained in $\hat{b}[\![\mathbf{x}]\!]$, where $\hat{q_n}\gamma = \langle \hat{p_r}, \hat{b} \ldots \rangle \in \hat{T}$. If $(\hat{b}[\![\mathbf{x}]\!])\beta$ contains none of $\mathbf{u}$, $\mathbf{uu}^+$, or $\mathbf{u}^+\mathbf{d}^+$, then for every instance $\dot{\mathbf{x}}$ of $\mathbf{x}$, each reference to $\dot{\mathbf{x}}$ occurs while the instance of λ_β that binds it is still active; this implies that all instances of $\mathbf{x}$ may be allocated on the stack. Otherwise, if $(\hat{b}[\![\mathbf{x}]\!])\beta$ contains one of $\mathbf{u}, \mathbf{uu}^+, \mathbf{u}^+\mathbf{d}^+$, we can say only that there might be an instance $\dot{\mathbf{x}}$ that is referenced following deactivation of the instance of λ_β that binds it.

2.11.3 Generalized Hierarchical Storage Management

Finally, we may repeat the abstraction of our reasoning about hierarchical storage management in the case of $\mathcal{E}_4$, exactly as we did for side-effects and stack allocation above. Again, the role under $\mathcal{E}_3$of $\hat{p_r} \ominus \hat{p_b}$, where $\hat{p_b}$ and $\hat{p_r}$ mark points at which a variable $\mathbf{x}$ is instantiated and referenced, respectively, is played under $\mathcal{E}_4$ by $\hat{b}[\![\mathbf{x}]\!]$, where λ_γ is a lambda expression within which $\mathbf{x}$ is referenced directly, and $\hat{b}$ is the second component of $(\mathcal{E}_4\hat{q_0})\gamma$.

2.12 Adding Flow-Sensitivity to the Analysis

Let us consider the example of Figure 20 again, in light of $\mathcal{E}_4$. Recall that the difficulty, under $\mathcal{E}_3$, is that the difference between the stack configuration that represents a reference to $\mathbf{x}$ within λ_γ, and the stack configuration that represents the point of $\mathbf{x}$'s instantiation, is so crudely approximated by $\ominus$ as to yield values near to $\perp_{\hat{P}}$ even when the arguments to $\ominus$ are relatively accurate. We addressed this problem by eliminating the use of $\ominus$ in $\mathcal{E}_4$ altogether, instead computing the difference of these two stack configurations directly within the semantic functions.

We have another problem, however. Let $\hat{q_n} = \mathcal{E}_4\,\hat{q_0}$ be the result of abstract interpretation of the program of Figure 20 under $\mathcal{E}_4$. Consider the value $\hat{d} = \hat{e}[\![\mathtt{t_3}]\!]$, where $\hat{q_n}\alpha = \langle \hat{p}, \hat{b}, \hat{e}, \hat{k}, \hat{r} \rangle$. Let $\hat{d} = \langle \langle \hat{\alpha}, \hat{b'} \rangle, \ldots \rangle$. The non-bottom values in $Conc_{\hat{D}}\hat{d}$ are closures of λ_γ, and therefore $\hat{\alpha} = \{\gamma\}$. Let $\hat{p'} = \hat{b'}[\![\mathbf{x}]\!]$. Recall that $\hat{p'}$ is a record of the interprocedural movements described by an instance of $\mathbf{x}$, from the point of its instantiation to the current state. Since no interprocedural movements take place between the closure of λ_γ and the assignment of the closure to $\mathtt{t_3}$ in statement S_5, $\hat{p'}$ is "primed" such that $\epsilon \in \hat{p'}\alpha$. At S_8, a recursive instance of λ_α is applied, and the value $\hat{d}$ makes a downward movement with respect to λ_α, before being

```
(define fact (lambda (n k)
   (if (= n 0)
       (k 1)
       (fact (1- n) (lambda (m) (k (* n m)))))))
```

Figure 27: Example of Overlapping Variable Lifetimes

joined with its previous value. (Recall that there is only one environment for all instances of λ_α, and at every evaluation of S_5, the value of $\mathtt{t_3}$ in this environment is "raised" in the lattice $\hat{D}$.) Therefore $\mathbf{d} \in \hat{p}'\alpha$. At S_{10}, a return from λ_α takes place, causing the value of $\hat{d}$ to make an upward movement before it is once more joined with its previous value. At this point, $\hat{p}'\alpha$ contains ϵ, $\mathbf{d}$, and $\mathbf{u}$. Repeating this cycle of call and return, we have that $\hat{p}'\alpha = \{\mathbf{dd}^+, \mathbf{d}, \epsilon, \mathbf{u}, \mathbf{uu}^+, \mathbf{u}^+\mathbf{d}^+\} = \Delta$. Again, it appears that the instances of $\mathtt{x}$ must be heap-allocated.

What has gone wrong? The trouble is that $\mathcal{E}_4$ is *flow-insensitive*. It is obvious from looking at the program text of Figure 20 that the closure of λ_γ that is applied at S_6 is the very one that is assigned into $\mathtt{t_3}$ at S_5. This is missed by $\mathcal{E}_4$, which knows of only one instance of $\mathtt{t_3}$ (representing all the instances of $\mathtt{t_3}$ to that point). At each recursive application of λ_α at S_8, the current (abstract) value of $\mathtt{t_3}$ undergoes a downward movement. When the next instance of S_5 is evaluated, the new value of $\mathtt{t_3}$ does not overwrite the old value of $\mathtt{t_3}$, because $\mathcal{E}_4$ maintains only one environment per lambda expression (and thus multiple assignments to a variable must have the effect of raising its value in the lattice $\hat{D}$, and not of overwriting the variable's value in the environment), but more importantly because there *are* distinct instances of $\mathtt{t_3}$, and while in this example each dies almost immediately after being assigned, in other examples it might survive across recursive invocations of the procedure that binds them. Consider the definition of `fact` in Figure 10 (the definition is reproduced in Figure 27 for convenience). During the evaluation of `(fact 10 (lambda (x) x))` (under $\mathcal{E}_2$), there are 11 instances of `n` and `k` live simultaneously (consider that the first multiplication of `m` by `n` does not occur until the final call to `fact` has been made). There is only one "location" in each environment (under $\mathcal{E}_4$) for the instances of `n`; the abstraction of the values of these instances are therefore joined to produce a single, representative member of $\hat{D}$, and the introduction of a new instance simply raises the lattice value; it cannot overwrite the value of `n` in the environment.

What is needed to give $\mathcal{E}_4$ a bit of flow-sensitivity? We must inform the abstraction of environments with the notion of multiple instances of variables, but we must add as little information content to the abstraction

$$\hat{\Lambda} = 2^{\Lambda}$$
$$\hat{N} = 2^{N}$$
$$\hat{B} = V \to \hat{P}$$
$$\hat{C} = \hat{\Lambda} \times \hat{B}$$
$$\hat{K} = \hat{N} \times \hat{B} \times \hat{P}$$
$$\hat{D} = \hat{C} \times \hat{K} \times \hat{PrimOp} \times \hat{Int} \times \hat{Bool}$$
$$\hat{E} = V \times \Delta \to \hat{D}$$
$$\hat{R} = \Lambda \to \hat{K}$$
$$\hat{T} = \hat{P} \times \hat{B} \times \hat{E} \times \hat{K} \times \hat{R}$$
$$\hat{Q} = N \to \hat{T}$$

Figure 28: Abstract Domains for $\mathcal{E}_5$

as possible, since we will pay for any increase in complexity at compile-time. Our approach (one of many possibilities) is to partition the instances of a variable x into 6 classes (one for each member of Δ), according to the movements they describe, from their points of instantiation, relative to the procedure that binds x. If λ_α is the binder of x, and p is the procedure string that records the interprocedural movements described by $\dot{\mathtt{x}}$ (an instance of x) from its instantiation to the current state, then we will abstract $\dot{\mathtt{x}}$ to the pair $\langle [\![\mathtt{x}]\!], Dir\ p\alpha \rangle$. Just as we used a pair $\langle [\![\mathtt{x}]\!], p \rangle$, where $p \in P$, to index the environment under $\mathcal{E}_2$, so we will use a pair $\langle [\![\mathtt{x}]\!], \delta \rangle$, where $\delta \in \Delta$ to index the environment under our modified (and last) version of the abstract semantics, $\mathcal{E}_5$. In order to achieve a measure of flow-sensitivity, we associate an environment with each statement, rather than with each lambda expression, as in $\mathcal{E}_3$ and $\mathcal{E}_4$. The domain equations for $\mathcal{E}_5$ are presented in Figure 28. The only changes from those of Figure 12 are to the definitions of $\hat{E}$ and $\hat{Q}$. Under $\mathcal{E}_5$ we will have one environment per statement, and the environment will map pairs in $V \times \Delta$ onto abstract values in $\hat{D}$.

We said, in complaining about the inaccuracy of $\mathcal{E}_4$ in the case of Figure 20, that we wished to overwrite the value of $\mathtt{t}_3$ rather than to join it with a gradually less accurate value, when forming the closures of λ_γ. We have, as yet, no justification for doing so under $\mathcal{E}_5$, for the act of partitioning the instances of $\mathtt{t}_3$ into classes is, of itself, no help in this regard: if an equivalence class under the partition represents more than one instance of $\mathtt{t}_3$, then we will still be forced to join values together, to simulate the action of assignment in our abstract semantics. The following theorems come to our rescue.

Theorem 27 *Let* $\dot{\mathbf{x}}$ *and* $\ddot{\mathbf{x}}$ *be two instances of* $\mathbf{x}$*, a variable bound by* λ_α*, and let* p_1 *and* p_2 *be the birth dates of* $\dot{\mathbf{x}}$ *and* $\ddot{\mathbf{x}}$ *respectively. Let* p_3 *be the procedure string of a state following the instantiation of both* $\dot{\mathbf{x}}$ *and* $\ddot{\mathbf{x}}$*. Then*

$$Dir(p_3 - p_1)\alpha = Dir(p_3 - p_2)\alpha = \epsilon$$

implies that $p_1 = p_2$ *(and therefore that* $\dot{\mathbf{x}} = \ddot{\mathbf{x}}$*).*

Proof: Let $Dir(p_3 - p_1)\alpha = \epsilon$ and $Dir(p_3 - p_2)\alpha = \epsilon$, and suppose that $p_1 \neq p_2$. By the definition of *Dir*,

$$Net(Trace(p_3 - p_1)\alpha) = Net(Trace(p_3 - p_2)\alpha) = \epsilon.$$

Assume without loss of generality that p_1 is a prefix of p_2. Then

$$Net(Trace(p_3 - p_1)\alpha) = Net(Trace((p_2 - p_1) + (p_3 - p_2))\alpha) = \epsilon$$

and therefore

$$Net(Trace(p_2 - p_1)\alpha) = \epsilon.$$

But $p_2 = \cdots\alpha^d$, since it is the birth date of an instance of λ_α. This means that $p_2 - p_1 = \cdots\alpha^d$, and $Net(Trace(p_2 - p_1))\alpha \neq \epsilon$, a contradiction. □

Theorem 28 *Let* $\dot{\mathbf{x}}$ *and* $\ddot{\mathbf{x}}$ *be two instances of* $\mathbf{x}$*, a variable bound by* λ_α*, and let* p_1 *and* p_2 *be the birth date of* $\dot{\mathbf{x}}$ *and* $\ddot{\mathbf{x}}$ *respectively. Let* p_3 *be the procedure string of a state following the instantiation of both* $\dot{\mathbf{x}}$ *and* $\ddot{\mathbf{x}}$*. Then*

$$Dir(p_3 - p_1)\alpha = Dir(p_3 - p_2)\alpha = \mathbf{d}$$

implies that $p_1 = p_2$ *(and therefore that* $\dot{\mathbf{x}} = \ddot{\mathbf{x}}$*).*

Proof: Let $Dir(p_3 - p_1)\alpha = \mathbf{d}$ and $Dir(p_3 - p_2)\alpha = \mathbf{d}$, and suppose that $p_1 \neq p_2$. By the definition of *Dir*,

$$Net(Trace(p_3 - p_1)\alpha) = Net(Trace(p_3 - p_2)\alpha) = \alpha^d.$$

Assume without loss of generality that p_1 is a prefix of p_2. Then

$$Net(Trace(p_3 - p_1)\alpha) = Net(Trace((p_2 - p_1) + (p_3 - p_2))\alpha) = \alpha^d$$

and since the α^d within $p_3 - p_2$ is not annihilated by *Net*,

$$Net(Trace(p_2 - p_1)\alpha) = \epsilon.$$

But $p_2 = \cdots\alpha^d$, since it is the birth date of an instance of λ_α. This means that $p_2 - p_1 = \cdots\alpha^d$, and $Net(Trace(p_2 - p_1))\alpha \neq \epsilon$, a contradiction. □

These theorems can be understood best with the help of the following definition (for illustrative purposes only).

$$RdEnv\colon \hat{E} \to V \to 2^{\Delta} \to \hat{D}$$
$$\equiv \lambda\hat{e}.\lambda v.\lambda s.\sqcup_{\hat{D}} \{\hat{e}\langle v, \delta\rangle \mid \delta \in s\}$$

$$WrEnv\colon \hat{E} \to V \to 2^{\Delta} \to \hat{D} \to \hat{E}$$
$$\equiv \lambda\hat{e}.\lambda v.\lambda s.\lambda\hat{d}.\text{if } s = \{\epsilon\} \text{ or } s = \{\mathbf{d}\}$$
$$\text{then } \hat{e}[\hat{d}/\langle v, \delta\rangle] \text{ where } s = \{\delta\}$$
$$\text{else } \hat{e}[\hat{d}//\langle v, \delta_1\rangle]\cdots[\hat{d}//\langle v, \delta_k\rangle] \text{ where } s = \{\delta_1, \ldots, \delta_k\}$$

Figure 29: *RdEnv* and *WrEnv*

$$Abs_X\colon (V \times P) \to P \to (V \times \Delta) \equiv \lambda\langle v, p_b\rangle.\lambda p_r.\langle v, Dir(p_r - p_b)\alpha\rangle$$
$$\text{where } v \text{ is bound by } \lambda_\alpha.$$

Abs_X maps a variable instance (represented by a pair in $V \times P$ of the variable and its birth date) and a procedure string representing the context of the abstraction, to an abstract variable instance (a pair in $V \times \Delta$ of the variable and a member of Δ, that summarizes the movements described by the variable instance, with respect to the procedure by which it is bound). This abstraction map is therefore "relative" in the same way that the abstraction maps for $\mathcal{E}_4$ were. Theorems 27 and 28 then say that for any state (let its procedure string be p_r), there is at most one instance of $\mathbf{x}$ (let its birth date be p_b) such that $Abs_X\langle[\![\mathbf{x}]\!], p_b\rangle p_r = \langle[\![\mathbf{x}]\!], \epsilon\rangle$, and likewise at most one instance such that $Abs_X\langle[\![\mathbf{x}]\!], p_b\rangle p_r = \langle[\![\mathbf{x}]\!], \mathbf{d}\rangle$.

The definitions of *RdEnv* and *WrEnv*, used to model the actions of reading and writing the environment under $\mathcal{E}_5$, are presented in Figure 2.12. Imagine an abstract state $\hat{q}$ such that $\hat{q}i = \langle\hat{p}, \hat{b}, \hat{e}, \ldots\rangle$, and let $\mathbf{x}$ be assigned the abstract value $\hat{d}$ at statement S_i, where $\mathbf{x}$ is bound by λ_α. The environment in effect after this assignment is given by $WrEnv\ \hat{e}[\![\mathbf{x}]\!]((\hat{b}[\![\mathbf{x}]\!])\alpha)\hat{d}$. While, under $\mathcal{E}_2$, an instance of $\mathbf{x}$ is identified by its birth date, here it is identified by the set $(\hat{b}[\![\mathbf{x}]\!])\alpha$ that summarizes its movements from the point of its instantiation to the current state, with respect to the procedure λ_α that binds it. If $(\hat{b}[\![\mathbf{x}]\!])\alpha = \{\mathbf{d}\}$ or $(\hat{b}[\![\mathbf{x}]\!])\alpha = \{\epsilon\}$, then this abstract instance of $\mathbf{x}$ represents only one concrete instance of $\mathbf{x}$, for every state in the concretization of $\hat{q}$, and we effect the assignment within *WrEnv* by "overwriting" the value of $\hat{e}$ at $\langle[\![\mathbf{x}]\!], \mathbf{d}\rangle$ or $\langle[\![\mathbf{x}]\!], \epsilon\rangle$. Otherwise, we raise the lattice value of $\hat{e}$ at $\langle[\![\mathbf{x}]\!], \delta\rangle$ for all $\delta \in (\hat{b}[\![\mathbf{x}]\!])\alpha$ by the value $\hat{d}$.

The abstraction maps for $\mathcal{E}_5$ are defined in Figure 30. The corresponding concretization maps are defined as described in subsection 2.11. Only the definitions of Abs_E and Abs_Q have changed from Figure 21. The revised def-

$$
\begin{aligned}
&Abs_{\Lambda} \equiv \lambda\alpha.\ \text{if } \alpha = \bot_{\Lambda} \text{ then } \{\} \text{ else } \{\alpha\}\\
&Abs_{N} \equiv \lambda i.\ \text{if } i = \bot_{N} \text{ then } \{\} \text{ else } \{i\}\\
&Abs_{B} \equiv \lambda b.\lambda p.\lambda v.Abs_{P}(p - (bv))\\
&Abs_{C} \equiv \lambda\langle\alpha, b\rangle.\lambda p.\langle Abs_{\Lambda}\alpha, Abs_{B}bp\rangle\\
&Abs_{K} \equiv \lambda\langle i, b, p, o\rangle.\langle Abs_{N}i, Abs_{B}bp, Abs_{P}p\rangle\\
&Abs_{D} \equiv \lambda x.\ \ \text{if } x = \bot_{D} \text{ then } \langle \bot_{\hat{C}}, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in C \text{ then } \langle Abs_{C}xp, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in K \text{ then } \langle \bot_{\hat{C}}, Abs_{K}xp, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in PrimOp \text{ then } \langle \bot_{\hat{C}}, \bot_{\hat{K}}, Abs_{PrimOp}x, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in Int \text{ then } \langle \bot_{\hat{C}}, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, Abs_{Int}x, \bot_{\hat{Bool}}\rangle\\
&\qquad\qquad \text{else if } x \in Bool \text{ then } \langle \bot_{\hat{C}}, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, Abs_{Bool}x\rangle\\
&Abs_{E} \equiv \lambda e.\lambda p.\lambda\langle v, \delta\rangle.\sqcup_{\hat{D}}\{Abs_{D}(e\langle v, p'\rangle)p \mid Dir(p - p')\alpha = \delta\}\\
&\qquad\qquad \text{where } v \text{ is bound by } \lambda_{\alpha}\\
&Abs_{R} \equiv \lambda r.\lambda p.\lambda\alpha.\ \sqcup_{\hat{K}}\{Abs_{K}(r(p' + \alpha^{d}))p \mid p' \in P\}\\
&Abs_{Q} \equiv \lambda\langle i, b, p, e, k, o, r\rangle.\lambda j.\ \text{if } i \neq j\\
&\qquad\qquad \text{then } \langle \bot_{\hat{P}}, \bot_{\hat{B}}, \bot_{\hat{E}}, \bot_{\hat{K}}, \bot_{\hat{R}}\rangle\\
&\qquad\qquad \text{else } \langle Abs_{P}p, Abs_{B}bp, Abs_{E}ep,\\
&\qquad\qquad\qquad Abs_{K}kp, Abs_{R}rp\rangle
\end{aligned}
$$

Figure 30: Abstraction Maps

initions of partial orderings and LUB operators over the abstract domains are defined in Figure 31 and 32. Again, only the definitions of $\sqsubseteq_{\hat{E}}, \sqsubseteq_{\hat{Q}}, \sqcup_{\hat{E}}$, and $\sqcup_{\hat{Q}}$ have changed, and those only slightly.

The definitions of the *Move* functions under $\mathcal{E}_5$ are presented in Figure 33. Only $Move_{\hat{E}}$ has changed, but its definition is markedly different from that under $\mathcal{E}_4$, for the reason that under $\mathcal{E}_5$, both the range and the domain of the function are, in effect, moved. If $\hat{e}\langle[\![\mathbf{x}]\!], \zeta\rangle$ is a value in the environment prior to these movements, and $\eta \in \hat{p}\alpha$ where $\mathbf{x}$ is bound by λ_{α}, then the environment that results from moving $\hat{e}$ by $\hat{p}$ will map the pair $\langle[\![\mathbf{x}]\!], \delta\rangle$ to a value greater than or equal to $\hat{e}\langle[\![\mathbf{x}]\!], \zeta\rangle$, where $\delta \in Cat\zeta\eta$. Intuitively, when the environment undergoes a movement described by $\hat{p}$, then the variable instances represented in the domain of the environment (as pairs in $V \times \Delta$) make movements defined by the *Cat* operator, and the values represented in the range of the environment (as members of $\hat{D}$) make movements defined by $Move_{\hat{D}}$. Suppose that $\langle[\![\mathbf{x}]\!], \zeta\rangle$ is in the domain of the environment, and that $\mathbf{x}$ is bound by λ_{α}. $\langle[\![\mathbf{x}]\!], \zeta\rangle$ therefore represents an equivalence class of instances of $\mathbf{x}$ (all those instances whose net

$$\sqsubseteq_{\hat{N}} \equiv \lambda\hat{i}.\lambda\hat{j}.\hat{i} \subseteq \hat{j}$$
$$\sqsubseteq_{\hat{\Lambda}} \equiv \lambda\hat{\alpha}.\lambda\hat{\beta}.\hat{\alpha} \subseteq \hat{\beta}$$
$$\sqsubseteq_{\hat{B}} \equiv \lambda\hat{b_1}.\lambda\hat{b_2}.(\hat{b_1}v) \subseteq (\hat{b_2}v)\ \forall v \in V$$
$$\sqsubseteq_{\hat{C}} \equiv \lambda\langle\hat{\alpha_1},\hat{b_1}\rangle.\lambda\langle\hat{\alpha_2},\hat{b_2}\rangle.(\hat{\alpha_1} \sqsubseteq_{\hat{\Lambda}} \hat{\alpha_2}) \wedge (\hat{b_1} \sqsubseteq_{\hat{B}} \hat{b_2})$$
$$\begin{aligned}\sqsubseteq_{\hat{K}} \equiv \lambda\langle\hat{i_1},\hat{b_1},\hat{p_1}\rangle.\lambda\langle\hat{i_2},\hat{b_2},\hat{p_2}\rangle.\ & (\hat{i_1} \sqsubseteq_{\hat{N}} \hat{i_2}) \\ & \wedge (\hat{b_1} \sqsubseteq_{\hat{B}} \hat{b_2}) \\ & \wedge (\hat{p_1} \sqsubseteq_{\hat{P}} \hat{p_2})\end{aligned}$$
$$\begin{aligned}\sqsubseteq_{\hat{D}} \equiv \lambda\langle\hat{c_1},\hat{k_1},\hat{f_1},\hat{z_1},\hat{x_1}\rangle.\lambda\langle\hat{c_2},\hat{k_2},\hat{f_2},\hat{z_2},\hat{x_2}\rangle.\ & (\hat{c_1} \sqsubseteq_{\hat{C}} \hat{c_2}) \\ & \wedge (\hat{k_1} \sqsubseteq_{\hat{K}} \hat{k_2}) \\ & \wedge (\hat{f_1} \sqsubseteq_{\hat{PrimOp}} \hat{f_2}) \\ & \wedge (\hat{z_1} \sqsubseteq_{\hat{Int}} \hat{z_2}) \\ & \wedge (\hat{x_1} \sqsubseteq_{\hat{Bool}} \hat{x_2})\end{aligned}$$
$$\sqsubseteq_{\hat{E}} \equiv \lambda\hat{e_1}.\lambda\hat{e_2}.(\hat{e_1}\langle v,\delta\rangle) \sqsubseteq_{\hat{D}} (\hat{e_2}\langle v,\delta\rangle)\ \forall v \in V, \forall\delta \in \Delta$$
$$\sqsubseteq_{\hat{R}} \equiv \lambda\hat{r_1}.\lambda\hat{r_2}.(\hat{r_1}\alpha) \sqsubseteq_{\hat{K}} (\hat{r_2}\alpha)\ \forall\alpha \in \Lambda$$
$$\begin{aligned}\sqsubseteq_{\hat{T}} \equiv \lambda\langle\hat{b_1},\hat{p_1},\hat{e_1},\hat{k_1},\hat{r_1}\rangle.\lambda\langle\hat{b_2},\hat{p_2},\hat{e_2},\hat{k_2},\hat{r_2}\rangle.\ & (\hat{b_1} \sqsubseteq_{\hat{B}} \hat{b_2}) \\ & \wedge (\hat{p_1} \sqsubseteq_{\hat{B}} \hat{b_2}) \\ & \wedge (\hat{e_1} \sqsubseteq_{\hat{E}} \hat{e_2}) \\ & \wedge (\hat{k_1} \sqsubseteq_{\hat{K}} \hat{k_2}) \\ & \wedge (\hat{r_1} \sqsubseteq_{\hat{R}} \hat{r_2})\end{aligned}$$
$$\sqsubseteq_{\hat{Q}} \equiv \lambda\hat{q_1}.\lambda\hat{q_2}.\lambda\alpha.(\hat{q_1}i) \sqsubseteq_{\hat{T}} (\hat{q_2}i)\ \forall i \in N$$

Figure 31: Partial Orderings

$$
\begin{aligned}
&\sqcup_{\hat{N}} \equiv \lambda \hat{n_1}.\lambda \hat{n_2}.\hat{n_1} \cup \hat{n_2} \\
&\sqcup_{\hat{\Lambda}} \equiv \lambda \hat{\alpha}.\lambda \hat{\beta}.\hat{\alpha} \cup \hat{\beta} \\
&\sqcup_{\hat{B}} \equiv \lambda \hat{b_1}.\lambda \hat{b_2}.\lambda v.(\hat{b_1} v) \sqcup_{\hat{P}} (\hat{b_2} v) \\
&\sqcup_{\hat{C}} \equiv \lambda \langle \hat{\alpha_1}, \hat{b_1} \rangle.\lambda \langle \hat{\alpha_2}, \hat{b_2} \rangle.\langle \hat{\alpha_1} \sqcup_{\hat{\Lambda}} \hat{\alpha_2}, \hat{b_1} \sqcup_{\hat{B}} \hat{b_2} \rangle \\
&\sqcup_{\hat{K}} \equiv \lambda \langle \hat{i_1}, \hat{b_1}, \hat{p_1} \rangle.\lambda \langle \hat{i_2}, \hat{b_2}, \hat{p_2} \rangle.\langle \begin{array}{l} \hat{i_1} \sqcup_{\hat{N}} \hat{i_2}, \\ \hat{b_1} \sqcup_{\hat{B}} \hat{b_2}, \\ \hat{p_1} \sqcup_{\hat{P}} \hat{p_2} \rangle \end{array} \\
&\sqcup_{\hat{D}} \equiv \lambda \langle \hat{c_1}, \hat{k_1}, \hat{f_1}, \hat{z_1}, \hat{x_1} \rangle.\lambda \langle \hat{c_2}, \hat{k_2}, \hat{f_2}, \hat{z_2}, \hat{x_2} \rangle.\langle \begin{array}{l} \hat{c_1} \sqcup_{\hat{C}} \hat{c_2}, \\ \hat{k_1} \sqcup_{\hat{B}} \hat{k_2}, \\ \hat{f_1} \sqcup_{\hat{B}} \hat{f_2}, \\ \hat{z_1} \sqcup_{\hat{B}} \hat{z_2}, \\ \hat{x_1} \sqcup_{\hat{B}} \hat{x_2} \rangle \end{array} \\
&\sqcup_{\hat{E}} \equiv \lambda \hat{e_1}.\lambda \hat{e_2}.\lambda \langle v, \delta \rangle.(\hat{e_1} \langle v, \delta \rangle) \sqcup_{\hat{D}} (\hat{e_2} \langle v, \delta \rangle) \\
&\sqcup_{\hat{R}} \equiv \lambda \hat{r_1}.\lambda \hat{r_2}.\lambda \alpha.(\hat{r_1} \alpha) \sqcup_{\hat{K}} (\hat{r_2} \alpha) \\
&\sqcup_{\hat{T}} \equiv \lambda \langle \hat{p_1}, \hat{b_1}, \hat{e_1}, \hat{k_1}, \hat{r_1} \rangle.\lambda \langle \hat{p_2}, \hat{b_2}, \hat{e_2}, \hat{k_2}, \hat{r_2} \rangle.\langle \begin{array}{l} \hat{p_1} \sqcup_{\hat{C}} \hat{p_2}, \\ \hat{b_1} \sqcup_{\hat{B}} \hat{b_2}, \\ \hat{e_1} \sqcup_{\hat{B}} \hat{e_2}, \\ \hat{k_1} \sqcup_{\hat{B}} \hat{k_2}, \\ \hat{r_1} \sqcup_{\hat{B}} \hat{r_2} \rangle \end{array} \\
&\sqcup_{\hat{Q}} \equiv \lambda \hat{q_1}.\lambda \hat{q_2}.\lambda i.(\hat{q_1} i) \sqcup_{\hat{T}} (\hat{q_2} i)
\end{aligned}
$$

Figure 32: LUB Operators Over the Abstract Domains

$$
\begin{aligned}
&Move_{\hat{B}} : \hat{B} \to \hat{P} \to \hat{B} \equiv \lambda\hat{b}.\lambda\hat{p}.\lambda v.(\hat{b}v) \oplus \hat{p} \\
&Move_{\hat{C}} : \hat{C} \to \hat{P} \to \hat{C} \equiv \lambda\langle\hat{\alpha}, \hat{b}\rangle.\lambda\hat{p}.\langle\hat{\alpha}, Move_{\hat{B}}\hat{b}\hat{p'}\rangle \\
&Move_{\hat{K}} : \hat{K} \to \hat{P} \to \hat{K} \equiv \lambda\langle\hat{i}, \hat{b}, \hat{p}\rangle.\lambda\hat{p'}.\langle\hat{i}, Move_{\hat{B}}\hat{b}\hat{p'}, \hat{p} \oplus \hat{p'}\rangle \\
&Move_{\hat{D}} : \hat{D} \to \hat{P} \to \hat{D} \equiv \lambda\langle\hat{c}, \hat{k}, \hat{f}, \hat{z}, \hat{x}\rangle.\lambda\hat{p}.\langle Move_{\hat{C}}\hat{c}\hat{p}, Move_{\hat{K}}\hat{k}\hat{p}, \hat{f}, \hat{z}, \hat{x}\rangle \\
&Move_{\hat{E}} : \hat{E} \to \hat{P} \to \hat{E} \equiv \lambda\hat{e}.\lambda\hat{p}.\lambda\langle x, \delta\rangle.\ \sqcup_{\hat{D}} \{Move_{\hat{D}}(\hat{e}\langle x, \zeta\rangle)\hat{p} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid \delta \in Cat\zeta\eta \text{ for some } \zeta \in \Delta, \eta \in \hat{p}\alpha\} \\
&Move_{\hat{R}} : \hat{R} \to \hat{P} \to \hat{R} \equiv \lambda\hat{r}.\lambda\hat{p}.\lambda\alpha.Move_{\hat{K}}(\hat{r}\alpha)\hat{p} \\
&Move_{\hat{T}} : \hat{T} \to \hat{P} \to \hat{T} \equiv \lambda\langle\hat{p}, \hat{b}, \hat{e}, \hat{k}, \hat{r}\rangle.\lambda\hat{p'}.\langle\hat{p} \oplus \hat{p'}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Move_{\hat{B}}\hat{b}\hat{p'}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Move_{\hat{E}}\hat{e}\hat{p'}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Move_{\hat{K}}\hat{k}\hat{p'}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Move_{\hat{R}}\hat{r}\hat{p'}\rangle
\end{aligned}
$$

Figure 33: Movement Functions for $\mathcal{E}_5$

movements with respect to λ_α are described by ζ). After the movement, $\langle[\![\mathbf{x}]\!], \zeta\rangle$ becomes part of each equivalence class $\langle[\![\mathbf{x}]\!], \delta\rangle$ such that $\delta \in Cat\zeta\eta$ for some $\eta \in \hat{p}\alpha$. As a special case, consider that in which $\hat{p} = Abs_P(\beta^d)$, $\alpha \neq \beta$ (that is, in which the movement described by $\hat{p}$ is downward into an instance of a lambda expression other than λ_α). Then $\hat{p}\alpha = \{\epsilon\}$, and $Cat\zeta\eta = \{\zeta\}$, for all $\eta \in \hat{p}\alpha$. Therefore $\langle[\![\mathbf{x}]\!], \zeta\rangle$ is unmoved in this case, as we would expect. As another example, suppose that $\hat{p}\alpha = \{\mathbf{u}\}$, and consider the class of variable instances represented by $\langle[\![\mathbf{x}]\!], \mathbf{d}\rangle$ (which class, as we showed in Theorem 28, contains at most one member). This variable instance "becomes" the instance $\langle[\![\mathbf{x}]\!], \epsilon\rangle$ after the movement $Move_{\hat{E}}\hat{e}\hat{p}$, since $Cat\ \mathbf{d}\ \mathbf{u} = \{\epsilon\}$.

The auxiliary functions *RdEnv* and *WrEnv*, used to read from and write to the environment in $\mathcal{E}_5$, are defined in Figure 2.12. Modeling the action of $\mathcal{E}_2$ closely, in $\mathcal{E}_5$ the (abstract) instance of a lexically visible variable will be identified by $(\hat{b}[\![\mathbf{x}]\!])\alpha$, where $\hat{b}[\![\mathbf{x}]\!]$ is the stack configuration that summarizes the movements made by the lexically visible instance of $\mathbf{x}$ from its instantiation to the current state $\hat{q}$, and λ_α is the procedure that binds $\mathbf{x}$. Consider the definition of *WrEnv*; the set $s = (\hat{b}[\![\mathbf{x}]\!])\alpha \subseteq \Delta$ isolates those movements that pertain to λ_α. If s contains only $\mathbf{d}$ or ϵ, then by Theorems 27 and 28, this abstract instance of $\mathbf{x}$ represents exactly one concrete instance of $\mathbf{x}$ (for every state in the concretization of $\hat{q}$), and we may model an assignment in *WrEnv* by "overwriting" the value of $\hat{e}\langle v, \delta\rangle$

where $s = \{\delta\}$; else, we simply raise the lattice value of $e\langle v, \delta\rangle$ for all $\delta \in s$, by the value being assigned.

The definitions of $\mathcal{S}_5$, $\mathcal{S}_5^*$, and $\mathcal{E}_5$ are given in Figures 34, 35, 36 and 37. Our alterations to $\mathcal{S}_4$ are restricted to the treatment of the environment. We may therefore apply Theorem 23 directly to $\mathcal{S}_5$, once we show that our abstraction of environments under $\mathcal{E}_5$ preserves the meaning of environments under $\mathcal{E}_2$. The following theorem suffices.

Theorem 29 *If* $Abs_E ep \sqsubseteq_{\hat{E}} \hat{e}$, $Abs_B bp \sqsubseteq_{\hat{B}} \hat{b}$ *and* $Abs_D dp \sqsubseteq_{\hat{D}} \hat{d}$, *then*

I. $Abs_D(e\langle[\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle)p \sqsubseteq_{\hat{D}} RdEnv\ \hat{e}[\![\mathbf{x}]\!]((\hat{b}[\![\mathbf{x}]\!])\alpha)$, *and*

II. $Abs_E(e[d/\langle[\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle])p \sqsubseteq_{\hat{E}} WrEnv\hat{e}[\![\mathbf{x}]\!]((\hat{b}[\![\mathbf{x}]\!])\alpha)\hat{d}$,

where $\mathbf{x}$ *is bound by* λ_α.

Proof:

I. By the definition of *RdEnv*,

$$RdEnv\hat{e}[\![\mathbf{x}]\!]((\hat{b}[\![\mathbf{x}]\!])\alpha) = \sqcup_{\hat{D}}\{\hat{e}\langle[\![\mathbf{x}]\!], \delta\rangle \mid \delta \in (\hat{b}[\![\mathbf{x}]\!])\alpha\}.$$

Let $Abs_P p \sqsubseteq_{\hat{P}} \hat{p}$. That $Abs_E ep \sqsubseteq_{\hat{E}} \hat{e}$ implies that

$$\sqcup_{\hat{D}}\{Abs_D(e\langle[\![\mathbf{x}]\!], p'\rangle)p \mid Dir(p - p')\alpha = \delta\} \sqsubseteq_{\hat{D}} \hat{e}\langle[\![\mathbf{x}]\!], \delta\rangle \text{ for all } \delta \in \Delta.$$

Therefore

$$Abs_D(e\langle[\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle)p \sqsubseteq_{\hat{D}} \hat{e}\langle[\![\mathbf{x}]\!], Dir(p - b[\![\mathbf{x}]\!])\alpha\rangle.$$

But

$$Dir(p - b[\![\mathbf{x}]\!])\alpha \in (\hat{b}[\![\mathbf{x}]\!])\alpha$$

since $Abs_B bp \sqsubseteq_{\hat{B}} \hat{b}$. Therefore

$$Abs_D(e\langle[\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle)p \sqsubseteq_{\hat{D}} \sqcup_{\hat{D}}\{\hat{e}\langle[\![\mathbf{x}]\!], \delta\rangle \mid \delta \in (\hat{b}[\![\mathbf{x}]\!])\alpha\}$$

and

$$Abs_D(e\langle[\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle)p \sqsubseteq_{\hat{P}} RdEnv\hat{e}[\![\mathbf{x}]\!]((\hat{b}[\![\mathbf{x}]\!])\alpha)$$

by the definition of *RdEnv*.

II.

Let $\hat{t} = \langle \hat{p}, \hat{b}, \hat{e}, \hat{k}, \hat{r} \rangle, i \in N$. Then $\mathcal{S}_5 : N \to \hat{T} \to \hat{Q}$ is defined, according to the form of S_i, as follows.

$$
\begin{array}{l}
S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_m\texttt{))}]\!] \text{ or } S_i = [\![\texttt{(set! x (call/cc f))}]\!] \Rightarrow \\
\quad \mathcal{S}_5 i \hat{t} = \hat{q}_c \sqcup_{\hat{Q}} \hat{q}_k \\
\quad \text{where } \hat{q}_c = \sqcup_{\hat{Q}} \Big\{ \lambda i'. \text{ if } i' \neq j \\
\qquad\qquad\qquad \text{then } \perp_{\hat{T}} \\
\qquad\qquad\qquad \text{else } \langle \hat{p} \oplus \hat{p}', \\
\qquad\qquad\qquad\quad (\mathit{Move}_{\hat{B}} \hat{b}' \hat{p}')[\lambda\alpha.\{\epsilon\}/[\![\texttt{z}_1]\!]] \cdots [\lambda\alpha.\{\epsilon\}/[\![\texttt{z}_n]\!]], \\
\qquad\qquad\qquad\quad \mathit{Move}_{\hat{E}} \hat{e}' \hat{p}', \\
\qquad\qquad\qquad\quad \mathit{Move}_{\hat{K}} \langle \{i\}, \hat{b}, \lambda\alpha.\{\epsilon\} \rangle \hat{p}', \\
\qquad\qquad\qquad\quad \mathit{Move}_{\hat{R}} (\hat{r}[\hat{k} // \mathit{Container}\ i]) \hat{p}') \\
\qquad\qquad\quad \text{where } \lambda_\alpha = [\![\texttt{(lambda (z}_1 \cdots \texttt{z}_m\texttt{) <z}_{m+1} \cdots \texttt{z}_n\texttt{> } S_j \cdots\texttt{)}]\!], \\
\qquad\qquad\qquad \hat{p}' = \mathit{Abs}_P(\alpha^d), \\
\qquad\qquad\quad \text{and} \quad \hat{e}' = \text{if } S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_m\texttt{))}]\!] \\
\qquad\qquad\qquad\qquad \text{then } \hat{e_m} \text{ where } \hat{e_0} = \hat{e} \\
\qquad\qquad\qquad\qquad\qquad \text{and} \quad \hat{e_l} = \mathit{WrEnv}\ \hat{e_{l-1}} [\![\texttt{z}_l]\!] \{\epsilon\} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{RdEnv}\ \hat{e} [\![\texttt{y}_l]\!] ((\hat{b} [\![\texttt{y}_l]\!]) \alpha_l)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{where } [\![\texttt{y}_l]\!] \text{ is bound by } \lambda_{\alpha_l},\ 1 \leq l \leq m \\
\qquad\qquad\qquad\qquad \text{else } \mathit{WrEnv}\ \hat{e} [\![\texttt{z}_1]\!] \{\epsilon\} \langle \perp_{\hat{C}}, \langle \{i\}, \hat{b}, \lambda\alpha.\{\epsilon\} \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad \perp_{\hat{PrimOp}}, \perp_{\hat{Int}}, \perp_{\hat{Bool}} \rangle \\
\qquad\qquad\quad \Big| \ \alpha \in \hat{\alpha} \Big\} \\
\qquad \text{where } \hat{e} [\![\texttt{f}]\!] = \langle \hat{c'}, \hat{k'}, \ldots \rangle \\
\qquad \text{and} \quad \hat{c'} = \langle \hat{\alpha}, \hat{b'} \rangle
\end{array}
$$

Figure 34: The Semantic Function $\mathcal{S}_5$ (Part I)

$$
\begin{array}{l}
\vdots \\
\text{and} \quad \hat{q_k} = \sqcup_{\hat{Q}} \Big\{ \lambda i'.\ \text{if } i' \neq j \\
\qquad\qquad\qquad\qquad \text{then } \bot_{\hat{T}} \\
\qquad\qquad\qquad\qquad \text{else } \mathit{Move}_{\hat{T}} \langle \hat{p}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \hat{b}', \\
\qquad\qquad\qquad\qquad\qquad\qquad \hat{e}', \\
\qquad\qquad\qquad\qquad\qquad\qquad \hat{r}\beta, \\
\qquad\qquad\qquad\qquad\qquad\qquad \hat{r}[\hat{k} // \mathit{Container}\ i] \rangle (\hat{\mathit{Inv}}\ \hat{p}') \\
\qquad\qquad\qquad\qquad\qquad \text{where } \hat{e}' = \text{if } S_i = [\![\texttt{(set! x (f y}_1\texttt{))}]\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{then } \mathit{WrEnv}\ \hat{e}[\![\texttt{z}]\!]\{\epsilon\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{RdEnv}\ \hat{e}[\![\texttt{y}_1]\!]((\hat{b}[\![\texttt{y}_1]\!])\alpha)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \texttt{y}_1 \text{ is bound by } \lambda_\alpha \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else } \mathit{WrEnv}\ \hat{e}[\![\texttt{z}]\!]\{\epsilon\} \langle \bot_{\hat{C}}, \langle \{i\}, \hat{b}, \lambda\alpha.\{\epsilon\} \rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \bot_{\hat{\mathit{PrimOp}}}, \bot_{\hat{\mathit{Int}}}, \bot_{\hat{\mathit{Bool}}} \rangle \\
\qquad\qquad\qquad\qquad \text{where } S_j = [\![\texttt{(set! z (call/cc g))}]\!] \\
\qquad\qquad\qquad\qquad \Big| \ j \in \hat{j} \Big\} \\
\qquad\qquad \text{where } \hat{e}[\![\texttt{f}]\!] = \langle \hat{c'}, \hat{k'}, \ldots \rangle \\
\qquad\qquad \text{and } \ \hat{k'} = \langle \hat{j}, \hat{b'}, \hat{p'} \rangle
\end{array}
$$

Figure 35: The Semantic Function $\mathcal{S}_5$ (Part II)

$$
\begin{aligned}
S_i = {} & [\![\texttt{(set! f (lambda}_\alpha\ \texttt{(x}_1 \ldots \texttt{x}_m\texttt{)}\ \texttt{<x}_{m+1} \cdots \texttt{x}_n\texttt{>}\ \cdots\texttt{))}]\!] \Rightarrow \\
& \mathcal{S}_5 i\hat{t} = \lambda i'.\ \text{if } i' \neq \mathit{Succ}\ i \\
& \qquad \text{then } \bot_{\hat{T}} \\
& \qquad \text{else } \langle \hat{p}, \\
& \qquad\quad \hat{b}, \\
& \qquad\quad \mathit{WrEnv}\ \hat{e}[\![\texttt{f}]\!]((\hat{b}[\![\texttt{f}]\!])\beta)\langle\langle\{\alpha\}, \hat{b}\rangle, \bot_{\hat{K}}, \bot_{\hat{PrimOp}}, \bot_{\hat{Int}}, \bot_{\hat{Bool}}\rangle \\
& \qquad\quad \hat{k}, \\
& \qquad\quad \hat{r}\rangle \\
& \qquad\quad \text{where f is bound by } \lambda_\beta
\end{aligned}
$$

$$
\begin{aligned}
S_i = {} & [\![\texttt{(if x (goto m) (goto n))}]\!] \Rightarrow \\
& \mathcal{S}_5 i\hat{t} = \lambda i'.\ \text{if } i' \notin \{m, n\} \\
& \qquad \text{then } \bot_{\hat{T}} \\
& \qquad \text{else } \hat{t}
\end{aligned}
$$

$$
\begin{aligned}
S_i = {} & [\![\texttt{(return x)}]\!] \Rightarrow \\
& \mathcal{S}_5 i\hat{t} = \sqcup_{\hat{Q}} \Big\{ \lambda i'.\ \text{if } i' \neq j \\
& \qquad\qquad \text{then } \bot_{\hat{T}} \\
& \qquad\qquad \text{else } \mathit{Move}_{\hat{T}} \langle \hat{p} \\
& \qquad\qquad\qquad\quad \hat{b}', \\
& \qquad\qquad\qquad\quad \mathit{WrEnv}\ \hat{e}[\![\texttt{y}]\!]((\hat{b}'[\![\texttt{y}]\!])\beta)(\mathit{RdEnv}\ \hat{e}[\![\texttt{x}]\!]((\hat{b}[\![\texttt{x}]\!])\alpha)) \\
& \qquad\qquad\qquad\quad \hat{r}(\mathit{Container}\ j), \\
& \qquad\qquad\qquad\quad \hat{r}\rangle(\hat{Inv}\ \hat{p}') \\
& \qquad\qquad\qquad\quad \text{where y is bound by } \lambda_\beta \\
& \qquad\qquad\qquad\quad \text{and} \quad \texttt{x} \text{ is bound by } \lambda_\alpha \\
& \qquad\quad \text{where } S_j = [\![\texttt{(set! y} \cdots\texttt{)}]\!] \\
& \qquad\quad \Big| \ j \in \hat{j} \Big\} \\
& \text{where } \hat{k} = \langle \hat{j}, \hat{b}', \hat{p}' \rangle
\end{aligned}
$$

Figure 36: The Semantic Function $\mathcal{S}_5$(Part III)

$$\mathcal{S}_5^* : \hat{Q} \to \hat{Q} \equiv \lambda \hat{q}.\, \sqcup_{\hat{Q}} \{\mathcal{S}_5 i(\hat{q}i) \mid i \in N\}$$

$$\mathcal{E}_5 : Q \to Q \equiv \lambda \hat{q}.\, \text{Let } \hat{q'} = \mathcal{S}_5^* \hat{q} \text{ in if } \hat{q'} \sqsubseteq_{\hat{Q}} \hat{q} \text{ then } \hat{q} \text{ else } \mathcal{E}_5(\hat{q} \sqcup_{\hat{Q}} \hat{q'})$$

Figure 37: The Semantic Functions $\mathcal{S}_5^*$ and $\mathcal{E}_5$

case 1: $(\hat{b}[\![\mathbf{x}]\!])\alpha = \{\epsilon\}$ or $\{\mathbf{d}\}$. Let

$$e' = e[d/\langle [\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle]$$

and

$$\hat{e'} = \hat{e}[\hat{d}/\langle [\![\mathbf{x}]\!], \delta\rangle]$$

where $(\hat{b}[\![\mathbf{x}]\!])\alpha = \{\mathbf{d}\}$. It is clear that

$$Abs_D(e'\langle [\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle)p \sqsubseteq_{\hat{D}} \hat{e'}\langle [\![\mathbf{x}]\!], \delta\rangle$$

because $Abs_E ep \sqsubseteq_{\hat{E}} \hat{e}$ and $Abs_D dp \sqsubseteq_{\hat{D}} \hat{d}$, and $Dir(p - b[\![\mathbf{x}]\!])\alpha = \delta$ since $Abs_B b \sqsubseteq_{\hat{B}} \hat{b}$. We must therefore show that the meaning of e' is preserved at all points other than $\langle [\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle$. Let $\dot{\mathbf{y}}$ be an instance of $\mathbf{y}$, and let p_b be the birth date of $\mathbf{y}$. If $\mathbf{y} \neq \mathbf{x}$, then clearly

$$Abs_D(e'\langle [\![\mathbf{y}]\!], p_b\rangle)p \sqsubseteq_{\hat{D}} \hat{e'}\langle [\![\mathbf{y}]\!], Dir(p - p_b)\alpha\rangle,$$

since $Abs_E ep \sqsubseteq_{\hat{E}} \hat{e}$. Assume therefore that $\mathbf{y} = \mathbf{x}$. By Theorems 27 and 28, if $Dir(p - p_b) = \delta$ then $b[\![\mathbf{x}]\!] = p_b$ and $\dot{\mathbf{y}}$ *is* the instance of $\mathbf{x}$ whose birth date is $b[\![\mathbf{x}]\!]$ (that is, the instance of $\mathbf{x}$ being assigned). This case was treated above. Else, if $Dir(p - p_b) \neq \delta$, then

$$Abs_D(e'\langle [\![\mathbf{y}]\!], p_b\rangle)p \sqsubseteq_{\hat{D}} \hat{e'}\langle [\![\mathbf{y}]\!], Dir(p - p_b)\alpha\rangle,$$

since this equation holds for e and $\hat{e}$, and e' and $\hat{e'}$ do not differ from e and $\hat{e}$ at the points $\langle [\![\mathbf{y}]\!], p_b\rangle$ and $\langle [\![\mathbf{y}]\!], Dir(p - p_b)\alpha\rangle$ respectively, by the fact that $p_b \neq b[\![\mathbf{x}]\!]$ and $Dir(p - p_b)\alpha \neq \delta$. Therefore

$$Abs_E(e[d/\langle [\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle]) \sqsubseteq_{\hat{E}} WrEnv\hat{e}[\![\mathbf{x}]\!]((\hat{b}[\![\mathbf{x}]\!])\alpha)\hat{d}$$

by the definition of $\sqsubseteq_{\hat{E}}$.

case 2: $(\hat{b}[\![\mathbf{x}]\!])\alpha \neq \{\epsilon\}$ and $(\hat{b}[\![\mathbf{x}]\!])\alpha \neq \{\mathbf{d}\}$. Let

$$e' = e[d/\langle [\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle]$$

and

$$\hat{e}' = \hat{e}[\hat{d}//\langle [\![\mathbf{x}]\!], \delta_1\rangle] \cdots [\hat{d}//\langle [\![\mathbf{x}]\!], \delta_k\rangle]$$

where $(\hat{b}[\![\mathbf{x}]\!])\alpha = \{\delta_1, \ldots, \delta_k\}$. This case is much simpler, because $\hat{e} \sqsubseteq_{\hat{E}} \hat{e}'$ by the definition of the notation $f[x//y]$. By assumption, $Abs_E ep \sqsubseteq_{\hat{E}} \hat{e}$, $Abs_D dp \sqsubseteq_{\hat{D}} \hat{d}$, and $Abs_B bp \sqsubseteq_{\hat{B}} \hat{b}$. Therefore $Dir(p - b[\![\mathbf{x}]\!])\alpha = \delta$ for some $\delta \in (\hat{b}[\![\mathbf{x}]\!])\alpha$, and it follows at once that

$$Abs_D(e'\langle [\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle)p \sqsubseteq_{\hat{D}} \sqcup_{\hat{D}}\{\hat{e'}\langle [\![\mathbf{x}]\!], \delta\rangle \mid \delta \in (\hat{b}[\![\mathbf{x}]\!])\alpha\}$$

and therefore that

$$Abs_E(e[d/\langle [\![\mathbf{x}]\!], b[\![\mathbf{x}]\!]\rangle])p \sqsubseteq_{\hat{E}} WrEnv\hat{e}[\![\mathbf{x}]\!]((\hat{b}[\![\mathbf{x}]\!])\alpha)\hat{d}.$$

□

Apart from their treatments of the environment, there is no difference between $\mathcal{S}_4$ and $\mathcal{S}_5$, and therefore Theorem 29 (to show that the meaning of environments are preserved by $\mathcal{S}_5$), and Theorem 23 (to show that all other components of states are preserved by $\mathcal{S}_5$) together constitute proof of the correctness of $\mathcal{S}_5$. Theorem 19 is likewise proof of the correctness of $\mathcal{E}_5$, because (apart from their respective invocations of $\mathcal{S}_3^*$, $\mathcal{S}_4^*$, and $\mathcal{S}_5^*$) there is no difference between $\mathcal{E}_3$, $\mathcal{E}_4$, and $\mathcal{E}_5$.

2.13 Examples of Analysis under $\mathcal{E}_5$

Let us now return to some examples to see what we may expect of the framework of analysis that we have constructed. First, consider the example of Figure 20 once more, to see if we have overcome the difficulties it presented for $\mathcal{E}_3$ and $\mathcal{E}_4$. It is easy to see that we have; in fact, it is enough to consider the evaluation of statement S_5, in which $\mathtt{t_3}$ is assigned a closure of λ_γ. Again, let $\hat{q_n} = \mathcal{E}_4\, \hat{q_0}$ be the result of abstract interpretation of the program of Figure 20 under $\mathcal{E}_4$, and let $\hat{q_n}\alpha = \langle \hat{p}, \hat{b}, \hat{e}, \hat{k}, \hat{r}\rangle$. We have that $(\hat{b}[\![\mathtt{t_3}]\!])\alpha = \{\epsilon\}$ when S_5 is first evaluated. Then by the definition of $WrEnv$, the closure of λ_α which is newly formed will overwrite the previous value of $\langle \mathtt{t_3}, \epsilon\rangle$ in the environment; when this closure is applied at statement S_6, it will have undergone no interprocedural movements. Therefore, if $\hat{b}'$ is the birth date map of a state in which a reference to `x` is made within λ_γ (that is, when evaluating `(set! t4 (car x))`), we will find that $(\hat{b}'[\![\mathbf{x}]\!])\gamma = \{\mathbf{d}\}$

```
(define accum-fn
        ( lambda_α (x) ( lambda_β (y) (set! x (+ x y)) x)))
(define apply-to-range ( lambda_γ (lo hi fn)
        (if (= lo hi)
            (fn lo)
            (begin (fn lo)
                   (apply-to-range (1+ lo) hi fn)))))
(define sum-of-integers ( lambda_σ (m n)
        (apply-to-range m n (accum-fn 0))))
(define list-of-sums ( lambda_ε (l1 l2)
        (if (null? l1)
            #f
            (cons (sum-of-integers (car l1) (car l2))
                  (list-of-sums (cdr l1) (cdr l2))))))
```

Figure 38: Example of Side-Effects and Object Lifetimes

```
(set! sum-of-integers ( lambda_σ (m n) <t1 t2>
      (set! t1 (accum-fn 0))
      (set! t2 (apply-to-range m n t1))
      (return t2)))
```

Figure 39: `sum-of-integers`, Rewritten in $\mathcal{L}$

and $(\hat{b}'[\![\mathtt{x}]\!])\alpha = \{\epsilon\}$. This implies, by Theorem 21, that `x` may be stack allocated.

Now let us turn our attention to the example of Figure 11; it is reproduced in Figure 38 for convenience. The procedure λ_σ (`sum-of-integers`) is rewritten in a form close to $\mathcal{L}$, for the purpose of illustration, in Figure 39.

The critical moment in evaluation is the assignment of the return value of `accum-fn` into $\mathtt{t}_1$; by exactly the reasoning used above, the closure of λ_β that is assigned into $\mathtt{t}_1$ overwrites the previous value of $\mathtt{t}_1$ in the environment, and therefore when `x` is referenced in λ_β, we will find that it has described no downward movements (none of $\mathbf{d}, \mathbf{dd}^+, \mathbf{u}^+\mathbf{d}^+$) with respect to λ_σ, and no upward movements (none of $\mathbf{u}, \mathbf{uu}^+, \mathbf{u}^+\mathbf{d}^+$) with respect to λ_σ, either. This implies both that λ_σ has no side-effects upon `x` (and `x` is the only mutable quantity in this example), and that `x` may be deallocated upon exit of the invocation of λ_σ which creates it (via a call to λ_α). In short, the analysis uncovers both the high-level parallelism of this example, in that the sum computed within each recursive application of λ_σ (`list-of-sums`) is seen to be independent of the others (since all are seen to be free of side-effects), and provides a precise description of the

```
(define cons (lambda (car cdr)
   (lambda (op val)
     (cond ((eq? op 'car) car)
           ((eq? op 'cdr) cdr)
           ((eq? op 'set-car!) (set! car val) car)
           ((eq? op 'set-cdr!) (set! cdr val) cdr)))))
(define car (lambda (x) (x 'car #f)))
(define cdr (lambda (x) (x 'cdr #f)))
(define set-car! (lambda (x y) (x 'set-car! y)))
(define set-cdr! (lambda (x y) (x 'set-cdr! y)))
```

Figure 40: `cons` in Terms of Closures

lifetime of each instance of `x`, for the purpose of its automatic deallocation (and ultimately, for the purpose of placing it within a hierarchical shared memory).

2.14 Mutable Data and Aliasing

It might appear to the reader familiar with the difficulties of analyzing and parallelizing Lisp programs, that in $\mathcal{L}$ we have put forth a subset of Scheme that sidesteps the most difficult issue of all: aliasing, particularly aliasing relationships that arise by the use of mutable, compound data objects, such as cons cells, user structures, vectors, atom property lists and hashtables. In fact, our abstractions of $\mathcal{L}$ provide very sharp analyses of such effects; it is simply a matter of casting such aliasing problems into the terminology we have been using, and of interpreting the results of the subsequent analysis.

Let us first consider (mutable) cons cells. A cons cell is a record of two fields, called `car` and `cdr`, either of which may be updated after the cell has been allocated. It is well known that the function `cons` and its auxiliary routines `car`, `cdr`, `set-car!`, and `set-cdr!` can be written using closures, as in Figure 40. (We forward this means of expressing `cons` only for the purpose of static analysis, and not as a means of implementing cons cells at run-time.)

The reader is now asked to consider the example of Figure 41. The example of Figure 11 has been rewritten in this figure, to make use of mutable cons cells instead of instances of λ_β, to accumulate a sum of integers. It is clear that the analysis of side-effects and object lifetimes applies equally well to this program as to that of Figure 11; in particular, the high-level parallelism of `list-of-sums` is discovered, and the lifetime of each cons cell is seen to be circumscribed by an instance of λ_σ. Theorems 6, 7 and

```
(define accum-fn
        ( lambda_α (x) (cons x #f)))
(define apply-to-range ( lambda_γ (lo hi y)
       (if (= lo hi)
           (set-car! y (+ (car y) lo))
           (begin (set-car! y (+ (car y) lo))
                  (apply-to-range (1+ lo) hi y)))))
(define sum-of-integers ( lambda_σ (m n)
        (apply-to-range m n (accum-fn 0))))
(define list-of-sums ( lambda_ϵ (l1 l2)
        (if (null? l1)
            #f
            (cons (sum-of-integers (car l1) (car l2))
                  (list-of-sums (cdr l1) (cdr l2))))))
```

Figure 41: Example of Side-Effects and Object Lifetimes

20 tell us that all of the dependences of a computation are uncovered by our analysis; any side-effects that arise through aliasing of cons cells, will therefore be revealed as side-effects upon the variables `car` and `cdr`, at any points at which such side-effects are visible.

To see the outcome of aliasing more clearly, consider Figure 42. The definitions of two mutable structures are shown; one has a field called `x`, the other a field called `y`. As with the cons cell, these fields are represented, for the purpose of static analysis, as free variables within closures. In the example, the variable `a` holds an instance of λ_α, and `b` and `c` hold instances of λ_β. In effect, `b` and `c` each "point" to `a`. The final two expressions of the example illustrate a dependence caused by this shared substructure, this aliasing of pointers. First `a` is reached via `b`, and is updated so that its `x` field has the value 2. This "indirect" update entails two visible side-effects: a use of the variables `y`, and a definition of the variable `x`. Second, `a` is reached via `c`, and its `x` field is read. Again, there are two visible side-effects, a use of `y` and a use of `x`. By Theorem 20, both of these side-effects are revealed by our analysis, and thus the aliasing of pointers is properly accounted for.

There is, however, much information provided by our analysis that is not available from a conventional alias analysis [14, 33]. Intuitively, this information is of two kinds: information concerning an object's lifetime, and the limits thereof (in terms of the net interprocedural movements it describes), and information concerning distinct instances of objects that arise from a single lexical construct (where the instances are distinguished by the movements they describe from an evaluation of that lexical construct,

```
(define make-struct-a
   (lambda_γ (x)
      (lambda_α (op val)
         (if (eq? op 'update)
             (begin (set! x val) x)
             x))))
(define make-struct-b
   (lambda_δ (y)
      (lambda_β (op val)
         (if (eq? op 'update)
             (begin (set! y val) y)
             y))))
(set! a (make-struct-a 1))
(set! b (make-struct-b a))
(set! c (make-struct-b a))
((b 'read #f) 'update 2)
((c 'read #f) 'read #f)
```

Figure 42: User Structures, in Terms of Closures

to a later point during the evaluation). For instance, in the example of Figure 38, we have been able to detect the freedom from side-effects of `sum-of-integers`, because the analysis is able to recognize each instance of λ_β as restricted in lifetime to the subtree of computation rooted at an instance of λ_σ, and to recognize the instance as distinct from all other instances of λ_β.

It is important to emphasize again that we are not suggesting that cons cells, user structures, etc., be implemented using closures, but rather that their lifetimes and the dependences that arise from their manipulation can be understood in terms of (the more general mechanism of) closures which capture free variables. This analogy is not entirely satisfactory, in its details, however. For example, in Figure 41, every application of the function `car` appears to our analysis to entail both a use and a definition, of both the variables `car` and `cdr`. This is really a problem of flow-sensitivity: the side-effects of an application of the closure that represents a cons cell depend entirely upon the argument that is passed to it; our analysis does not take this into account, but rather attributes all of the possible effects of a procedure to each of its points of application.

There are several ways around this difficulty; the first is simply to extend the semantics of $\mathcal{L}$ to accommodate compound mutable data directly; this presents little technical difficulty, for as we have seen, their implications for dependence and lifetime analysis are less general that those of closures that

capture free variables. The other means of addressing the problem is to increase the flow-sensitivity of the analysis functions, so that they might take into account the differing conditions under which a procedure is applied. The latter approach is appealing for the reason that it would improve the accuracy of the analysis generally (not simply in the case of aliased, mutable data). If, for example, instead of joining all of the environments in which a particular procedure is invoked, we analyze the procedure (that is, perform an abstract evaluation of the body of the procedure) once for every point in the program at which it is applied, and if we increase the sensitivity of the analysis to treat simple expressions such as `(eq? op 'car)` where `op` has a constant value (and recognize control paths that cannot be taken, when they depend upon the outcome of such a comparison), then we may easily sharpen the framework sufficiently to reveal that the procedures `car` and `cdr` make no modifications to variables, whereas `set-car!` and `set-cdr!` do. This technique assumes that the procedures to which we choose to give such a flow-sensitive analysis are not recursive; but this is true of all of the procedures we have used to define `cons`, `car`, `cdr`, `set-car!`, `set-cdr!`, and user structures.

There is a simple, mechanical means by which we may further improve the accuracy of our analysis, when applied to mutable objects such as returned by `cons` (as defined in Figure 40). Suppose that we rewrite expressions of the form `(cons A B)` as

```
((lambda (car cdr) (lambda (op val) ... )) A B)
```

where the identifier `cons` is literally replaced by the definition given in Figure 40. We proceed with analysis as usual. This has the effect of partitioning the cons cells created by the program into classes according to their (lexical) points of creation. The advantage of this is very simple: side-effects upon the `car` and `cdr` variables bound by one lexical occurrence of `cons` will be unrelated to those upon the `car` and `cdr` variables bound by other occurrences (since these occurrences will be distinct lambda expressions). If we do not effect such a transformation, then the analysis may reveal dependences that are caused by cons cells that originate from lexically distinct invocations of `cons`, and which therefore could not possibly refer to the same memory locations. The same techniques may be applied, of course, to lambda expressions that are used to simulate other forms of mutable data (e.g., user structures).

There are some mutable objects, whose precise behavior it is unnecessary, or complex, to simulate exactly during interprocedural analysis, via closures. For example, while hashtables and vectors may (in their function) be described exactly in $\mathcal{L}$, an approximate description may be more practical, and equally accurate, for our analysis. As an example of such an

```
(define cons
   (lambda (cxr)
      (lambda (val)
         (if #f (begin (set! cxr val) cxr) cxr))))
(define car (lambda (x) (x #f)))
(define cdr (lambda (x) (x #f)))
(define set-car! (lambda (x y) (x y)))
(define set-cdr! (lambda (x y) (x y))))
```

Figure 43: An Abstraction of `cons`, for Analysis

approximation, we could redefine `cons` and its auxiliary functions as per Figure 43.

This is a strange-looking definition indeed. First, notice that the `(if #f ... )` construct is treated by all of our abstract semantics as if both the true and false branches were possible outcomes. Second, notice that the value of the argument to `x` in `car` and `cdr` is unimportant, but on the other hand it causes the `car` and `cdr` variables to be treated as if `#f` were possible values for them. Better would be if a constant $\perp_D$ were provided at the source level, so that we could write `(if` $\perp_D$ `...)` and `(x` $\perp_D$`)` instead of `(if #f ...)` and `(x #f)`, respectively. The important thing about the definition is that it preserves the dependence behavior of our previous definition of `cons`. There is a loss of accuracy, in that where `car` and `cdr` were distinct variables in the environment previously, they are now represented by one variable, and thus their values will be joined in the lattice $\hat{D}$. Nevertheless, any side-effects upon the variables `car` and `cdr` (when using the definition of Figure 40) will be reflected as side-effects upon the variable `cxr` (when using the definition of Figure 43); and likewise, the lifetime of cons cells as determined by use of the less accurate definition of Figure 43 will be at least as great as that determined using Figure 40. These facts may be confirmed formally without difficulty. We may form similar abstractions of the dependence behavior of vectors, hashtables, atom property lists, and so on. For each such object, the correct abstractions of its behavior describe a lattice of approximation, that is quite interesting in itself (but beyond the scope of this work). Before leaving these thoughts behind, it is worth pointing out that the abstraction of dependence behavior in the above style, leads to an elegant handling of the problem of interprocedural analysis in the face of separate compilation. The idea is a simple one: the dependence implications of a separately compiled module may be abstracted into procedures, just as we have abstracted `cons` into the form of Figure 43, and analyzed in lieu of the entire module, for the sake of efficiency in compilation. Furthermore, such abstractions provide a natural

means of representing the dependence consequences of a subroutine call, when the procedure being invoked may vary from run to run (according to, for example, a link-time decision). This promises to be a rich area of investigation.

It should be clear at this point that the techniques described in this section are applicable to a wide variety of programming language constructs. For instance, it is straightforward to create a semantics for a suitable subset of C or Pascal in terms of procedure strings, and to abstract this semantics using stack configurations, in order to analyze the dependences and lifetimes of, say, dynamically allocated. The essential insights of this section are the way in which side-effects and object lifetimes can be reasoned about in terms of procedure strings, and the way in which these strings, and the reasoning that applies to them, can be abstracted to stack configurations. Because most programming languages in wide use lack such radically general features as Scheme's first-class procedures and continuations, the application of these techniques to such languages is, in many instances, a mere restriction to less difficult situations (and, as we will see in subsection 2.16, these restrictions can sometimes be used to improve the accuracy or efficiency of the analysis).

2.15 Management of a Hierarchical, Shared Memory

We have, from time to time, been discussing a hierarchical strategy for management of dynamically allocated objects. Under this strategy, each object is associated with a procedure instance. This procedure instance is guaranteed to have outlive the object, i.e., to be active prior to the object's allocation, and subsequent to the last reference to the object; as a consequence, the object may be deallocated safely upon its exit.

This strategy is intended as directly analogous to one for the placement of data within a hierarchical shared memory. Ultimately, the goal of the analysis framework we have designed is to facilitate the automatic parallelization of a program; let us suppose that this parallelism is realized in the following very simple way: where the sequential execution of the program describes a procedure calling tree, the parallel version describes exactly the same tree, except that for every node that has been successfully parallelized, the node's children are evaluated simultaneously instead of sequentially. (This is a simple model of parallelization, but realistic enough, as we could certainly rewrite the sequential and parallel versions of a program in a form that corresponds to this model.) See Figure 44. In it, a procedure calling tree is depicted. Assume that after parallelization, the descendants of λ_2 are to be evaluated simultaneously, as are the descendants of λ_{11}; these two nodes are highlighted in the figure.

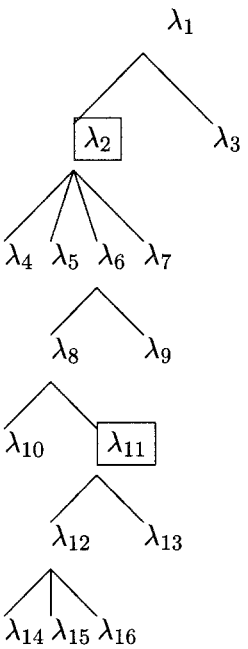

Figure 44: A Procedure Calling Graph

Now suppose that we use the following (again, realistic enough) model of parallel execution within a shared memory machine: upon execution of a parallelized node of the program, the machine is partitioned into d submachines, where d is the out-degree (number of children) of the node. When, within one such submachine, another parallel node is encountered, the submachine is further partitioned into submachines, and so on. The process of subdivision ends when a submachine contains only one processor, and the subcomputation performed by that processor is executed sequentially. To each submachine, assume there corresponds a (lowest) level in the hierarchical memory which is visible to all processors in the submachine. Any data which must be shared among the processors of the submachine must be placed at this level of the hierarchy, or at a higher (more widely visible) level; data to be shared only among the processors of the submachine may be placed at exactly this level of the hierarchy. The execution depicted in Figure 44 represents two divisions of the machine into submachines: once at λ_2 (where the entire machine is partitioned into four submachines), and once at λ_{11} (where one of the submachines is further partitioned into two submachines).

When cast in these terms, the problem of placing each dynamically allocated object within the memory hierarchy is identical to the problem of hierarchical deallocation we have been discussing; we must place every object at a level of the hierarchy such that it is visible to the entire submachine that executes the subtree of computation that delimits the object's lifetime. Exactly as before, this entails locating a node that is above (an ancestor of) the subtree of computation that contains the object's lifetime; this node corresponds to a level in the memory hierarchy at which the object may be safely placed. Of course, we would like to find the lowest such node for the sake of reducing latency and congestion in the memory hierarchy. Adapting the techniques of subsection 2.10.3 directly, any parallelized procedure through which the object makes no upward movement will suffice; when the object is allocated, we place it such that it is visible to the submachine on which the innermost instance of that procedure is executing; this guarantees that the object has sufficient visibility.

Referring to Figure 44, consider a variable instance $\dot{\mathbf{x}}$ that is bound by λ_{14}, and captured as a free variable by a closure formed within λ_{14}. Let λ_x be the nearest ancestor of λ_{14} through which $\dot{\mathbf{x}}$ describes no upward movements. If $\lambda_x = \lambda_{14}$ or λ_{12}, then $\dot{\mathbf{x}}$ may be placed in the hierarchy such that it is visible only to the submachine that executes the subtree of computation rooted at λ_{12} (this may be the private memory of a processor). If $\lambda_x = \lambda_{11}$ or λ_8 or λ_6, then $\dot{\mathbf{x}}$ may be placed such that it is visible only to the submachine that executes the subtree of computation rooted at λ_6. Finally, if $\lambda_x = \lambda_2$ or λ_1, then $\dot{\mathbf{x}}$ must be placed at the global level. As with

our artificial version of this problem, there are several possible strategies. One is to form, at compile-time, a set X of the procedures through which x makes no upward movements, and to allocate $\dot{\mathrm{x}}$ by traversing links of the calling tree at run-time, until the first member of X is encountered; this will correspond to a level of the memory hierarchy at which $\dot{\mathrm{x}}$ may safely be allocated. Alternatively, using additional information about the structure of the calling graph, we may make a purely static decision about which level of the hierarchy at which to place $\dot{\mathrm{x}}$, or perhaps simply how many levels "upward" in the hierarchy, from the point at which it is created, it must be placed to have adequate visibility.

2.16 In the Absence of `call/cc`

It is worth asking what penalty is paid, in the accuracy of our compile-time analysis, because of Scheme's feature of first-class continuations. There are at least two major improvements that can be made to the analysis, in its absence. We begin with a revised exact semantics ($\mathcal{E}_6$) for the subset of $\mathcal{L}$ that does not include `call/cc` (see Figures 45 and 46). The domains for this semantics are the same as for $\mathcal{E}_2$ (see Figure 6).

There is but one important change from the definition of $\mathcal{E}_2$ (apart from the deletion of any text that pertains to first-class continuations): the evaluation of a `return` form now results in a state whose procedure string is $p + \alpha^u$, where p is the procedure string of the state prior to the return, and λ_α is the procedure being deactivated. This is a perfect complement to the $p + \alpha^d$ construction that describes a procedure invocation, and raises the question, why could we not compute the procedure string described by a return so easily in the presence of `call/cc`? In effect, the new definition says that if p' is the procedure string in effect when a procedure λ_α is applied, and p is the procedure string in effect at the point of return from that invocation, then $Inv(p - p') = \alpha^u$ always (in the absence of `call/cc`). It is easy to construct an example to show that this statement does not hold in the presence of the procedure re-activations made possible by `call/cc`. For instance, let $p = p' + \alpha^d\alpha^u\beta^u\beta^d\alpha^d$, and suppose that p describes the following scenario: in a state whose procedure string is p', and in which procedure λ_β is active, λ_α is applied via `call/cc`, the continuation created by `call/cc` is captured in a global variable, control returns from λ_α, and likewise from λ_β. Then the continuation created while λ_α was active is retrieved from the global variable, and applied, resulting in a state whose procedure string is p.

Now consider a return from the (re)activated instance of λ_α: the suffix $Inv(p - p') = \alpha^u\beta^u\beta^d$ will be appended to p. (We may verify that

$$Net(p + Inv(p - p')) = Net\, p',$$

Let $q = \langle i, p, b, e, k, o, r \rangle \in Q$. Then $\mathcal{S}_6 : Q \to Q$ is defined as follows:

$S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_m\texttt{))}]\!]$ or $S_i = [\![\texttt{(set! x (call/cc f))}]\!] \Rightarrow$
if $e\langle [\![\texttt{f}]\!], b[\![\texttt{f}]\!] \rangle = \langle \alpha, b' \rangle \in C$
then $\mathcal{S}_6 q = \langle\ j,$
$p + \alpha^d,$
$b'[p + \alpha^d / [\![\texttt{z}_1]\!]] \cdots [p + \alpha^d / [\![\texttt{z}_n]\!]],$
$e',$
$\langle i, b, p, o \rangle,$
$p + \alpha^d,$
$r[k/o] \rangle$
where $\lambda_\alpha = [\![\texttt{(lambda (z}_1 \cdots \texttt{z}_m\texttt{) <z}_{m+1} \cdots \texttt{z}_n\texttt{> } S_j \cdots\texttt{)}]\!]$
and $e' = e[e\langle [\![\texttt{y}_1]\!], b[\![\texttt{y}_1]\!] \rangle / \langle [\![\texttt{z}_1]\!], p + \alpha^d \rangle] \cdots$
$[e\langle [\![\texttt{y}_m]\!], b[\![\texttt{y}_m]\!] \rangle / \langle [\![\texttt{z}_m]\!], p + \alpha^d \rangle]$

$S_i = [\![\texttt{(set! f (lambda}_\alpha \texttt{ (x}_1 \cdots \texttt{x}_m\texttt{) <x}_{m+1} \cdots \texttt{x}_n\texttt{> } \cdots \texttt{))}]\!] \Rightarrow$
$\mathcal{S}_6 q = \langle Succ\, i, p, b, e[\langle \alpha, b \rangle / \langle [\![\texttt{f}]\!], b[\![\texttt{f}]\!] \rangle], k, o, r \rangle$

$S_i = [\![\texttt{(if x (goto m) (goto n))}]\!] \Rightarrow$
$\mathcal{S}_6 q = \langle$ if $e\langle [\![\texttt{x}]\!], b[\![\texttt{x}]\!] \rangle = true$ then m else $n, p, b, e, k, o, r \rangle$

$S_i = [\![\texttt{(return x)}]\!] \Rightarrow$
$\mathcal{S}_6 q = \langle Succ j, p + \alpha^u, b', e[e\langle [\![\texttt{x}]\!], b[\![\texttt{x}]\!] \rangle / \langle [\![\texttt{y}]\!], b'[\![\texttt{y}]\!] \rangle], r o', o', r \rangle$
where $S_j = [\![\texttt{(set! y } \cdots\texttt{)}]\!]$,
$\alpha = Container\, i,$
and $k = \langle j, b', p', o' \rangle$

$S_i = [\![\texttt{(end)}]\!] \Rightarrow$
$\mathcal{S}_6 q = q$

Figure 45: The Semantic Function $\mathcal{S}_6$

$\mathcal{E}_6 : Q \to Q \equiv \lambda q.$ Let $q' = \mathcal{S}_6 q$
in if $q' = q$ then q else $\mathcal{E}_6 q'$

Figure 46: The Semantic Function $\mathcal{E}_6$

as per Theorem 3, as follows:

$$\begin{aligned} Net(p + Inv(p - p')) &= Net(p' + \alpha^d\alpha^u\beta^u\beta^d\alpha^d + \alpha^u\beta^u\beta^d) \\ &= Net(p' + \beta^u\beta^d) \\ &= Net\ p' \end{aligned}$$

because λ_β is the procedure that applies λ_α, and thus $p' = \cdots\beta^d$.) The important point is that such a re-activation is necessary to create a situation in which $Inv(p-p') \neq \alpha^u$ at the point of return from λ_α. We can formalize this result as follows.

Theorem 30 *Let $q = \langle i, p, b, e, k, o, r\rangle$ be a state during evaluation under $\mathcal{E}_6$, such that $p \neq \epsilon$, where $k = \langle j, b', p', o'\rangle$, and $\alpha =$ Container i. Then $Inv(p - p') = \alpha^u$.*

Proof: By induction on the number of states in the evaluation sequence. Let q be the first state during evaluation with a non-empty procedure string p. We have therefore that $p = \alpha^d$ for some $\alpha \in \Lambda$, and $p' = \epsilon$. In this case $Net(p - p') = \alpha^d$ and $Inv(p - p') = \alpha^u$ trivially.

Now assume the theorem is true for sequences of fewer than n states, and let q be the n^{th} state during evaluation. If q results from an application of λ_α, then $p = p' + \alpha^d$, $Net(p - p') = \alpha^d$, and $Inv(p - p') = \alpha^u$. Otherwise, if λ_α is active in q, and q results from the return from an instance of λ_β, then let p'' be the procedure string of the state in which the application of this instance of λ_β occurs, and let p''' be the procedure string of the state immediately prior to q. By the definition of $\mathcal{S}_6$, $p = p''' + \beta^u$. By induction, $Inv(p''' - p'') = \beta^u$, and therefore $Net(p''' - p'') = \beta^d$. Likewise, by induction, $Inv(p'' - p') = \alpha^u$ and therefore $Net(p'' - p') = \alpha^d$. By the choices of p, p', p'', and p''',

$$\begin{aligned} Net(p - p') &= Net((p'' - p') + (p''' - p'') + (p - p''')) \\ &= Net(Net(p'' - p') + Net(p''' - p'') + Net(p - p''')) \\ &= Net(\alpha^d + \beta^d + \beta^u) \\ &= Net(\alpha^d) \\ &= \alpha^d \end{aligned}$$

and therefore $Inv(p - p') = \alpha^u$. Otherwise (if q does not result from procedure application or return) then no interprocedural movements take place in the evaluation step that leads to q, and $Inv(p - p') = \alpha^u$ by induction. □

This result has enormous significance for the accuracy of our static analysis, because it implies that we may use a construct analogous to $p +$

Let $\hat{t} = \langle \hat{p}, \hat{b}, \hat{e}, \hat{k}, \hat{r} \rangle, i \in N$. Then $\mathcal{S}_7 : N \to \hat{T} \to \hat{Q}$ is defined, according to the form of S_i, as follows.

$$
\begin{array}{l}
S_i = [\![\texttt{(set! x (f y}_1 \cdots \texttt{y}_n\texttt{))}]\!] \Rightarrow \\
\quad \mathcal{S}_5 i \hat{t} = \sqcup_{\hat{Q}} \Big\{ \lambda i'.\ \text{if } i' \neq j \\
\qquad\qquad \text{then } \perp_{\hat{T}} \\
\qquad\qquad \text{else } \langle \hat{p} \oplus \hat{p}', \\
\qquad\qquad\quad (\mathit{Move}_{\hat{B}} \hat{b}' \hat{p}')[\lambda\alpha.\{\epsilon\}/[\![\texttt{z}_1]\!]] \cdots [\lambda\alpha.\{\epsilon\}/[\![\texttt{z}_n]\!]], \\
\qquad\qquad\quad \mathit{Move}_{\hat{E}} \hat{e}' \hat{p}', \\
\qquad\qquad\quad \langle \{i\}, \hat{b}, \hat{p} \rangle, \\
\qquad\qquad\quad \hat{r}[\hat{k} // \mathit{Container}\ i] \rangle \\
\qquad\quad \text{where } \lambda_\alpha = [\![\texttt{(lambda (z}_1 \cdots \texttt{z}_m\texttt{) <z}_{m+1} \cdots \texttt{z}_n\texttt{> } S_j \cdots\texttt{)}]\!], \\
\qquad\qquad \hat{p}' = \mathit{Abs}_P(\alpha^d), \\
\qquad\quad \text{and} \quad \hat{e}' = \hat{e_m} \text{ where } \hat{e_0} = \hat{e} \\
\qquad\qquad\qquad\qquad \text{and} \quad \hat{e_l} = \mathit{WrEnv}\ \hat{e_{l-1}}[\![\texttt{z}_l]\!]\{\epsilon\} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{RdEnv}\ \hat{e}[\![\texttt{y}_l]\!]((\hat{b}[\![\texttt{y}_l]\!])\alpha_l)) \\
\qquad\qquad\qquad\qquad\qquad\quad \text{where } [\![\texttt{y}_l]\!] \text{ is bound by } \lambda_{\alpha_l},\ 1 \leq l \leq m \\
\qquad\quad \Big| \ \alpha \in \hat{\alpha} \Big\} \\
\qquad \text{where } \hat{e}[\![\texttt{f}]\!] = \langle \hat{c'}, \ldots \rangle \\
\qquad \text{and} \quad \hat{c'} = \langle \hat{\alpha}, \hat{b'} \rangle
\end{array}
$$

Figure 47: The Semantic Function $\mathcal{S}_7$ (Part I)

(namely, $\hat{p} \oplus \mathit{Abs}_P(\alpha^u)$) to model procedure return, in the absence of `call/cc`. In Figures 47, 48 and 49 are defined an abstract interpretation (called $\mathcal{E}_7$) of the subset of $\mathcal{L}$ treated by $\mathcal{E}_6$, that makes use of this observation. The domains for $\mathcal{E}_7$ are exactly as for $\mathcal{E}_5$, and the correctness of $\mathcal{E}_7$ follows immediately from the correctness of $\mathcal{E}_5$ and Theorem 30. The analysis of side-effects and object lifetimes based upon $\mathcal{E}_7$ is more accurate in its treatment of upward movements than $\mathcal{E}_5$, for the reason that as an object X moves upward (via a return from λ_α) under $\mathcal{E}_7$, it is subject to the movement $\mathit{Move}\ X(\mathit{Abs}_P(\alpha^u))$, rather than the movement $\mathit{Move}\ X(\hat{\mathit{Inv}}\ \hat{p'})$ (see the meaning of a return form under $\mathcal{E}_5$, in Figure 36). Therefore the effect of any inaccuracy which accumulates in $\hat{p'}$ under $\mathcal{E}_5$, is eliminated entirely under $\mathcal{E}_7$, since $\mathit{Abs}_P(\alpha^u)$ is (by Theorem 30) as accurate an abstraction of $\mathit{Inv}(p - p')$ as possible, in the absence of `call/cc`.

There is also a considerable simplification in the treatment of continu-

$$
\begin{array}{l}
S_i = [\![\texttt{(set! f (lambda}_\alpha\ \texttt{(x}_1 \ldots \texttt{x}_m\texttt{)}\ \texttt{<x}_{m+1} \cdots \texttt{x}_n\texttt{>}\ \cdots\texttt{))}]\!] \Rightarrow \\
\quad \mathcal{S}_7 i \hat{t} = \lambda i'.\ \text{if } i' \neq \mathit{Succ}\ i \\
\qquad\qquad \text{then } \perp_{\hat{T}} \\
\qquad\qquad \text{else } \langle \hat{p}, \\
\qquad\qquad\qquad \hat{b}, \\
\qquad\qquad\qquad \mathit{WrEnv}\ \hat{e}[\![\texttt{f}]\!]((\hat{b}[\![\texttt{f}]\!])\beta)\langle\langle\{\alpha\}, \hat{b}\rangle, \perp_{\hat{PrimOp}}, \perp_{\hat{Int}}, \perp_{\hat{Bool}}\rangle \\
\qquad\qquad\qquad \hat{k}, \\
\qquad\qquad\qquad \hat{r}\rangle \\
\qquad\qquad\qquad \text{where f is bound by } \lambda_\beta
\end{array}
$$

$$
\begin{array}{l}
S_i = [\![\texttt{(if x (goto m) (goto n))}]\!] \Rightarrow \\
\quad \mathcal{S}_7 i \hat{t} = \lambda i'.\ \text{if } i \notin \{m, n\} \\
\qquad\qquad \text{then } \perp_{\hat{T}} \\
\qquad\qquad \text{else } \hat{t}
\end{array}
$$

$$
\begin{array}{l}
S_i = [\![\texttt{(return x)}]\!] \Rightarrow \\
\quad \mathcal{S}_7 i \hat{t} = \sqcup_{\hat{Q}} \Big\{ \lambda i'.\ \text{if } i' \neq j \\
\qquad\qquad\qquad \text{then } \perp_{\hat{T}} \\
\qquad\qquad\qquad \text{else } \langle \hat{p}', \\
\qquad\qquad\qquad\qquad \hat{b}', \\
\qquad\qquad\qquad\qquad \mathit{Move}_E\Big(\mathit{WrEnv}\ \hat{e}[\![\texttt{y}]\!]((\hat{b}'[\![\texttt{y}]\!])\beta)(\mathit{RdEnv}\ \hat{e}[\![\texttt{x}]\!]((\hat{b}[\![\texttt{x}]\!])\alpha)) \Big) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Abs}_P(\gamma^u) \\
\qquad\qquad\qquad\qquad \hat{r}\gamma, \\
\qquad\qquad\qquad\qquad \hat{r}\rangle \\
\qquad\qquad\qquad\qquad \text{where y is bound by } \lambda_\beta \\
\qquad\qquad\qquad\qquad \text{and } \quad \texttt{x} \text{ is bound by } \lambda_\alpha \\
\qquad\qquad\qquad \text{where } S_j = [\![\texttt{(set! y } \cdots\texttt{)}]\!] \\
\qquad\qquad\qquad \text{and } \quad \mathit{Container}\ j = \gamma \\
\qquad\qquad\qquad \Big| \ j \in \hat{j} \Big\} \\
\quad \text{where } \hat{k} = \langle \hat{j}, \hat{b}', \hat{p}' \rangle
\end{array}
$$

Figure 48: The Semantic Function $\mathcal{S}_7$(Part II)

$$\mathcal{S}_7^* : \hat{Q} \to \hat{Q} \equiv \lambda \hat{q}.\, \sqcup_{\hat{Q}} \{\mathcal{S}_7 i(\hat{q}i) \mid i \in N\}$$

$$\mathcal{E}_7 : Q \to Q \equiv \lambda \hat{q}.\, \text{Let } \hat{q'} = \mathcal{S}_7^* \hat{q} \text{ in if } \hat{q'} \sqsubseteq_{\hat{Q}} \hat{q} \text{ then } \hat{q} \text{ else } \mathcal{E}_7(\hat{q} \sqcup_{\hat{Q}} \hat{q'})$$

Figure 49: The Semantic Functions $\mathcal{S}_7^*$ and $\mathcal{E}_7$

ations ($\hat{K}$) and restoration functions ($\hat{R}$) in $\mathcal{E}_7$, which makes it yet more accurate than $\mathcal{E}_5$. The justification for this simplification comes from the following theorem.

Theorem 31 *Let $q_i = \langle i, p, b, e, k, o, r \rangle$ be a state during the evaluation of $\mathcal{E}_6\, q_0$. Then*

$$Net(p - o) = \epsilon.$$

Proof: By induction on i. If $i = 0$, then $p = o = \epsilon$ trivially. Else, suppose that the theorem holds for $i < n$, let $i = n$, and let $q_{n+1} = \langle i', p', b', c', k', o' \rangle$. If q_{n+1} results from a procedure application in state q_n, then $p' = o'$ and $Net(p - o) = \epsilon$. Else, if q_{n+1} results from a return from procedure λ_γ in state q_n, then $p' = p + \alpha^u$. Let $q_j = \langle i'', p'', b'', c'', k'', o'' \rangle$ be the state in which the this instance of λ_γ was applied. Then

$$\begin{aligned} Net(p' - o') &= Net((p'' + (p - p'') + \gamma^u) - o') \\ &= Net((p'' + Net(p - p'') + \gamma^u) - o') \\ &= Net((p'' + \gamma^d + \gamma^u) - o') \end{aligned}$$

since by Theorem 30, $Inv(p - p'') = \gamma^u$. Therefore

$$\begin{aligned} Net(p' - o') &= Net(p'' - o'') \\ &= Net(p'' - o'') \end{aligned}$$

since o' and o'' are equal to the birth date of the procedure instance to which control returns in q_{n+1}. Then, by induction,

$$Net(p' - o') = \epsilon.$$

□

Now consider the continuation k associated with a procedure instance. Under $\mathcal{E}_2$, k is "saved" in a restoration map $r \in R$, subjected to inter-procedural movements during the computation that follows, and retrieved whenever control returns to the procedure instance. The important point is this: that whenever control returns to the procedure instance, the continuation k associated with that instance has undergone a net movement given by $Net(p - o)$, where p is the current procedure string and o is the birth date of the procedure instance. But under $\mathcal{E}_6$, $Net(p - o) = \epsilon$ always, and so the net effect of the movement experienced by a continuation (whenever the continuation is used) is empty. Since stack configurations represent only *Net* movements, under $\mathcal{E}_7$ we therefore dispense entirely with the use of $Move_K$ and $Move_R$. The result is that $\mathcal{E}_7$ is more accurate than $\mathcal{E}_5$ in its approximation to the continuations of a program. While this is not important for the analysis of side-effects and object lifetimes, it may be significant if we make use of the continuations directly, for example, to construct an approximation to the procedure calling graphs that may be described by the program's execution.

But alas, `call/cc` and first-class continuations are firmly implanted in the Scheme definition. Or are they? We might indulge in a moment's wishful thinking, and consider CPS (*continuation passing style*) conversion [5] as a way out of our difficulties. When a program is rewritten in continuation passing style, to its every lambda expression is added a parameter. The value of this parameter is a closure of one argument, called the *continuation* of the procedure. In lieu of its normal return sequence, the procedure applies its continuation to the value it would return. For example, the function `fact` is shown before and after CPS conversion, in Figure 2.16. The converted function is invoked as `(fact (lambda`$_\gamma$ `(x) x) 10)`. One advantage of CPS conversion is that we may define `call/cc` simply as

```
(lambda (k f) (f k k))
```
[21]

Continuation passing style would seem to be the solution! We may use it to implement first-class continuations (that is, continuations created via `call/cc`) in terms of closures, and therefore analyze the resulting program using $\mathcal{E}_7$ rather than $\mathcal{E}_5$.

To see what's wrong with this, let's consider the procedure strings described by the evaluation of `(fact (lambda`$_\gamma$ `(x) x) 5)`, where `fact` is in CPS, as per Figure 2.16. It is easy to see that when the top-level continu-

[21] This definition leaves a bit to be desired, as it does not permit `f` to be a continuation, since we have said that continuations are procedures of one argument. We may remedy this by making continuations procedures of two arguments, that simply ignore their first argument.

```
(define fact (lambda (n)
  (if (= n 0)
      1
      (* n (fact (1- n))))))
(define fact (lambda_α (k n)
  (if (= n 0)
      (k 1)
      (fact (lambda_β (m) (k (* n m)))
            (1- n)))))
```

Figure 50: `fact`, Before and After CPS Conversion

ation (λ_γ) is applied, the procedure string in effect is

$$\alpha^d\alpha^d\alpha^d\alpha^d\alpha^d\alpha^d\beta^d\beta^d\beta^d\beta^d\beta^d\beta^d.$$

In short, the evaluation of a program converted to CPS describes only *downward* movements (until the top-level continuation is applied, and the entire string unwinds). If, therefore, we were ever to ask if a variable could be stack-allocated, the answer would always be yes, because to our analysis it appears that once activated, a procedure remains active to the end of the computation (and thus any variables it binds enjoy indefinite extent, even though allocated on the stack). Likewise, our analysis of side-effects would conclude that a reference to a variable instance $\dot{\mathtt{x}}$ induces a side-effect in every procedure that is activated between the instantiation of $\dot{\mathtt{x}}$ and the state in which the reference occurs, because each such procedure appears to be active at the point of reference to $\dot{\mathtt{x}}$. This is to be expected, since the means by which upward movements are introduced into procedure strings (procedure return) is replaced by further procedure application in CPS. Put another way, if we were to convert a program to CPS, and execute it according to the operational semantics of procedure activation implicit in our analysis framework, the program would continually allocate stack space as it ran, while deallocating none. It would then be vacuous to observe that all its variables could be stack-allocated. This is not an indictment of CPS; it is simply to say that the model of procedure activation and deactivation it assumes differs from that built into the semantics of procedure strings.

The upshot of this seems to be that a real price is paid, in the accuracy of our compile-time analysis, for the power of `call/cc`.

2.17 Environment Pruning

There is a final improvement to both the accuracy and efficiency of the abstract semantics that we will discuss only informally. In the absence of

`call/cc`, it is possible to reduce the amount of information passed across procedure boundaries (during the analysis) by the following observation. When a procedure λ_α is invoked by λ_β, λ_β need transmit to the initial state of execution within λ_α, only the values of those variable instances in the environment prior to the application, that will be referenced during the subcomputation initiated by the application. If λ_β is the function `(lambda (x y) (+ x y))`, the only variable instances whose values are relevant are the fresh instances of `x` and `y`; if, however, λ_β is a more involved procedure, which invokes yet further procedures, then an appreciable fraction of the environment at the point of application may be needed during the subcomputation. We can be more precise about what variable instances might be accessed during the subcomputation. Let $\hat{c}$ be the (abstract) closure of λ_α that is applied by λ_β, and let $\dot{\mathtt{x}}$ be an instance of a variable `x`, such that $\dot{\mathtt{x}}$ is instantiated prior to the application of $\hat{c}$. If $\dot{\mathtt{x}}$ is to be accessed during the subcomputation initiated by the application of $\hat{c}$, then it must either occur free in $\hat{c}$, or in a closure that is accessible through a variable instance that occurs free in $\hat{c}$, or in a closure that is accessible through a variable instance that occurs free in a closure that is accessible through a variable instance that occurs free in $\hat{c}$, etc. (This transitive dependence is the same as that described by Theorem 6.) Let $\hat{e_0}$ be the (abstract) environment that contains only bindings for the parameters to $\hat{c}$, and the free variable instances captured by $\hat{c}$. $\hat{e_0}$ is a subset of the environment in effect when $\hat{c}$ is applied, extended with bindings for the parameters of λ_α; the variable instances in the domain of $\hat{e_0}$ are therefore mapped to the values they have at the commencement of execution within $\hat{c}$. Let $\hat{e_1}$ be the environment which is created by extending $\hat{e_0}$ to include any variable instances that occur free within closures that are found among the values in $\hat{e_0}$. That is, if $\dot{\mathtt{y}}$ has the closure c' as its value in $\hat{e_0}$, and c' has captured a free variable instance $\dot{\mathtt{z}}$, then $\dot{\mathtt{z}}$ is added to $\hat{e_1}$, and is mapped to the value it has at the point of application of $\hat{c}$. We continue generating environments $\hat{e_i}$ in this way until no further extension is possible. (We are forming the transitive closure of $\hat{e_0}$, under the relation "occurs as a free variable instance in a closure found in the environment".) It is easy to show that the resulting environment (call it $\hat{e_*}$) contains every variable instance whose value prior to the application of $\hat{c}$ might be needed during the subcomputation initiated by the application. This observation works both to improve the accuracy of our static analysis, since it reduces the portion of the environment at each call that is subjected to the *Move* functions, and to reduce the expense of analysis, assuming that the time spent to compute the transitive relation described above is less than is spent manipulating an unnecessarily bulky environment. Consider the example of `(lambda (x y) (+ x y))` - admittedly a trivial function. Assuming that `x` and `y` cannot take closures as their values, $\hat{e_*}$ will be quickly computed, and if this function is invoked from a

state with a large environment, an appreciable savings will be realized.

An analogous trimming of the environment can be performed when a procedure is deactivated. We observe that for a variable instance to persist beyond the lifetime of a procedure instance, it must either occur free in the caller of the procedure, or in a closure that is accessible to the caller, or in a closure that is accessible from a closure that is accessible to the caller, etc. To return again to our trivial example, instances of `x` and `y` cannot persist beyond the deactivation of `(lambda (x y) (+ x y))`, because they cannot occur free in the caller, nor are they captured by any closures.

3 The Automatic Parallelization of Scheme Programs

We have seen how interprocedural analysis is used in Parcel to assess the dependence structure and object lifetimes of a Scheme program. In this section we will see how the compiler puts this information to use in restructuring the program for execution on a shared-memory multiprocessor. Our assumptions concerning the target architecture will be few: we will envision it as a number of identical processors sharing a memory. In [3], the individual transformations that are discussed below, were presented in their technical details, in terms of the control flow and dependence graphs that might be manipulated by a compiler. Here, the goal will instead be to portray the compilation process in its entirety. We will proceed by following several example programs, as they are subjected to the restructuring transformations of Parcel, and we will concentrate upon the intuition underlying each transformation, its contribution to the shaping of the program for efficient, parallel execution. The figures in the text below are produced by Parcel, and are simply human-readable renderings of the compiler's data structures, depicted at intervals during compilation, with the goal of providing "snapshots" of the restructuring process.

Quicksort seems an ideal algorithm with which to introduce the transformations performed by Parcel, for the reason that it is probably familiar to the reader, it performs some simple but representative list manipulations, and it includes a tail-recursive procedure that will serve to introduce Parcel's treatment of iterative computation. It is necessary to see how iterative computation is treated, before we can move to Parcel's treatment of recursive (not merely tail-recursive) computation. Parcel has been designed to be an optimizing compiler for parallel shared memory architectures, and not merely a compiler that detects parallelism; in the restructuring of quicksort, we will see a variety of transformations of which it is capable, that contribute to the speed of the object codes it produces, but which are not parallelizing transformations per se.

```
(define sortby
  (lambda (f l)
     (if (null? l)
         '()
         (let
            ((l-and-r (splitby f (cdr l) (car l) '() '())))
            (append
               (sortby f (car l-and-r))
               (list (car l))
               (sortby f (cdr l-and-r))))) ))
(define splitby
  (lambda (f x partition left right)
    (cond ((null? x) (cons left right))
          ((> (f partition) (f (car x)))
           (splitby f (cdr x) partition
                    (cons (car x) left) right))
          (#t
           (splitby f (cdr x) partition
                    left (cons (car x) right))))))
(sortby id (read))
```

Figure 51: Quicksort

3.1 The Program Representation

The compiler begins with the definition of quicksort given in Figure 51. The algorithm consists of two procedures, `sortby` and `splitby`. `sortby` takes a procedure `f` and a list `l`, and produces a sorted list. The procedure `f`, when applied to an element of `l`, is expected to return a number; this value is used to compare the element to other elements of `l`. If `l` is a list of numbers, then the identity procedure may be used; in this example, it is assumed that such is the list read from input (via `read`), and thus `f` is given the value `id` (the identity procedure) by the top-level invocation of `sortby`. `splitby` divides a list `x` into two lists, `left` and `right`, of those elements less than the partition element, and those greater than or equal to the partition element, respectively. The parameters `left` and `right` are initially null; at each recursive invocation of `splitby`, the first element of `x` is added either to the head of `left` or `right`. `sortby` uses `splitby` to effect one such division of `l`, using the first element of `l` as the partition. It then applies itself recursively to the two resulting sublists, and concatenates the sorted results, placing the partition element between them.

Parcel treats a core subset of Scheme that includes only a handful of special forms: `lambda`, `define`, `cond`, `set!`, etc. All other special forms that are visible to the user are provided as macros. The version of the example program that is actually parsed by Parcel is therefore the macro-

```
(define sortby
  (lambda (f l)
    (cond
      ( (null? l)
        '() )
      ( #t
        ( (lambda (l-and-r)
            (append
              (sortby f (car l-and-r))
              (list (car l))
              (sortby f (cdr l-and-r))) )
          (splitby f (cdr l) (car l) '() '()) )
        ) ) ))
(define splitby
  (lambda (f x partition left right)
    (cond
      ( (null? x)
        (cons left right) )
      ( (> (f partition) (f (car x)))
        (splitby f (cdr x) partition
                 (cons (car x) left) right) )
      ( #t
        (splitby f (cdr x) partition
                 left (cons (car x) right)) ) ) ))
(sortby id (read))
```

Figure 52: Quicksort Program, after Macro-Expansion

expanded version of Figure 52. Only a few changes have occurred in the definition of `sortby`: an `if` form has been rewritten as a `cond`, and a `let` has been expanded into a nested `lambda` expression, in the usual way.

From here, the compiler begins its work. It will first parse the program, and rewrite it in a language similar to $\mathcal{L}$, as defined in Section 2. See Figures 53 and 54. Alas, we have left the orderly world of Scheme syntax, and entered the murky realm of compiler data structures made manifest by a simple pretty-printer. There are four `lambda` expressions depicted in this figure, and the compiler has named them `$-$`, `$-$-splitby`, `$-$-sortby`, and `$-$-sortby-&`. There are two `lambda` expressions known to the compiler, that are not written explicitly by the user, but are part of every program; these are called `$` and `$-$`. `$` is the outermost `lambda` expression, by which, conceptually, the globally defined variables made available by the system to the programmer, are bound. For example, `car`, `cdr`, `append`, and `call/cc` are bound by `$`. The action of `$` is to apply `$-$`, which is the `lambda` expression that represents the top level of the user's program, and by which, conceptually, all of his top-level variables are bound. `$` is not

depicted here, because it is boring and will remain so throughout compilation. The compiler constructs the name `$-$-sortby-&` by concatenation of the name of the containing lambda expression (`$-$-sortby`) with the suffix `-&`, in the hope that the name will be suggestive of the lexical position of the inner lambda expression.

There is, visible to the compiler but not to the user, an environment which surrounds that of `$`, called the `nil` environment. The environment of `$` and the `nil` environment are identical, except that variables in the `nil` environment cannot be altered by the user program, while, for example, the user may write `(set! car call/cc)` and affect the definition of `car` that is used by his code (that is, the variable `car` bound by `$`). The `nil` environment simply permits the compiler to refer to top-level variables without worrying that the user will clobber their values.

Consider the definition of `$-$`. The syntax

```
(lambda () <sortby splitby t-56 t-57> ...  )
```

means that `$-$` is a `lambda` expression of no arguments, that has four local variables (`sortby`, `splitby`, `t-56` and `t-57`). Its action is to form closures of `$-$-sortby` and `$-$-splitby`, to call `read` to fetch a list of numbers, to apply `sortby` to this list, and to return the result. The formation of a closure of `$-$-sortby` will be denoted as `#<$-$-sortby>`, and likewise for `lambda` expressions by other names. It makes for easier reading, if we concentrate upon one `lambda` expression at a time, and summarize nested ones in this way; the compiler, too, treats the program one `lambda` expression at a time. The closures of `$-$-sortby` and `$-$-splitby` are passed to the procedure `id` before being assigned into a variable for the reason that every expression treated by the compiler, through most of the restructuring phase of compilation, is of the form `(set! l (f a b ... c))` where `l`, `f`, `a`, `b` and `c` are variables, constants or closures, but not further applications or special forms. The resemblance to the language $\mathcal{L}$ of Section 2 should be clear.

Let us move on to `$-$-splitby`. It has a long list of local variables, all compiler-generated temporaries. There are many intrinsic procedures applied here (`car`, `cdr`, `cons`, `>`, `null?`), but at this point the compiler does not know that such intrinsics are being applied; it sees only references to variables bound by `$`, and as we mentioned, the user may modify such variables. After parsing the program, the compiler launches into an interprocedural analysis based upon the techniques of Section 2, and after this analysis, it will be aware that the *variable* `car` has the *procedure* `car` as its value, at all points within `$-$-splitby`, and its representation of the program will change to reflect this knowledge: the reference to the variable

```
($-$ =
  (lambda ()
    <sortby splitby t-56 t-57>
    (set! sortby (id #<$-$-sortby>))
    (set! splitby (id #<$-$-splitby>))
    (set! t-57 (read))
    (set! t-56 (sortby id t-57))
    (return t-56) ))
($-$-splitby =
  (lambda (f x partition left right)
    <t-44 t-45 t-46 t-47 t-48 t-49 t-50 t-51 t-52 t-53 t-54 t-55>
    (set! t-45 (null? x))
    (cond
      ( t-45
        (set! t-44 (cons left right)) )
      ( else
        (set! t-47 (f partition))
        (set! t-49 (car x))
        (set! t-48 (f t-49))
        (set! t-46 (> t-47 t-48))
        (cond
          ( t-46
            (set! t-50 (cdr x))
            (set! t-52 (car x))
            (set! t-51 (cons t-52 left))
            (set! t-44 (splitby f t-50 partition t-51 right)) )
          ( else
            (set! t-53 (cdr x))
            (set! t-55 (car x))
            (set! t-54 (cons t-55 right))
            (set! t-44 (splitby f t-53 partition left t-54))))))
    (return t-44) ))
```

Figure 53: The Initial Representation of Quicksort (Part 1)

```
($-$-sortby =
  (lambda (f l)
    <t-25 t-26 t-27 t-28 t-37 t-38>
    (set! t-26 (null? l))
    (cond
      ( t-26
        (set! t-25 (id '())))
      ( else
        (set! t-27 (id #<$-$-sortby-&>))
        (set! t-37 (cdr l))
        (set! t-38 (car l))
        (set! t-28 (splitby f t-37 t-38 '() '()))
        (set! t-25 (t-27 t-28)) ) )
    (return t-25) ))
($-$-sortby-& =
  (lambda (l-and-r)
    <t-30 t-31 t-32 t-33 t-34 t-35 t-36>
    (set! t-34 (car l-and-r))
    (set! t-31 (sortby f t-34))
    (set! t-35 (car l))
    (set! t-32 (list t-35))
    (set! t-36 (cdr l-and-r))
    (set! t-33 (sortby f t-36))
    (set! t-30 (append t-31 t-32 t-33))
    (return t-30) ))
```

Figure 54: The Initial Representation of Quicksort (Part 2)

`car` will be replaced by a reference to the "constant" (procedure) `car`. The same applies to all applications of intrinsic procedures within the program. Unfortunately, our printed representation of the compiler's data structures will not reflect this fact, but we will be clear about the meaning of such intrinsics when it is important to the discussion.

`$-$-sortby` and `$-$-sortby-&` are two procedures (and not one) only because of the definition of `let`, and not because there is a compelling reason for the computation to be divided, interprocedurally, in this way. The recursive calls to `$-$-sortby` occur within `$-$-sortby-&`. There are several transformations within Parcel that apply only to self-recursive procedures (procedure that invoke themselves directly). An artificial procedure boundary such as exists between `$-$-sortby` and `$-$-sortby-&` is an impediment to such transformations. We will return to this momentarily.

The first action of the compiler, once having built a representation of the source code, is to perform an interprocedural analysis based upon the results of Section 2. Let us assume therefore, that the analysis has been performed, and that for every expression of the program we have a *def* and *use* set; that is, a set of mutable quantities that may be defined and used as a result of evaluating the expressions. These sets will reflect both the local (visible) and interprocedural (remote) side-effects of the expression. (We are using *side-effect* here as per Definition 1.)

3.2 Preparatory Optimizations

Before restructuring the program for parallel execution, Parcel attempts to reorganize the computation to facilitate the discovery of parallelism, and to perform any traditional optimizations that are not at odds with the aim of automatic parallelization. The goals of this preparatory restructuring are straightforward: to eliminate spurious or artificial dependences, to enhance the visibility of the computation to the compiler, and generally to reduce and simplify the code without introducing additional dependences. From every procedure of the program, Parcel generates two versions: a sequential and a parallel (we will return to this). The preparatory phase of optimization is designed to be consistent with both; therefore the version of each procedure that emerges from this phase serves as the starting point for further refinement into both the parallel and sequential versions.

This phase of optimization is organized as a battery of individual optimizations which are applied to the program repeatedly, until no further improvements occur. This organization was chosen because the application of one transformation may create conditions that enable another, and so on; we wish to allow such propagation of transformations to occur until the program stabilizes into a fixpoint. We have not attempted to demonstrate

```
($-$-sortby =
  (lambda (f l)
    <t-25 t-26 t-27 t-28 l-and-r t-30 t-31
     t-32 t-33 t-34 t-35 t-36 t-37 t-38>
    (set! t-26 (null? l))
    (cond
      ( t-26
        (set! t-25 '()) )
      ( else
        (set! t-37 (cdr l))
        (set! t-38 (car l))
        (set! t-28 (splitby id t-37 t-38 '() '()))
        (set! l-and-r t-28)
        (set! t-34 (car l-and-r))
        (set! t-31 (sortby id t-34))
        (set! t-35 (car l))
        (set! t-32 (list t-35))
        (set! t-36 (cdr l-and-r))
        (set! t-33 (sortby id t-36))
        (set! t-30 (append t-31 t-32 t-33))
        (set! t-25 t-30) ) )
    (return t-25) ))
```

Figure 55: `$-$-sortby-&` is Merged into `$-$-sortby`

formally that such a fixpoint must be reached, but it is not difficult to reason informally about the interaction between transformations, and to arrange for a monotonicity in their net effect upon the program.

3.2.1 Contour Merging

Whenever it is possible to do so without an increase in the program size, Parcel expands procedures in-line. This means, in essence, that a closure that is applied in only one place, is open-coded at that single point of application, provided that the lexical environment at the point of application contains all of the variable bindings that occur free in the closure. This optimization is called *contour merging*, for the reason that it eliminates the needless lexical contours that arise from the use of `let`, `let*`, `letrec` and other binding forms. As we will see, when it is applied more generally than to the mere elimination of lexical contours, it is quite a powerful tool for enhancing the visibility of computation to the compiler, and for allowing optimizations to be applied to larger units of computation. It is easy to formulate a general test for the legality of contour merging in terms of the

stack configurations computed during interprocedural analysis. Intuitively, if each free variable of the closure makes no *net* (upward or downward) movements with respect to the procedure that binds it, from the point where it is bound to the point of application of the closure in which it occurs free, *and* if the free variable is in the lexical scope at the point of application, then the procedure may be expanded in-line at the point of application, as all of its free variable references will be to the correct bindings. When we say that a variable is "in the lexical scope" at a certain point, we are ignoring the possibility that it is shadowed by another variable of the same identifier. Parcel pays no attention to the identifiers associated with variables once the program has been parsed: it considers a variable to be in the lexical scope at a point in the program, if it is bound by a lambda expression that surrounds that point, textually. In effect, all variables are renamed to unique identifiers when the program is parsed.

To return to our example, Parcel discovers that the above condition applies (trivially) to the procedure `$-$-sortby-&`, and the procedure is expanded in-line at the point of its application. See Figure 55. The compiler has also applied dead code elimination, and so has deleted the formation of the closure of `$-$-sortby-&`. That is, the definition of `t-27` in Figure 54 is discovered to be useless and is eliminated.

Let us be more formal about the condition under which contour merging is correct, since it is a very useful transformation. Assume that we have performed interprocedural analysis using $\mathcal{E}_5$ or $\mathcal{E}_7$ as defined in Section 2, and let $\hat{c}$ be an (abstract) closure. Suppose that $\dot{\mathtt{x}}$ is a free variable instance in $\hat{c}$, and let $\hat{p}$ be the stack configuration that describes the movements that $\dot{\mathtt{x}}$ makes between the point at which it is instantiated, and a point of application of $\hat{c}$. If $\hat{p}\alpha = \{\epsilon\}$ where `x` is bound by λ_α (and if `x` is in the lexical scope at the point of application), then the same instance of `x` is visible at the points at which $\hat{c}$ is closed and applied. If this is true of every free variable instance in $\hat{c}$, then it may be expanded in-line at the point of application. Of course, the compiler must also determine that there is but one lambda expression applied at this point. This information is available directly from the results produced by $\mathcal{E}_5$ and $\mathcal{E}_7$: an abstract closure is represented as a set of lambda expression indices, and a function from variables to stack configurations; if the former has only one member, then it represents (concrete) closures of only one lambda expression.

There is a simple but important special case of this test: if $\hat{c}$ is closed at the top level of the program, that is, directly within the lambda expression `$` or `$-$`, then all of its free variables will be bound by by `$-$` or `$`, and we will always have that $\hat{p}\alpha = \{\epsilon\}$ for every free variable `x` in $\hat{c}$. Therefore $\hat{c}$ can be in-lined at any point where it is applied.

```
($-$-splitby =
  (lambda (f x partition left right)
    <t-44 t-45 t-46 t-47 t-48 t-49 t-50 t-51 t-52 t-53 t-54 t-55>
    (exit-block
      (repeat
        (cond
          ( x
            (set! t-48 (car x))
            (set! t-46 (> partition t-48))
            (cond
              ( t-46
                (set! t-52 t-48)
                (set! left (cons t-52 left)) )
              ( else
                (set! t-55 t-48)
                (set! right (cons t-55 right)) ) )
            (set! x (cdr x)) )
          ( else
            (go l-74:) ) ))
      (l-74: (set! t-44 (cons left right)) (return t-44))) ))
```

Figure 56: Tail-Recursion is Eliminated from `$-$-splitby`

3.2.2 Tail-Recursion Elimination

We would like to see the procedure call to `$-$-splitby` within `$-$-sortby` disappear by contour merging, but it won't happen so easily, for `$-$-splitby` is recursive, and therefore in-lining it at all its points of application would be a (serious) violation of our rule against increasing the program size. However, the compiler discovers that the procedure is tail-recursive, and transforms it into a loop. See Figure 56. The syntax of this figure requires some explanation. An expression of the form

```
(exit-block EXPR
            (L1: A1 A2 ···)
            (L2: B1 B2 ···)
                 ⋮
            (Ln: Z1 Z2 ···))
```

indicates that `EXPR` will contain branches (`go` forms) to the labels L_1 through L_n. When such a `go` form is evaluated, the expressions following the target label are evaluated from left to right, and control leaves the `exit-block` form. An expression of the form `(repeat EXPR)` indicates that `EXPR` is evaluated repeatedly; the repetition ceases only by an explicit branch out of the `repeat` form.

We will return shortly to the conditions under which tail-recursion elimination is correct. The mechanics of the transformation once it is determined to be applicable, are simple: each of the parameters is assigned the value to which it would be bound on a tail-recursive call, and a branch is made to the top of the procedure. Some temporary variables may be needed to effect this updating of the parameters. The reader may have noticed several subtle optimizations performed by the compiler in producing the code of Figure 56. First, the naive translation of tail-recursion would have produced some vacuous assignments to the effect of `(set! left left)` and `(set! right right)`, but these have been cleaned up following the transformation. The danger of such an assignment, is that it may create the appearance that `left` or `right` is conditionally computed (within an `if`-structure), whereas its value is actually unchanged. In the case of this procedure, no such spurious dependence would result, because `left` and `right` really *are* conditionally dependent upon `t-46`. In any event, it seems prudent to delete useless code early, before it is transformed into something that the compiler is unable to eliminate.

The second optimization that has been performed, is the "floating" of the invariant expression `(car l)` out of the inner `cond` expression. This expression is computed along both paths of the `cond` form, and is therefore not conditionally dependent upon `t-46`. Similarly, the expression `(set! x (cdr x))` was found on both branches of the inner `cond` after tail-recursion elimination, and was therefore floated out of the conditional block. The variable `t-46` is dependent upon `x` (via `t-48` and `partition`). If we did not float the expression `(set! x (cdr x))` out of the inner `cond` form, it would appear that `x` was conversely dependent upon `t-46`. This additional dependence would prevent the compiler from parallelizing the computation of both the values of `x` and the values of `t-46`, whereas we will see below that, having performed this transformation, both of these computations may be made parallel.

The conditional branch on `t-45` (the value of which was `(null? x)`) has been replaced by a conditional branch, with the logical sense reversed, on the variable `x`. When it is considered that `null?` is, effectively, boolean negation[22] this transformation is seen to be very simple. We emphasize that this transformation is triggered not by an occurrence of the variable `null?`, the value of which can be overwritten by the user; rather, interprocedural flow analysis has revealed that the intrinsic procedure `null?` is applied in computing `t-45`. The outcome would have been the same if the user had assigned the procedure `null?` into the variable `call/cc`, and had written `(call/cc x)` as the exit condition.

[22] As of [41], the empty list and boolean false (`#f`) may be treated as indistinguishable by an implementation of Scheme.

```
(define fact (lambda (n k)
   (if (= n 0)
       (k 1)
       (fact (1- n)
             (lambda (m) (k (* m n)))))))
```

Figure 57: A Continuation-Passing Version of Factorial

Finally, the compiler has recognized that `f` has as its value the identity procedure (`id`), and it therefore treats applications of `f` as simple (identity) assignments, and eliminates them where they are superfluous. Assignments of the form `(set! a b)` in the printed representation of the compiler's data structures are, internally, expressions of the form `(set! a (id b))`, where `id` represents not the identifier `id` but the procedure `id` (a constant). The pretty-printer produces the simpler form, when it sees an application of the identify begin.

Tail-recursion elimination, as we have described it, cannot be applied to every procedure that is determined to be tail-recursive by the compiler. For example, consider the familiar continuation-passing version of `fact`, shown in Figure 57. This procedure is arguably tail-recursive, but it is incorrect to rewrite it as a loop in which `n` is updated to have the value `(1- n)`, and in which `k` is repeatedly assigned to (exactly) the same closure: `(lambda (m) (k (* m n)))`. The result would be an infinite loop for `(fact x)` where `x` is greater than zero, because `k` would be made a recursive procedure with no exit condition. The problem is obviously that variable binding and assignment have meanings that are not, in general, interchangeable.

Under what conditions can tail-recursion elimination be performed? To re-use the location to which a variable instance is bound, it must be that the instance is no longer needed. Since we would re-use the locations associated with the parameters and local variables of a procedure upon every tail-recursive call to the procedure, we require that the lifetimes of these variables be restricted to the time between the point at which the procedure is invoked, and the point at which it invokes itself tail-recursively (or returns). In other words, considering the procedure as a loop, we require that the lifetimes of its bound variables be restricted to a single iteration of the loop. As in the case of contour merging, this condition can be neatly expressed in terms of the stack configurations computed by the interprocedural analysis described in Section 2. The condition may be stated this way: let λ_α be the tail-recursive procedure under consideration, let $\dot{\mathtt{x}}$ be an instance of a variable bound by λ_α, and let $\hat{c}$ be an (abstract) closure in which $\dot{\mathtt{x}}$ occurs free. Let $\hat{p}$ be the stack configuration that describes the interprocedural movements made by $\dot{\mathtt{x}}$ from its point of instantiation to a

```
($-$-sortby =
  (lambda (f l)
    <t-25 t-26 t-27 t-28 l-and-r t-30 t-31
     t-32 t-33 t-34 t-35 t-36 t-37 t-38>
    (cond
      ( l
        (set! t-37 (cdr l))
        (set! t-38 (car l))
        (set! l-and-r (splitby id t-37 t-38 '() '()))
        (set! t-34 (car l-and-r))
        (set! t-31 (#self-closure# id t-34))
        (set! t-35 t-38)
        (set! t-32 (list t-35))
        (set! t-36 (pcdr l-and-r))
        (set! t-33 (#self-closure# id t-36))
        (set! t-25 (append t-31 t-32 t-33)) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 58: A Common Subexpression is Eliminated in `$-$-sortby`

point at which it is referenced within $\hat{c}$. If $\hat{p}\alpha = \{\epsilon\}$, then this reference occurs while the instance of λ_α that binds $\dot{\mathbf{x}}$ is still active, for otherwise, by Theorem 21, $\hat{p}\alpha$ would contain one of $\mathbf{u}$, $\mathbf{uu}^+$ or $\mathbf{u}^+\mathbf{d}^+$. Furthermore, at the point of reference to $\dot{\mathbf{x}}$, no additional instances of λ_α are active (for otherwise $\hat{p}\alpha$ would contain one of $\mathbf{d}$ or $\mathbf{dd}^+$). If this condition holds for all variables bound by λ_α, and for every closure in which those variables occur free, then we may perform tail-recursion elimination in confidence that, at the point where they would be re-bound by a tail-recursive call, the parameters and local variables of the tail-recursive procedure are dead and may overwritten instead.

3.2.3 Common Subexpression Elimination

In Figure 55, the expression `(car l)` is computed into both `t-38` and `t-35`. The compiler remedies this by eliminating one of the computations, and replacing references to `t-35` by references to `t-38`. See Figure 58. This leaves the identity assignment `(set! t-35 t-38)` which is eliminated by forward substitution shortly.

The conditions under which common subexpression may be eliminated are a bit slippery, and do not lend themselves as easily to specification in terms of stack configurations. To be sure, the analysis of Section 2 permits

us to show easily that expressions have no side-effects, and this is a necessary for common subexpression elimination. Unfortunately it is not sufficient. Consider two occurrences of the expression `(cons a b)`. Although they are identical and have no side-effects (in the sense of Definition 1), the semantics of `eq?` dictate that they remain as distinct applications of `cons`, if it is possible that their results will be compared using `eq?`, or if they might be updated using `set-car!` or `set-cdr!`. As another example, consider the procedure

```
(define f (lambda (x) (lambda (y) (set! x (+ x y))))).
```

The procedure `f` (that is, the outer lambda expression) is side-effect free, but two identical invocations of it must not be treated as common subexpressions, because they create distinct instances of `x`. The test for the legality of common subexpression elimination must therefore include a criterion for the "exact equivalence" of two subexpressions, that accounts for such circumstances. In short, if the evaluation of a subexpression results in the creation of new data objects, and the objects created in one evaluation are discernible from those created in another evaluation, then to replace several (lexical) instances of the expression with one is incorrect, even if the expression is free of side-effects (by Definition 1).

The version of `$-$-sortby` in Figure 58 has a few peculiarities. First, the recursive calls to `sortby` have been replaced by the forms `(#self-closure# id t-34)` and `(#self-closure# id t-36)`. As mentioned above, several transformations performed by Parcel apply only to *self-recursive* lambda expressions: closures which make applications of themselves (the same lambda expression, the same environment). We see, in the printed representation of the compiler's data structures, the symbol `#self-closure#` when it has discovered a recursive procedure invocation that satisfies these conditions, and that occurs directly within the body of the procedure (not within another, lexically contained, lambda expression). Such an application may be considered when performing tail-recursion elimination, recursion splitting (to be introduced below) and other transformations that apply only to self-recursive procedures. The requirement that the recursive application occur directly within the body of a procedure, and not within a lexically contained procedure, increases the importance of contour merging.

The expression `(cdr l-and-r)` in Figure 55 has been rewritten as `(pcdr l-and-r)` in Figure 58. Parcel's run-time system makes use of some unusual list representations; we will return to this later. Among these representations is one using which it is less costly to take the `cdr` of a list that is known to be null-terminated (a *proper* list) than one which may be non-null-terminated (an *improper* list). In this case, the compiler discov-

ers that `l-and-r` is always a proper list, since the variable `right` within `$-$-sortby` is always a proper list, and it replaces the application of the more general intrinsic procedure `cdr` with one of the intrinsic procedure `pcdr`, that applies only to proper lists. This is, in effect, a reduction in strength [10].

3.2.4 More Contour Merging

Once it has rewritten the tail-recursive procedure `$-$-splitby` in iterative form, the compiler is able to expand it in-line, at its point of application within `$-$-sortby`, for the reason that there is now but one application of `$-$-splitby` in the program. See Figure 59.

We skip now to the version of the program that emerges from the preparatory restructuring phase we have been discussing; this version is presented in Figure 60. The benefits of iterative application of tail-recursion and contour merging, among the other preparatory transformations, now become apparent. First, the variable `l-and-r`, which previously held the cons cell that paired the two sublists returned by `$-$-splitby`, is gone entirely. The method of the compiler is apparent from Figure 59. The variable `t-44` is assigned the pair of `left` and `right`, and this pair becomes the value of `l-and-r`. Immediately afterwards, `t-34` is assigned the `car` and `t-36` the `cdr` of this pair. The compiler first replaces the right-hand sides of these assignments by `left` and `right`, respectively, and then discards the assignment to `l-and-r` as dead code.

Finally, the temporaries `t-52` and `t-55`, which were used in the iterative computation of `left` and `right`, have been eliminated by forward substitution. The result is a very clean organization of the quicksort algorithm, that is ideal both for compilation into sequential form, and for further restructuring into parallel form. Parcel does just this: the version of the program in Figure 60 is subjected to two distinct sets of transformations, one of which leads to an optimized sequential version of the program, the other which leads to an optimized parallel version. The run-time system makes use of these two versions, so that additional parallel activity can be created when the machine is underutilized, on the one hand, while allowing each processor to execute an optimized, sequential version of the code when adequate parallelism exists, on the other hand. We will follow the progress of the program, as it is restructured by Parcel into parallel form.

3.3 Exit-Loop Translation

Parcel has two central algorithms for automatic parallelization: *exit-loop translation* and *recursion splitting*. In fact, recursion splitting includes exit-loop translation as a subalgorithm, as we will see when we consider another

```
($-$-sortby =
  (lambda (f l)
    <t-25 l-and-r t-31 t-32 t-33 t-34 t-35 t-36 t-37 t-38
     f x partition left right t-44 t-46 t-48 t-52 t-55>
    (cond
      ( l
        (set! t-37 (cdr l))
        (set! t-38 (car l))
        (set! f id)
        (set! x t-37)
        (set! partition t-38)
        (set! left '())
        (set! right '())
        (exit-block
          (repeat
            (cond
              ( x
                (set! t-48 (car x))
                (set! t-46 (> partition t-48))
                (cond
                  ( t-46
                    (set! t-52 t-48)
                    (set! left (cons t-52 left)) )
                  ( else
                    (set! t-55 t-48)
                    (set! right (cons t-55 right)) ) )
                (set! x (cdr x)) )
              ( else
                (go l-74:) ) ))
          (l-74:
            (set! t-44 (cons left right))
            (set! l-and-r t-44)
            (set! t-34 (car l-and-r))
            (set! t-31 (#self-closure# id t-34))
            (set! t-35 t-38)
            (set! t-32 (list t-35))
            (set! t-36 (pcdr l-and-r))
            (set! t-33 (#self-closure# id t-36))
            (set! t-25 (append t-31 t-32 t-33)))) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 59: `$-$-splitby` is Merged into `$-$-sortby`

```
($-$-sortby =
  (lambda (f l)
    <t-25 t-31 t-32 t-33 t-38 x left right t-46 t-48>
    (cond
      ( l
        (set! t-38 (car l))
        (set! x (cdr l))
        (set! left '())
        (set! right '())
        (exit-block
          (repeat
            (cond
              ( x
                (set! t-48 (car x))
                (set! t-46 (> t-38 t-48))
                (if
                  t-46
                  (set! left (cons t-48 left))
                  (set! right (cons t-48 right)))
                (set! x (cdr x)) )
              ( else
                (go l-25:) ) ))
          (l-25:
            (set! t-31 (#self-closure# id left))
            (set! t-32 (list t-38))
            (set! t-33 (#self-closure# id right))
            (set! t-25 (append t-31 t-32 t-33)))) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 60: The Quicksort Program, after Preparatory Transformations

example program, below. We concentrate first upon exit-loop translation. These transformations are presented in [3] in terms of control flow graphs, and the details of the transformations as described there differ substantially from those that were ultimately implemented in Parcel, although the goals of the transformations remain the same. During the compiler's development, we discovered several ways in which these transformations could be simplified and generalized, and in which the transformed programs could be made more efficient. The changes in our approach will be apparent in the stepwise evolution of the program depicted below.

3.3.1 Replacing Exits with Recurrences

We will restrict our attention to the procedure `$-$-sortby`, as the procedure `$-$` is uninteresting and is unaffected by further transformations. `$-$-sortby` (as of Figure 60) contains a loop derived from `$-$-splitby`. This loop is like a Pascal `while` or `repeat` structure, in that the number of iterations to be performed is not known prior to execution of the loop, but is rather determined by a condition computed in every iteration. In this case, the exit condition is simply the variable `x`; when `x` becomes empty, the loop is exited by a branch to `L-25`. In general, such a loop might contain many branches which send control from the loop to various points in the code that follows the loop. If we replace every such exiting branch by an assignment to a boolean variable that indicates when the exit condition has been satisfied, then the loop may be easily rewritten as a `while` loop.[23]

See Figure 61. Again, there is some new syntax to explain. An expression of the form

```
(do (i n) EXPR)
```

denotes a loop in which EXPR is evaluated `n` times, and `i` assumes the values `0` to `n-1` in successive iterations. That the number of iterations (`n`) is replaced in Figure 61 by `??` indicates that the compiler is manipulating a `do` loop for which there is, as yet, no expression for the number iterations.

The variable `t-59` is initialized to false (`#f`) before this loop begins, and remains false until the original loop would have been exited. In place of the exit branch from the loop, the compiler has written the expression

```
(set! t-59 (#or t-59 i-60)).
```

For the moment, assume that the operator `#or` simply returns its second argument. We will explain its meaning more precisely below. The values of `t-59` after every iteration of the loop describe a sequence of the form

[23] In general, we will have to record which branch was taken in exiting the loop, and perform a multiway branch after the derived `while` loop is performed, to simulate the action of exiting the original loop.

```
($-$-sortby =
  (lambda (f l)
    <t-25 t-31 t-32 t-33 t-38 x left right t-46 t-48 t-59 i-60>
    (cond
      ( l
        (set! t-38 (car l))
        (set! x (cdr l))
        (set! left '())
        (set! right '())
        (set! t-59 #f)
        (do
          (i-60 ??)
          (cond
            ( x
              (set! t-48 (car x))
              (set! t-46 (> t-38 t-48))
              (if
                t-46
                (set! left (cons t-48 left))
                (set! right (cons t-48 right)))
              (set! x (cdr x)) )
            ( else
              (set! t-59 (#or t-59 i-60)) ) ))
        (set! t-31 (#self-closure# id left))
        (set! t-32 (list t-38))
        (set! t-33 (#self-closure# id right))
        (set! t-25 (append t-31 t-32 t-33)) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 61: Exit Branches are Eliminated from `$-$-sortby`

```
#f, #f, ... #f, N
```

where there is one `#f` for every iteration that would be performed by the loop of Figure 60. In short, exit-loop translation works by reorganizing the computation such that the sequence of values assigned to `t-59` is computed in parallel, and that the first non-`#f` value within this sequence is located, also in parallel. This value is the number of iterations of the loop; call this number N. It then reorganizes the rest of the loop (the portion of the loop that has nothing to do with `t-59`) into a conventional `do` structure, the number of iterations of which is N. This `do` loop is subjected to further parallelizing transformations, that are applicable only to loops whose number of iterations is computed prior to their execution.

Of course, the procedure of Figure 61 is not yet a legal translation of `$-$-sortby`, since the compiler has written a `do` loop for which the number of iterations is unknown. It must be remembered that these figures provide windows into the restructuring process, and in the case of exit-loop translation, the transformed loop must pass through some awkward intermediate states before emerging as a finished product. We will try to augment the printed representations of these intermediate states with insight into their significance.

3.3.2 Variable Expansion

The first thing to be done is to isolate the computation within the loop that is relevant to the variable `t-59`; to do this in turn requires several steps. The compiler first applies *variable expansion* [47, 3] to every variable that is computed within the loop. Variable expansion, or *scalar expansion* as it is usually called in the literature on vectorization of Fortran, is, roughly speaking, a technique whereby N assignments to a single location replaced by N assignments into a vector of length N. There are several reason for such a transformation. First, if N processors are, ultimately, to compute the N values assigned to a variable within a loop simultaneously, there must be N memory locations into which they may write their results; variable expansion provides these N locations, where there was merely one location previously. Second, after this transformation, the value of a variable at every point during the loop's execution will be recorded in a vector. We may therefore break the transformed loop into one which contains the definitions of the variable, and others which make use of the variable, since these uses will have been replaced by references into the vector. We will see an example of such a division of a loop, shortly.

See Figure 62. More explanations of syntax are due. The notation `x[i.j]` indicates the value of the expanded variable `x` after the $\texttt{j}^{\text{th}}$ assignment during the $\texttt{i}^{\text{th}}$ iteration of the loop. `x[i.0]` is therefore the value of `x` before

```
($-$-sortby =
  (lambda (f l)
    <t-25 t-31 t-32 t-33 t-38 x left right t-46 t-48 t-59 i-60>
    (cond
      ( l
        (set! t-38 (car l))
        (set! x (cdr l))
        (set! left '())
        (set! right '())
        (set! t-59 #f)
        (do (i-60 ??)
          (cond
            ( x[i-60.0]
              (set! t-48[i-60.1] (car x[i-60.0]))
              (set! t-46[i-60.1] (> t-38 t-48[i-60.1]))
              (cond
                ( t-46[i-60.1]
                  (set! left[i-60.1]
                        (cons t-48[i-60.1] left[i-60.0]))
                  (set! right[i-60.1] right[i-60.0]) )
                ( else
                  (set! right[i-60.1]
                        (cons t-48[i-60.1] right[i-60.0]))
                  (set! left[i-60.1] left[i-60.0]) ) )
              (set! x[i-60.1] (cdr x[i-60.0]))
              (set! t-59[i-60.1] t-59[i-60.0]) )
            ( else
              (set! t-59[i-60.1] (#or t-59[i-60.0] i-60))
              (set! x[i-60.1] x[i-60.0])
              (set! left[i-60.1] left[i-60.0])
              (set! right[i-60.1] right[i-60.0]) ) ))
        (set! t-31 (#self-closure# id left))
        (set! t-32 (list t-38))
        (set! t-33 (#self-closure# id right))
        (set! t-25 (append t-31 t-32 t-33)) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 62: Variables are Expanded in `$-$-sortby`

any assignments occur to it, during the $\texttt{i}^{\text{th}}$ iteration. In general, along some path through the body of a loop there may be several assignments (say, `k` of them) to a variable. Parcel arranges, by an algorithm described in [3], that the number of assignments to an expanded variable be equal along every path through the body of the loop. For example, identity assignments of the form

```
(set! right[i-60.1] right[i-60.0])
```

and

```
(set! left[i-60.1] left[i-60.0])
```

have been added to the loop in Figure 62 in order that there be one assignment to `left`, and one to `right`, on every path through the body of the loop. Therefore `x[i+1.0]` refers to the same position within the expanded variable `x` as `x[i.k]`, where `k` is the number of assignments to `x` along every path through the transformed loop. That is, the value of `x` after the last assignment to it in iteration `i`, is the same as its starting value in iteration `i+1`. There are several actions which must be taken by the code, in order to complete the process of variable expansion. First, the vectors into which the variables have been expanded must be allocated, and the first locations of these vectors must be assigned the initial value of the variables, prior to the execution of the loop. Second, after the loop has executed, the variable must be assigned the value of the last position of the vector (to give it the final value it would have had after the original loop). Some of these actions may not be necessary for a particular variable; for instance, a variable may be unused after the loop terminates. The code to perform these actions has not yet been added to the procedure, but will be below.

The reader might well ask how we plan to allocate these expanded variables (vectors) as contiguous blocks of storage, when we don't yet know how many iterations the loop has, and we therefore don't know how long the vector should be. Furthermore, it seems that there is a nasty circularity here: we need to know N to allocate these vectors in order that N may be computed. Clearly, we have some more rewriting to do, before we arrive at a sensible translation of the original loop.

3.3.3 Loop Distribution

Consider Figure 63. The `do` loop of Figure 62 has been broken into six loops, each of which computes the values of only one of the variables computed in the original loop. This technique is called *loop distribution* or *loop fission* [47, 3]. In this transformation, variable expansion acts to record all the values that are assumed by a variable during the loop, so that

```
($-$-sortby =
  (lambda (f l)
    <t-25 t-31 t-32 t-33 t-38 x left right t-46 t-48 t-59 i-60>
    (cond
      ( l
        (set! t-38 (car l))
        (set! x (cdr l))
        (set! left '())
        (set! right '())
        (set! t-59 #f)
        (do (i-60 ??)
            (if x[i-60.0]
                (set! x[i-60.1] (cdr x[i-60.0]))
                (set! x[i-60.1] x[i-60.0])))
        (do (i-60 ??)
            (if x[i-60.0] (set! t-48[i-60.1] (car x[i-60.0]))))
        (do (i-60 ??)
            (if x[i-60.0] (set! t-46[i-60.1] (> t-38 t-48[i-60.1]))))
        (do (i-60 ??)
            (if x[i-60.0]
                (if t-46[i-60.1]
                    (set! left[i-60.1] (cons t-48[i-60.1] left[i-60.0]))
                    (set! left[i-60.1] left[i-60.0]))
                (set! left[i-60.1] left[i-60.0])))
        (do (i-60 ??)
            (if x[i-60.0]
                (if t-46[i-60.1]
                  (set! right[i-60.1] right[i-60.0])
                  (set! right[i-60.1] (cons t-48[i-60.1] right[i-60.0])))
                (set! right[i-60.1] right[i-60.0])))
        (do (i-60 ??)
            (if x[i-60.0]
                (set! t-59[i-60.1] t-59[i-60.0])
                (set! t-59[i-60.1] (#or t-59[i-60.0] i-60))))
        (set! t-31 (#self-closure# id left))
        (set! t-32 (list t-38))
        (set! t-33 (#self-closure# id right))
        (set! t-25 (append t-31 t-32 t-33)) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 63: Loops are Distributed in `$-$-sortby`

the computation of those values may be isolated from other computation in the loop (that may make use of these values).

3.3.4 Reordering the Subloops

We mentioned above that the goal of exit-loop translation is to extract that portion of the loop that is pertinent to the computation of its exit condition, so that we might determine the number of iterations of the loop. In the case of our example, this may be restated as the portion of the loop that is pertinent to the computation of `t-59`. The compiler determines which of the loops created by distribution are relevant to the computation of `t-59`, by reordering them such that the one in which `t-59` is computed is preceded by as few others as possible, respecting the dependences among variables, of course. Those which remain above that in which `t-59` is computed, belong to the computation of the exit condition. See Figure 64. In this case, it would appear that only the loops in which `x` and `t-59` are computed are relevant to the exit condition of the loop.

3.3.5 Eliminating Unused Computation

In forming the loop of Figure 61, the compiler replaces each branch from the loop by an an assignment to `t-59`; after this assignment, control flows directly to the bottom of the loop. This adds a control path to the loop body, of course, and when variable expansion is applied, identity assignments are added along this path. However, the computation along these (former) exit paths is useless, except in the case of `t-59`, and must be eliminated. See Figure 65. In each case, this leaves us with a conditional node (a branch on `x[i-60.0]`) one of whose outgoing edges has been deleted. The compiler is quick to recognize this as dead code and eliminate it. We may view this transformation in the following way: the loop in which `x` is computed, for example, is meaningful only for `t-59` iterations, whatever `t-59` turns out to be; we are not interested in the loop's behavior after the first `t-59` iterations are performed. However, by the very definition of `t-59` (the iteration number in which the exit condition is first satisfied, where iterations are counted from 0), it is impossible for the variable `x` to become false (that is, for the exit condition to be satisfied) during the first `t-59` iterations. Therefore, for the meaningful iterations of the loop in which `x` is computed, the branch on `x` and the identity assignment `(set! x[i-60.1] x[i-60.0])` are inert, they contribute nothing and may be eliminated. Indeed, they must be eliminated if the compiler is to recognize the recurrence described by `x`, for otherwise the computation of `x` appears to have a complex dependence structure.

The compiler has added an index variable and number of iterations to each of the loops that follows that in which `t-59` is computed. The number

```
($-$-sortby =
  (lambda (f l)
    <t-25 t-31 t-32 t-33 t-38 x left right t-46 t-48 t-59 i-60>
    (cond
      ( l
        (set! t-38 (car l))
        (set! x (cdr l))
        (set! left '())
        (set! right '())
        (set! t-59 #f)
        (do (i-60 ??)
            (if x[i-60.0]
              (set! x[i-60.1] (cdr x[i-60.0]))
              (set! x[i-60.1] x[i-60.0])))
        (do (i-60 ??)
            (if x[i-60.0]
                (set! t-59[i-60.1] t-59[i-60.0])
                (set! t-59[i-60.1] (#or t-59[i-60.0] i-60))))
        (do (i-60 ??)
            (if x[i-60.0] (set! t-48[i-60.1] (car x[i-60.0]))))
        (do (i-60 ??)
            (if x[i-60.0] (set! t-46[i-60.1] (> t-38 t-48[i-60.1]))))
        (do (i-60 ??)
            (if x[i-60.0]
                (if t-46[i-60.1]
                    (set! left[i-60.1] (cons t-48[i-60.1] left[i-60.0]))
                    (set! left[i-60.1] left[i-60.0]))
                (set! left[i-60.1] left[i-60.0])))
        (do (i-60 ??)
          (if x[i-60.0]
              (if t-46[i-60.1]
                  (set! right[i-60.1] right[i-60.0])
                  (set! right[i-60.1] (cons t-48[i-60.1] right[i-60.0])))
              (set! right[i-60.1] right[i-60.0])))
        (set! t-31 (#self-closure# id left))
        (set! t-32 (list t-38))
        (set! t-33 (#self-closure# id right))
        (set! t-25 (append t-31 t-32 t-33)) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 64: Distributed Loops are Reordered in `$-$-sortby`

```
($-$-sortby =
  (lambda (f l)
    < t-25 t-31 t-32 t-33 t-38 x left right
      t-46 t-48 t-59 i-60 i-61 i-62 i-63 i-64 >
    (cond
      ( l
        (set! t-38 (car l))
        (set! x (cdr l))
        (set! left '())
        (set! right '())
        (set! t-59 #f)
        (do (i-60 ??) (set! x[i-60.1] (cdr x[i-60.0])))
        (do (i-60 ??)
          (if x[i-60.0]
              (set! t-59[i-60.1] t-59[i-60.0])
              (set! t-59[i-60.1] (#or t-59[i-60.0] i-60))))
        (do (i-61 t-59) (set! t-48[i-61.1] (car x[i-61.0])))
        (do (i-62 t-59) (set! t-46[i-62.1] (> t-38 t-48[i-62.1])))
        (do (i-63 t-59)
            (if t-46[i-63.1]
                (set! left[i-63.1]
                      (cons t-48[i-63.1] left[i-63.0]))
                (set! left[i-63.1] left[i-63.0])))
        (do (i-64 t-59)
          (if t-46[i-64.1]
              (set! right[i-64.1] right[i-64.0])
              (set! right[i-64.1]
                    (cons t-48[i-64.1] right[i-64.0]))))
        (set! t-31 (#self-closure# id left))
        (set! t-32 (list t-38))
        (set! t-33 (#self-closure# id right))
        (set! t-25 (append t-31 t-32 t-33)) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 65: Exit Path Computations are Eliminated in `$-$-sortby`

of iterations is given as `t-59` itself. A distinct index variable has been given to each of these `do` loops, in order that there be no artificial dependences between the loops, that would prevent several of them from being executed simultaneously.

3.3.6 The Parallel Computation of the Number of Iterations

Granted, that Figure 65 does not appear to be a legal translation of the original procedure `$-$-sortby`; we have interrupted the compiler while in the midst of performing a lengthy transformation. Still, the procedure at this point has a clear intuitive meaning, that gives insight into the generality of exit-loop translation. Consider for a moment only the first two `do` loops of Figure 65, and suppose that we ignore that they are written as `do` loops, but think of them as signifying the following computation. Let `x[0.0]` have the initial value of `x` (that is, the value of `(cdr l)`, as per the figure), and let `t-59[0.0]` have the initial value of `t-59` (`#f`, as per the figure). Given `x[0.0]`, we may perform one iteration of the loop that computes `t-59`; this will give a value for `t-59[0.1]` (and recall that `t-59[0.1]` and `t-59[1.0]` refer to the same vector element). Then `t-59[0.1]` is either `#f` (if `x[0.0]` is non-null) or `0` (if `x[0.0]` is null). Assuming `x[0.0]` is non-null, we may then execute one iteration of the loop in which `x` is computed. This gives us a value for `x[1.0]`, and we may again perform an iteration of the loop in which `t-59` is computed. Once again, the outcome will be either that `t-59[1.1]` is `#f` (if `x[1.0]` is non-null) or `1` (if `x[1.0]` is null). We may repeat this indefinitely, until a numeric value is obtained for some `t-59[N.1]`; this value will be `N` itself, the iteration number in which the exit condition is first satisfied.

As we have described it, this process is very sequential. A simple observation, however, leads straightforwardly to its parallelization. The sequence `x[0.0]`, `x[1.0]`, ..., is a simple recurrence relation whose terms may be computed in parallel with good speedup, depending upon the representation that s-expressions are given in memory. The i^{th} term of this sequence is given by

```
x[i.0] = (list-tail x[0.0] i),
```

and the computation of these terms can be made quite parallel; in effect, we can compute an application of `list-tail` in constant or near-constant time, given the proper representation of `x` in memory. We will return to this. Likewise, given the sequence `x[0.0]`, `x[1.0]`, ..., the sequence `t-59[0.0]`, `t-59[1.0]`, ..., is also a simple recurrence relation, a variation on the *first-one* problem of finding the first one in a boolean string, the terms of which can be computed in parallel with good speedup. We may modify our interpretation of the first two loops in Figure 65, then, by

viewing them instead as describing the following computation: first, k terms of the sequence `x[0.0]`, `x[1.0]`, ... are computed in parallel. Using these k terms, we compute k terms of the sequence `t-59[0.0]`, `t-59[1.0]`, ... We then examine the `t-59` terms to see if there is a non-null term among them, and to find the leftmost such term, if so. If there is such a term, the number of iterations of the original loop has been discovered, and we are done. Otherwise, we repeat the process by computing k more `x` terms, and k more `t-59` terms, and so on until the number of iterations has been found.

At last, we may explain the meaning of `#or`: it is a a binary operator, defined by the following four equations:

```
(#or #f #f) = #f
(#or #f j)  = j
(#or i #f)  = i
(#or i j)   = i.
```

Assume that `i` and `j` are integers. Then `#or` takes two arguments which are either boolean or integer values, and returns either a boolean or integer value. It is easily verified that this operator is associative. If we reduce a vector of boolean and integer values using this operator, it will return the leftmost integer found within the vector, or `#f` if none is found. In short, this operator is used by parallel prefix [32], to reduce a vector (`t-59` in our example) of boolean and integer values, in order that the least iteration for which the exit condition is satisfied may be found in parallel. The somewhat laborious details of the computation of the number of iterations using this technique are given in [3]. In the example before us, the compiler will be able to produce a closed-form expression for the number of iterations; but were it necessary to compute this number through the use of `#or`, the recurrence would be restructured and rewritten in parallel form by the compiler in a later phase. We will see several examples of such restructuring of recurrences, for parallel solution, momentarily.

This, then, is the mechanism underlying exit-loop translation. It depends, in this case, upon `x` describing a recurrence relation with a parallel solution. In general, Parcel requires of a variable that contributes to the exit condition of the loop, either that it describe such a recurrence relation, or a computation with even simpler dependences. For example, each of the terms `x[0.1]`, `x[1.1]`, etc., might be the return value of another procedure, which is determined by interprocedural analysis to be side-effect free. This requirement is imposed, not because it is impossible to give a legal translation to the loop if, for example, the terms of `x` had to be computed sequentially; but rather as a heuristic intended to prevent the generation of an inefficient restructured version of the loop that contains too little

parallelism to recover the expense of expanded variables, distributed loop control, etc., in the computation of the number of iterations. The philosophy embodied in Parcel is that if a loop or procedure is resistant to automatic parallelization, it is better to hope for more natural parallelism in the procedures that call, and are called by, the resistant computation, than to force the issue.

There are several important observations to be made at this point. First, as part of the process of computing `t-59`, the intermediate values of `x` are computed and saved; they need not be recomputed when they are used in the loops which follow the computation of the number of iterations, and thus the parallelism introduced by this technique entails very little redundant computation. Second, in some cases, and the example before us is such a case, the recurrence relation that defines the number of iterations of the loop has a closed-form solution. In this case, the solution is simply `t-59 = (length x)`. As we will see momentarily, this fact is not lost to Parcel.

3.3.7 Marking Doalls and Recurrences

See Figure 66. In the next several figures, the compiler will make each of the observations that we have mentioned in the above paragraphs. First, it marks the loop in which `x` is computed as an induction sequence (a simple recurrence in which the $\texttt{i}^{\text{th}}$ term has a closed-form solution in terms of `i` and the value `x[0.0]`), and the loop in which `t-59` is computed as a recurrence relation (as mentioned above, a simple variation on the boolean first-one recurrence). The names `do-induction` and `do-recurrence` indicate this discovery. Since each of the loops that contributes to the computation of the number of iterations can be made parallel, exit-loop translation has succeeded. It remains only to rewrite the computation of `t-59` in as efficient a form as possible, and to continue with the parallelization and optimization of the rest of the procedure.

3.3.8 Closed-Form Solution for the Number of Iterations

See Figure 67. Simple induction sequences over s-expressions and integers, such as that described here by `x`, are recognized by Parcel, because they occur so frequently, and can be solved in closed form much more efficiently than by the k-terms-at-a-time approach described above. The compiler has simply written `(set! t-59 (length x))`, and the loop in which `t-59` was computed has disappeared. The loop in which `x` is computed is needed by other loops in this figure, and cannot be deleted; it is rewritten to have `t-59` as its number of iterations. There is some dead code in the figure, that will be eliminated shortly.

```
($-$-sortby =
  (lambda (f l)
    < t-25 t-31 t-32 t-33 t-38 x left right t-46
      t-48 t-59 i-60 i-61 i-62 i-63 i-64 t-65 t-66 >
    (cond
      ( l
        (set! t-38 (car l))
        (set! x (cdr l))
        (set! left '())
        (set! right '())
        (set! t-59 #f)
        (do-induction (i-60 ??) (set! x[i-60.1] (cdr x[i-60.0])))
        (do-recurrence
          (i-60 ??)
          (if x[i-60.0]
              (set! t-59[i-60.1] t-59[i-60.0])
              (set! t-59[i-60.1] (#or t-59[i-60.0] i-60))))
        (doall (i-61 t-59) (set! t-48[i-61.1] (car x[i-61.0])))
        (doall (i-62 t-59) (set! t-46[i-62.1] (> t-38 t-48[i-62.1])))
        (do-rem-recurrence
          (i-63 t-59)
          (if t-46[i-63.1]
              (set! t-65[i-63.1] (cons t-48[i-63.1] t-65[i-63.0]))
              (set! t-65[i-63.1] t-65[i-63.0])))
        (do-rem-recurrence
          (i-64 t-59)
          (if t-46[i-64.1]
              (set! t-66[i-64.1] t-66[i-64.0])
              (set! t-66[i-64.1] (cons t-48[i-64.1] t-66[i-64.0]))))
        (set! right (append2 right t-66))
        (set! left (append2 left t-65))
        (set! t-31 (#self-closure# id left))
        (set! t-32 (list t-38))
        (set! t-33 (#self-closure# id right))
        (set! t-25 (append t-31 t-32 t-33)) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 66: Recurrences and Parallel Loops are Identified in `$-$-sortby`

```
($-$-sortby =
  (lambda (f l)
    < t-25 t-31 t-32 t-33 t-38 x left right t-46
      t-48 t-59 i-60 i-61 i-62 i-63 i-64 t-65 t-66 i-67 >
    (cond
      ( l
        (set! t-38 (car l))
        (set! x (cdr l))
        (set! left '())
        (set! right '())
        (set! t-59 #f)
        (set! t-59 (length x))
        (do-induction (i-67 t-59) (set! x[i-67.1] (cdr x[i-67.0])))
        (doall (i-61 t-59) (set! t-48[i-61.1] (car x[i-61.0])))
        (doall (i-62 t-59) (set! t-46[i-62.1]
                                 (> t-38 t-48[i-62.1])))
        (do-rem-recurrence (i-63 t-59)
          (if t-46[i-63.1]
              (set! t-65[i-63.1] (cons t-48[i-63.1] t-65[i-63.0]))
              (set! t-65[i-63.1] t-65[i-63.0])))
        (do-rem-recurrence (i-64 t-59)
          (if t-46[i-64.1]
              (set! t-66[i-64.1] t-66[i-64.0])
              (set! t-66[i-64.1] (cons t-48[i-64.1] t-66[i-64.0]))))
        (set! right (append2 right t-66))
        (set! left (append2 left t-65))
        (set! t-31 (#self-closure# id left))
        (set! t-32 (list t-38))
        (set! t-33 (#self-closure# id right))
        (set! t-25 (append t-31 t-32 t-33)) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 67: A Closed-Form Solution for `t-59` is Found

3.3.9 Restructuring the Recurrences

Let us return to Figure 66 to consider the computation of the variables `t-48`, `t-46`, `left`, and `right`. The loop in which `t-48` is computed requires little explanation; its iterations are independent of one another, and so the compiler has marked it as a `doall` loop, so that its iterations may be executed simultaneously. The same is true of the loop in which `t-46` is computed.

The loops in which `left` and `right` are computed have been rewritten, using `t-65` and and `t-66` instead of `left` and `right`, respectively. They are called `do-rem-recurrence` loops for the purposes of displaying the compiler's view of them, because they describe recurrence relations with parallel solutions, whose *remote terms* are the only terms that are needed. The remote term of the sequence `left[0.0]`, `left[1.0]`, ..., `left[N.0]` is the final term, `left[N.0]`. The procedure has been rewritten in this way, so that in the event that the original loop had been surrounded by other loops, the recurrences described by `left` and `right` would be parallelized at several of these nest levels, and not merely at the innermost level. Consider, for example, the assignment `(set! left (append2 left t-65))`. (`append2` is simply a special case of `append` that takes exactly two arguments.) If this assignment appeared inside yet another `do` loop that surrounded all of Figure 66, then the compiler would distribute loop control around the assignment, isolating it from the rest of the outer `do` loop, and the recurrence relation it defines would be given a parallel translation as well.

Now consider Figure 68. The loop in which `t-65` is computed has been rewritten, so that it now places either `t-48[i.1]` or the constant `#?` into the i^{th} position of `t-65`, depending upon the value of `t-64[i.1]`; the loop in which `t-66` is computed is rewritten similarly. Intuitively, if a cell was added to `left` in the i^{th} iteration of the original loop, the the `car` of this cell is placed in `t-65[i.1]`, and the marker `#?` is placed in `t-65[i.1]` otherwise. The procedure `cons-rem-rec` takes two arguments: an input vector and an output vector, which may be the same. It expects its input vector to be filled with legitimate values and `#?` markers, as are `t-65` and `t-66`, and it produces a list of only the non-`#?` values, in the reverse order of their occurrence in the input vector. This list is pointed to by the last position of the output vector, which represents the *remote term* of the recurrence being solved. The procedure works in parallel, and is part of the Parcel run-time library. Its workings are described in [3].

The loop in which `x` was computed in Figure 67 has been replaced by a call to the routine `cdr-ind`. This procedure takes a vector `v`, the first position of which is assumed to point to a list, and and integer, call it `k`.

```
($-$-sortby =
  (lambda (f l)
    < t-25 t-31 t-32 t-33 t-38 x left right t-46 t-48
      t-59 i-60 i-61 i-62 i-63 i-64 t-65 t-66 i-67 >
    (cond
      ( l
        (set! t-38 (car l))
        (set! x (cdr l))
        (set! left '())
        (set! right '())
        (set! t-59 (length x))
        (set! x (allocate x t-59))
        (set! t-46 (allocate #f t-59))
        (set! t-48 (allocate #f t-59))
        (set! t-65 (allocate #f t-59))
        (set! t-66 (allocate #f t-59))
        (cdr-ind x 1)
        (doall (i-61 t-59)
          (set! t-48[i-61.1] (car x[i-61.0]))
          (set! t-46[i-61.1] (> t-38 t-48[i-61.1])))
        (doall (i-63 t-59)
          (if t-46[i-63.1]
              (set! t-65[i-63.1] t-48[i-63.1])
              (set! t-65[i-63.1] #?)))
        (cons-rem-rec t-65 t-65)
        (doall (i-64 t-59)
          (if t-46[i-64.1]
              (set! t-66[i-64.1] #?)
              (set! t-66[i-64.1] t-48[i-64.1])))
        (cons-rem-rec t-66 t-66)
        (set! t-66 (restore t-66 t-59))
        (set! t-65 (restore t-65 t-59))
        (set! right (append2 right t-66))
        (set! left (append2 left t-65))
        (set! t-31 (#self-closure# id left))
        (set! t-32 (list t-38))
        (set! t-33 (#self-closure# id right))
        (set! t-25 (append t-31 t-32 t-33)) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 68: Recurrences are Restructured for Parallel Execution

Upon return, the i^{th} position of the vector points to (cd^{ik}r v[0.0]), the ik^{th} cell of v[0.0]. The length of the vector v, and not the length of the list v[0.0], determines the number of terms of the induction sequence to be computed.

3.3.10 Allocating and Initializing Expanded Variables

The procedures `allocate` and `restore` perform the initial and final actions, respectively, upon an expanded variable. `allocate` takes two arguments: an initial value (the value to be given to `x[0.0]`, where `x` is the expanded variable; in other words, the value of `x` prior to the loop in which `x` is expanded), and the number `N` of iterations of the loop in which `x` is expanded. Recall that Parcel adds identity assignments as needed, to insure that the number of assignments to a variable `x` is invariant over all paths through the loop, when it expands `x` in a loop. Let this number of assignments be `W`. `allocate` creates a vector of length `NW+1` locations on the run-time stack to hold the values of `x` at all points during the loop's execution. There are `NW+1` locations because there is one initial value (`x[0.0]`), and `W` values per each of `N` iterations. `allocate` then assigns to `x[0.0]` the value of its first argument. `restore` simply returns the value of the final position of `x` (`x[N.0]` or, equivalently, `x[N-1.W]`); its arguments are `x` and `N`. The value of `W` is a compile-time constant; `allocate` and `restore` are compiled in-line at code generation, and the value of `W` is built directly into the code that is emitted.

3.3.11 Loop Fusion

The version of `$-$-sortby` in Figure 68 is, at last, a complete and legal translation of the original `$-$-sortby` in Figure 54.

The final parallel version of `$-$-sortby` is given in Figure 69. In order to reduce the overhead of starting and stopping parallel loops, the compiler has fused the loop in which `t-48` is computed with that in which `t-46` is computed. The loops in which `t-65` and `t-65` are computed should be similarly fused, but Parcel uses a naive "undistribution" algorithm, and applies it only to loops which are unaltered from the form they had immediately prior to distribution, whereas the loops in which `t-65` and `t-66` are computed have been derived from the loops in which the variables `left` and `right` were originally computed.

3.3.12 Cobegin Insertion

A simple but significant step of parallelization has been performed by the compiler in arriving at Figure 69. Consider the final `doall` loop of the figure; it has only two iterations. The syntax

```
($-$-sortby =
  (lambda (f l)
    < t-25 t-31 t-32 t-33 t-38 x left right t-46 t-48 t-59
      i-60 i-61 i-62 i-63 i-64 t-65 t-66 i-67 i-72 >
    (cond
      ( l
        (set! right '())
        (set! left '())
        (set! x (cdr l))
        (set! t-59 (length x))
        (set! t-66 (allocate #f t-59))
        (set! t-65 (allocate #f t-59))
        (set! t-48 (allocate #f t-59))
        (set! t-46 (allocate #f t-59))
        (set! x (allocate x t-59))
        (cdr-ind x 1)
        (set! t-38 (car l))
        (doall (i-61 t-59)
          (set! t-48[i-61.1] (car x[i-61.0]))
          (set! t-46[i-61.1] (> t-38 t-48[i-61.1])))
        (set! t-32 (list t-38))
        (doall (i-64 t-59)
          (if t-46[i-64.1]
              (set! t-66[i-64.1] #?)
              (set! t-66[i-64.1] t-48[i-64.1])))
        (doall (i-63 t-59)
          (if t-46[i-63.1]
              (set! t-65[i-63.1] t-48[i-63.1])
              (set! t-65[i-63.1] #?)))
        (cons-rem-rec t-66 t-66)
        (set! right (restore t-66 t-59))
        (cons-rem-rec t-65 t-65)
        (set! left (restore t-65 t-59))
        (doall (i-72 2)
          (mway i-72
            (0 (set! t-31 (#self-closure# id left)))
            (1 (set! t-33 (#self-closure# id right)))))
        (set! t-25 (append t-31 t-32 t-33)) )
      ( else
        (set! t-25 '()) ) )
    (return t-25) ))
```

Figure 69: The Final, Parallel Version of `$-$-sortby`

```
(define
  tak
  (lambda (x y z)
    (cond
      ( (not (< y x))
        z )
      ( #t
        (tak (tak (1- x) y z)
             (tak (1- y) z x)
             (tak (1- z) x y)) ) ) ))
(write (tak 18 12 6))
```

Figure 70: The Procedure `tak`

```
(mway m
   (0 ExprA ... )
   (1 ExprB ... )
      ⋮
   (m-1 ExprM ... ))
```

indicates a multi-way branch on the value of `m`; it is similar to the `switch` form of C. It is used to select one of `m` sequences of expressions for evaluation. This `doall` loop might more clearly be written as

```
(cobegin
   (set! t-31 (#self-closure# id left))
   (set! t-33 (#self-closure# id right))).
```

It permits the recursive invocations of `$-$-sortby` to be executed concurrently. Since each of these recursive invocations will itself contain parallelism (both the parallelism that was extracted from the loop that partitions each sublist to be sorted, and the parallel recursive calls to `$-$-sortby`) a significant degree of parallelism can be obtained from this procedure at runtime. The compiler inserts such `cobegin` constructs by grouping together invocations of user procedures and/or loops that can be evaluated simultaneously. That is, expressions over intrinsic procedures are not candidates for inclusion in such a form. This is simply a heuristic intended to prevent parallel activity which does not pay back the expense of its creation.

3.4 Recursion Splitting

We next consider a very simple recursive procedure, which is nonetheless not merely tail-recursive. To parallelize this procedure, Parcel will apply a

```
($-$
  =
  (lambda ()
    <tak t-29 t-30>
    (set! tak #<$-$-tak>)
    (set! t-30 (tak 18 12 6))
    (set! t-29 (write t-30))
    (return t-29) ))
($-$-tak
  =
  (lambda (x y z)
    <t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28>
    (set! t-22 (< y x))
    (cond
      ( t-22
        (set! t-26 (1- x))
        (set! t-23 (#self-closure# t-26 y z))
        (set! t-27 (1- y))
        (set! t-24 (#self-closure# t-27 z x))
        (set! t-28 (1- z))
        (set! t-25 (#self-closure# t-28 x y))
        (set! t-20 (#self-closure# t-23 t-24 t-25)) )
      ( else
        (set! t-20 z) ) )
    (return t-20) ))
```

Figure 71: The Initial Representation of `tak`

technique introduced in [3] called *recursion splitting*, a general technique for rewriting a recursive computation as a pair of loops, so that the latter may be subjected to further transformations, as were applied to the quicksort example above.

The program we will be considering, following macro expansion, is given in Figure 70. `tak` is a simple function over integers that contains four recursive calls. In Figure 71 is given the program as seen by Parcel, following parsing, interprocedural analysis, and the preparatory restructuring described in subsection 3.2. This simple function proves impervious to the preparatory transformations. As is the case with every program that it treats, Parcel introduces a lambda expression called `$-$` which represents the top level of the user's program, and by which his global variables are bound, conceptually. Henceforth, we will confine our attention to `$-$-tak`.

The compiler has marked each of the recursive calls within `$-$-tak` as

self-recursive; this permits the application of tail-recursion elimination and recursion splitting to the procedure, provided that other conditions necessary to their application are satisfied. Indeed, there is a tail-recursive call to `$-$-tak`; the compiler could use this fact to obtain a loop from the procedure body, and subject this loop to exit-loop translation as was done in the case of `$-$-splitby` above. This would prove to be an error, however, because exit-loop translation would fail when applied to the loop, for the reason that `x` would describe a recurrence of the form

```
x[i.1] = (tak (1- x[i.0]) y z),
```

and the compiler would fail to parallelize such a recurrence, and would therefore fail to parallelize the computation of the number of iterations of the loop it had created. Moreover, if it first performed tail-recursion elimination, it would be unable to perform recursion splitting afterwards, as the remaining recursive calls would be within the loop introduced by tail-recursion elimination, and recursion splitting does not treat such recursive calls. Parcel therefore refrains from performing tail-recursion elimination upon a procedure if to do so would leave recursive calls to the procedure within a loop.

3.4.1 Overview

The idea behind recursion splitting is simple. The compiler first selects a set of recursive calls to the procedure, such that there is at most one member of the set along any path through the procedure; this set is called a *fence*. In the case of `$-$-tak`, there are four recursive calls to the procedure, but all occur along a single control path through the procedure body; any fence will therefore have just one member. The fence is then used to divide the procedure into two loops, called the *forward* and *backward* loops. The forward loop contains all of the computation that occurs between the entrance to the procedure and the members of the fence, and the backward loop contains all of the computation between the members of the fence and the return from the procedure. These two loops will have the same number of iterations, and this number will be determined by exit-loop translation of the forward loop. In the evaluation of an application of the original procedure, the parameters and local variables of the procedure are recorded on the stack at each recursive call, and when this recursive call returns, these variables are restored from the stack. To simulate this pushing and popping of parameters and locals, expanded variables will be used. Each parameter and local variable will be represented by a vector of length (roughly) `N`, where `N` is the number of iterations of the forward loop. Whenever a variable would have been "pushed" by a member of the fence (a selected recursive call), it is instead written into a vector by the

```
($-$-tak
  =
  (lambda (x y z)
    <t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28 t-44 i-45>
    (set! t-44 #f)
    (do
      (i-45 ??)
      (set! t-22 (< y x))
      (cond
        ( t-22
          (set! t-26 (1- x))
          (set! x t-26) )
        ( else
          (set! t-44 (#or t-44 i-45)) ) ))
    (set! t-20 z)
    (do
      (i-45 t-44)
      (set! t-23 t-20)
      (set! t-27 (1- y))
      (set! t-24 (#self-closure# t-27 z x))
      (set! t-28 (1- z))
      (set! t-25 (#self-closure# t-28 x y))
      (set! t-20 (#self-closure# t-23 t-24 t-25)))
    (return t-20) ))
```

Figure 72: Forward and Backward Loops are Formed in `$-$-tak`

forward loop, and when it would have been "popped" at a return from a member of the fence, it is instead read from the vector by the backward loop. Intuitively, the iteration spaces of the forward and backward loops run in opposite directions; variables "pushed" in the first iteration of the forward loop are "popped" in the last iteration of the backward loop, and those "pushed" in the second iteration of the forward loop are "popped" in the next-to-last iteration of the backward loop, and so on.

3.4.2 Forming the Forward and Backward Loops

It is easiest to appreciate the transformation by example. Consider Figure 72. The computer has chosen the set containing the first recursive call in Figure 71 as the fence. It therefore divides the procedure into two loops at this recursive call. The forward loop contains only the updating of the parameter `x`, and the computation of `t-44`, which should be familiar to the reader from the discussion of exit-loop translation above, as the number of

iterations of the forward loop. As usual, we have interrupted the compiler at an awkward moment for a view of its data structures. At this point, no variable expansion has occurred, and consequently this is far from a legal translation of the program. Not to worry.

The backward loop (the second `do` loop of Figure 72) contains the bulk of the computation from the original procedure. It is has no assignments to `t-44`; that is, it contained no exit branches that were rewritten as assignments to `t-44`, as did the forward loop. The number of iterations of the backward loop, like the forward loop, will ultimately be `t-44`. The return value of `$-$-tak` (`t-20`) is computed iteratively in the backward loop, just as the parameters were computed iteratively in the forward loop.

Recursion splitting proceeds from here in two major steps. First, the compiler applies exit-loop translation to the forward loop. Indeed, it has already begun, by replacing loop exits by assignments to `t-44`. If exit-loop translation succeeds, then recursion splitting succeeds, and both the forward and backward loops are subjected for further parallelization, just as the loop that arose from `splitby` was parallelized, in the quicksort example of subsection 3.1. Otherwise, recursion splitting fails, and another fence will be tried. When the possible fences have been exhausted, then recursion splitting fails finally, and other sources of parallelism within the procedure will be sought.

3.4.3 Exit-Loop Translation of the Forward Loop

We focus our attention, then, upon exit-loop translation of the forward loop. See Figure 73. The variables defined in the forward loop are expanded. We mentioned two purposes for the expansion of variables in discussing the quicksort example above. First, we said, it permits (or facilitates) the computation in parallel of the values that are assigned to a variable in successive iterations of a loop, by providing a distinct memory location into which each such value may be written. Second, it permits the production of a variable's successive values to be isolated from the consumption of those values, by recording them in a vector, to which the consuming computation may refer. Put another way, we may distribute loop control around the computation of each variable in a loop (see Figure 63) only because the successive values assigned to the variable (in one subloop derived by distribution of the original loop) will be held in a vector, for consumption during the computation of another variable (in a second derived subloop). In this latter capacity, expanded variables act as communication media between subcomputations. In recursion splitting, this function is extended: not only do expanded variables communicate values between the subloops that are derived by distribution of the forward loop, they act also to communicate these values into the backward loop, replacing

```
($-$-tak
  =
  (lambda (x y z)
    <t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28 t-44 i-45>
    (set! t-44 #f)
    (do
      (i-45 ??)
      (set! t-22[i-45.1] (< y x[i-45.0]))
      (cond
        ( t-22[i-45.1]
          (set! t-26[i-45.1] (1- x[i-45.0]))
          (set! x[i-45.1] t-26[i-45.1])
          (set! t-44[i-45.1] t-44[i-45.0]) )
        ( else
          (set! t-44[i-45.1] (#or t-44[i-45.0] i-45))
          (set! x[i-45.1] x[i-45.0]) ) ))
    (set! t-20 z)
    (do
      (i-45 t-44)
      (set! t-23 t-20)
      (set! t-27 (1- y))
      (set! t-24 (#self-closure# t-27 z x[[i-45.0]]))
      (set! t-28 (1- z))
      (set! t-25 (#self-closure# t-28 x[[i-45.0]] y))
      (set! t-20 (#self-closure# t-23 t-24 t-25)))
    (return t-20) ))
```

Figure 73: Variables Defined in the Forward Loop are Expanded

the function of the stack in the original procedure. In the case of Figure 73, the variable `x` is defined (and therefore expanded) in the forward loop, and the references to `x` in the backward loop have been replaced by references to the vector which will record the values assigned to `x` in the loop. There is an important point concerning the references to `x` in the backward loop: they are written as

```
x[[i-45.0]].
```

This subscript form is used when references are made in the backward loop (or in a loop derived from the backward loop) to a variable defined in the forward loop, and is equivalent to

```
x[(1- (- N i-45)).0]
```

where `N` is the number of iterations of the forward and backward loops, the value to be assigned to `t-44` in this case. Intuitively, an expanded variable that is defined in the forward loop is read "backward" in the backward loop, for precisely the reason that it is replacing the function of the stack. Alternatively, one may view the iteration spaces of the forward and backward loop as having opposite orientations, as mentioned above: the index variable of the forward loop counts recursive calls via members of the fence, while the index variable of the backward loop counts returns from these calls, and the returns occur in the reverse order of the corresponding calls, by the nature of recursion.

Before proceeding with the transformation, the compiler pauses to perform some optimizations, much like the preparatory optimizations discussed in subsection 3.2. The reason is that, as when performing tail-recursion elimination, some temporary variables are needed (in general) to update the parameters of the procedure, when forming the forward and backward loops. The compiler first writes the most "general" form of the forward and backward loops, and then attempts to improve them, by eliminating the manipulation of these temporary quantities where possible, by floating invariant computations out of conditional structures, etc. See Figure 74. The variable `t-26`, used previously in the updating of `x`, is eliminated. `t-23` is similarly eliminated from the backward loop.

We proceed with exit-loop translation of the forward loop exactly as though it was the entirety of the computation. The next step, it will be recalled, is to distribute the forward loop, with the aim of isolating the portion of the forward loop that is relevant to the computation of its number of iterations, or equivalently in this case, to the computation of `t-44`. See Figure 75. The compiler has distributed the forward loop into three loops, and has reordered these such that as few as possible precede that in

```
($-$-tak
  =
  (lambda (x y z)
    <t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28 t-44 i-45>
    (set! t-44 #f)
    (do
      (i-45 ??)
      (set! t-22[i-45.1] (< y x[i-45.0]))
      (cond
        ( t-22[i-45.1]
          (set! x[i-45.1] (1- x[i-45.0]))
          (set! t-44[i-45.1] t-44[i-45.0]) )
        ( else
          (set! t-44[i-45.1] (#or t-44[i-45.0] i-45))
          (set! x[i-45.1] x[i-45.0]) ) ))
    (set! t-20 z)
    (do
      (i-45 t-44)
      (set! t-27 (1- y))
      (set! t-24 (#self-closure# t-27 z x[[i-45.0]]))
      (set! t-28 (1- z))
      (set! t-25 (#self-closure# t-28 x[[i-45.0]] y))
      (set! t-20 (#self-closure# t-20 t-24 t-25)))
    (return t-20) ))
```

Figure 74: The Forward Loop is Cleaned Up Before Proceeding

```
($-$-tak
  =
  (lambda (x y z)
    <t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28 t-44 i-45>
    (set! t-44 #f)
    (do
      (i-45 ??)
      (if
        t-22[i-45.1]
        (set! x[i-45.1] (1- x[i-45.0]))
        (set! x[i-45.1] x[i-45.0])))
    (do (i-45 ??) (set! t-22[i-45.1] (< y x[i-45.0])))
    (do
      (i-45 ??)
      (if
        t-22[i-45.1]
        (set! t-44[i-45.1] t-44[i-45.0])
        (set! t-44[i-45.1] (#or t-44[i-45.0] i-45))))
    (set! t-20 z)
    (do
      (i-45 t-44)
      (set! t-27 (1- y))
      (set! t-24 (#self-closure# t-27 z x[[i-45.0]]))
      (set! t-28 (1- z))
      (set! t-25 (#self-closure# t-28 x[[i-45.0]] y))
      (set! t-20 (#self-closure# t-20 t-24 t-25)))
    (return t-20) ))
```

Figure 75: The Forward Loop is Distributed, and the Subloops Reordered

```
($-$-tak
  =
  (lambda (x y z)
    <t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28 t-44 i-45>
    (set! t-44 #f)
    (do (i-45 ??) (set! x[i-45.1] (1- x[i-45.0])))
    (do (i-45 ??) (set! t-22[i-45.1] (< y x[i-45.0])))
    (do
      (i-45 ??)
      (if
        t-22[i-45.1]
        (set! t-44[i-45.1] t-44[i-45.0])
        (set! t-44[i-45.1] (#or t-44[i-45.0]))))
    (set! t-20 z)
    (do
      (i-45 t-44)
      (set! t-27 (1- y))
      (set! t-24 (#self-closure# t-27 z x[[i-45.0]]))
      (set! t-28 (1- z))
      (set! t-25 (#self-closure# t-28 x[[i-45.0]] y))
      (set! t-20 (#self-closure# t-20 t-24 t-25)))
    (return t-20) ))
```

Figure 76: Exit Path Computations are Deleted in `$-$-tak`

which `t-44` is computed. It turns out that all of the computation in the forward loop is relevant to the computation of `t-44`. Had the compiler not eliminated the temporaries `t-26` and `t-23`, the loops in which `x` and `t-20` are computed, would each contain assignments to two variables and not merely one. This would defeat the recognition of the recurrence described by `x`; the recurrence described by `t-20` is intractable to the compiler in any event.

The next step is to delete any inert computation that arose from the (former) exit paths of the forward loop. See the discussion of Figure 65 for an explanation of this transformation, and see Figure 76 for its outcome, in the case of `$-$-tak`.

At this point, the compiler examines the first three loops of Figure 76 to decide if each can be made parallel, either because it describes a familiar recurrence relation with a parallel solution, or because it is simply a loop whose iterations are independent of one another. See Figure 77. The compiler has found the computations of `x`, `t-22`, and `t-44` to be an induction sequence, a parallel loop, and the familiar recurrence described by `#or`, re-

```
($-$-tak
  =
  (lambda (x y z)
    <t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28 t-44 i-45>
    (set! t-44 #f)
    (do-induction (i-45 ??) (set! x[i-45.1]
                                          (1- x[i-45.0])))
    (doall (i-45 ??) (set! t-22[i-45.1] (< y x[i-45.0])))
    (do-recurrence
      (i-45 ??)
      (if
        t-22[i-45.1]
        (set! t-44[i-45.1] t-44[i-45.0])
        (set! t-44[i-45.1] (#or t-44[i-45.0]))))
    (set! t-20 z)
    (do
      (i-45 t-44)
      (set! t-27 (1- y))
      (set! t-24 (#self-closure# t-27 z x[[i-45.0]]))
      (set! t-28 (1- z))
      (set! t-25 (#self-closure# t-28 x[[i-45.0]] y))
      (set! t-20 (#self-closure# t-20 t-24 t-25)))
    (return t-20) ))
```

Figure 77: Parallel Loops (from the Forward Loop) are Recognized

```
($-$-tak
  =
  (lambda (x y z)
    <t-20 t-22 t-23 t-24 t-25 t-26
     t-27 t-28 t-44 i-45 i-46 i-47>
    (set! t-44 #f)
    (set! t-44 (pos-diff x y))
    (do-induction
      (i-46 t-44)
      (set! x[i-46.1] (1- x[i-46.0])))
    (doall (i-47 t-44) (set! t-22[i-47.1] (< y x[i-47.0])))
    (set! t-20 z)
    (do
      (i-45 t-44)
      (set! t-27 (1- y))
      (set! t-24 (#self-closure# t-27 z x[[i-45.0]]))
      (set! t-28 (1- z))
      (set! t-25 (#self-closure# t-28 x[[i-45.0]] y))
      (set! t-20 (#self-closure# t-20 t-24 t-25)))
    (return t-20) ))
```

Figure 78: A Closed-Form Solution for `t-44` is Found

spectively. At this point, recursion splitting has succeeded. As before, the compiler will attempt to rewrite the computation of `t-44` in as efficient a form as possible, and will then proceed with the parallelization of the rest of the procedure.

See Figure 78. Once again, a closed-form solution to the recurrence described by `t-44` has been found, in terms of `x` and `y`. The expression `(pos-diff x y)` is `(- x y)` if `x` is greater than or equal to `y`, and zero otherwise. There is some dead code in this figure; in particular, the entire loop in which `t-22` is computed is now dead code, since this variable has no uses. Parcel will ultimately recognize this and delete it.

Before moving on to the backward loop, the compiler adds code to allocate and initialize the vectors which represent expanded variables, and to restore the final positions of these vectors to the variables, following the loop. See Figure 79, and the discussion of Figure 68 for an explanation of the functions `allocate` and `restore`. The computation of the values of `x` has been rewritten as an invocation of the procedure `add2-ind`. This procedure takes two arguments: a vector `v` and an integer `k`. `v[0.0]` is assumed to be initialized to an integer value. Upon return, `v[i.0]` contains the value `v[0.0]+ik`, for every element of the vector. This computation

```
($-$-tak
  =
  (lambda (x y z)
    <t-20 t-22 t-23 t-24 t-25 t-26 t-27
     t-28 t-44 i-45 i-46 i-47 t-48>
    (set! t-44 #f)
    (set! t-44 (pos-diff x y))
    (set! x (allocate x t-44))
    (set! t-22 (allocate #f t-44))
    (add2-ind x -1)
    (doall (i-47 t-44) (set! t-22[i-47.1] (< y x[i-47.0])))
    (set! t-48 x)
    (set! x (restore x t-44))
    (set! t-20 z)
    (set! x (reallocate t-48))
    (do
      (i-45 t-44)
      (set! t-27 (1- y))
      (set! t-24 (#self-closure# t-27 z x[[i-45.0]]))
      (set! t-28 (1- z))
      (set! t-25 (#self-closure# t-28 x[[i-45.0]] y))
      (set! t-20 (#self-closure# t-20 t-24 t-25)))
    (return t-20) ))
```

Figure 79: The Restructuring of the Forward Loop is Completed

may be performed in parallel, of course.

3.4.4 Variable Expansion and the Bottom of Recursion

Recursion splitting requires some fairly detailed manipulations of the procedure; a good example is in handling the computation that occurs at the "bottom" of recursion. In this case, the assignment `(set! t-20 z)` is the entire computation performed at the bottom of recursion, and establishes the first in the sequence of values taken by `t-20`, the variable whose final value is returned by `$-$-tak`. In general, however, there might be a more complex computation at the bottom of recursion, that involves any of the parameters and local variables of the procedure. In particular, the value of `x` might be required for this computation. When, however, execution of the forward loop is completed, `x` points to a vector of values; it is the last of these values that must be assigned to `x`, so that the computation at the bottom of recursion may be performed sensibly. This is the purpose of the expression `(set! x (restore x t-44))`. However, the vector which holds the values of `x` as computed in the forward loop is needed by the backward loop; it is therefore saved in the variable `t-48`, and the function `reallocate` is used to assign this vector to `x`, just prior to the backward loop. In reality, `reallocate` is an identity function, it does nothing; but for reasons that have to do with the implementation details of Parcel, a function `reallocate` is used, to inform the compiler of the purpose of this expression, for later optimizations.

3.4.5 Parallelization of the Backward Loop

The remainder of the parallelization process is easy. The backward loop does not need to be subjected to exit-loop translation: its number of iterations is `t-44`, the same as that of the forward loop. First, the variables defined within the backward loop are expanded. See Figure 80. Next, the backward loop is distributed. See Figure 81. The loops that result from distribution are classified as parallel, recurrences, and so on: see Figure 82. All of the loops, except the last, are seen to be parallel, and are marked as `doall` forms accordingly. At this point, there are needlessly many loops, and the compiler remedies this by "undistributing" as much as possible, a transformation usually called *loop fusion* [47]. See Figure 83.

The version of the procedure that emerges from recursion splitting is given in Figure 84. The expanded variables of the backward loop are allocated with routines analogous to `allocate` and `restore`, called `allocate-r` and `restore-r` respectively. The vectors that represent expanded variables at run-time are marked as arising either from a forward loop (the default case, as would apply also to the loop in Figure 60) or a backward loop created by recursion splitting. These markings are the only difference be-

```
($-$-tak
  =
  (lambda (x y z)
    <t-20 t-22 t-23 t-24 t-25 t-26 t-27
     t-28 t-44 i-45 i-46 i-47 t-48>
    (set! t-44 #f)
    (set! t-44 (pos-diff x y))
    (set! x (allocate x t-44))
    (set! t-22 (allocate #f t-44))
    (add2-ind x -1)
    (doall (i-47 t-44) (set! t-22[i-47.1] (< y x[i-47.0])))
    (set! t-48 x)
    (set! x (restore x t-44))
    (set! t-20 z)
    (set! x (reallocate t-48))
    (do
      (i-45 t-44)
      (set! t-27[i-45.1] (1- y))
      (set! t-24[i-45.1] (#self-closure# t-27[i-45.1]
                                         z x[[i-45.0]]))
      (set! t-28[i-45.1] (1- z))
      (set! t-25[i-45.1] (#self-closure# t-28[i-45.1]
                                         x[[i-45.0]] y))
      (set!
        t-20[i-45.1]
        (#self-closure# t-20[i-45.0]
                        t-24[i-45.1] t-25[i-45.1])))
    (return t-20) ))
```

Figure 80: Variables Defined in the Backward Loop are Expanded

tween the vectors created by `allocate` and `allocate-r`. The distinction is important to some recurrence solution routines, which take two arguments, one a vector to be read, the other a vector to be written. In the event that the input vector represents a variable expanded in the forward loop, and the output vector a variable expanded in the backward loop, the proper subscript functions must be selected, based upon the types of these vectors, so that the input vector is read in the correct "direction".

The final parallel version of `$-$-tak` is given in Figure 85. A nesting of parallel loops has occurred because the compiler has wrapped two recursive calls to `$-$-tak` in a "`cobegin`" form, expressed as a `doall` loop of two iterations. It is very informative to follow the parallel evaluation of this version of the procedure, to appreciate to what a flood of parallelism it gives rise, as it makes recursive calls within parallel loops.

3.5 High-Level (Coarse-Grained) Parallelism

It might have occurred to the reader to object that the first two program examples we have considered hardly call for a machinery of interprocedural analysis so elaborate as that we constructed in Section 2. These simple programs contained few procedures and no interprocedurally visible side-effects; we must turn to a more involved example if we are to see how the analysis facilitates the discovery of high-level (coarse-grained) parallelism.

We have chosen the `boyer` benchmark from the Gabriel benchmark suite [21], for the reason that it comprises a number of procedures, and makes use of simple, interprocedural side-effects. In particular, we will focus our attention upon the procedures `one-way-unify`, `one-way-unify-lst`, `rewrite`, `rewrite-args`, and `rewrite-with-lemmas`. See figures 86 and 87. These procedures, as they are seen by the compiler following parsing, are shown in figures 88, 89, 90, 91, and 92. We have made some small but significant changes to the benchmark as it is found in [21], for the purpose of illustrating Parcel's treatment of side-effects; these changes have no effect upon the values computed by the benchmark. In the original benchmark there are two global variables, `temp-temp` and `unify-subst`, both of which are used in a somewhat local manner. We have made `unify-subst` a local variable of `rewrite-with-lemmas`, as its value is overwritten every time `rewrite-with-lemmas` is invoked. We have therefore made `one-way-unify` a local procedure of `rewrite-with-lemmas` (since `unify-subst` occurs free in its body), and likewise with `one-way-unify-lst` (since it calls `one-way-unify`). Similarly, the variable `temp-temp` is used momentarily within `one-way-unify` to hold a temporary quantity, to avoid twice evaluating an expression. We have therefore given `one-way-unify` a local variable by the same name, although we could as easily have written a `let`

form to introduce temp-temp.

```
($-$-tak
  =
  (lambda (x y z)
    < t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28 t-44
      i-45 i-46 i-47 t-48 i-49 i-50 i-51 i-52 i-53 >
    (set! t-44 #f)
    (set! t-44 (pos-diff x y))
    (set! x (allocate x t-44))
    (set! t-22 (allocate #f t-44))
    (add2-ind x -1)
    (doall (i-47 t-44) (set! t-22[i-47.1] (< y x[i-47.0])))
    (set! t-48 x)
    (set! x (restore x t-44))
    (set! t-20 z)
    (set! x (reallocate t-48))
    (do (i-49 t-44) (set! t-28[i-49.1] (1- z)))
    (do
      (i-50 t-44)
      (set! t-25[i-50.1]
            (#self-closure# t-28[i-50.1]
                            x[[i-50.0]] y)))
    (do (i-51 t-44) (set! t-27[i-51.1] (1- y)))
    (do
      (i-52 t-44)
      (set! t-24[i-52.1] (#self-closure# t-27[i-52.1]
                                         z x[[i-52.0]])))
    (do
      (i-53 t-44)
      (set!
        t-20[i-53.1]
        (#self-closure# t-20[i-53.0]
                        t-24[i-53.1] t-25[i-53.1])))
    (return t-20) ))
```

Figure 81: The Backward Loop is Distributed

```
($-$-tak
  =
  (Lambda (x y z)
    < t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28 t-44
      i-45 i-46 i-47 t-48 i-49 i-50 i-51 i-52 i-53 >
    (set! t-44 #f)
    (set! t-44 (pos-diff x y))
    (set! x (allocate x t-44))
    (set! t-22 (allocate #f t-44))
    (add2-ind x -1)
    (doall (i-47 t-44) (set! t-22[i-47.1] (< y x[i-47.0])))
    (set! t-48 x)
    (set! x (restore x t-44))
    (set! t-20 z)
    (set! x (reallocate t-48))
    (doall (i-49 t-44) (set! t-28[i-49.1] (1- z)))
    (doall
      (i-50 t-44)
      (set! t-25[i-50.1] (#self-closure# t-28[i-50.1]
                                         x[[i-50.0]] y)))
    (doall (i-51 t-44) (set! t-27[i-51.1] (1- y)))
    (doall
      (i-52 t-44)
      (set! t-24[i-52.1] (#self-closure# t-27[i-52.1]
                                         z x[[i-52.0]])))
    (do
      (i-53 t-44)
      (set!
        t-20[i-53.1]
        (#self-closure# t-20[i-53.0]
                        t-24[i-53.1] t-25[i-53.1])))
    (return t-20) ))
```

Figure 82: Parallel Loops and Recurrences are Recognized

```
($-$-tak
  =
  (lambda (x y z)
    < t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28 t-44
      i-45 i-46 i-47 t-48 i-49 i-50 i-51 i-52 i-53 >
    (set! t-44 #f)
    (set! t-44 (pos-diff x y))
    (set! x (allocate x t-44))
    (set! t-22 (allocate #f t-44))
    (add2-ind x -1)
    (doall (i-47 t-44) (set! t-22[i-47.1] (< y x[i-47.0])))
    (set! t-48 x)
    (set! x (restore x t-44))
    (set! t-20 z)
    (set! x (reallocate t-48))
    (doall
      (i-49 t-44)
      (set! t-27[i-49.1] (1- y))
      (set! t-24[i-49.1] (#self-closure# t-27[i-49.1]
                                         z x[[i-49.0]]))
      (set! t-28[i-49.1] (1- z))
      (set! t-25[i-49.1] (#self-closure# t-28[i-49.1]
                                         x[[i-49.0]] y)))
    (do
      (i-53 t-44)
      (set!
        t-20[i-53.1]
        (#self-closure# t-20[i-53.0]
                        t-24[i-53.1] t-25[i-53.1])))
    (return t-20) ))
```

Figure 83: Parallel Loops are Coalesced

```
($-$-tak
  =
  (lambda (x y z)
    < t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28 t-44
      i-45 i-46 i-47 t-48 i-49 i-50 i-51 i-52 i-53 >
    (set! t-44 (pos-diff x y))
    (set! x (allocate x t-44))
    (add2-ind x -1)
    (set! t-20 (allocate-r z t-44))
    (set! t-24 (allocate-r #f t-44))
    (set! t-25 (allocate-r #f t-44))
    (set! t-27 (allocate-r #f t-44))
    (set! t-28 (allocate-r #f t-44))
    (doall
      (i-49 t-44)
      (set! t-27[i-49.1] (1- y))
      (set! t-24[i-49.1] (#self-closure# t-27[i-49.1]
                                         z x[[i-49.0]]))
      (set! t-28[i-49.1] (1- z))
      (set! t-25[i-49.1] (#self-closure# t-28[i-49.1]
                                         x[[i-49.0]] y)))
    (do
      (i-53 t-44)
      (set!
        t-20[i-53.1]
        (#self-closure# t-20[i-53.0]
                        t-24[i-53.1] t-25[i-53.1])))
    (set! t-20 (restore-r t-20 t-44))
    (return t-20) ))
```

Figure 84: After Recursion Splitting

```
($-$-tak
  =
  (lambda (x y z)
    < t-20 t-22 t-23 t-24 t-25 t-26 t-27 t-28 t-44
      i-45 i-46 i-47 t-48 i-49 i-50 i-51 i-52 i-53 i-54 >
    (set! t-44 (pos-diff x y))
    (set! x (allocate x t-44))
    (add2-ind x -1)
    (set! t-20 (allocate-r z t-44))
    (set! t-24 (allocate-r #f t-44))
    (set! t-25 (allocate-r #f t-44))
    (set! t-27 (allocate-r #f t-44))
    (set! t-28 (allocate-r #f t-44))
    (doall
      (i-49 t-44)
      (set! t-28[i-49.1] (1- z))
      (set! t-27[i-49.1] (1- y))
      (doall
        (i-54 2)
        (mway
          i-54
          (0 (set! t-24[i-49.1]
                   (#self-closure# t-27[i-49.1]
                                   z x[[i-49.0]])))
          (1 (set! t-25[i-49.1]
                   (#self-closure# t-28[i-49.1]
                                   x[[i-49.0]] y))))))
    (do
      (i-53 t-44)
      (set!
        t-20[i-53.1]
        (#self-closure# t-20[i-53.0]
                        t-24[i-53.1] t-25[i-53.1])))
    (set! t-20 (restore-r t-20 t-44))
    (return t-20) ))
```

Figure 85: The Final (Parallel) Version of `$-$-tak`

```
(define
  rewrite
  (lambda (term)
    (cond
      ( (atom? term)
        term )
      ( #t
        (rewrite-with-lemmas
          (cons (car term) (rewrite-args (cdr term)))
          (getprop (car term) 'lemmas)) ) ) ))
(define
  rewrite-args
  (lambda (lst)
    (cond
      ( (null? lst)
        #f )
      ( #t
        (cons (rewrite (car lst)) (rewrite-args (cdr lst))) ) ) ))
```

Figure 86: The Procedures `rewrite` and `rewrite-args`

These modifications were made by hand for the reason that Parcel is not equipped to change the status of a variable from global to local (under the assumption that a programmer will avoid global variables when they are not needed), although the conditions under which such a transformation may be applied are easily expression in terms of procedure strings and stack configurations.

Let's first consider the variable `unify-subst`. It occurs as a free variable in the procedure `one-way-unify`, where it is modified as well as used. Nevertheless, analysis by $\mathcal{E}_5$ or $\mathcal{E}_7$ (as defined in Section 2) reveals that the procedures `rewrite`, `rewrite-args`, and `rewrite-with-lemmas` have no side-effects upon this variable. Recall the test embodied in Theorem 7 for side-effects: the closure which captures `unify-subst` makes no (net) downward movement into `rewrite`, `rewrite-args` or `rewrite-with-lemmas` before being applied. As a diversion, we might consider annotating this program using the "type" system of side-effects described in [23]. We would find that, because it is captured by a closure which modifies it, the variable `unify-subst` induces side-effects in all the routines that are (indirect) callers of `one-way-unify`, including `rewrite`, `rewrite-args` and `rewrite-with-lemmas`. This program is therefore an example in which the automatic side-effect analysis of Parcel has greater accuracy than is possible for the user to achieve manually, using the system of [23].

Let us turn to the procedure `rewrite-args` of Figure 91. This is a simple and somewhat typical procedure, that might have been the result of macro-

```
(define
  rewrite-with-lemmas
  (lambda (term lst)
    (define unify-subst #f)
    (define
      one-way-unify
      (lambda (term1 term2)
        (define temp-temp)
        (cond
          ( (atom? term2)
            (cond
              ( (set! temp-temp (assq term2 unify-subst))
                (equal? term1 (cdr temp-temp)) )
              ( #t
                (set! unify-subst (cons (cons term2 term1) unify-subst))
                #t ) ) )
          ( (atom? term1)
            #f )
          ( (eq? (car term1) (car term2))
            (one-way-unify-lst (cdr term1) (cdr term2)) )
          ( #t
            #f ) ) ))
    (define
      one-way-unify-lst
      (lambda (lst1 lst2)
        (cond
          ( (null? lst1)
            #t )
          ( (one-way-unify (car lst1) (car lst2))
            (one-way-unify-lst (cdr lst1) (cdr lst2)) )
          ( #t
            #f ) ) ))
    (cond
      ( (null? lst)
        term )
      ( (one-way-unify term (cadr (car lst)))
        (rewrite (apply-subst unify-subst (caddr (car lst)))) )
      ( #t
        (rewrite-with-lemmas term (cdr lst)) ) ) ))
```

Figure 87: The Procedure `rewrite-with-lemmas`, and its Subroutines

```
($-$-rewrite-with-lemmas
  =
  (lambda (term lst)
    < unify-subst one-way-unify one-way-unify-lst
      t-139 t-140 t-141 t-142 t-143 t-144 t-145 t-146 t-147 >
    (set! unify-subst (id #f))
    (set! one-way-unify
          (id #<$-$-rewrite-with-lemmas-one-way-unify>))
    (set! one-way-unify-lst
          (id #<$-$-rewrite-with-lemmas-one-way-unify-lst>))
    (set! t-140 (null? lst))
    (cond
      ( t-140
        (set! t-139 (id term)) )
      ( else
        (set! t-143 (car lst))
        (set! t-142 (cadr t-143))
        (set! t-141 (one-way-unify term t-142))
        (cond
          ( t-141
            (set! t-146 (car lst))
            (set! t-145 (caddr t-146))
            (set! t-144 (apply-subst unify-subst t-145))
            (set! t-139 (rewrite t-144)) )
          ( else
            (set! t-147 (cdr lst))
            (set! t-139 (rewrite-with-lemmas term t-147)) ) ) ) )
    (return t-139) ))
```

Figure 88: The Procedure `$-$-rewrite-with-lemmas` After Parsing

```
($-$-rewrite-with-lemmas-one-way-unify-lst
  =
  (lambda (lst1 lst2)
    <t-132 t-133 t-134 t-135 t-136 t-137 t-138>
    (set! t-133 (null? lst1))
    (cond
      ( t-133
        (set! t-132 (id #t)) )
      ( else
        (set! t-135 (car lst1))
        (set! t-136 (car lst2))
        (set! t-134 (one-way-unify t-135 t-136))
        (cond
          ( t-134
            (set! t-137 (cdr lst1))
            (set! t-138 (cdr lst2))
            (set! t-132 (one-way-unify-lst t-137 t-138)) )
          ( else
            (set! t-132 (id #f)) ) ) ) )
    (return t-132) ))
```

Figure 89: `$-$-rewrite-with-lemmas-one-way-unify-lst`

expanding an expression like `(mapcar rewrite lst)`. It is recursive, but not tail-recursive, and so the compiler will treat it by recursion splitting. See Figure 93. Here, the forward and backward loops have been formed. As always, the forward loop will be subjected to exit-loop translation; the `#or` expression is placed along what was formerly the path taken at the "bottom" of recursion. Ultimately, `t-204` will hold the number of iterations of this procedure. The progress of recursion splitting is straightforward. The variables defined in the forward loop are expanded (Figure 94), some traditional optimizations are applied to clean up unneeded temporary variables (Figure 95), the forward loop is distributed (Figure 96), and the resulting subloops are reordered so that the one in which `t-204` is computed is preceded by as few as possible (Figure 97). Next, any computation that was not performed by the original loop, that falls along what were previously exit paths from the loop, is eliminated (Figure 98). Each of the subloops that was created by distribution of the forward loop is then examined, to see if it is a `doall` loop or a familiar recurrence relation (Figure 99). At this point, we see that the compiler has succeeded in uncovering parallelism that is quite coarse in granularity: the call to `$-$-rewrite` within the forward loop has been placed in a `doall` loop. Since `$-$-rewrite` may ultimately invoke `$-$-rewrite`, `$-$-rewrite-args`, and `$-$-rewrite-with-lemmas` recursively, this can give rise to a flurry of nested parallel activity at runtime.

```
($-$-rewrite-with-lemmas-one-way-unify
  =
  (lambda (term1 term2)
    <temp-temp t-120 t-121 t-122 t-123 t-124
     t-125 t-126 t-127 t-128 t-129>
    (set! t-121 (atom? term2))
    (cond
      ( t-121
        (set! temp-temp (assq term2 unify-subst))
        (cond
          ( temp-temp
            (set! t-126 (cdr temp-temp))
            (set! t-120 (equal? term1 t-126)) )
          ( else
            (set! t-127 (cons term2 term1))
            (set! unify-subst (cons t-127 unify-subst))
            (set! t-120 (id #t)) ) ) )
      ( else
        (set! t-122 (atom? term1))
        (cond
          ( t-122
            (set! t-120 (id #f)) )
          ( else
            (set! t-124 (car term1))
            (set! t-125 (car term2))
            (set! t-123 (eq? t-124 t-125))
            (cond
              ( t-123
                (set! t-128 (cdr term1))
                (set! t-129 (cdr term2))
                (set! t-120 (one-way-unify-lst t-128 t-129)) )
              ( else
                (set! t-120 (id #f)) ) ) ) ) ) )
    (return t-120) ))
```

Figure 90: The Procedure `$-$-rewrite-with-lemmas-one-way-unify` After Parsing

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-107 t-108 t-109 t-110 t-111>
    (set! t-107 (null? lst))
    (cond
      ( t-107
        (set! t-106 (id #f)) )
      ( else
        (set! t-110 (car lst))
        (set! t-108 (rewrite t-110))
        (set! t-111 (cdr lst))
        (set! t-109 (rewrite-args t-111))
        (set! t-106 (cons t-108 t-109)) ) )
    (return t-106) ))
```

Figure 91: The Procedure `$-$-rewrite-args` After Parsing

```
($-$-rewrite
  =
  (lambda (term)
    <t-97 t-98 t-99 t-100 t-101 t-102 t-103 t-104>
    (set! t-98 (atom? term))
    (cond
      ( t-98
        (set! t-97 (id term)) )
      ( else
        (set! t-101 (car term))
        (set! t-103 (cdr term))
        (set! t-102 (rewrite-args t-103))
        (set! t-99 (cons t-101 t-102))
        (set! t-104 (car term))
        (set! t-100 (getprop t-104 'lemmas))
        (set! t-97 (rewrite-with-lemmas t-99 t-100)) ) )
    (return t-97) ))
```

Figure 92: The Procedure `$-$-rewrite` After Parsing

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111 t-204 i-205>
    (set! t-204 #f)
    (do
      (i-205 ??)
      (cond
        ( lst
          (set! t-110 (car lst))
          (set! t-108 (rewrite t-110))
          (set! t-111 (cdr lst))
          (set! lst t-111) )
        ( else
          (set! t-204 (#or t-204 i-205)) ) ))
    (set! t-106 #f)
    (do (i-205 t-204)
        (set! t-109 t-106)
        (set! t-106 (cons t-108 t-109)))
    (return t-106) ))
```

Figure 93: The Forward and Backward Loops are Formed

As might have been expected, the compiler discovers that `t-204` may be computed directly by the expression `(length lst)`; see Figure 100. Before moving on to the backward loop, the compiler fuses loops where possible, among those that originated from the forward loop (Figure 101), adds code to allocate and restore the expanded variables of the forward loop (Figure 102), and translates any recognized recurrences among the subloops originating from the forward loop (Figure 103).

The treatment of the backward loop is much simpler, by contrast: the variable `t-106` is expanded (Figure 104), the recurrence it describes is recognized (Figure 99), and finally this recurrence is rewritten as a call to the run-time procedure `cons-rem-ind` (Figure 106). `cons-rem-ind` is a simple version of `cons-rem-rec`, used in the translation of `quicksort` in subsection 3.3.9. Its two arguments are an input vector `x` of values, and an output vector `y`. A list of length equal to that of `x` is constructed, whose top level contains the values in `x`, in reverse order; this list is appended to the head of the value `x[0.0]`, and the result is pointed to by `y[N.0]`, where `N` is the number of iterations of the loop in which `x` and `y` are expanded. This procedure, like `cons-rem-rec`, solves a recurrence for its remote term; it is for this reason that only the last position of `y` is given a value. The final version of `$-$-rewrite-args` is shown in Figure 107.

There are two important points to be made by this example. First,

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111 t-204 i-205>
    (set! t-204 #f)
    (do
      (i-205 ??)
      (cond
        ( lst[i-205.0]
          (set! t-110[i-205.1] (car lst[i-205.0]))
          (set! t-108[i-205.1] (rewrite t-110[i-205.1]))
          (set! t-111[i-205.1] (cdr lst[i-205.0]))
          (set! lst[i-205.1] t-111[i-205.1])
          (set! t-204[i-205.1] t-204[i-205.0]) )
        ( else
          (set! t-204[i-205.1] (#or t-204[i-205.0] i-205))
          (set! lst[i-205.1] lst[i-205.0])
          (set! t-108[i-205.1] t-108[i-205.0]) ) ))
    (set! t-106 #f)
    (do
      (i-205 t-204)
      (set! t-109 t-106)
      (set! t-106 (cons t-108[[i-205.1]] t-109)))
    (return t-106) ))
```

Figure 94: Variables Defined in the Forward Loop are Expanded

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111 t-204 i-205>
    (set! t-204 #f)
    (do
      (i-205 ??)
      (cond
        ( lst[i-205.0]
          (set! t-110[i-205.1] (car lst[i-205.0]))
          (set! t-108[i-205.1] (rewrite t-110[i-205.1]))
          (set! lst[i-205.1] (cdr lst[i-205.0]))
          (set! t-204[i-205.1] t-204[i-205.0]) )
        ( else
          (set! t-204[i-205.1] (#or t-204[i-205.0] i-205))
          (set! lst[i-205.1] lst[i-205.0])
          (set! t-108[i-205.1] t-108[i-205.0]) ) ))
    (set! t-106 #f)
    (do (i-205 t-204) (set! t-106 (cons t-108[[i-205.1]] t-106)))
    (return t-106) ))
```

Figure 95: The Forward Loop is Cleaned up Before Proceeding

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111 t-204 i-205>
    (set! t-204 #f)
    (do
      (i-205 ??)
      (if
        lst[i-205.0]
        (set! lst[i-205.1] (cdr lst[i-205.0]))
        (set! lst[i-205.1] lst[i-205.0])))
    (do (i-205 ??)
        (if lst[i-205.0]
            (set! t-110[i-205.1] (car lst[i-205.0]))))
    (do
      (i-205 ??)
      (if
        lst[i-205.0]
        (set! t-108[i-205.1] (rewrite t-110[i-205.1]))
        (set! t-108[i-205.1] t-108[i-205.0])))
    (do
      (i-205 ??)
      (if
        lst[i-205.0]
        (set! t-204[i-205.1] t-204[i-205.0])
        (set! t-204[i-205.1] (#or t-204[i-205.0] i-205))))
    (set! t-106 #f)
    (do (i-205 t-204) (set! t-106 (cons t-108[[i-205.1]] t-106)))
    (return t-106) ))
```

Figure 96: The Forward Loop is Distributed

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111 t-204 i-205>
    (set! t-204 #f)
    (do
      (i-205 ??)
      (if
        lst[i-205.0]
        (set! lst[i-205.1] (cdr lst[i-205.0]))
        (set! lst[i-205.1] lst[i-205.0])))
    (do
      (i-205 ??)
      (if
        lst[i-205.0]
        (set! t-204[i-205.1] t-204[i-205.0])
        (set! t-204[i-205.1] (#or t-204[i-205.0] i-205))))
    (do (i-205 ??)
        (if lst[i-205.0]
            (set! t-110[i-205.1] (car lst[i-205.0]))))
    (do
      (i-205 ??)
      (if
        lst[i-205.0]
        (set! t-108[i-205.1] (rewrite t-110[i-205.1]))
        (set! t-108[i-205.1] t-108[i-205.0])))
    (set! t-106 #f)
    (do (i-205 t-204) (set! t-106 (cons t-108[[i-205.1]] t-106)))
    (return t-106) ))
```

Figure 97: Subloops of the Forward Loop are Reordered

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111 t-204 i-205 i-206 i-207>
    (set! t-204 #f)
    (do (i-205 ??) (set! lst[i-205.1] (cdr lst[i-205.0])))
    (do
      (i-205 ??)
      (if
        lst[i-205.0]
        (set! t-204[i-205.1] t-204[i-205.0])
        (set! t-204[i-205.1] (#or t-204[i-205.0] i-205))))
    (do (i-206 t-204)
        (set! t-110[i-206.1] (car lst[i-206.0])))
    (do (i-207 t-204)
        (set! t-108[i-207.1] (rewrite t-110[i-207.1])))
    (set! t-106 #f)
    (do (i-205 t-204)
        (set! t-106 (cons t-108[[i-205.1]] t-106)))
    (return t-106) ))
```

Figure 98: Exit-Path Computations are Eliminated

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111 t-204 i-205 i-206 i-207>
    (set! t-204 #f)
    (do-induction (i-205 ??)
                  (set! lst[i-205.1] (cdr lst[i-205.0])))
    (do-recurrence
      (i-205 ??)
      (if
        lst[i-205.0]
        (set! t-204[i-205.1] t-204[i-205.0])
        (set! t-204[i-205.1] (#or t-204[i-205.0] i-205))))
    (doall (i-206 t-204)
           (set! t-110[i-206.1] (car lst[i-206.0])))
    (doall (i-207 t-204)
           (set! t-108[i-207.1] (rewrite t-110[i-207.1])))
    (set! t-106 #f)
    (do (i-205 t-204)
        (set! t-106 (cons t-108[[i-205.1]] t-106)))
    (return t-106) ))
```

Figure 99: Doall Loops and Recurrences are Recognized

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111
     t-204 i-205 i-206 i-207 i-208>
    (set! t-204 #f)
    (set! t-204 (length lst))
    (do-induction (i-208 t-204)
                  (set! lst[i-208.1] (cdr lst[i-208.0])))
    (doall (i-206 t-204)
           (set! t-110[i-206.1] (car lst[i-206.0])))
    (doall (i-207 t-204)
           (set! t-108[i-207.1] (rewrite t-110[i-207.1])))
    (set! t-106 #f)
    (do (i-205 t-204)
        (set! t-106 (cons t-108[[i-205.1]] t-106)))
    (return t-106) ))
```

Figure 100: A Closed-Form Solution is found for t-204

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111
     t-204 i-205 i-206 i-207 i-208>
    (set! t-204 #f)
    (set! t-204 (length lst))
    (do-induction (i-208 t-204)
                  (set! lst[i-208.1] (cdr lst[i-208.0])))
    (doall
      (i-206 t-204)
      (set! t-110[i-206.1] (car lst[i-206.0]))
      (set! t-108[i-206.1] (rewrite t-110[i-206.1])))
    (set! t-106 #f)
    (do (i-205 t-204)
        (set! t-106 (cons t-108[[i-205.1]] t-106)))
    (return t-106) ))
```

Figure 101: Subloops of the Forward Loop are Fused

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111 t-204
     i-205 i-206 i-207 i-208 t-209>
    (set! t-204 #f)
    (set! t-204 (length lst))
    (set! lst (allocate lst t-204))
    (set! t-108 (allocate #f t-204))
    (set! t-110 (allocate #f t-204))
    (do-induction (i-208 t-204)
                  (set! lst[i-208.1] (cdr lst[i-208.0])))
    (doall
      (i-206 t-204)
      (set! t-110[i-206.1] (car lst[i-206.0]))
      (set! t-108[i-206.1] (rewrite t-110[i-206.1])))
    (set! t-209 t-108)
    (set! t-108 (restore t-108 t-204))
    (set! t-106 #f)
    (set! t-108 (reallocate t-209))
    (do (i-205 t-204)
        (set! t-106 (cons t-108[[i-205.1]] t-106)))
    (return t-106) ))
```

Figure 102: `allocate` and `restore` Forms are Introduced

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111 t-204
     i-205 i-206 i-207 i-208 t-209>
    (set! t-204 #f)
    (set! t-204 (length lst))
    (set! lst (allocate lst t-204))
    (set! t-108 (allocate #f t-204))
    (set! t-110 (allocate #f t-204))
    (cdr-ind lst 1)
    (doall
      (i-206 t-204)
      (set! t-110[i-206.1] (car lst[i-206.0]))
      (set! t-108[i-206.1] (rewrite t-110[i-206.1])))
    (set! t-209 t-108)
    (set! t-108 (restore t-108 t-204))
    (set! t-106 #f)
    (set! t-108 (reallocate t-209))
    (do (i-205 t-204)
        (set! t-106 (cons t-108[[i-205.1]] t-106)))
    (return t-106) ))
```

Figure 103: Recurrences from the Forward Loop are Translated

from Parcel's perspective, there is no difference between the extraction of coarse- and fine-grained parallelism: within `$-$-rewrite-args`, which during its execution may initiate a lengthy and interprocedurally complex subcomputation at each invocation it makes of `$-$-rewrite`, the compiler uncovered parallelism by applying exactly the techniques that were applied to `tak`, an "innermost" procedure. Of course, the scheduling implications of coarse- and fine-grained parallelism may be different, but this matter is left to the run-time system in Parcel, which is presented with a parallel and sequential version of each procedure, and has a flexible mechanism for selecting between them according to the utilization of processors at run-time.

Second, consider the collection of procedures represented in Figures 88 through 92. Suppose that of these, the compiler is successful in discovering parallelism only within `$-$-rewrite-args`. Nevertheless, because of recursion among these procedures, this may well give a satisfactory, if not abundant, degree of parallelism at run-time. The point is simply this: it is not necessary for the compiler to detect parallelism within every procedure of a program, in order to be successful in parallelizing the program as a whole.

```
($-$-rewrite-args
  =
  (lambda (lst)
    <t-106 t-108 t-109 t-110 t-111 t-204
     i-205 i-206 i-207 i-208 t-209>
    (set! t-204 #f)
    (set! t-204 (length lst))
    (set! lst (allocate lst t-204))
    (set! t-108 (allocate #f t-204))
    (set! t-110 (allocate #f t-204))
    (cdr-ind lst 1)
    (doall
      (i-206 t-204)
      (set! t-110[i-206.1] (car lst[i-206.0]))
      (set! t-108[i-206.1] (rewrite t-110[i-206.1])))
    (set! t-209 t-108)
    (set! t-108 (restore t-108 t-204))
    (set! t-106 #f)
    (set! t-108 (reallocate t-209))
    (do
      (i-205 t-204)
      (set! t-106[i-205.1]
            (cons t-108[[i-205.1]] t-106[i-205.0])))
    (return t-106) ))
```

Figure 104: Variables Defined in the Backward Loop are Expanded

```
($-$-rewrite-args
  =
  (lambda (lst)
    < t-106 t-108 t-109 t-110 t-111 t-204 i-205
      i-206 i-207 i-208 t-209 i-210 t-211 >
    (set! t-204 #f)
    (set! t-204 (length lst))
    (set! lst (allocate lst t-204))
    (set! t-108 (allocate #f t-204))
    (set! t-110 (allocate #f t-204))
    (cdr-ind lst 1)
    (doall
      (i-206 t-204)
      (set! t-110[i-206.1] (car lst[i-206.0]))
      (set! t-108[i-206.1] (rewrite t-110[i-206.1])))
    (set! t-209 t-108)
    (set! t-108 (restore t-108 t-204))
    (set! t-106 #f)
    (set! t-108 (reallocate t-209))
    (do-rem-induction
      (i-210 t-204)
      (set! t-211[i-210.1]
            (cons t-108[[i-210.1]] t-211[i-210.0])))
    (set! t-106 (append2 t-106 t-211))
    (return t-106) ))
```

Figure 105: Doalls and Recurrences from the Backward Loop are Recognized

```
($-$-rewrite-args
  =
  (lambda (lst)
    < t-106 t-108 t-109 t-110 t-111 t-204 i-205
      i-206 i-207 i-208 t-209 i-210 t-211 >
    (set! t-204 #f)
    (set! t-204 (length lst))
    (set! lst (allocate lst t-204))
    (set! t-108 (allocate #f t-204))
    (set! t-110 (allocate #f t-204))
    (cdr-ind lst 1)
    (doall
      (i-206 t-204)
      (set! t-110[i-206.1] (car lst[i-206.0]))
      (set! t-108[i-206.1] (rewrite t-110[i-206.1])))
    (set! t-209 t-108)
    (set! t-108 (restore t-108 t-204))
    (set! t-106 #f)
    (set! t-108 (reallocate t-209))
    (set! t-211 (allocate-r #f t-204))
    (cons-rem-ind t-108 t-211)
    (set! t-211 (restore-r t-211 t-204))
    (set! t-106 (append2 t-106 t-211))
    (return t-106) ))
```

Figure 106: Recurrences from the Backward Loop are Translated

```
($-$-rewrite-args
  =
  (lambda (lst)
    < t-106 t-108 t-109 t-110 t-111 t-204 i-205
      i-206 i-207 i-208 t-209 i-210 t-211 >
    (set! t-204 (length lst))
    (set! lst (allocate lst t-204))
    (set! t-108 (allocate #f t-204))
    (set! t-110 (allocate #f t-204))
    (cdr-ind lst 1)
    (doall
      (i-206 t-204)
      (set! t-110[i-206.1] (car lst[i-206.0]))
      (set! t-108[i-206.1] (rewrite t-110[i-206.1])))
    (set! t-211 (allocate-r #f t-204))
    (cons-rem-ind t-108 t-211)
    (set! t-106 (restore-r t-211 t-204))
    (return t-106) ))
```

Figure 107: The Final, Parallel Version of `$-$-rewrite-args`

3.6 Organization of the Compiler

Having now seen them performed upon a number of procedures, the reader could probably sketch the algorithms for exit-loop translation and recursion splitting, informally. We will do so now.

Algorithm 1 (Exit-Loop Translation:)

1. *Select an index variable* `i` *and a variable* `n` *to hold the number of iterations of the loop.*
2. *Replace each exit branch from the loop by an assignment of the form* `(set! n (#or n i))`*; let control flow from this assignment to the bottom of the loop.*
3. *Perform variable expansion upon all variables defined in the loop.*
4. *Perform loop distribution.*
5. *Reorder the resulting subloops so that as few as possible precede that in which* `n` *is computed.*
6. *Mark each of the loops which precedes that in which* `n` *is computed as a recurrence relation for which a parallel solution is available, or a doall loop. Fail if any cannot be so marked.*
7. *Let* $\mathtt{v}_1, \ldots, \mathtt{v}_k, \mathtt{n}$ *be the variables which are computed in the loops which precede that in which* `n` *is computed (inclusive of that in which* `n` *is computed). If* `n` *(which is computed in terms of* $\mathtt{v}_1$ *through* $\mathtt{v}_k$*) describes a recurrence for which a closed form solution exists, emit that solution in place of the loop which computes* `n`*; else rewrite the computation of* $\mathtt{v}_1, \ldots, \mathtt{v}_k, \mathtt{n}$ *so that the first non-null value of* `n` *is found in parallel (as a "first-one" recurrence).*
8. *Make* `n` *the number of iterations of each of the remaining loops (those that follow that in which* `n` *is computed). Treat these loops by recurrence and doall recognition, recurrence translation, loop fusion, etc.*

Algorithm 2 (Recursion Splitting:)

1. *Let* `p` *be the procedure at hand. Select a fence* F*, a set of self-recursive calls to* `p`*, such that there is at most one member of* F *along any control path through the body of* `p`*.*

```
Parser
Interprocedural Analysis
Preparatory Optimizations
   Contour Merging
   Tail-recursion Elimination
   Expression Simplification / Strength Reduction
   Invariant Floating
   Copy Propagation
   Common Subexpression Elimination
   Dead Code Elimination
Parallelizing Transformations
   Exit Loop Translation
   Recursion Splitting
   Cobegin Insertion
```

Figure 108: The Organization of the Parcel Compiler

2. *Split the procedure into a forward and backward loop, using the members of the fence as the points of division.*

3. *Perform exit-loop translation upon the forward loop, as per Algorithm 1; at Step 3 of that algorithm, replace every reference made in iteration* `i` *of the backward loop to a variable defined in the forward loop, by a reference to the last value assumed by that variable in iteration* `n-i` *of the forward loop. Fail if exit-loop translation fails.*

4. *Expand the variables defined in the backward loop, and apply loop distribution, recurrence and doall recognition, loop fusion, etc.*

The organization of Parcel as a whole is given in Figure 108. As mentioned above, the preparatory optimizations are applied in a cycle, until the program stabilizes into a version which is unaffected by further attempts at optimization. Exit-loop translation is applied to the loops of a procedure, from the innermost loops to the outermost.

3.7 S-expressions in Parcel

The Parcel compiler permits the user to specify whether cons cells will be regarded as mutable or immutable. If they are regarded as mutable, then they are implemented in the conventional way, as a record of two fields, and their dependence implications are analyzed as described in subsection 2.14. In that case, however, recurrences over list data will not be recognized by

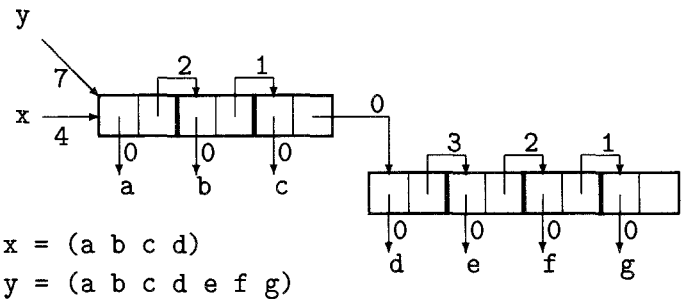

Figure 109: Two S-expressions Using Parcel's Representation

the compiler, so that some of Parcel's run-time functionality will go unused. On the other hand, if cons cells are regarded as immutable, then an alternative representation is given to them, that facilitates the parallelization of code which performs list-manipulating operations (`car`, `cdr`, `cons`, `append`, etc.); this representation is infeasible in the presence of destructive list operations. (Better would be for the compiler, by analysis, to partition the cons cells of the program into those which may be operated upon destructively, and those which are not. The appropriate representation would then be chosen accordingly.)

It is worth mentioning at the outset, that list-manipulation does not dominate modern Lisp code the way it might, say, programs written in Lisp 1.5; this owes to the data structuring facilities available in modern dialects (e.g., `define-structure`, object-oriented extensions). It is important, however, to a balanced strategy of parallelization, to address recurrence relations, such as those defined over s-expressions, which if neglected, introduce sequential bottlenecks into otherwise nicely parallel code.

Each pointer in this representation, whether a variable, a car or cdr pointer, etc., comprises three fields: a tag, a length, and an address. The tag indicates that the object pointed to is a proper (nil-terminated) list, an improper (non-nil terminated) list, or an object of a different type altogether. The length field may have a non-zero value only if the object pointed to is a list; its meaning will be explained shortly. The address field is the location in memory of the object pointed to (except in the case of immediate data). Figure 109 shows two s-expressions, `x` and `y`, constructed using such cons cells. Beside each arc (pointer) in the diagram is shown the corresponding length field. `x` and `y` have length 4 and 7 respectively.

A heavy line separating two cells indicates that the cells occupy adjacent memory locations (where the unit of memory is taken to be a single cons cell). Therefore, y occupies two contiguous blocks of memory, one of length 3, the other of length 4.[24]

The reader may have noticed that the meaning of the length field associated with a cdr pointer seems to be different from that of the length field associated with a car or variable pointer (such as x or y). The length field of a car or variable pointer indicates the number of cells in the top level of the s-expression pointed to; that of a cdr pointer indicates the number of cells "remaining" (to the "right") in the contiguous block containing the cdr pointer. For instance, the cell whose car is d in Figure 109 has a cdr pointer with a length field of 3; there are three cells to the right of this cell in the contiguous block containing it.

This representation permits two lists to share a cell without sharing the entire subexpression rooted at that cell. Furthermore, because our version of the comparator eq? examines length fields as well as addresses, such a cell will appear *not* to be eq? with the same cell in another s-expression that shares it, but that does not share the entire sub-expression rooted at it. For instance, (eq? (cdr x) (cdr y)) returns #f, where x and y are as shown in Figure 109. When we examine the mechanics of append in this representation, we will see that this behavior preserves the conventional semantics of eq?.

All pointers to atoms have a length field of zero; #f is the pointer whose tag indicates a proper list, and whose length is zero.

Another feature of this representation is that we may access any cell in the top level of a list in time proportional to the number of contiguous blocks of which the list is composed, and not to the number of cells in its top level. Recall, for example, that the procedure cdr-ind, described in subsection 3.3, must compute $(\texttt{cd}^{\texttt{i}}\texttt{k}\texttt{r}\ \texttt{v[0.0]})$ for $0 \leq \texttt{i} \leq \texttt{N} - 1$, where N is the length of its input vector v. Assuming that v[0.0] has relatively few blocks in its top level, this operation will take roughly constant time for all i. Furthermore, the routines cons-rem-rec and cons-rem-ind described above produce single, contiguous blocks. The hope, then, is that lists which are to be consumed by cdr-ind and similar routines, will have been produced by routines such as cons-rem-rec, cons-rem-ind, etc., although experimentation is needed to see if such hope is warranted.

A minor benefit of the representation is that the predicate equal? which compares s-expressions for isomorphism may be speeded by including a

[24]This representation bears a resemblance to *cdr-coding* [42] and *vector-coding* [25], but has both a different motivation and different properties. Our motivation is not to save memory, but rather to facilitate the parallel creation and access of lists.

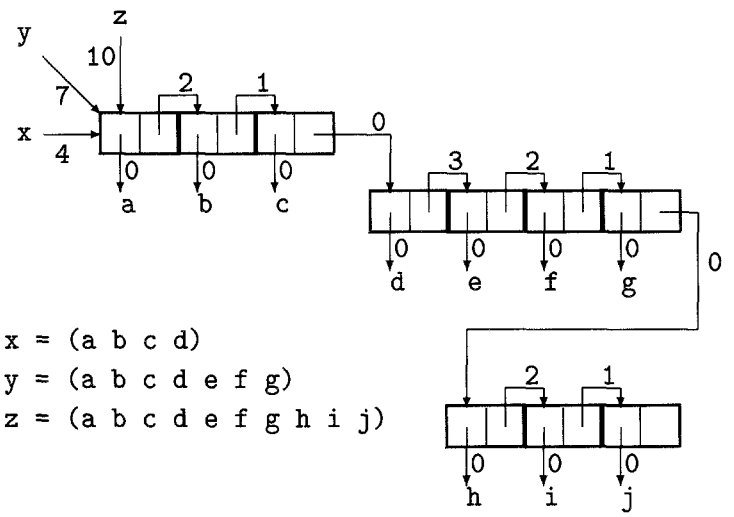

Figure 110: The Result of appending to y

comparison of lengths at every car-wise traversal. In fact, that the length of each list is available in constant time is of general utility; for example, many a loop in the restructured code produced by the compiler has as its number of iterations, the length of a list, or a simple function of the length; likewise, many of the techniques for solving recurrences involving operations upon lists make use of the length fields (for example, `cdr-ind`, `cons-rem-rec` and `cons-rem-ind` operate upon the length fields of their inputs and outputs).

The unusual sublist sharing permitted by this representation helps to regain some efficiency advantages sacrificed by forbidding `rplacd` (and, of course, does so without the side-effects for which `rplacd` is well known). Let's consider two common operations involving `rplacd`: the (destructive) addition of cells to the end of a list, and the (destructive) elimination of cells from the end of a list. In both cases, the disadvantage of aliased side-effects is offset by the fact that no copying of cells is necessary. Let's examine the analogous operations in Parcel's representation (that is, the non-destructive counterparts of these operations). Figure 110 shows the result of performing the operation `(set! z (append y '(h i j)))`, where y is as shown in Figure 109. No new cells have been created: the list `'(h i`

`j)` is merely tacked onto the end of `y` by walking to the end of `y`, discovering that its final cdr is unused, and placing the pointer to `'(h i j)` in this cdr pointer. A pointer to `y`, with a length field of 10, is returned as the value of `z`. Furthermore, the operation requires only time proportional to the number of contiguous blocks of which `y` is composed (in this case, 2). The variable `y` is *not* altered, as the end of `y` is defined, not by the presence of a null cdr pointer, but by its length and tag. As pointed out above, the cells common to `y` and `z` will appear to `eq?` to have been copied, just as in the conventional version of `append`. Of course, if the final cell of `y` was in use, as would be the case if we tried the operation `(set! w (append y '(k l m)))` *after* forming `z` as above, or if `y` were an improper list, `append` would copy the cells of `y`, as does the conventional `append`.

The second use of `rplacd` we would like to emulate (non-destructively) is the removal of cells from the end of a list. It is easy to see that by merely subtracting from the length of a list, we delete cells from its end. Thus, in Figure 109 above, it could be that `x` is the result of performing `(firstn y 4)`, where `(firstn a b)` returns the first `b` cells of `a`. For example, `(firstn s (1- (length s)))` returns a list consisting of all but the last cell of `s` (and in constant time). As before, `s` is not affected and the conventional meaning of `eq?` is preserved.

Unfortunately, by composing `append`'s and `firstn`'s, it is possible to violate the usual meaning of `eq?`. For example, the expression

```
(eq? (firstn (append x y) (length x)) x)
```

can return true using Parcel's representation, but cannot return true using a conventional representation, if `x` is non-empty.

It should be clear that `rplacd` is difficult to perform using this representation, as it may affect the length of every list sharing a cell, and to update all such pointers is impractical. A more subtle problem exists with `rplaca`. Consider the result of `(set! w (append y y))`, where `y` is as shown in Figure 109. This will produce a list of length 14, with only 7 cells in its top level! See Fig 111. Now, the operation `(rplaca w 'oops)` would change *two* cells in the top level of `w`, and one in the top level of `y`, which is clearly not what would happen with a conventional representation (where only one cell in `w` would be altered, and none in `y`). In short, this representation is probably not suitable for use with the `rplac` operations.

3.8 Relation to Previous Work

The approaches taken by various investigators to the problem of parallelism in Lisp may be divided according to two orthogonal criteria: the

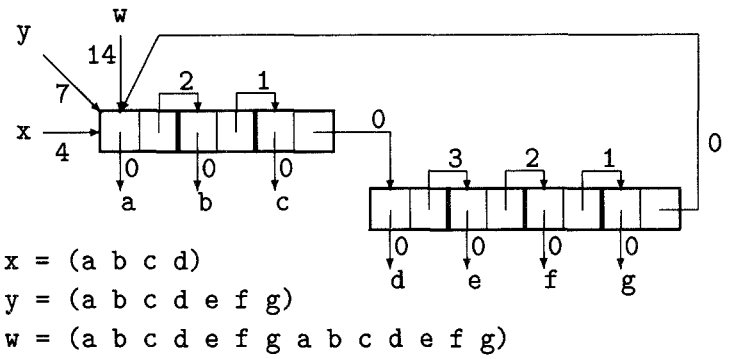

Figure 111: An Unusual Case of Sublist Sharing

mechanism(s) by which parallelism is exploited, and the degree to which parallelism is exploited automatically. In Multilisp [24, 38], a flexible mechanism called a `future` is used to permit the overlapping of the production of a quantity with its consumption. It might be, for example, that a function returns a quantity which will not be needed (in fully evaluated form), for some time following the function's return; by enclosing the expression whose value is to be returned in a `future`, the overlap may be turned into speedup. Another example of such a mechanism for explicit parallelism is the `qlambda` construct of Qlisp [22]. Using `qlambda`, one may spawn an asynchronous process which will run in the lexical environment in which the `qlambda` is closed. The `pcall` statement (described in both [22] and [24]) is similar in power to the `cobegin`-style parallelism discussed in subsection 3.3, but there in no form in Multilisp or Qlisp analogous to the `doall` construct of which Parcel makes such wide use, nor to the many recurrence solution routines that are part of its run-time system. Both `future` and `qlambda` are less rigidly structured than any of the constructs for parallelism employed by Parcel. On the one hand, this has an advantage in expressiveness: a `future` obeys the rules of *indefinite extent* that pertain to all Scheme objects, and thus fits neatly into the language, especially in the absence of side-effects. On the other hand, the modes of parallelism used in Parcel are machine-oriented, and are designed for efficiency in implementation. The formation of a `future` entails (in general) both closure formation and scheduling overhead, and introduces at most one additional, parallel strand of execution. The starting of a `doall` loop may result in many concurrent streams of execution, and (in the case of the machines for which Parcel is targeted) consumes but a few instructions of overhead. The α construct ("apply-to-all") of Connection Machine Lisp [7] may be seen

as a special case of a `doall` loop. CM Lisp's β operator has two variants. The first is similar to the `reduce` operator above (β requires both commutativity and associativity of the operator of reduction); the other variant of β has no analogue in Parcel (it is specific to the manipulation of *xector*s). SIMD machines execute such forms efficiently when conditional branching within the function being mapped (or the operator of reduction) is limited; ordinarily, this restricts one to the mapping of primitive functions, or simple user functions in which conditional execution is controlled by *mode vectors* (boolean vectors which "turn off" processors not participating in a computation).

Most work on parallelizing Lisp to date has left the job of identifying and exploiting parallelism to the user. All of the constructs described above, for example, come from dialects of Lisp which have been extended for the expression of parallelism. Some work has been done on the automatic insertion of forms such as `future` and `qlambda`; see [35] and [34]. These approaches leave the structure of the program relatively unaltered. In Curare [33], the problem of parallelizing tail-recursive functions which operate destructively upon list structures is addressed. As mentioned in subsection 2.14, Parcel's interprocedural analysis of object lifetimes and side-effects is of greater generality than methods based upon conventional alias analysis (such as that described in [33]), because while aliasing relations are subsumed by the Parcel analysis of side-effects, it is able to distinguish among instances of dynamically created objects, and to limit the visibility of side-effects according to the lifetimes of the objects involved. This is essential to the automatic extraction of high-level parallelism. Of course, much work has been done on program transformation and optimization; the approaches taken may be divided broadly into two categories. In [31] and [39] techniques for the automatic parallelization of Fortran programs are discussed; these operate upon a program represented as a control-flow or dependence graph, and may be seen as extensions of traditional techniques for program optimization. Such is the approach described in this paper, and in [33]. Another category of program transformation operates more directly upon the *syntax* of a program, and makes use of pattern matching in lieu of use-definition and control-flow information (i.e., in lieu of semantic analysis); for this reason, such techniques are applied primarily to functional (side-effect free) languages. See [20]. A mixed strategy for parallelism detection, namely a compiler which accepts a language that includes constructs for parallelism, is feasible as well; see [37]. The nondeterminism that results from the addition of annotations for parallelism may complicate the analysis of dependences sufficiently that sequential and parallelizing optimizations are severely inhibited. The interesting trade-offs of this interaction appear to be relatively unexplored.

Finally, the reader may wish to see [5] and [29] for work on the compilation of Scheme for sequential machines. Many Scheme compilers work by conversion of the input program to *continuation passing style.* This approach is not taken in Parcel, for the reason that many transformations performed in our compiler, including traditional optimizations such as invariant floating, global common subexpression elimination, and particularly parallelizing transformations, are most naturally expressed in terms of a conventional control flow model (i.e., a control flow graph), and are facilitated by increased *visibility* of the computation. Rather than coalescing units of control flow into larger, nested structures over which transformations can be applied, CPS conversion appears to cause fragmentation of the computation that limits the scope of optimization and restructuring. Indeed, great effort is spent in Parcel in merging procedure contours, so that the compiler can manipulate procedure bodies that are as large as possible, and as free from branching and procedure application as possible. On the other hand, many traditional optimizations have been recast into the CPS framework (see the above references), and it may be that the transformations we are performing could be so recast as well.

4 Preliminary Performance Results

We have constructed a code generator and parallel run-time system for the Alliant FX/8 [2], an 8-processor shared memory multiprocessor, in order to test the compilation strategy of the Parcel compiler. The run-time system consists of a parallel stop-and-copy garbage collector, a microtask scheduler, and a library of parallel recurrence solution routines, in addition to the usual (sequential) functionality of a Scheme run-time system, such as I/O. The table in Figure 112 lists the running time of an Alliant FX/8 under normal, daytime loading, executing a few of the Gabriel benchmarks [21] as compiled by Parcel. At the time of this writing, the run-time system is still under development, and isn't able to execute the entire Gabriel suite of benchmarks; in any event, a detailed study of the run-time behavior of the object codes produced by the Parcel compiler is called for, and is beyond the scope of this work. Nevertheless, the reader may compare these running times to those of commercially developed Lisp compilers for sequential machines, to appreciate the efficiency of the object codes produced by the Parcel compiler.

	Parcel - FX/8
boyer	1.80 + 0.00
dderiv	0.48 + 0.00
deriv	0.58 + 0.00
idiv2	0.28 + 0.00
rdiv2	0.25 + 0.00
tak	0.11 + 0.00

Figure 112: Preliminary Performance Figures for Parcel - CPU+GC Seconds

5 Conclusions

We have presented a comprehensive approach to the interprocedural analysis and automatic parallelization of Scheme programs. There are a number of conclusions to be drawn from this work.

First, we conclude that automatic parallelization can be profitably applied to languages other than Fortran. In fact, the simplicity and clarity of their semantics make Scheme programs ideal as input to a parallelizing compiler.

Second, we conclude that the heavy use of procedures by Scheme programmers, and in the implementation of the advanced features of the language, means that aggressive interprocedural analysis is essential to the successful optimization of Scheme programs for parallel and sequential execution. To answer this requirement, we have introduced *procedure strings* and *stack configurations* as a natural and powerful framework in which to reason about object lifetimes and interprocedural side-effects. Because it restricts the visibility of side-effects according to the lifetimes of mutable objects, the system of interprocedural analysis we have constructed is able to reveal high-level parallelism in programs that make use of side-effects. It is likewise well suited to problems of memory management, both the problem of placing objects on a stack where their lifetimes permit, and of placing objects in a hierarchical shared memory according to the visibility required of them. In fact, the generality of Scheme's semantics would allow us easily to use this framework for the analysis of object lifetimes and side-effects in, for example, C programs. In short, it can provide the theoretical basis for parallelization and memory management of programs that manipulate pointers and dynamically allocated storage, since these manipulations can be reasoned about in terms of Scheme's more general feature of first-class procedures. The framework also accommodates Scheme's first-class contin-

uations naturally; this gives hope that it will apply to other, usually more restricted mechanisms for causing interprocedural movement of control.

Third, we conclude that in parallelizing Scheme (or Lisp) programs a compiler must treat control structures more complex than the conventional `do` loop of Fortran. In particular, `while` and `repeat` structures (which may arise from tail-recursion) and recursion (other than tail-recursion) are rich sources of parallelism, but the extraction of this parallelism often requires extensive transformation of the program, as the examples we have presented demonstrate. The techniques of *exit-loop translation* and *recursion splitting* we have introduced are a natural extension of the techniques for parallelizing Fortran programs developed by Kuck and his colleagues, to the control structures found commonly in Scheme programs. In fact, when augmented with numerous "sequential" optimizations performed in Parcel, exit-loop translation and recursion splitting may be seen as the bridge over which Scheme programs must pass to be eligible for restructuring by the techniques that have been so well-developed for Fortran (or straightforward adaptations of those techniques). Like our framework of interprocedural analysis, exit-loop translation and recursion splitting are directly applicable to other languages that provide iterative structures and recursion.

Finally, we conclude that agressive "sequential" optimizations are important to the successful parallelization of Scheme programs, for two reasons. First, transformations which do not introduce parallelism on their own, may nonetheless facilitate parallelization by simplifiying code, eliminating spurious control and data dependences, and rearranging computations so that they are more "visible" to the compiler. Second, the use of several versions of procedures (one parallel, one sequential) is an effective means of balancing the opposing requirements of creating parallel activity, when the target machine is underutilized, and executing efficient sequential code in each processor, when the target machine is saturated with parallel activity. If the parallelized procedures produced by Parcel were not complemented by optimized sequential ones, the performance of its object codes would be unbalanced and awkward for the run-time system to manage, as the degree of parallelism would be grossly out of proportion to the target machine size, and the extensive restructuring performed in parallelizing a program would come back as sheer overhead during execution.

6 Acknowledgements

The author wishes to acknowledge David Padua for the many hours he gave to the creation, expression and correction of this work. The author also wishes to acknowledge Todd Allen, Michael Burke, Ronald Cytron, Perry

Emrath, Samuel Kamin, Clyde Kruskal, David Kuck, Tim McDaniel, Sam Midkiff, Uday Reddy, and David Sehr for their encouragement and their many insightful comments and suggestions.

References

1. *Butterfly Parallel Processor Overview.* BBN Laboratories Inc., Cambridge, Massachusetts (1985).

2. *FX/Series Architecture Manual.* Alliant Computer Systems Corporation, Acton, Massachusetts (January 1986).

3. Harrison III, Williams Ludwell. *Compiling Lisp for Evaluation on a Tightly Coupled Multiprocessor.* Technical Report 565, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign (March 1986).

4. Harrison III, Williams Ludwell and Padua, David A. Parcel: project for the automatic restructuring and concurrent evaluation of lisp. In *Proceedings of the 1988 International Conference on Supercomputing*, Association for Computing Machinery (July 1988).

5. Steele Jr., Guy L. *RABBIT: a Compiler for Scheme.* Technical Report AI Memo 474, Massachusetts Institute of Technology (May 1978).

6. Steele Jr., Guy L. *Common Lisp: the Language.* Digital Press (1984).

7. Steele Jr., Guy L. and Hillis, W. D. Connection machine lisp: fine-grained parallel symbolic processing. In *Proceedings of the 1986 Conference on Lisp and Functional Programming* (August 1986) 279–297.

8. Steele Jr., Guy L. and Sussman, Gerald Jay. *The Revised Report on Scheme.* Technical Report AI Memo 452, Massachusetts Institute of Technology (January 1978).

9. Abelson, Harold and Sussman, Gerald J. *Structure and Interpretation of Computer Programs. The MIT Electrical Engineering and Computer Science Series*, MIT Press, Cambridge, Massachusetts (1985).

10. Aho, Alfred V. and Ullman, Jeffrey D. *Principles of Compiler Design.* Addison Wesley Publishing Company, Reading, Massachusetts (1979).

11. Allison, Lloyd. *A Practical Introduction to Denotational Semantics. Cambridge Computer Science Texts 23*, Cambridge University Press, Cambridge (1986).

12. Banerjee, Uptal D. *Data Dependence in Ordinary Programs.* Master's thesis, University of Illinois at Urbana-Champaign (November 1976).

13. Banerjee, Uptal D. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign (October 1979).

14. Burke, Michael. *An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data Flow Analysis.* Technical Report RC 12702 (#58665), IBM T.J. Watson Research Center (September 1987).

15. Burn, G. L. *Abstract Interpretation and the Parallel Evaluation of Functional Languages.* PhD thesis, Imperial College, University of London (march 1987).

16. Church, Alonzo. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, New Jersey (1941).

17. Cousot, P. and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages* (January 1977) 238–252.

18. Cousot, P. and Cousot, R. Systematic design of program analysis frameworks. In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages* (January 1979) 269–282.

19. Cytron, Ronald G. and Ferrante, Jeanne. What's in a name? or the value of renaming for parallelism detection and storage management. In *Proceedings of the 1987 International Conference on Parallel Processing* (August 1987) 19–27.

20. Darlington, John and Burstall, Richard M. A system which automatically improves programs. *Acta Informatica*, 6, 41 (1976).

21. Gabriel, Richard P. *Performance and Evaluation of Lisp Systems.* MIT Press, Cambridge, Massachusetts (1985).

22. Gabriel, Richard P. and McCarthy, John. Queue-based multiprocessing lisp. In *Proceedings of the 1984 Conference on Lisp and Functional Programming* (January 1984) 25–44.

23. Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A. *FX-87 Reference Manual.* Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology (January 1987).

24. Halstead, Robert H. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7, 4 (October 1985) 501–538.

25. Hansen, W. J. Compact list representation: definition, garbage collection and system implementation. *Communications of the ACM*, 12, 9 (September 1969).

26. Hecht, M. S. *Flow Analysis of Computer Programs.* Elsevier North-Holland (1977).

27. Hillis, W. Daniel. *The Connection Machine.* MIT Press, Cambridge, Massachusetts (1985).

28. Hudak, Paul and Young, Jonathan. A collecting interpretation of expressions (without powerdomains). In *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages* (January 1988).

29. Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. Orbit: an optimizing compiler for scheme. In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction* (July 1986) 162–175.

30. Kuck, David J., Davidson, Edward S., Lawrie, Duncan H., and Sameh, Ahmed H. Supercomputing today and the cedar approach. *Science*, 231 (February 1986) 967–974.

31. Kuck, David J., Kuhn, Robert H., Leasure, Bruce, and Wolfe, Michael J. The structure of an advanced vectorized for pipelined processors. In *Fourth International Computer Softward and Applications Conference* (October 1980).

32. Ladner, R. E. and Fischer, M. J. Parallel prefix computation. *Journal of the ACM* (October 1980) 831–838.

33. Larus, J. and Hilfinger, P. N. Restructuring lisp programs for concurrent execution (summary). In *Conference Record of the ACM SIGPLAN Symposium on Parallel Programming* (1988).

34. Marti, J. and Fitch, J. The bath concurrent lisp machine. In *EUROCAM '83 (Lecture Notes in Computer Science)*, Springer Verlag (1983).

35. McGehearty, P. F. and Krall, E. J. Potentials for parallel execution of common lisp programs. In *Proceedings of the 1986 International Conference on Parallel Processing* (1986) 696–702.

36. Midkiff, Samuel P. and Padua, David A. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36, 12 (December 1987) 1485–1495.

37. Midkiff, Samuel P. and Padua, David A. *The Further Concurrentization of Parallel Programs.* Technical Report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign (1988).

38. Miller, James Slocum. *MultiScheme: A Parallel Processing System Based on MIT Scheme.* PhD thesis, Massachusetts Institute of Technology (1987).

39. Padua, David A. and Wolfe, Michael J. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29, 12 (December 1986).

40. Pfister, G. F., Brantley, D. A., *et al.* The ibm research parallel processor prototype (rp3). In *Proceedings of the 1985 International Conference on Parallel Processing* (1985) 764–771.

41. Rees, J., Clinger, W., *et al.* Revised revised revised report on the algorithmic language scheme. *SIGPLAN Notices*, 21, 12 (December 1986) 37–76.

42. Roads, C. B. *3600 Technical Summary.* Symbolics Corporation, Cambridge, Massachusetts (February 1983).

43. Scott, Dana S. *Domains for Denotational Semantics.* Technical Report, Carnegie-Mellon University (June 1982).

44. Stoy, J. E. *Denotational Semantics: the Schott-Strachey Approach to Programming Language Theory.* MIT Press (1977).

45. Triolet, Remi. *Contributions to Automatic Parallelization of Fortran Programs with Procedure Calls.* PhD thesis, University of Paris VI (I.P.) (1984).

46. Wegman, Mark and Zadeck, Kenneth. Constant propagation with conditional branches. In *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages* (January 1985) 291–299.

47. Wolfe, Michael J. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign (October 1982).

7 Vita

Williams Ludwell Harrison III was born on June 2, 1960 in Lafayette, Indiana. He graduated from Wheaton Central High School in 1978, and from the University of Illinois at Urbana-Champaign in 1983, obtaining a Bachelor of Arts in English Literature and Political Science. He entered the graduate school of the University of Illinois at Urbana-Champaign in the Fall of 1983. His doctoral research was directed and supported by Professor David A. Padua. He is now with the Center for Supercomputing Research and Development of the University of Illinois at Urbana-Champaign as a Senior Software Engineer, and the Department of Computer Science as an Adjunct Assistant Professor.