Dynamic Language Parallelization

Lorenz F. Huelsbergen

Technical Report #1178

September, 1993

DYNAMIC LANGUAGE PARALLELIZATION

By
Lorenz F. Huelsbergen

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN – MADISON
1993

DYNAMIC LANGUAGE PARALLELIZATION

Lorenz F. Huelsbergen, Ph.D.
University of Wisconsin–Madison 1993

Dynamic language parallelization is a new method, for the automatic parallelization of imperative programs, that finds parallelism *during* program execution. Dynamic parallelization uncovers more parallelism—and better selects useful parallelism—than is statically possible at compile time. It requires only inexpensive compile-time analyses, allows separate compilation, and admits interactive programming environments.

This thesis describes the design and implementation of the first dynamic parallelization techniques for imperative higher-order languages such as ML, Scheme, and Lisp. Prototype implementations, in an optimizing ML compiler on a shared-memory parallel computer, confirm the thesis that dynamic language parallelization is feasible, inexpensive, and often effective. The dynamic techniques address parallelization in the presence of four language attributes that inhibit static parallelization: imperative higher-order functions, side effects to dynamic structures, expressions with variable amounts of computation, and automatic storage reclamation.

λ-*Tagging* dynamically propagates information about a function's side effects with the function's physical run-time representation. A λ-tagging compiler can insert checks to λ-tags that select parallel evaluation only when λ-tag side-effect information indicates that parallel evaluation is safe.

Dynamic resolution determines at run time when updates to a dynamic data structure may safely occur in parallel. It dynamically detects shared data, and correctly coordinates access to this data at run time. Dynamic resolution can automatically parallelize some non-trivial functions that elude static parallelization (*e.g*, a destructive list-based sort).

Dynamic granularity estimation maintains size approximations on dynamic data structures (*e.g.*, lists) at run time. Dynamically, the program consults these approximations to decide when parallel evaluation of an expression will always speed the program's execution. A compiler can statically identify expressions whose evaluation cost always depends on structure sizes, and can insert checks to data sizes that select parallel evaluation when beneficial.

A *concurrent garbage collector* reclaims a program's spent storage in parallel with the program's computation proper. The thesis describes the design and implementation of the first concurrent copying collector that does not require special hardware or operating systems support. The collector relies on the language or compiler to identify all program accesses to mutable data. Measurements of the collector's implementation indicate that it removes all perceptible garbage-collection pauses from a program's execution.

To my mother and father

Acknowledgements

Foremost, I thank my advisor. Jim Larus provided the insight, encouragement, and patience without which this thesis would not exist.

The other members of the thesis committee—Charles Fischer, Tom Reps, Susan Horwitz and Arnold Johnson—challenged, and improved, the underpinnings of dynamic language parallelization. My further research in this area will certainly embody their ideas. Readers Charles Fischer, Tom Reps, John Reppy, and Phil Pfeiffer greatly improved the content, exposition, and style.

Todd Proebsting unwittingly taught me new ways of approaching and solving problems. Tom Ball took the time to carefully read and critique the initial drafts and the conference submissions that are now the technical core of the thesis. Discussions with programming-languages students Paul Adams, Sam Bates, Satish Chandra, Doug Hahn, Steve Kurlander, and G. Ramalingam spurred many an idea or improvement.

Andrew Appel, Greg Morrisett, John Reppy, and David Tarditi provided assistance with the New Jersey implementation of Standard ML. Sarita Adve and Jeff Hollingsworth answered my computer architecture and operating systems questions, respectively.

For two months in the Spring of 1993, the FL language group at IBM Almaden—John Williams, Alex Aiken, Ed Wimmers, and TK Lakshman—introduced me to another language and its inner workings. During this period, Alex Aiken suggested an operational approach to counting program time steps; this considerably simplified the development in Chapter 5.

I would like to thank my teachers at Grinnell College and at Wisconsin. In particular, Anita Solow, John Stone, Gene Herman, Emily and Tom Moore, Eric Bach, and Charles Fischer imparted wisdom upon me.

I gratefully acknowledge the support of the NSF (Grant CCR-9101035), the University of Wisconsin (Graduate School Grant), and DARPA (Fellowship in Parallel Processing).

Finally, I thank my family: My parents instilled in me the value of an education and gave me the tools and means to attain it; my brothers were (and are) a constant source of perspective and encouragement.

Contents

Chapter 1

Introduction

A program's text is a static description of dynamic computations. Automatic program parallelization performed at compile time is fundamentally limited—in order to preserve a program's semantics, compilers must statically compute a conservative approximation of the program's dynamic behavior. *Dynamic language parallelization* combines static program analysis with dynamic information about the program's actual computation. Parallelization decisions are made on the fly. Dynamic parallelization uncovers more parallelism—and better selects useful parallelism—than is statically possible. It requires only inexpensive compile-time analyses, allows separate compilation of program modules, and admits interactive programming environments.

The design of dynamic parallelization techniques for imperative languages and the implementation of these techniques in an optimizing compiler for a parallel machine, confirm the thesis that dynamic language parallelization is feasible, inexpensive, and effective.

1.1 Dynamic Language Parallelization

This work studies the automatic parallelization, at run-time, of imperative dynamic languages. *Dynamic languages* are general-purpose; their programs construct and modify dynamic data structures, manipulate functions as values, perform general I/O, and automatically manage their storage. This thesis develops dynamic parallelization for the ML language [78, 79], but the underlying concepts also apply to other languages with dynamic features (*e.g.*, to languages in the Algol [84], Lisp [76], Prolog [22], and Simula [17] families).

The next sections describe the new parallelization techniques, methods for measuring their efficacy, and a rationale for the implicit parallelization of imperative languages.

1.1.1 Dynamic Parallelization Techniques

Dynamic language parallelization comprises a family of run-time *parallelization techniques*. A dynamic parallelization technique is a *hybrid*: a *static component* computes inexpensive, but partial, information about the program at compile time; a *dynamic component* gathers basic information about the program at run time and augments the precomputed static information with this dynamic information. Using this

combined information, the run-time system makes semantically-correct (*safe*) parallelization decisions during the program's execution that preserve the program's meaning. Figure 1.1 depicts the static-dynamic components in a conventional compiler and run-time system.

Dynamic language parallelization supplements—not supplants—existing static analyses. In doing so, it not only improves parallelization, but also reduces the cost of static analyses since only partial information about the program's dynamic behavior need be computed at compile time. Dynamic methods stand to substantially improve implicit parallelization because they have complete access to the program's state. Techniques that are entirely static must base their parallelization decisions on conservative, hence imprecise, approximations to this state. Dynamic techniques need not rely on crude static estimates, but may rather defer decisions until run time when more precise estimates exist. Precise approximations enable precise parallelization decisions; these, in turn, yield more parallelism and can better select parallelism for a specific machine.

Utilizing run-time information is not free—maintaining and manipulating dynamic information introduces overheads into the program's execution. For the dynamic techniques of this thesis, however, empirical evidence suggests that the speed improvement resulting from the additional (dynamic) parallelism can offset their run-time overhead.

I have designed and implemented new dynamic techniques that address four problems facing automatic language parallelization: analyzing imperative higher-order functions, allowing concurrent[1] updates to dynamic data structures, deducing expression granularities, and reclaiming storage concurrently. The implementations of the dynamic techniques are prototypes; though amenable to full compiler automation, they focus on the efficient implementation of a technique's dynamic component since this component governs the technique's effectiveness.

Imperative Higher-Order Functions

λ-tagging (Chapter 3) is a new technique that dynamically identifies and parallelizes an imperative program's *functional* subcomputations—computations that do not produce side effects. λ-tagging annotates a function's run-time representation with a tag describing the imperative operations that the function may perform. Examining a λ-tag at run time selects parallel evaluation when it is safe to do so. λ-tagging overcomes the difficulty of statically tracking functional values—often created dynamically—through complex data structures and unpredictable control flow.

Updates to Dynamic Data

Dynamic resolution (Chapter 4) is a new technique that detects when concurrent modification of a data structure is possible. Static techniques can only imprecisely approximate the structure of a program's dynamic data. Precise characterization of this structure is, however, critical to effective parallelization. Dynamic resolution directly consults a program's dynamic data structures in order to decide when concurrent structure updates are safe.

[1]The terms "parallel" and "concurrent" are used synonymously in this thesis.

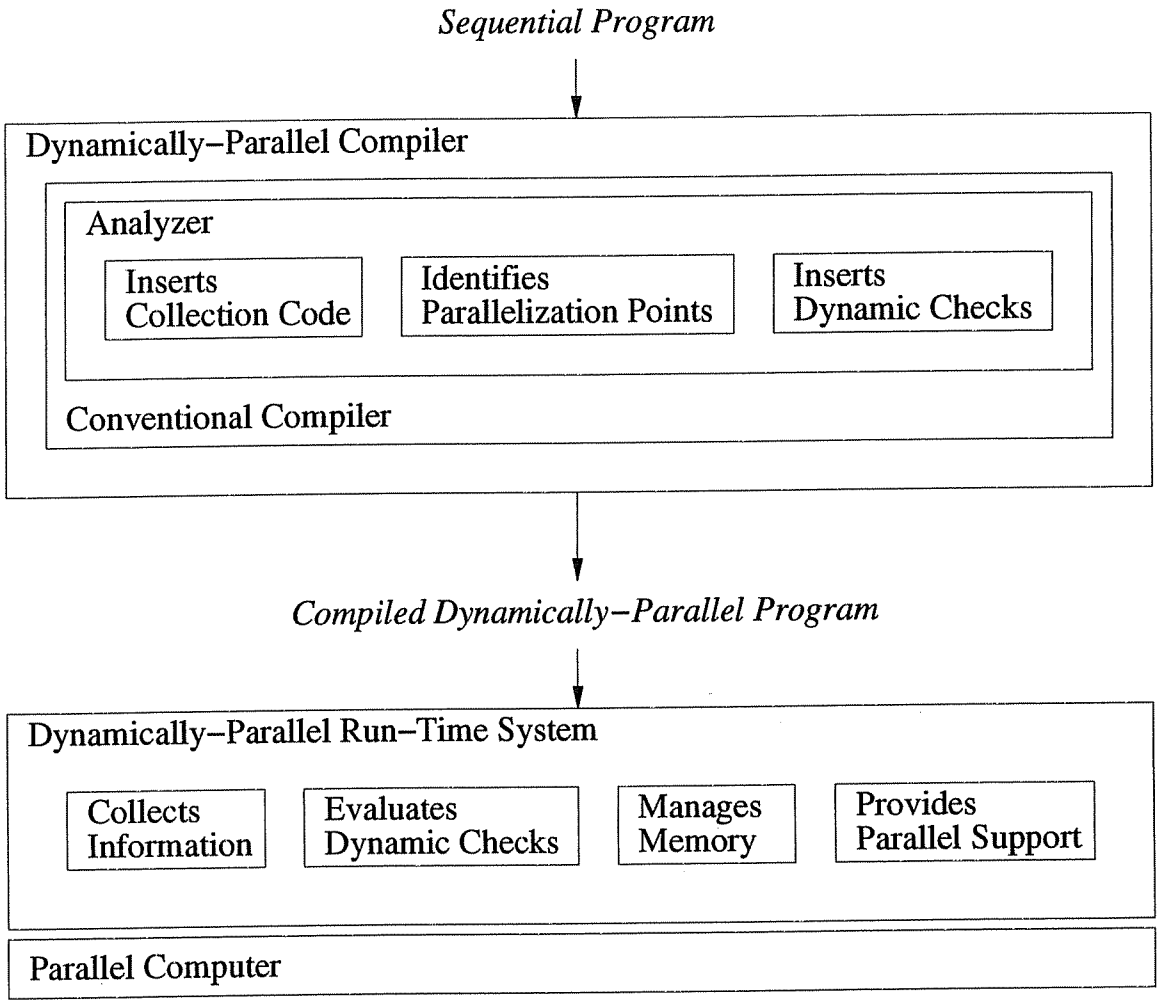


Figure 1.1: A dynamically-parallel compiler is a conventional sequential compiler with additional static analyses that compute partial information about the program. The dynamically-parallel run-time system gathers dynamic information that augments these (partial) analyses. The static component of a dynamic parallelization technique consists of compiler analyses that identify program points at which dynamic information can guide parallelization. Checks on the dynamic information are inserted at these points at compile time. The dynamic component in the run-time system collects information *per* the static component's instruction. Checks to combined static-dynamic information select parallel evaluation only when it is safe. Since the program's state is mostly memory, the dynamic component may also manage memory.

Expression Granularity

Dynamic granularity estimation (Chapter 5) is a new technique that matches a program's parallelism to the underlying machine's granularity at run time. Since computation in dynamic languages builds data structures of variable size, the time complexities of the program's functions often hinge on the size of the data to which they are applied. Dynamic granularity estimation maintains a bounded estimate of a datum's size at run time. Statically, this technique identifies functions whose application costs vary with their parameter's data sizes. Dynamic examination of the run-time size estimates then selects parallel evaluation only when beneficial.

Concurrent Storage Reclamation

A new *concurrent garbage collector*[2] (Chapter 6) reclaims a sequential program's discarded storage in parallel with the program's computation proper. This collector is the first concurrent *copying* collector that does not need special hardware or operating system support. It does, however, require static support (language or compiler) for efficiency, I describe the design and implementation of this collector as well as a design extension that can reclaim the storage of parallel programs.

1.1.2 Measuring Dynamic Parallelization

Dynamic parallelization is more powerful than static parallelization. A demonstration of this does not, however, require direct comparison of a dynamic technique to its static counterpart(s). This is fortunate since only few static approaches have been implemented, and these often require experimental systems (*e.g.*, [68]).

I measure dynamic-parallelization's efficacy with two (new) criteria: *necessity* and *efficiency*. A dynamic parallelization technique, $\mathcal{D}$, is *necessary* when programs exist for which $\mathcal{D}$ finds parallelism and *no* static technique finds this parallelism. Necessity entails all existing *and* future static parallelization techniques. To show that $\mathcal{D}$ is necessary, it suffices to find a program P that $\mathcal{D}$ can parallelize, but for which static parallelization is impossible.[3] Necessity arguments are qualitatively strong—they are independent of the underlying systems.

Efficiency measures the performance of a dynamic technique $\mathcal{D}$ relative to other (static) parallelization techniques. Foremost, I compare the sequential execution time of a program P to the execution time of P dynamically parallelized with $\mathcal{D}$. For $\mathcal{D}$ to be effective, $\mathcal{D}$ must—given a (small) number of processors—improve P's execution time relative to its sequential execution time. Additionally, I measure $\mathcal{D}$'s efficiency relative to an explicitly parallel version of the program.[4] This latter measure provides an indication of $\mathcal{D}$'s performance relative to static parallelization. Efficiency measures are quantitative; they are particular to a specific program, input data set, compiler, and computer.

---

[2] Garbage collection it is not a new dynamic technique *per se*; most conventional garbage collectors, sequential and concurrent, are entirely dynamic.

[3] Such a P usually contains a conditional C with a predicate whose value is unknown at compile time. To ensure correctness, a static technique must assume that either branch of C is dynamically taken. A dynamic technique, however, can detect—for individual instances of C—which branch is actually taken.

[4] Explicitly parallel versions were coded manually and utilize no dynamic information. Since in general $\mathcal{D}$ may be necessary, an explicitly parallel program is always supplied data for which its operation is known *a priori* to be correct.

1.2 Rationale

In this section, I provide rationale for pursuing the implicit (automatic) parallelization of imperative dynamic languages. The argument is in the context of dynamic languages.

1.2.1 Imperative versus Functional Languages

Functional languages (*e.g.*, Pure Lisp [76], FP [13], Haskell [52]) forbid direct programmer manipulation of state. The resulting absence of side effects enables equational reasoning about functional programs [13]. It also readily exposes parallelism; expressions whose subexpressions are evaluated may safely evaluate in parallel. Some common algorithms, however, are difficult to express without mutable state [93]. More severely, functional implementations of many algorithms are often inefficient.

Although parallelism is easily extracted from functional languages, the task of selecting expressions with granularities suitable for parallel evaluation remains [40]. Furthermore, functional languages assume an infinite space in which to place fresh values. Since machines are finite, these languages rely on automatic storage reclamation to recycle spent storage. Garbage collection must not become a bottleneck in a parallel system. This work addresses the issues of identifying expressions with suitable granularities and of garbage collection in parallel language implementation.

Imperative languages (*e.g.*, Scheme [95], ML [78, 79]) routinely allow modification of the program's state through assignment operators.[5] This makes algorithm specification expressive and efficient. Shared mutable state in parallel systems also provides a convenient mechanism for interprocessor communication. However, a language's gain in imperative expressiveness complicates reasoning about its programs. Nevertheless, I adopt the view that access to mutable state is necessary to utilize realizable computers efficiently and that assignment is characteristic of efficient general-purpose languages (*cf.* [31]).

In the presence of side effects, parallel evaluation of imperative expressions may produce indeterminate results because of uncoordinated concurrent access of mutable data. Coordinating correct parallel access to common mutable data is a critical problem facing the parallelization of imperative languages. The techniques of this thesis directly approach this goal.

1.2.2 Implicit versus Explicit Parallelism

Parallel language constructs (*e.g.*, threads, synchronization mechanisms, and communication channels) allow programmers to introduce *explicit parallelism* into a program. Explicit parallelization speeds programs. However, explicit parallelization comes at great difficulty and cost. In addition to programming a solution to the problem at hand, the programmer is now also responsible for explicitly specifying its parallel solution. Furthermore, explicitly parallel programs often exploit idiosyncrasies of the target architecture. Such programs are not portable; nor are they simple to maintain. Finally, the use of explicit constructs must avoid common parallelization pitfalls (*e.g.*, race conditions, dead or live lock), lest indeterminate behavior ensue.

[5] Functional programs can simulate "state" by rebuilding a data structure to incorporate "updates" (*cf.* [59]). However, this strategy restricts parallelism since it must sequentially thread all "mutable" data structures through the computation.

Implicit parallelism is parallelism that is extracted automatically from the program by a compiler.[6] Implicit parallelism is desirable because the language implementation—not the programmer—matches the program's parallelism to the target machine. With multiple, machine-specific compilers for a language, programs are potentially portable across a spectrum of uni- and multi-processors. An implicitly parallel language implementation preserves the language's semantics and sidesteps programmer parallelization errors entirely. As this work shows, implicit parallelization also reveals parallelism whose explicit demarcation is cumbersome and subtle.

Note that implicit parallelization does not imply the automatic parallelization of inherently sequential algorithms or programs. Programs must contain parallelism. A recursive divide-and-conquer programming style, if consistently applied, exposes abundant parallelism (*cf.* [83])—yet this programming style does not require the programmer to reason about concurrent evaluation. For a program written in this style, the language implementation should identify and coordinate as much useful parallelism as possible.

1.3 Thesis Overview

Chapter 2 describes the language under consideration for dynamic parallelization, the compiler that implements it, and the target parallel machine. Beyond description of the systems, this chapter also identifies sources of parallelism in the language, characterizes the side effects of the language's imperative features, and defines the kinds of conflicts that parallel evaluation of these imperative features can cause.

Chapters 3–6 constitute the core of the thesis. Each core chapter describes one of the four dynamic techniques (§1.1.1 above). Since the individual techniques are orthogonal, these chapters are—for the most part—independent.

Chapter 7 concludes with a summary of the new techniques and provides a perspective view of dynamic language parallelization.

1.4 Notes

This section examines previous work on run-time parallelization and various parallel systems for dynamic languages. Subsequent chapters supply further comparison to related work.

Run-Time Parallelization

Dynamic language parallelization is in its infancy. Run-time parallelization [72, 101, 102, 92, 123] exists for static array-based languages (*e.g.*, Fortran).[7] These methods parallelize loop nests containing indirect (and hence statically unknown) array references by dynamically *pre-executing* a loop to find a parallel schedule for its iterations.[8] A good parallel schedule absorbs the cost of pre-execution. Lu and Chen [73, 72] extend these array techniques to loops with simple pointer calculations. However,

[6] In the case of dynamic language parallelization, the compiler *and* run-time system cooperatively find implicit parallelism.

[7] [63] surveys run-time techniques that optimize sequential programs.

[8] For regular access to direct array indicies, static methods of scheduling parallel loop bodies have met with success (*e.g.*, [66, 4, 3, 121]). Not surprisingly, static parallelization performs well for programs with statically-predictable behavior. I do not study array parallelism in this thesis.

none of these techniques is general: procedure invocation is not supported; indirect array indicies and data structures are constrained to remain fixed during the loop's execution. Furthermore, compiler automation of these techniques requires extensive static analysis.

Dynamic language parallelization, as proposed here, *incrementally* collects information about a program's dynamic data and computation structure. During parallel evaluation, the costs of this collection are distributed. Available run-time information guides decisions at parallelization points; complex evaluation schedules are unnecessary.

Implicitly Parallel Systems

Harrison's PARCEL [45, 44] and Larus's Curare [68, 67] statically transform sequential Scheme [95] programs for parallel execution. Both compilers perform an interprocedural side-effect analysis. They do not use run-time information. As is characteristic of static analyses, PARCEL and Curare compute a conservative semantics-preserving estimate of a program's dynamic behavior. Therefore, they usually only find small amounts of parallelism in the presence of modifications to large dynamic structures. In contrast, dynamic techniques construct and consult sharp approximations to a program's run-time behavior and can consequently find more parallelism. PARCEL's and Curare's analyses are over entire program texts; their time complexities (*i.e.*, compile-time costs) hinge on a program's size. This expense hinders interactive development [67]. The dynamic techniques of this thesis, on the other hand, only require inexpensive static analysis—analysis that is often local to a function's definition. That is, all interprocedural information is obtained at run time.

Katz's ParaTran [62, 111] models a Scheme program as a parallel database. Access to a value is a *transaction* under this model. Parallel evaluation proceeds optimistically; upon detection of a transaction (access) conflict, *rollback* restores the computation to a stable state. Examination of *timestamps* on values detects conflicts dynamically. A transaction t commits when timestamps indicate that further transactions will not conflict t. ParaTran's rollback and timestamp mechanisms introduce large overheads into a program's execution. It is unclear whether Scheme programs contain enough conflict-free parallelism to offset these costs. ParaTran relies on extensive static analysis and has not been implemented on a parallel machine.

Gray [40] and Boyle *et al.* [18] automatically parallelize Pure Lisp [76]. Pure Lisp is entirely side-effect free. This simplifies implicit parallelization since shared mutable state does not exist. However, the parallel programs produced by these approaches create large numbers of small threads; scheduling overheads plague these approaches. The difficulty lies with static analyses that cannot, with enough precision, determine how much computation an expression contains. As this thesis demonstrates (Chapter 5), incorporation of dynamic information partially alleviates this problem.

Functional dataflow languages (*e.g.*, Id [87, 12]) automatically evaluate in parallel on dataflow machines [27]. Id's extension to concurrent M-structures [15] provides mutable state for the efficient expression of certain algorithms, but renders Id imperative and indeterminate. Automatic techniques for the determinate use of state in Id do not yet exist.

Explicitly Parallel Systems

Reppy's Concurrent ML (CML) extends ML with explicit synchronous communication [96, 97]. In CML, programmers construct powerful higher-order synchronization and communication abstractions from message-passing operations based on CSP [50]. Other research also introduces explicit communication into ML (*e.g.*, [94]) and explores the semantic issues that these mechanisms entail [16]. Cooper *et al.* designed a portable platform, called MP [82, 24], for creating explicitly parallel ML programs and have ported it to several parallel machines. MP provides thread creation, parallel memory allocation, and synchronization facilities (see 2.4). Similar packages exist for many languages (*e.g.*, C [23], Scheme [57]).

Halstead's MultiLisp [42] was the first parallel Lisp. It extends Lisp with explicit language constructs: `pcall` evaluates the arguments in a function application in parallel; (`future` *e*) immediately returns a placeholder for expression *e* and begins *e*'s concurrent evaluation—when the value of an undetermined future is required, evaluation suspends until the future's computation completes. Mul-T [65] and Multi-Scheme [77] are derivative systems that emphasize performance. Concurrent garbage collection [42, 43] and efficient thread creation [81, 80, 40] are also studied in these systems.

Qlisp [36, 39] is another early Lisp with parallel language constructs. It too burdens the programmer with the tasks of identifying suitable parallelism and, in the presence of side effects, of reasoning about determinate behavior.

Chapter 2

Language, Compiler, and Machine

This chapter describes the language under consideration for dynamic parallelization, the compiler and run-time systems that implement it, and the target parallel machine. It also gives the method used for the experiments.

2.1 Notation

If A and B are sets, then $A \cup B$ is their union, $A \cap B$ is their intersection, and $A \backslash B$ is their difference. The empty set is denoted by $\emptyset$, and $\mathrm{Fin}(A)$ denotes the set of finite subsets of A. If f is a map, then the domain and range of f are $\mathrm{Dom}(f)$ and $\mathrm{Rng}(f)$. A finite map from A to B is a partial map with finite domain. Denote the set of finite maps from A to B as

$$A \xrightarrow{fin} B$$

where any $f \in A \xrightarrow{fin} B$ can be written in the form:

$$\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$$

The empty map is written $\{\}$. If f and g are maps, then $f \pm g$ is the map with f modified by g and has the domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and the values:

$$(f \pm g)(a) = \begin{cases} g(a) & \text{if } a \in \mathrm{Dom}(g) \\ f(a) & \text{otherwise} \end{cases}$$

A *sequent* of the form $A \vdash \mathit{phrase} \rightarrow B$ holds, with respect to A, if $\mathit{phrase} \rightarrow B$ where $\rightarrow$ is some ternary relation between A, *phrase*, and B. An inference rule has the form

$$\frac{P_1 \cdots P_n}{C}$$

where $n > 0$. Successful inference of the premises, P_i, infers the conclusion C. The premises are either sequents or mathematical side conditions.

2.2 λ_v-S Language

λ_v-S [122] is the call-by-value λ-calculus, λ_v [91, 98], extended with operators on mutable state. λ_v-S is an imperative dynamic language; it forms the core of Standard ML (SML) [78, 79]. Here, I present the base syntax and relevant semantics of λ_v-S. I then extend it with ML syntax. The individual parallelization techniques of the following chapters further extend or restrict the λ_v-S language as necessary.

2.2.1 λ_v-S Syntax

Exposition of the λ_v-S syntax follows that of Felleisen and Friedman [32]. The ground terms of λ_v-S are variables and constants:

x	$\in$	VAR	variables
b	$\in$	CONST = BCONST $\cup$ FCONST	constants
		BCONST = {`()`,`true`,`false`,`0`,`1`,...}	base constants[1]
		FCONST = {`+`,`-`,`not`,...}	function constants

Terms are expressions ($e \in$ EXP)

e	::=	v	value
	|	$e\ e$	application
	|	`let` $x = e$ `in` e	let
	|	`if` e `then` e `else` e	if
	|	`ref` e	allocate
	|	`set` $e\ e$	store
	|	`get` e	fetch

and values ($v \in$ VAL $\subset$ EXP):

v	::=	b	constant
	|	x	variable
	|	$\lambda x.e$	λ-abstraction

Although application and λ-abstraction derive the `let` and `if` terms directly, they are included here as terms for the following reasons. Inclusion of the `let` term admits programmer definition of polymorphic functions [26]. This is subsequently used (§3) to automatically infer a term's result type and its possible side effects. Since conditionals introduce imprecision into analyses, I explicitly include the `if` term to expose its analytical consequences.

Operators on mutable state (the imperatives: `ref`, `set`, and `get`) are also included as expression terms since their treatment is central to the development of dynamic parallelization. However, they are also viewed as function constants when this simplifies the exposition.

Variable x is *free* in term e if x occurs in e and e does not contain a λ-abstraction or a `let` term that *binds* x. Otherwise, when a λ-abstraction or a `let` term in e binds x, variable x is *bound* in e. The free

[1] In addition to the `int` and `bool` base constants, the base constant `()` denotes the only value of the `unit` type.

variables in term e, FV(e), are defined inductively by:

$$\begin{aligned}
\mathrm{FV}(b) &= \emptyset \\
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(\lambda x.e) &= \mathrm{FV}(e) \backslash \{x\} \\
\mathrm{FV}(e_1\ e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(\texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2) &= \mathrm{FV}(e_1) \cup (\mathrm{FV}(e_2) \backslash \{x\}) \\
\mathrm{FV}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \cup \mathrm{FV}(e_3) \\
\mathrm{FV}(\texttt{ref}\ e) &= \mathrm{FV}(e) \\
\mathrm{FV}(\texttt{set}\ e_1\ e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(\texttt{get}\ e) &= \mathrm{FV}(e)
\end{aligned}$$

I assume that the program's variables bound in λ-abstractions and `let` terms have been uniquely renamed. This avoids the unintentional capture of free variables. The textual replacement of term e for variable x in term e' is denoted as $e'[e/x]$.

2.2.2 λ_v-S Semantics

Formal semantics for λ_v-S can be found in [122, 112]. Two aspects of the semantics are particularly relevant to parallelization: the operation of the imperative forms and the order of expression evaluation.

The notation $e \rightarrow v$ denotes the evaluation of expression e to a value v.

Evaluation Order

Call-by-value λ_v-S is an eager language. Value terms are *irreducible* and evaluate to themselves. Evaluation of expression terms proceeds from left to right. Let $e_i \rightarrow v_i$. In an application term, (e_1 e_2), expression e_1 evaluates before e_2. After v_2 is obtained, the functional v_1 is applied to v_2. The imperative expression terms similarly evaluate their arguments from left to right. The conditional

(`if` e_1 `then` e_2 `else` e_3)

is *non-strict*; that is, it does not evaluate all of its subexpressions. The boolean value *true* for v_1 selects evaluation of e_2 and returns v_2; *false* evaluates e_3 and returns v_3. A `let` term, (`let` x `=` e_1 `in` e_2), first produces v_1 and binds it to x before evaluating e_2 and returning v_2.

Imperatives and Conflicts

The imperative expression terms create (`ref`), update (`set`), and retrieve (`get`) the contents of *reference values*. A reference value is a mutable value whose contents is a λ_v-S value. The imperatives operate as follows (where $e_i \rightarrow v_i$):

(`ref` e_1) creates a new reference value r and initializes r to v_1. The result of a `ref` expression is the reference value r.

(`set` e_1 e_2) changes the contents of reference value v_1 to v_2. The result of a `set` expression is always the unit constant `()`.

(`get` e_1) fetches and returns the contents of reference value v_1.

The expression `(set x (get y))`, for example, fetches the contents v of the reference value bound to `y` and stores v in the reference value bound to `x`.

The notion of *conflict* captures the possibility of imperative interactions between expressions. Expressions e_1 and e_2 *conflict* if either e_1 or e_2 `sets` *some* reference value and the other accesses (`sets` or `gets`) *some* reference value.[2] Since actual evaluation of e_1 and e_2 may cause no imperative interactions—if, for example, e_1 and e_2 access disjoint sets of reference values—this definition of conflict captures the *possibility* of such interactions.

Conflicts do not lead to indeterminate behavior under sequential left-to-right evaluation. However, if expressions e_1 and e_2 conflict *and* access a common reference value r, naive parallel evaluation of e_1 and e_2 may produce indeterminate behavior because multiple access orders involving r are possible. For example, naive parallel evaluation of `(set x 0)` and `(set x 1)` yields either the value `0` *or* the value `1` for `x` (assuming atomic stores to reference values).

2.2.3 Extended λ_v-S

This section extends the syntax of λ_v-S to a (large) subset of ML. Syntactic additions include multiple `let` bindings, function definitions, recursive datatypes, patterns, and control structures. Since ML statically *infers* the types of a program's expressions (*cf.* §3.1), I first describe type notation and the implications of statically typing λ_v-S. I then compare extended λ_v-S to ML.

Static Type Inference

Algorithms to statically infer the type of λ_v-S expressions exist (*e.g.*, [26]). A λ_v-S program P is *well-typed* if a static type for P exists. A well-typed program cannot fault during execution due to a type inconsistency. In this thesis, I only consider λ_v-S programs that are well-typed.

The notation $e : \tau$ denotes that expression e has type τ. For example, `5 : int` and `(not x) : bool`. A λ-abstraction is a function and maps a value of type τ_1 to a value of type τ_2. It has a *function type*

[2] Concurrent allocation of new reference values with `ref` does not introduce conflicts; it is a simple implementation matter for multiple processors to concurrently create new values (*e.g.*, [43, 67, 82]).

written as $\tau_1 \rightarrow \tau_2$. For example, the function that returns the boolean *true* when its integer parameter is negative has type:

$$(\lambda \texttt{x.x < 0}) : \texttt{int} \rightarrow \texttt{bool}$$

Type inference places slight restrictions on λ_v-S programs. For example, in a conditional

$$(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3)$$

static type inference constrains the types of the conditional's subexpressions to $e_1 : \texttt{bool}$ and $e_{2,3} : \tau$.

Type inference extends to structured datatypes. I introduce further type notation as necessary.

Value Binding

In λ_v-S, a `let` expression *binds* a value to an identifier. ML syntax allows multiple value bindings within a `let`; *i.e.*, nested `let` expressions may be flattened. For example, in

```
let val x = y
    val xsquared = x * x
in
    e
end
```

`x` is first bound to the value of `y` from the `let`'s enclosing scope. The identifier `xsquared` is then bound to the square of `x`. The identifiers `x` and `xsquared`, in addition to bindings inherited from the enclosing scope, are accessible in e.

Recursive Functions and Polymorphism

The `val rec` syntax defines recursive functions.[3] The definition

$$\texttt{val rec } f = e$$

binds identifier f to the value of e—which must always be a λ-abstraction—and makes this binding of f accessible in e. Additional syntax further simplifies function definition. The λ-abstraction $\lambda x.e$ is written as `(fn` x `=>` e`)`. The `fun` syntax

$$\texttt{fun } f \ x = \cdots f \cdots$$

names and defines a recursive function f of a single parameter, x. This syntax is equivalent to the `val rec` binding

$$\texttt{val rec } f = (\texttt{fn } x \texttt{ => } \cdots f \cdots)$$

Currying provides multiple parameters to functions. For example,

$$\texttt{fun } g \ x \ y = \cdots g \cdots$$

defines a function g of two parameters, x and y. The binding

$$\texttt{val rec } g = (\texttt{fn } x \texttt{ => } (\texttt{fn } y \texttt{ => } \cdots g \cdots))$$

is g's `val rec` equivalent. As an example, consider the function that *composes* two higher-order functions:

```
fun compose f g = (fn x => f (g x))
```

[3] The call-by-value Y combinator implements recursion [98]:

$$Y_v \equiv \lambda f.\lambda x.(\lambda g.f(\lambda x.g\ g\ x))(\lambda g.f(\lambda x.g\ g\ x))\ x$$

The function compose, after application of its arguments f and g, returns a new function. Subsequent application of this new function computes f (g x), *e.g.*, the application

```
compose (fn x => x - 1) (fn x => x * x)
```

creates an anonymous function of a single parameter that computes $f(y) = y^2 - 1$.

The compose function has *polymorphic* type

$$\texttt{compose} : (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$$

that is, individual instances of an application of compose may compose functions of different types. Polymorphism is indicated by the appearance of *type variables* (α, β, γ) in compose's type. The type of compose's first functional parameter f is $\beta \rightarrow \gamma$; functional parameter g is of type $\alpha \rightarrow \beta$. Upon application of its two parameters, compose creates and returns a function of type $\alpha \rightarrow \gamma$. The above example applies compose to functions of type int $\rightarrow$ int. Therefore, the static type-inference algorithm *instantiates* the type variables α, β, and γ to int in this particular application. The result type of the above application is therefore int $\rightarrow$ int.

Basic Types

It is a straightforward matter to augment the ground types of λ_v-S (int, bool, and unit) with ML's additional ground types, *i.e.*, the real and string types.

References

A ref expression (ref e), where $e : \tau$, creates a mutable reference value (§2.2.2) of type:

$$\texttt{(ref } e\texttt{)}: \tau\ \texttt{ref}$$

Static typing imposes restrictions on values stored in reference values [112]. The only restriction relevant to this thesis is that the type of a reference value is fixed; that is, updates to a reference value r of type τ ref may only place values of type τ into r.

Extra syntax provides concise access to reference values. The ! operator fetches a reference value: !$e \equiv$ (get e). The infix assignment operator := performs a set: (e_1 := e_2) $\equiv$ (set e_1 e_2).

Tuples

Tuples are aggregates of values and introduce dynamic data structures into the language. The syntax $(e_1, \ldots, e_n)$ creates an n-tuple. A tuple expression first evaluates its subexpressions in left-to-right order $(e_i \rightarrow v_i)$ and then forms a tuple of the n result values v_i. A tuple has the *product type*:

$$(e_1 : \tau_1, \ldots, e_n : \tau_n) : (\tau_1 * \ldots * \tau_n)$$

For example, (true,(1,2,3)): (bool * (int * int * int)) is a binary tuple—it pairs a boolean value with a ternary tuple of integers.

Extension to records (tuples with labelled components) and reference arrays (aggregates of reference values) is straightforward.

Algebraic Recursive Datatypes

ML's datatype syntax permits programmer declaration of new recursive datatypes. Such datatypes give rise to dynamic data structures. In languages with higher-order functions and reference values as here,

dynamic data structures cause the parallelization difficulties addressed in this thesis.

The *list* type constructor, declared with

```
datatype α list = Nil | Cons of (α * α list)
```

is a central datatype. Type variable α parameterizes the `list` datatype and enables the `list` *constructors* (`Nil` and `Cons`) to build a family of homogeneous lists. The identifier `Nil` is a nullary constructor that constructs the empty list. `Cons` is a binary constructor that, when applied to a tuple with first component $s_1 : \alpha$ and second component $s_2 : \alpha$ `list`, builds a new list with head element s_1 and tail s_2. For example, the expression

```
Cons(1,Cons(2,Cons(3,Nil)))
```

is of type `int list` and constructs a three-element list of integers. Since lists are ubiquitous in dynamic languages, additional syntax aids in their construction and access: $[e_1,\ldots,e_n]$ creates an n element list; and `[]` denotes the constructor `Nil`. The infix operator `::` denotes the `Cons` constructor, *i.e.*, `Cons`$(e_1,e_2) \equiv (e_1 :: e_2)$. The binary infix operator `@` appends two lists. Selector functions `hd` and `tl` respectively return the head and tail of a non-empty list.

Pattern Matching

A *pattern* is a data template. If a datum *matches* a pattern, the *variables* in the pattern are bound to the corresponding components of the datum. A match *fails* if the datum and pattern do not concur. For example, the pattern `(x,y)` in

```
let val (x,y) = e in ... end
```

is matched to the value of e, which must be a binary tuple (v_1, v_2). This match binds identifier `x` to v_1 and identifier `y` to v_2 in the scope of the `let`.

Pattern matching is useful for defining functions—a series of n patterns can select from a function's n cases. The `|` symbol separates pattern-case pairs. Matching proceeds serially from left to right. A successful match causes evaluation of the corresponding function case. The matching process faults if no pattern matches the function's argument value. Using patterns, a function to compute the lengths of lists is:[4]

```
fun length [] = 0
  | length (x::xs) = 1 + (length xs)
```

When `length` is applied to the empty list, the first pattern (`[]`) matches and `length` returns `0`; otherwise, `length`'s argument matches the `::` list constructor in the second pattern (`x::xs`). This match binds `x` to the head element of the list and `xs` to the tail. Since the `length` function does not require a binding for a list's head element, the pattern (`x::xs`) is more informatively written as (`_::xs`) where the *wildcard* (`_`) matches—but does not bind—anything.

The keyword `as` decomposes a datum d with respect to a pattern while retaining a binding for d proper, *e.g.*, the pattern (`l as (0::xs)`) matches a list whose head element is the integer `0`; a successful match binds `l` to the entire list and `xs` to the list's tail.

---

[4] An equivalent definition of `length` that does not use patterns is:

```
fun length l = (if l = [] then 0 else 1 + (length (tl l)))
```

As with the variables bound by λ-abstractions or `let` terms, I assume that the program's pattern variables have unique names (renaming if necessary).

Control Flow

Mechanisms for explicit expression sequencing are desirable in imperative languages. The *sequence* $(e_1; \ldots ; e_n)$ evaluates e_i before e_j, $i < j$, and returns e_n's value, v_n. Additionally, the boolean junctions $\wedge$ and $\vee$ are available as ML's short-circuit infix operators: `andalso` and `orelse`, respectively.

Comparison to ML

Extended λ_v-S comprises most of ML. λ_v-S and dynamic parallelization do not address two aspects of ML: the module facility and the exception mechanism.

Modules encourage programming with abstract datatypes and enable separate compilation of program components. Although parallelization in the presence of modules is not the central concern of this thesis, modules do not impede dynamic parallelization—indeed, as opposed to static parallelization methods, dynamic techniques support modular programming since parallelization information propagates seamlessly across module boundaries at run time.

ML's exception mechanism allows programmers to declare, raise, and handle exceptions. Exceptions are imperative and, as such, can be treated as a form of side effect. For simplicity, I develop dynamic parallelization for programs that do not use exceptions. Note that this restriction does not preclude the dynamic parallelization of the functions within an ML program that do not raise or handle exceptions.

2.3 Parallelism in λ_v-S

The application term, $e \equiv (e_1\ e_2)$, is the primary source of parallelism in λ_v-S. If e_1 and e_2 do not conflict, then e_1 and e_2 may safely evaluate in parallel; that is, their parallel evaluation is *safe*. The application proper occurs after parallel evaluation of e's subexpressions.

The symbol $||$ stands for parallelism. It is a compiler-produced annotation—attached to an expression by the compiler only when parallel evaluation is known to be safe—and not a language mechanism. It is used in the following contexts: $e_{||}$ indicates that expression e (potentially) contains parallel subexpressions; $f_{||}$ where f is a function indicates that the argument expressions to f evaluate in parallel; $(e_1\ ||\ e_2)$ indicates the parallel evaluation of an application term's subexpressions; $(e_1, \ldots, e_2)_{||}$ denotes parallel evaluation of n-tuples. The separator $;_{||}$ in a sequence indicates that the expressions it separates evaluate in parallel.

2.4 ML Compiler

Implementation is based on the Standard ML of New Jersey ML compiler (SML/NJ) [11, 9]. SML/NJ is an optimizing compiler that—except for some C routines in the run-time system—is written in ML. SML/NJ compiles to a continuation-passing style [9, 107, 64] and, from this, generates native code for many computer architectures. The compiler can also produce portable C code with the *sml2c* [110] code

generator. Version 0.73 of SML/NJ and *sml2c* was used to implement the techniques described in this thesis.

The MP [82, 24] queue-based multiprocessing platform provides concurrency mechanisms (thread creation, synchronization, and run-time support[5]) for SML/NJ. MP primitives suffice to build the concurrency constructs necessary for implementing dynamic parallelization.

2.5 Shared-Memory Multiprocessor

Since dynamic languages build large pointer-based structures, a global address space simplifies the layout and access of such structures. The techniques of this thesis, therefore, assume that the target machine's parallel processors access a common global address space.

Implementation is on a 20 processor Sequent Symmetry with 40MB of shared memory. Each of the Sequent's 20-MHZ 80386 processors connects via a 64KB two-way set-associative cache to a bus to the shared memory. The word size of the machine is 32 bits. Word-size memory reads and writes are atomic. The Symmetry's operating system is a UNIX variant.

2.6 Experimental Measurements

Reported program execution times are the average of at least three trials and include the time required for garbage collection. All measurements were taken on an idle machine with standard compiler-optimization settings.

[5]Garbage collection is performed sequentially by a single processor in MP. The lack of concurrent collection capability prompted the design of the concurrent garbage collector (Chapter 6).

Chapter 3

λ-Tagging

Languages with both higher-order functions and imperative features pose difficulties for static compiler parallelization, because many different higher-order functions may reach, and may be applied in, a given expression during the program's evaluation. Since higher-order languages manipulate functions as values, statically determining a good approximation to the set of functions actually applied in a higher-order application expression is difficult in practice [85, 104].[1] Therefore, existing static parallelization systems (*e.g.*, [45, 68]) cannot precisely analyze—and hence do not effectively parallelize—program expressions involving higher-order imperative functions.

This chapter describes the design and implementation of a dynamic technique called λ-*tagging* that uncovers parallelism in the presence of imperative higher-order functions. λ-tagging dynamically propagates information about a function's potential side effects with the function's run-time representation. The `map` function of Figure 3.1 illustrates the problem and serves as an example of automatic parallelization using λ-tags.[2]

If a compiler cannot deduce the identity or behavior of the function f in an application expression e, the compiler must err conservatively and assume that f's application always exhibits worst-case behavior. Even when the set F of higher-order functions reaching e is statically known, static analyses conservatively approximate the side effects of an $f \in F$ as an upper bound of the side effects of *all* the functions in F. However, a dynamic instance of e may actually invoke a side-effect-free function. This, in turn, may expose dynamic parallelism that static methods are unable to utilize.

[1] In general, finding the *exact* set of higher-order functions reacing an application is undecidable.

[2] The `map` function is representative of a large class of functions that obscure parallelism by receiving and applying higher-order parameters (*e.g.*, sorting algorithms parameterized with comparison predicates and generalized data-traversal operations).

```
fun map f [] = []
  | map f (x::xs) = (f x) :: (map f xs)
```

Figure 3.1: The `map` function applies its higher-order parameter `f` to every element of its list parameter and returns a new list containing the results of the individual applications.

```
fun mapParallel f [] = []
  | mapParallel f (x::xs) = (f x) ::|| (mapParallel f xs)
```

Figure 3.2: Parallel version of map. A compiler may safely substitute mapParallel for map *only if* it statically ascertains that f does not produce side effects. The parallel infix list constructor $::_{||}$ forks its arguments as parallel threads and performs the Cons construction when the threads join.

```
fun mapDynamic f [] = []
  | mapDynamic f (x::xs) = if (safe f) then
                               (f x) ::|| (mapDynamic f xs)
                           else
                               (f x) :: (mapDynamic f xs)
```

Figure 3.3: Dynamically-parallel version of the map function using λ-tags. The predicate safe examines a function's λ-tag and returns *true* if the function cannot exhibit interfering side effects.

For example, application of map may safely use a parallel version (mapParallel of Figure 3.2) if map's higher-order parameter f cannot produce interfering side effects, *e.g.*, when f is functional. In the expression

$$e \equiv \texttt{map (nth n [...,g,...,h,...]) 1}$$

let the functions bound to g and h be functional and imperative, respectively. Expression e is an example of a program that static compilers cannot parallelize. The function nth uses its first argument, integer n, to select the n^{th} element of its second argument, a list. In e, nth's second argument is a list of functions. If n's value is unknown at compile time, a conventional compiler may not transform map into mapParallel because nth can select an imperative function from its list—imperative functions as arguments to mapParallel can cause indeterminate behavior.

As the above example illustrates, static approaches forego valuable parallelism when instances of statically-unknown functions are dynamically functional. Dynamic parallelization, however, often finds this statically-undetectable parallelism. By the definition of *necessity* (§1.1.2), this example demonstrates that—in the presence of imperative higher-order functions—dynamic parallelization techniques such as λ-tagging are necessary.

λ-tagging sidesteps the problem of not knowing a function's identity at compile-time by examining *tags* on functions at run-time. Such a tag is called a λ-*tag* and describes the possible side effects that an application of its function produces. Statically, λ-tagging infers a function's potential side effects from the program text and attaches an initial λ-tag to the function. In the case of a dynamically-created function f, static λ-tag information describes the λ-tag *to be dynamically created* upon f's dynamic instantiation. λ-tags propagate dynamically with functions' run-time representations. Checks to λ-tags—automatically inserted at compile time—dynamically determine when parallel evaluation is safe. In this manner, λ-tags enable dynamic detection of parallel computations involving statically-unknown higher-order functions.

Figure 3.3 depicts a dynamically-parallel version of map. A λ-tagging compiler can automatically generate mapDynamic from map's conventional definition (Figure 3.1). On entry to mapDynamic, the potential side effects of higher-order parameter f (described by f's λ-tag) select parallel or sequential mapping. Note that the expression from which the actual higher-order function originates is irrelevant.

Further optimization makes `mapDynamic` more efficient—a compiler may replace the recursive application of `mapDynamic` in the conditional's *false* branch with that of sequential `map` since `f`'s side effects are invariant in `mapDynamic`; similarly, application of `mapParallel` may replace that of `mapDynamic` in the *true* branch. This can substantially reduce the number of checks to λ-tags that `mapDynamic` performs.

This chapter develops λ-tagging for the λ_v-S language. I first describe λ-tagging's static (§3.1) and dynamic (§3.2) components. I then describe the λ-tagging implementation (§3.4) and discuss empirical results (§3.5).

3.1 Static Component

Static type and side-effect (*effect*) inferencing methods [108, 109, 61, 75] are used to assign initial λ-tags to a program's functions at compile time. After introducing the algebras and notation used by static effect inferencing, I describe how this static effect information is used for λ-tag parallelization.

3.1.1 Effects

An expression's *type* describes *what* e computes. An expression's *effect* [75] describes *how* it produces a result; *i.e.*, effects capture the potential side effects that can occur in computing an expression's value. Here, I follow Talpin and Jouvelot [108] and describe an algebra of effects and types for λ_v-S.

The ground terms of the algebra EFFECT are *effect variables* and *effect constants*:

$$\begin{array}{lll} \varepsilon \in \text{EFFECTVAR} & & \text{effect variables} \\ \kappa \in \text{EFFECTCONST} = \{\bot_\epsilon, \mathsf{read}, \mathsf{write}\} & & \text{effect constants} \end{array}$$

The constant $\bot_\epsilon$ denotes the absence of effects; that is, a purely-functional expression has effect $\bot_\epsilon$. The effect constants **read** and **write** denote the effects produced upon application of the `get` and `set` operators of λ_v-S, respectively.[3]

The effect terms ($\epsilon \in$ EFFECT) are:

$$\begin{array}{lcll} \epsilon & ::= & \varepsilon & \text{effect variable} \\ & | & \kappa & \text{effect constant} \\ & | & \epsilon \sqcup \epsilon & \text{effect join} \end{array}$$

Since an effect ϵ may include effect variables (ε), effects are polymorphic. The least upper-bound operator $\sqcup$ combines effects. Equality among effect terms is modulo the axioms of the $\sqcup$ operator:

$$\begin{array}{ll} (\epsilon \sqcup \epsilon') \sqcup \epsilon'' = \epsilon \sqcup (\epsilon' \sqcup \epsilon'') & \text{associativity} \\ \epsilon \sqcup \epsilon' = \epsilon' \sqcup \epsilon & \text{commutativity} \\ \epsilon \sqcup \epsilon = \epsilon & \text{idempotency} \\ \epsilon \sqcup \bot_\epsilon = \epsilon & \text{unity} \end{array}$$

[3] Augmenting the set of effect constants—to admit languages with IO or exceptions, for example—is straightforward. I omit such extension here.

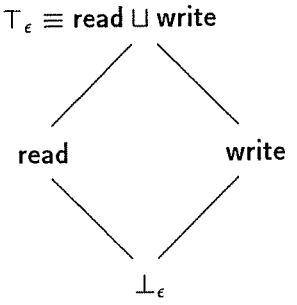


Figure 3.4: Constant-effect lattice built from $\sqsubseteq$, $\sqcup$, and EFFECTCONST.

The effect read $\sqcup$ write is the *maximum* effect that a λ_v-S expression can exhibit. The maximum effect is denoted $\top_\epsilon \equiv$ read $\sqcup$ write. The $\sqsubseteq$ relation introduces *subeffects*. Define $\sqsubseteq$ as:

$$\epsilon' \sqsubseteq \epsilon \iff \epsilon = \epsilon \sqcup \epsilon'$$

For example, the read effect is a (proper) subeffect of read $\sqcup\, \varepsilon_0$; that is, read $\sqsubseteq$ read $\sqcup\, \varepsilon_0$. The effects $\epsilon \equiv$ read and $\epsilon' \equiv$ write are *incomparable* ($\epsilon \not\sqsubseteq \epsilon' \wedge \epsilon' \not\sqsubseteq \epsilon$). The $\sqsubseteq$ relation induces a partial order on the effect terms. Figure 3.4 depicts the lattice of *constant effects* constructed from $\sqsubseteq$, $\sqcup$, and the effect constants. As we shall see, the elements of this lattice are the only effects carried dynamically in a function's λ-tag.

By itself, the algebra EFFECT cannot completely describe the effect of λ_v-S expressions. This is because λ-abstractions (functions) harbor side effects. The *direct effect* of an expression e is the effect of *evaluating* e to a value v. The *latent effect* of e is the effect produced in *subsequent application* of e's value v to further arguments. For example, evaluation of the following expression

$$e_{inc} \equiv \texttt{fn x => (set x (1 + (get x)))}$$

exhibits no direct effect in producing the value v_{inc}; that is, it has direct effect $\bot_\epsilon$. This is because λ-abstractions are values in λ_v-S and simply evaluate to themselves. However, subsequent application of v_{inc} invokes `get` and `set` (to increment an integer reference value) and therefore produces e_{inc}'s latent effect read $\sqcup$ write.

A standard polymorphic type system (*e.g.*, [26]) serves to characterize an expression's latent effect. The algebra TYPE accommodates the latent effects of higher-order imperative functions. Ground types are *type variables* and *type constants*:

α, β, γ	$\in$	TYPEVAR	type variables
ι	$\in$	TYPECONST $= \{\texttt{unit}, \texttt{int}, \texttt{bool}, \ldots\}$	type constants

A type ($\tau \in$ TYPE) is defined by:

τ	$::=$	ι	constant type
	$\mid$	α	variable type
	$\mid$	τ `ref`	reference type
	$\mid$	$\tau \xrightarrow{\epsilon} \tau$	function type

$$\texttt{ref} \;\; ! \;\; \langle \alpha \xrightarrow{\perp_\varepsilon} \alpha\,\texttt{ref},\ \perp_\epsilon \rangle$$

$$\texttt{set} \;\; ! \;\; \langle \alpha\,\texttt{ref} \xrightarrow{\perp_\varepsilon} \alpha \xrightarrow{\mathsf{write}} \texttt{unit},\ \perp_\epsilon \rangle$$

$$\texttt{get} \;\; ! \;\; \langle \alpha\,\texttt{ref} \xrightarrow{\mathsf{read}} \alpha,\ \perp_\epsilon \rangle$$

Figure 3.5: Effect of the λ_v-S imperative operators.

```
val rec compose = (fn f => (fn g => (fn x => f (g x))))
```

Figure 3.6: The compose function.

Note that the function type $\tau \xrightarrow{\epsilon} \tau$ is labelled with its latent effect ϵ.

Given the EFFECT and TYPE algebras, it is now possible to denote the complete effect of a λ_v-S expression. An expression e's effect is a pair: $\langle \tau, \epsilon \rangle$. This pair consists of e's type τ (which contains e's latent effect, if any) and e's direct effect ϵ. The notation

$$e \,!\, \langle \tau, \epsilon \rangle$$

states that e has type τ and direct effect ϵ. When $\epsilon = \perp_\epsilon$, I often omit it and write $e \,!\, \tau$ for e's effect.

Example Static Effects

Figure 3.5 gives the static effects of the `ref`, `get`, and `set` operators. Foremost, note that the direct effect of these operators is always the empty effect $\perp_\epsilon$. This is because the trivial evaluation of (function) constants to themselves is without effect. However, subsequent application of their values can incur side effects. Applying `ref` to an argument of type α simply creates a new reference value of type $\alpha\,\texttt{ref}$ and produces no visible side effects (see §2.2.2). Update (`set`) of a reference value of type $\alpha\,\texttt{ref}$ returns type `unit`; in doing so, it produces its latent **write** effect since it overwrites a mutable value. Note that this **write** occurs only after application of `set`'s second parameter (of type α). A `get` operation on a value of type $\alpha\,\texttt{ref}$ exhibits a **read** effect since it retrieves the contents (type α) of a mutable value.

As with its type, the effect of a function may also be polymorphic. For example, the `map` function (Figure 3.1) has effect:

$$\texttt{map} \,!\, (\alpha \xrightarrow{\varepsilon} \beta) \xrightarrow{\perp_\varepsilon} \alpha\,\texttt{list} \xrightarrow{\varepsilon} \beta\,\texttt{list}$$

Here, the effect variable ε represents the effect of `map`'s higher-order parameter. This effect is produced after application provides two arguments to `map`. The `compose` function of Figure 3.6 has effect:

$$(\beta \xrightarrow{\varepsilon_0} \gamma) \xrightarrow{\perp_\varepsilon} (\alpha \xrightarrow{\varepsilon_1} \beta) \xrightarrow{\perp_\varepsilon} (\alpha \xrightarrow{\varepsilon_0 \sqcup \varepsilon_1} \gamma)$$

The latent effect of `compose`'s result function, $(\alpha \xrightarrow{\varepsilon_0 \sqcup \varepsilon_1} \gamma)$, is the least upper bound of the effects of its functional parameters `f` and `g`. These parameters, respectively, have the variable types $(\beta \xrightarrow{\varepsilon_0} \gamma)$ and $(\alpha \xrightarrow{\varepsilon_1} \beta)$ with latent effects ε_0 and ε_1. Their combined latent effect, $\varepsilon_0 \sqcup \varepsilon_1$, captures all potential side effects that can occur in computing the composition `f (g x)`. Note that `compose`'s latent effect (in its type) contains all effects that `compose` *and* its result values can ever exhibit.

Effect Inference Rules

The inference rules of Figure 3.7 statically deduce a conservative approximation to the effect of a λ_v-S expression. These rules associate an environment and an effect with an expression.

An effect environment $\mathcal{E} \in \text{ENV}$ is a finite map from λ_v-S identifiers ($\text{ID} = \text{VAR} \cup \text{CONST}$) to types:

$$\mathcal{E} \in \text{ENV} = \text{ID} \xrightarrow{fin} \text{TYPE}$$

A sequent of the form $\mathcal{E} \vdash e \,!\, \langle \tau, \epsilon \rangle$ holds if one can infer, with respect to environment $\mathcal{E}$, that e has type τ and direct effect ϵ.

The inference rule (**var**) deduces that identifier x has type τ and no direct effect if x maps to τ in the environment $\mathcal{E}$. This reflects an identifier's status as a value term in λ_v-S. The (**abs**) rule creates latent effects. The direct effect of a λ-abstraction's body—in an environment extended with formal parameter x mapped to type τ—becomes a latent effect in the λ-abstraction's function type. The (**app**) rule is used to infer, for an application $(e\ e')$, a direct effect composed from three effects: the direct effects ϵ and ϵ' of evaluating e and e', respectively; and e's latent effect ϵ''. The (**sub**) rule increases (with respect to $\sqsubseteq$) an expression's effect. This rule says that if e has direct effect ϵ and $\epsilon \sqsubseteq \epsilon'$, one can then also infer the direct effect ϵ' for e's direct effect. The (**ref**) rule states that the direct effect of creating a reference value is the effect of evaluating `ref`'s argument. The (**get**) and (**set**) rules, respectively, combine the `read` and `write` effect constants with the effect of evaluating their argument(s).

The effect of the conditional is deduced via the (**if**) rule. Its effect consists of the (direct) effect of its predicate and the effects of *both* conditional branches. This potentially introduces imprecision into effects since a dynamic instance of a conditional expression evaluates only a single branch. Hence, an expression may be assigned an effect that it never produces. Dynamic parallelization with λ-tagging, however, propagates function effects at run time and thereby circumvents some of this imprecision. For example, in the expression

$$e \equiv \texttt{map (if p then f else g) l}$$

let `f` be effect-free and let `g` be imperative: $\texttt{f} \,!\, \tau \xrightarrow{\perp_\xi} \tau'$ and $\texttt{g} \,!\, \tau \xrightarrow{\top_\xi} \tau'$. Assuming that the predicate `p` has no effect, the static rules deduce the conditional's latent effect as the least upper bound of `f` and `g`'s latent effects:

$$\texttt{(if p then f else g)} \,!\, \langle \tau \xrightarrow{\top_\xi} \tau', \perp_\epsilon \rangle$$

In e, this conservative maximum effect prohibits static compiler transformation of `map` to `mapParallel` (Figure 3.2). With dynamic λ-tagging, however, a dynamic instance of e propagates the λ-tag from either `f` or `g` into the dynamically-parallel version of `map` (`mapDynamic` of Figure 3.3).

Two inference rules are required for `let` expressions: **let** and **e-let**. This requirement permits definition of functions with polymorphic types and effects; a polymorphic type and effect arises for a function only in a `let` binding [26]. The **let** rule assigns a polymorphic type and effect to the expression being bound (*e.g.*, a function). The **e-let** rule, on the other hand, is used to infer the type of the expression once (at the binding); therefore, the binding variable is not polymorphic in the body of the `let`. To decide which rule to use, it is necessary to syntactically differentiate expressions as *expansive* or *non-expansive* [112]. An expression is non-expansive if it is a variable or a λ-abstraction; all other expressions are expansive. Non-expansive expressions in `let` bindings have polymorphic effect (and type) and can, therefore, be multiply instantiated in the `let` body. Polymorphism is achieved by syntactically copying the expression into the `let` body; the non-expansive (**let**) rule replaces occurrences of the bound

$$\frac{x \mapsto \tau \in \mathcal{E}}{\mathcal{E} \vdash x \,!\, \langle \tau, \bot_\epsilon \rangle} \quad \textbf{(var)}$$

$$\frac{\mathcal{E} \pm \{x \mapsto \tau\} \vdash e \,!\, \langle \tau', \epsilon \rangle}{\mathcal{E} \vdash (\lambda x.e) \,!\, \langle \tau \xrightarrow{\epsilon} \tau', \bot_\epsilon \rangle} \quad \textbf{(abs)}$$

$$\frac{\mathcal{E} \vdash e \,!\, \langle \tau \xrightarrow{\epsilon''} \tau', \epsilon \rangle \quad \mathcal{E} \vdash e' \,!\, \langle \tau, \epsilon' \rangle}{\mathcal{E} \vdash (e\ e') \,!\, \langle \tau', \epsilon \sqcup \epsilon' \sqcup \epsilon'' \rangle} \quad \textbf{(app)}$$

$$\frac{\mathcal{E} \vdash e \,!\, \langle \tau, \epsilon \rangle \quad \epsilon \sqsubseteq \epsilon'}{\mathcal{E} \vdash e \,!\, \langle \tau, \epsilon' \rangle} \quad \textbf{(sub)}$$

$$\frac{\mathcal{E} \vdash e \,!\, \langle \tau, \epsilon \rangle}{\mathcal{E} \vdash (\texttt{ref}\ e) \,!\, \langle \tau\ \texttt{ref}, \epsilon \rangle} \quad \textbf{(ref)}$$

$$\frac{\mathcal{E} \vdash e \,!\, \langle \tau\ \texttt{ref}, \epsilon \rangle}{\mathcal{E} \vdash (\texttt{get}\ e) \,!\, \langle \tau, \epsilon \sqcup \mathsf{read} \rangle} \quad \textbf{(get)}$$

$$\frac{\mathcal{E} \vdash e \,!\, \langle \tau\ \texttt{ref}, \epsilon \rangle \quad \mathcal{E} \vdash e' \,!\, \langle \tau, \epsilon' \rangle}{\mathcal{E} \vdash (\texttt{set}\ e\ e') \,!\, \langle \texttt{unit}, \epsilon \sqcup \epsilon' \sqcup \mathsf{write} \rangle} \quad \textbf{(set)}$$

$$\frac{\mathcal{E} \vdash e \,!\, \langle \texttt{bool}, \epsilon \rangle \quad \mathcal{E} \vdash e' \,!\, \langle \tau, \epsilon' \rangle \quad \mathcal{E} \vdash e'' \,!\, \langle \tau, \epsilon'' \rangle}{\mathcal{E} \vdash (\texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'') \,!\, \langle \tau, \epsilon \sqcup \epsilon' \sqcup \epsilon'' \rangle} \quad \textbf{(if)}$$

$$\frac{\neg\text{expansive}\,(e) \quad \mathcal{E} \vdash e \,!\, \langle \tau, \bot_\epsilon \rangle \quad \mathcal{E} \vdash e'[e/x] \,!\, \langle \tau', \epsilon' \rangle}{\mathcal{E} \vdash (\texttt{let}\ x\ =\ e\ \texttt{in}\ e') \,!\, \langle \tau', \epsilon' \rangle} \quad \textbf{(let)}$$

$$\frac{\text{expansive}\,(e) \quad \mathcal{E} \vdash e \,!\, \langle \tau, \epsilon \rangle \quad \mathcal{E} \pm \{x \mapsto \tau\} \vdash e' \,!\, \langle \tau', \epsilon' \rangle}{\mathcal{E} \vdash (\texttt{let}\ x\ =\ e\ \texttt{in}\ e') \,!\, \langle \tau', \epsilon \sqcup \epsilon' \rangle} \quad \textbf{(e-let)}$$

Figure 3.7: Effect inference rules for λ_v-S.

identifier x in the `let` body, e', with the text of the binding expression, e. The expansive (**e-let**) rule simply infers the effect of the binding expression and assigns it to the binding identifier. The following expression illustrates how the **let** rule introduces polymorphism:

```
let fun id x = x
in
    ··· (id 5) ··· (id true) ···
end
```

Since the identity function `id` is non-expansive, it can have the polymorphic type $\alpha \xrightarrow{\perp_\S} \alpha$. This allows `id`'s type to be multiply instantiated (to the different types `int` $\xrightarrow{\perp_\S}$ `int` and `bool` $\xrightarrow{\perp_\S}$ `bool`) in the `let` body.

Effect Algorithm

This effect inference system is decidable; a procedure exists—algorithm $\mathcal{I}$ [108, 109]—that infers the *minimal* observable effects of expressions. An expression e's minimal observable effect is the least effect with respect to the $\sqsubseteq$ relation that includes all effects that e can produce. Algorithm $\mathcal{I}$ is consistent [108, 109] with respect to the effect inference system.

3.1.2 Using Static Effect Information

Given the static effect information for a λ_v-S program, it is now possible to assign a λ-tag for every function in the program. These λ-tags will propagate with the functions at run time and will be examined to make safe parallelization decisions.

I first describe (static) λ-tag assignment. Then, I give an algorithm for generating dynamically-parallel functions given that higher-order functions carry dynamic λ-tags. Finally, I describe an optimization that can reduce the number of times a program checks λ-tags.

λ-Tag Assignment

Static effect information reveals whether a function is purely functional (has no effect), potentially functional (applies functions with unknown effects, *i.e.*, variable effects) or imperative (`sets` or `gets` mutable values). This effect information is used to give every function f in the program a λ-tag $\in$ EFFECT that safely approximates the effects that f can produce. A function f's λ-tag is permanently assigned at compile time when possible. Otherwise, every dynamic evaluation instance of f forms a consistent λ-tag for f. The following description of λ-tag assignment assumes that static effect information has been computed for all expressions in the program. Following this description, I give some sample λ-tag assignments.

λ-tag assignment proceeds in two phases: λ-tag assignment first computes the effect for a function's λ-tag (*computation* phase); then, λ-tag assignment determines when (compile or run time) to assign the effect to the function's λ-tag (*determination* phase).

The λ-tag propagated at run time is always a *constant effect.* A constant effect is an effect built entirely from the effect constants ($\kappa \in$ EFFECTCONST); *i.e.*, it is an element in the effect lattice of

Figure 3.4. Furthermore, the λ-tag effect is the least constant effect that describes *all* the effects in its function's type.

To explain the computation of a function's λ-tag, I introduce further notation. A *tagged* function f is noted

$$f \equiv (\lambda^{\psi} x.e) \,!\, \tau \xrightarrow{\epsilon} \tau'$$

where $\psi \in \text{EFFECT}$ is f's λ-tag. Since application of f can return functions (*e.g.*, when f is curried), f's result type τ' may contain further latent effects. These latent effects appear only when the result of applying f is then applied to arguments. The λ-tag for f incorporates these additional effects as well. This allows for efficient implementation of the run-time checks to λ-tags (§3.1.3 below) that need to determine whether f's application *and* result value are effect free.

The effect to be carried by a function's λ-tag is computed as follows. The general function

$$f \equiv (\lambda^{\psi} x.e) \,!\, \tau \xrightarrow{\epsilon} \tau'$$

has the λ-tag effect

$$\psi = \mathit{Latent}^*(\tau \xrightarrow{\epsilon} \tau')$$

where a type's recursive latent effect, Latent^*, is computed thus:

$$\mathit{Latent}^*(\tau) \;=\; \begin{cases} \epsilon \sqcup \mathit{Latent}^*(\tau'') & \text{if } \tau = \tau' \xrightarrow{\epsilon} \tau'' \\ \mathit{Latent}^*(\tau') & \text{if } \tau = \tau' \,\texttt{ref} \\ \bot_{\epsilon} & \text{otherwise} \end{cases}$$

If type τ contains a function type, $\mathit{Latent}^*(\tau)$, gathers the latent effect of this function type and, recursively, the latent effects in the function's return type. Note that if the effect system deduces that the f's parameter x contains latent effects (*i.e.*, that x itself is a function or contains functions), these effects always appear in f's latent effect or as latent effects in f's return type.

Upon static computation of function f's λ-tag effect ψ, it remains to determine whether to assign ψ to f's λ-tag at compile or run time. Although conservative λ-tags can always be assigned statically, assignment at run time can potentially produce better (*i.e.*, more precise) λ-tags. This, in turn, can result in the detection of more parallelism.

Static λ-tag assignment is precise—and consequently occurs—when ψ for function $f \equiv (\lambda^{\psi} x.e)$ does not contain effect variables; that is, when $\nexists \varepsilon \in \text{EFFECTVAR}$ such that $\varepsilon \sqsubseteq \psi$. In this case ψ must be a constant effect and can be assigned statically. Otherwise, when $\exists \varepsilon \in \text{EFFECTVAR}$ such that $\varepsilon \sqsubseteq \psi$, the effect ψ contains effect variables; ψ is then a *variable* effect. When ψ is a variable effect, a precise static λ-tag assignment is not possible because the presence of variable effects in ψ indicate the application, within f, of functions with unknown effect. In this case the static assignment of the maximum effect $\top_{\epsilon}$ as the λ-tag is valid, albeit conservative. This conservative approximation, however, can often be avoided by instead assigning the λ-tag dynamically.

When the λ-tag effect ψ for function f contains effect variables, assignment of f's λ-tag at run time may yield a precise λ-tag. This is because the effects of the functions responsible for ψ's variable effects are present (on their λ-tags) at run time. If every effect variable ε_i in ψ represents the effect of some function g_i and every g_i's λ-tag is available when f evaluates, the combined effect of the g_i's λ-tags can be dynamically incorporated into the λ-tag for f. In other words, if f's effect ψ is variable due to the application of function variables free in f, then dynamic assignment of f's λ-tag is possible. Dynamic

$$\texttt{id} \equiv (\lambda^{\perp_\epsilon}\,\texttt{x.x}) \,!\, \alpha \xrightarrow{\perp_\epsilon} \alpha$$

$$\texttt{zero} \equiv \left(\lambda^{\mathsf{write}}\,\texttt{x.set x 0}\right) \,!\, \texttt{int ref} \xrightarrow{\mathsf{write}} \texttt{unit}$$

$$\texttt{compose} \equiv \left(\lambda^{\epsilon \sqcup \epsilon'}\texttt{f}.\lambda^{\epsilon \sqcup \epsilon'}\texttt{g}.\lambda^{\epsilon \sqcup \epsilon'}\texttt{x.f (g x)}\right) \,!\, (\beta \xrightarrow{\epsilon} \gamma) \xrightarrow{\perp_\epsilon} (\alpha \xrightarrow{\epsilon'} \beta) \xrightarrow{\perp_\epsilon} (\alpha \xrightarrow{\epsilon \sqcup \epsilon'} \gamma)$$

$$\texttt{map} \equiv \left(\begin{array}{l}\lambda^{\epsilon}\,\texttt{f}.\lambda^{\epsilon}\,\texttt{l.if l = [] then []} \\ \qquad\qquad \texttt{else f (hd l)::(map f (tl l))}\end{array}\right) \,!\, (\alpha \xrightarrow{\epsilon} \beta) \xrightarrow{\perp_\epsilon} \alpha\,\texttt{list} \xrightarrow{\epsilon} \beta\,\texttt{list}$$

Figure 3.8: Examples of λ-tag effects immediately *after* their static computation.

$$\texttt{id} \equiv (\lambda^{\perp_\epsilon}\,\texttt{x.x}) \,!\, \alpha \xrightarrow{\perp_\epsilon} \alpha$$

$$\texttt{zero} \equiv \left(\lambda^{\mathsf{write}}\,\texttt{x.set x 0}\right) \,!\, \texttt{int ref} \xrightarrow{\mathsf{write}} \texttt{unit}$$

$$\texttt{compose} \equiv (\lambda^{\mathsf{T}_\epsilon}\texttt{f}.\lambda^{\mathsf{T}_\epsilon}\texttt{g}.\lambda^{\mathsf{f} \sqcup \mathsf{g}}\,\texttt{x.f (g x)}) \,!\, (\beta \xrightarrow{\epsilon} \gamma) \xrightarrow{\perp_\epsilon} (\alpha \xrightarrow{\epsilon'} \beta) \xrightarrow{\perp_\epsilon} (\alpha \xrightarrow{\epsilon \sqcup \epsilon'} \gamma)$$

$$\texttt{map} \equiv \left(\begin{array}{l}\lambda^{\mathsf{T}_\epsilon}\,\texttt{f}.\lambda^{\mathsf{f}}\,\texttt{l.if l = [] then []} \\ \qquad\qquad \texttt{else f (hd l)::(map f (tl l))}\end{array}\right) \,!\, (\alpha \xrightarrow{\epsilon} \beta) \xrightarrow{\perp_\epsilon} \alpha\,\texttt{list} \xrightarrow{\epsilon} \beta\,\texttt{list}$$

Figure 3.9: Examples of λ-tag effects *after* determining when to assign the effects.

assignment combines the λ-tags from the functions that are free in f (and hence bound with an assigned λ-tag when f evaluates) into f's λ-tag.

For example, the function

$$f \equiv (\lambda^{\psi} x.\,\texttt{get}\ (g\ x)) \,!\, \alpha \xrightarrow{\epsilon \sqcup \mathsf{read}} \beta$$

has the λ-tag effect $\psi = \varepsilon \sqcup \mathsf{read}$, where the variable effect ε stems from g's (unknown) effect. The function f is therefore a candidate for dynamic λ-tag assignment because ψ contains the effect variable ε. Since g carries a λ-tag at run-time that is available for inspection when f evaluates, a consistent λ-tag for f can be assigned dynamically. This λ-tag for f is written $\psi = g \sqcup \mathsf{read}$ to reflect the run-time inclusion of g's effect. For the dynamic instances of f where g is functional, f carries the precise λ-tag read.

Example λ-tags after their static computation are given in Figure 3.8. The same λ-tags after determination of when (run or compile time) to assign the effect are in Figure 3.9. An effect that will be dynamically assigned contains the function identifiers of the functions whose effects are to be included at run time. The λ-tags for `id` and `zero` are static (since they do not contain effect variables) and are

$$\mathcal{R}_\lambda \;:\; \textsc{Exp} \times \mathrm{Fin}(\textsc{Var}) \to \textsc{Exp}_{||} \times (\textsc{Var} \xrightarrow{fin} \textsc{Effect})$$

$$\begin{aligned}
\mathcal{R}_\lambda(e, F) = \; & \mathit{case}\ \ e\ \ \mathit{of} \\
& b,\, x \Rightarrow (e,\, \{f \mapsto \bot_\epsilon \mid f \in F\}) \\
& (e_1\ \ e_2) \Rightarrow \mathit{let}\ (e_i', S_i') = \mathcal{R}_\lambda(e_i, F),\ 1 \le i \le 2 \\
& \qquad\qquad T = \left\{ S \in F \xrightarrow{fin} C \mid (e_1'\ ||\ e_2') \text{ is safe with respect to } S \right\} \\
& \qquad \mathit{in}\ \ \mathit{if}\ T = \emptyset\ \mathit{then}\ ((e_1'\ \ e_2'),\, S_1' \sqcup S_2') \\
& \qquad\qquad \mathit{else}\ ((e_1'\ ||\ e_2'),\, (\bigsqcup T) \sqcup S_1' \sqcup S_2') \\
& \lambda\, x.e \Rightarrow \mathit{let}\ (e', S') = \mathcal{R}_\lambda(e, F \backslash \{x\}) \\
& \qquad \mathit{in}\ \ (\lambda\, x.e',\, S' \pm \{x \mapsto \bot_\epsilon\})
\end{aligned}$$

Figure 3.10: λ-tag restructuring algorithm $\mathcal{R}_\lambda$.

therefore identical in both figures. The `compose` and `map` functions, however, assign (some) λ-tags dynamically. After application of its functional parameters, `compose` constructs a λ-tag for its return value from the λ-tags of its parameters. Note that the function returned after supplying a single argument to `compose` has effect $\top_\epsilon$ (assigned statically) because the effect of `compose`'s second argument (formal `g`) is yet unknown. Upon receiving its first parameter `f`, the `map` function constructs the λ-tag for its inner function from the λ-tag bound to `f`. Functions created by the applications `(map id)` and `(map zero)`, for example, carry the λ-tags $\bot_\epsilon$ and **write**, respectively.

3.1.3 Parallelization with λ-Tags

In addition to the static effect information, λ-tags supply dynamic effect information about functions. A program can now be *restructured.* Restructuring inserts parallelism annotations—and the necessary λ-tag checks to dynamically ensure their correctness—into the program.

Restructuring occurs at the function level. Let $f \equiv (\lambda x_1.\lambda x_2.\ldots\lambda x_n.e)$ be the function of interest. $X = \{x_1, \ldots, x_n\}$ is the set of f's formal parameters and $H \subseteq X$ is the set of f's higher-order parameters that are first order.[4] Higher-order parameters to f that are not first order are not included in H and are assumed to have maximum effect $\top_\epsilon$.[5] Let $C = \{\bot_\epsilon, \mathsf{read}, \mathsf{write}, \mathsf{read} \sqcup \mathsf{write}\}$ be the set of constant

[4] Define a function g to be *first order* if it has type $\tau \to \tau'$ and τ and τ' do not contain function types. That is, g is first order when it does not use its parameter in a function context (*e.g.*, application) and g does not create and return a function.

[5] The restriction that the set H only contain first-order functions is necessary for the following reason. A parameter that is higher-order (and without effect) can be applied to a function h and thereby produce h's effect. However, the origin of h depends on the higher-order parameter, and it is not possible to statically identify—and hence to statically insert a check to—this function h. For example, the parameter y of the function

$$(\lambda x.\lambda y.\lambda z.x\ y\ z) : (\alpha \to (\beta \to \gamma)) \to \alpha \to \beta)$$

can be a function (when α is instantiated to a function type), but the type of parameter x obscures this fact. By only considering first-order parameters to a function, all of the function's parameters that can be applied in the function are statically identified as such by their type.

effects.

Figure 3.10 contains the λ-tag restructuring algorithm $\mathcal{R}_\lambda$. To restructure a function, apply algorithm $\mathcal{R}_\lambda$ to function f's body e and its set H of higher-order parameters. $\mathcal{R}_\lambda(e, H)$ returns an annotated expression $e_{||}$ and a *safety map* S where

$$S \in \text{VAR} \xrightarrow{fin} \text{EFFECT}$$

is a map from H to the constant effects C. This map indicates the conditions that must exist on entry to f for the parallel body $e_{||}$ to be safely used in lieu of e; that is, an element $h \mapsto \epsilon \in S$ indicates that the higher-order parameter h to function f must have an effect (on its λ-tag) that is $\sqsubseteq \epsilon$ for safe evaluation of $e_{||}$.

Restructuring function f proceeds in a bottom-up fashion. Algorithm $\mathcal{R}_\lambda$ descends into an expression e with the set F of variables, visible in e, that represent the parameters to f that are first-order functions. If e is a constant or variable, no parallelization is possible and $\mathcal{R}_\lambda$ simply returns e along with the constant map from F to the effect $\bot_\epsilon$.

Parallelism annotations arise only in application expressions (§2.3). For an application $(e_1\ e_2)$, its subexpressions e_i are first recursively restructured. This returns new, potentially annotated, expressions e_i' $(1 \leq i \leq 2)$ and the corresponding safety maps S_i'. For an $h \in F$, such a map S_i' contains the greatest effect that h may have for the parallelism in e_i to be safe. After restructuring the application's subexpressions, $\mathcal{R}_\lambda$ restructures the application proper. For all maps $S \in F \xrightarrow{fin} C$ from the higher-order parameters F reaching the application to the effect constants C, algorithm $\mathcal{R}_\lambda$ examines $(e_1'\ e_2')$ in conjunction with the static effect information for $(e_1'\ e_2')$. If the effect information for $(e_1'\ e_2')$, augmented with effect bindings in S, reveals that e_1 cannot conflict with e_2 under parallel evaluation, then $\mathcal{R}_\lambda$ retains S in the set T of such maps. After examining all such maps S, the annotated application $(e_1'\ ||\ e_2')$ is returned if T contains a map; that is, if conditions exist under which $(e_1'\ ||\ e_2')$ is safe. The safety map returned in this case is the least upper bound[6] of the maps in T and the maps S_i'. Otherwise, when no safety map $S \in F \xrightarrow{fin} C$ exists such that e_1 and e_2 can safely evaluate in parallel, $\mathcal{R}_\lambda$ returns an (unannotated) application of the restructured subexpressions e_1' and e_2'. Finally, a λ-abstraction is restructured by restructuring the λ-abstraction's body. Before doing so, $\mathcal{R}_\lambda$ removes identifiers identical to the λ-abstraction's variable from the set of higher-order parameters reaching the λ-abstraction.

As an example of restructuring with λ-tag information, consider the concrete function:

$$f \equiv \texttt{(fn g => fn h => fn x => (g x) (h x))}$$

$\mathcal{R}_\lambda$ applied to f's body and higher-order parameters, $\mathcal{R}_\lambda(\texttt{(g x) (h x)}, \{\texttt{g},\texttt{h}\})$, yields the annotated body `(g || x) || (h || x)` and the safety map $\{\texttt{g} \mapsto \textsf{read}, \texttt{h} \mapsto \textsf{read}\}$. Evaluation of this parallel body is therefore valid when both g's λ-tag $\sqsubseteq$ **read** and h's λ-tag $\sqsubseteq$ **read**. That is, when g and h do not `set` reference values. This leads directly to a dynamically-parallel version of f:

```
(fn g => fn h => fn x => if (reads g) andalso (reads h) then
                             (g || x) || (h || x)
                         else
                             (g x) (h x))
```

The predicate `reads` examines the λ-tag on its argument (a function) and returns *true* if the effect on this λ-tag is $\sqsubseteq$ **read**.

[6]The least upper bound of two maps is formed by applying $\sqcup$ pointwise.

```
fun mapDynamic f [] = []
  | mapDynamic f (x::xs) = if (safe f) then
                               (f x) ::|| (mapParallel f xs)
                           else
                               (f x) :: (map f xs)
```

Figure 3.11: Dynamically-parallel `map` with invariant-effect optimization.

Parallelism that $\mathcal{R}_\lambda$ demarcates should not (and need not) always be used. In the above example, evaluating the variables `g` and `x` (and similarly, `h` and `x`) in parallel is likely to slow, rather than speed, evaluation because the cost of evaluating a variable (or constant) is negligible whereas parallel-thread creation incurs overhead. Therefore, it is necessary to apply heuristics (*e.g.*, Gray's *quickness* [40]) to select an expression's useful parallelism.[7] Useful parallelism in the above example is `(g x) || (h x)`.

3.1.4 Invariant-Effect Optimization

λ-tag examination at run time incurs costs. Removal of redundant checks to λ-tags can improve λ-tagging's performance. A check c to function f's λ-tag may be safely removed if the compiler can prove that a prior check c' to f's λ-tag always asserts c's condition before evaluation reaches c (note that once assigned, a λ-tag's effect does not change). *Loop-invariant* effects, in particular, provide significant optimization opportunity. Static compiler invariance analyses suffice to implement this optimization [9, 2, 35].

Figure 3.11 illustrates invariant-effect optimization of the dynamically-parallel version of `map`. A compiler can discover that the effect of `map`'s higher-order parameter `f` is invariant within `map`. The recursive applications of `mapDynamic` can therefore be replaced by the applications of sequential `map` (Figure 3.1) and `mapParallel` (Figure 3.2). Contrast this optimized version of `mapDynamic` with the unoptimized version in Figure 3.3. The number of dynamic λ-tag checks performed when `mapDynamic` is applied to a list l drops from $|l|$ to one with this optimization.

3.2 Dynamic Component

Construction of λ-tagging's dynamic component is straightforward. Mechanisms to *propagate*, *manipulate*, and to *examine* λ-tags are needed.

3.2.1 λ-Tag Propagation

λ-tags propagate with a function's run-time representation; *i.e.*, with its *closure*. A function's λ-tag can be either stored in its closure or merged with its closure's address. The former is simple to implement because it only requires an additional field in a closure to hold a λ-tag. The latter is more efficient because retrieval of a function's closure is not necessary to obtain the function's effect; examination of

[7] Chapter 5 develops a technique that, for some expressions, dynamically determines whether their parallel evaluation is worthwhile.

```
fun split l pivot p =
        let fun split' [] less greater = (less,greater)
              | split' (x::xs) less greater =
                    if (p x pivot) then
                        split' xs (x::less) greater
                    else
                        split' xs less (x::greater)
        in
            split' l [] []
        end

fun qs p [] = []
  | qs p (x::xs) =
         let val (l,g) = split xs x p
         in
             (qs p l) @ (x::(qs p g))
         end
```

Figure 3.12: Sequential higher-order quicksort.

the closure's address suffices. In a parallel system, this can potentially reduce contention for the contents of shared closures. However, this latter approach requires that there be space in the closure's address for the λ-tag effect encoding.

3.2.2 λ-Tag Manipulation

The compiler and run-time system provide primitives to place, retrieve, and combine λ-tags:

`(setTag` f ϵ`)` places the effect ϵ in function f's λ-tag.

`(getTag` f`)` retrieves and returns the effect from function f's λ-tag.

`(combineEffects` ϵ ϵ'`)` returns the constant effect $\epsilon \sqcup \epsilon'$.

3.2.3 λ-Tag Examination

For every effect $\kappa \in$ EFFECTCONST the compiler and run-time system provide a predicate (*e.g.*, `safe`, `reads`, and `writes`) that accepts a single function argument and returns *true* only if the function's λ-tag effect is $\sqsubseteq \kappa$.

```
val rec compose =
    (setTag (fn f =>
             (setTag (fn g =>
                      (setTag (fn x => f (g x))
                              (combineEffects (getTag f) (getTag g))))
                     ReadWrite))
            ReadWrite)
```

Figure 3.13: λ-tag assignment for compose.

```
val rec qsDynamic =
    (setTag (fn p =>
             (setTag (fn [] => [] |
                         (x::xs) =>
                             if (reads p) then
                                 let val (l,g) = split xs x p
                                 in
                                     (qsDynamic p l) @|| (x::(qsDynamic p g))
                                 end
                             else
                                 let val (l,g) = split xs x p
                                 in
                                     (qsDynamic p l) @ (x::(qsDynamic p g))
                                 end
                     (getTag p)))
            ReadWrite)
```

Figure 3.14: Dynamically-parallel quicksort with λ-tag assignment.

3.3 Examples

Automatically λ-tagged and restructured versions of compose (Figure 3.6) and quicksort (qs, Figure 3.12) are in Figures 3.13 and 3.14, respectively. The static λ-tag-assignment phase inserts the setTag, getTag, and combineEffects primitives.[8]

The compose function contains no useful parallelism because application of f must follow that of g (control dependence). Dynamically-parallel qsDynamic sorts sublists in parallel when the comparison predicate p cannot perform set operations. Restructuring finds no useful parallelism in the quicksort's auxiliary function split. Invariant-effect optimization (§3.1.4) can produce a version of qsDynamic that performs the (reads p) check only once since p, and hence its effect, does not change during the sort. With this optimization, the recursive calls are then to the sequential qs or to an automatically-generated, but not dynamic, parallel version of qs.

[8] Static *function escape analysis* (*e.g.*, [9]) can sometimes statically determine that a function f cannot be used in any higher-order application expression. When this is the case, f need not be assigned a λ-tag.

3.4 Implementation

Static effect inferencing (§3.1.1), λ-tag assignment (§3.1.1), and restructuring (§3.1.3) were implemented in ML for ML's λ_v-S subset (§2.2). A heuristic, similar to Gray's *quickness*[9] [40], statically selects viable parallelism. Automatic invariant-effect optimization (§3.1.4) was not implemented. A restructured parallel λ_v-S program is manually converted to its ML equivalent for execution with a λ-tagging run-time system.

λ-tagging's dynamic component was implemented in the SML/NJ system. A small integer encodes the constant effects that a λ-tag carries. The SML/NJ compiler was modified to allocate an additional word in every function closure to hold its λ-tag. Routines to access λ-tags (`getTag` and `setTag`) and to combine the effects encoded therein (`combineEffects`) were also added to the compiler. These routines were directly implemented using existing (non-standard ML) primitives present in SML/NJ that provide unrestricted access to ML data structures. Since the implementation of these routines is at a high level in the compiler, they require tens of machine instructions to execute. By building low-level primitives for these routines into the compiler's back end, λ-tagging performance could be further improved. However, the current implementation of these routines is simple and sufficiently efficient to demonstrate that λ-tagging is viable.

3.5 Results

Programs measured were quicksort (`qs`, Figure 3.12) and a symbolic matrix-multiplication routine (`mm`). The `qs` function sorted a list of 10000 integers using the effect-free integer $<$ relation as its comparison predicate. This predicate is a higher-order parameter to `qs`. The `mm` routine is parameterized by two matrices and by two higher-order functions: addition and multiplication functions for individual matrix elements. Parallelism in `mm` stems entirely from parallelism in the `map` function—`mm` uses `map` to transpose matrices and to perform the matrix multiplication proper. The `mm` routine was applied to 100×100 integer matrices—effect-free integer addition and multiplication functions were supplied as the higher-order parameters to `mm`.

Timings for `qs` and `mm` are in Figures 3.15 and 3.16. Up to four versions of each program were measured: sequential, explicitly parallel, λ-tagging, and optimized λ-tagging. Absolute speedup is not the point of these execution times; rather, the point is the differences in execution time between the λ-tag versions and the sequential and explicitly parallel versions of a program that demonstrate λ-tagging's efficiency. The optimized λ-tag version was built by manually performing effect-invariance analysis (§3.1.4) to remove redundant λ-tag checks. The explicitly parallel version of a program was built from the unoptimized λ-tagging version by removing all λ-tag checks and enabling all λ-tag parallelism. Since the resulting explicitly parallel program is potentially unsafe, it was always supplied higher-order parameters that were effect free. The sequential and explicitly parallel versions used the unmodified run-time system; *i.e.*, closures did not contain the extra word for the λ-tag.

As is evident from Figure 3.15, automatic parallelization of `qs` with unoptimized λ-tags introduces

[9]The *quickness* heuristic statically computes a lower bound on the number of applications an expression always performs. As such, it is extremely conservative.

	Processors							
	1	2	3	4	5	6	7	8
Sequential	31.3	–	–	–	–	–	–	–
Explicit Parallel	49.1	20.8	17.7	15.0	12.8	12.2	13.1	15.0
λ-tags	50.5	21.9	17.7	15.5	13.1	13.5	14.4	15.5

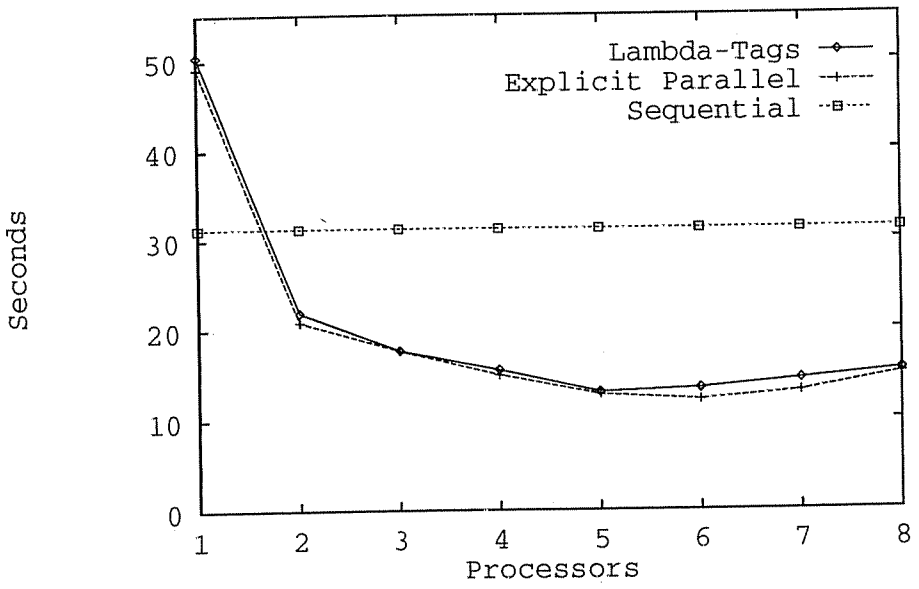


Figure 3.15: λ-tag times for quicksort (qs).

	Processors							
	1	2	3	4	5	6	7	8
Sequential	142.5	–	–	–	–	–	–	–
Explicit Parallel	168.8	91.0	61.2	44.7	36.0	28.9	24.0	23.8
λ-tags	216.9	118.7	82.8	64.8	52.0	47.1	43.0	38.5
Optimized λ-tags	172.2	94.7	66.5	48.1	39.0	31.4	28.0	25.1

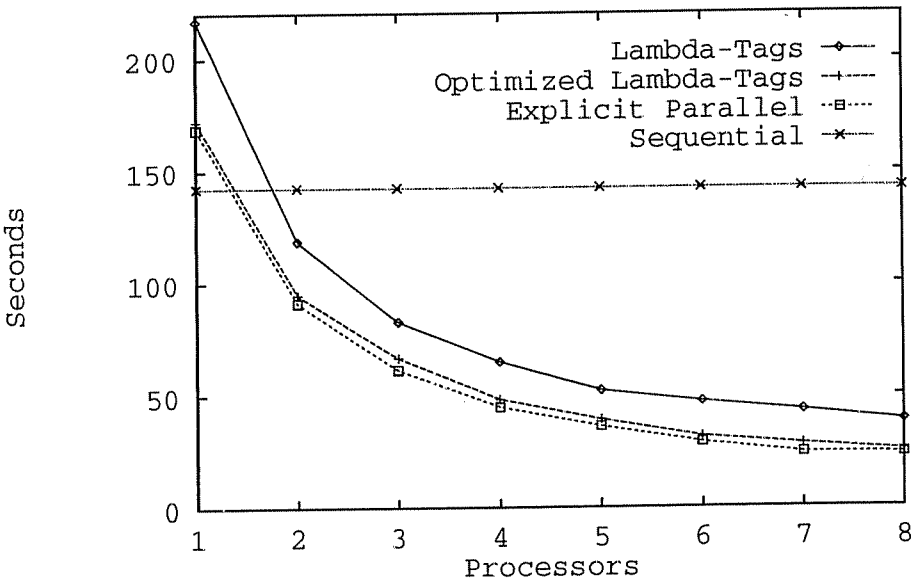


Figure 3.16: λ-tag times for symbolic matrix multiply (mm).

negligible overhead into the program's execution. Hence, effect-invariance optimization is not necessary for `qs` and is not shown. Automatic parallelization of `mm` with unoptimized λ-tags, however, incurs overhead—on a single processor as much as 29% of the explicitly parallel time. This is because all parallelism in `mm` arises in applications of `map`. Without λ-tags, the compiler produces an efficient loop for `map`. With λ-tag checks, however, the compiler is unable to compile `map` (now `mapDynamic` of Figure 3.3) into an equally efficient loop. Even with this inefficiency, unoptimized λ-tagging improves on the program's sequential execution time with just two processors. Further improvements occur with additional processors. For `mm` with redundant λ-tag-check elimination (Figure 3.11), the cost of optimized λ-tag parallelization is less than 10% of the explicitly parallel execution time (for all tested processor counts).

Beyond eight processors, times for parallel versions of both programs level and then rise. I surmise that this is not due to a lack of parallelism in the programs, but rather to insufficient memory bandwidth.[10] That is, the Sequent's bus is able to sustain memory requests for ML programs only for a small number of processors. However, this effect is not related to λ-tagging, but rather to the hardware implementation of the computer's shared-memory system. Additionally, both programs generate many small threads whose scheduling costs outweigh their computation.[11] This, too, can restrict continued speedup with additional processors.

3.6 Other λ-Tag Uses

The λ-tagging idea—propagating information with functions at run time—is general. In addition to side-effect information, a function's λ-tag can carry other information about the function's properties.

For example, in a lazy (call-by-name) language (see [60, 34]), λ-tags can dynamically carry *strictness information*[12] with functions. Precise strictness information permits (parallel and sequential) optimization: the strict arguments in a function application may evaluate in parallel; in a sequential implementation, strict arguments may be fully evaluated immediately, instead of being evaluated only when they are actually used. Another use for λ-tags is in load-balancing a parallel language system. A function f can carry a λ-tag that describes the (approximate) cost of applying f; applications of functions containing little computation relative to their parallel-scheduling cost should evaluate sequentially. Resource requirements (*e.g.*, memory demands) can also be encoded in a function's λ-tag.

3.7 Notes

λ-tagging was previously published in [53].

All previous work concerning imperative function analysis in higher-order languages is solely static. These analyses fall into two categories: inference of a program's effect from the language's static semantics and a static approximation of the program's dynamic semantics.

---

[10] Morrisett and Tolmach [82] also observed this effect for explicitly parallel ML programs using the same run-time system and computer.

[11] Chapter 5 describes a technique that can dynamically estimate the benefit of evaluating an expression in parallel.

[12] A function $f \equiv (\lambda x.e)$ is *strict* in parameter x if application of f always uses the value bound to x.

Lucassen and Gifford's FX language [75, 74, 38] introduced effects and effect inference in the framework of *polymorphic effect systems*. Formalization of FX's effect system [61] lead to Talpin and Jouvelot's algorithm $\mathcal{I}$ [108, 109] that infers an expression's effect. These systems differ from the effect inference system used for λ-tag assignment (§3.1.1) in two respects: they have an *allocation effect* and they infer the *region* in which an effect occurs. Reference value creation with `ref` produces an allocation effect. In implementations where allocations do not conflict—as in the λ-tag implementation—an allocation effect is unnecessary. It can, however, be added to the inference system for λ-tags if necessary. In addition to an expression's type and effect, FX (and algorithm $\mathcal{I}$) also consider and infer the expression's region. A region specifies *where* reference values are located in the store. Reference value creation associates a reference value with a region. Base effects (*e.g.*, **read** and **write**) are no longer constants, but are unary functions parameterized by the region in which they occur (*e.g.*, **write**(ρ) indicates a **write** effect to region ρ). Region inference can detect that accesses to different reference values in disjoint regions do not conflict. This, in turn, permits parallelization of these accesses. Furthermore, region inference reveals *effect masking* (*e.g.*, [108]). An expression masks its effect if the reference values it modifies are internal to the expression. That is, if the expression creates a reference value, modifies it (perhaps repeatedly), but does not relinquish this value to the enclosing context, then the expression is referentially transparent and its evaluation cannot conflict with that of other expressions. Region information could improve λ-tag assignment since a function that completely masks its effect can be assigned the pure effect $\perp_\epsilon$.

The second class of static analyses for higher-order imperative languages approximates the program's dynamic behavior at compile time. These methods statically require access to the entire program text to perform an interprocedural analysis since they approximate the program's entire behavior at compile time. This limits their practical use for large programs. λ-tagging's static component, on the other hand, performs its analysis at the function level and only requires an environment containing the types and effects of prior function definitions. λ-tagging, therefore, admits separate and interactive compilation of large programs. Neirnyck [85] describes a static side-effect analysis for an imperative call-by-value language similar to λ_v-S. She uses abstract interpretation [25, 1] to determine alias information. Her analysis can be used to (conservatively) decide if two expressions can interfere. Shivers [104, 105] attributes the difficulty of higher-order language analysis to the lack of an explicit static control-flow graph at compile time. He presents a technique that statically (conservatively) recovers the control-flow graph of a higher-order Scheme program. Shivers develops several sequential optimizations based on this control-flow information. Two parallel systems—Harrison's PARCEL [45, 44] and Larus's Curare [68, 67]—extract static parallelism from Scheme programs. Both systems primarily target first-order programs; they do not effectively track higher-order functions embedded in dynamic data.

Though diverse in their methodology, all static approaches to higher-order side-effect analysis introduce imprecision into their result for identical reasons: multiple higher-order functions may reach function applications via many control-flow paths. Static techniques and systems must—and do—approximate a function reaching an application as the least upper bound of the functions that can actually reach the application. The dynamic λ-tagging technique described in this chapter avoids this imprecision since it examines the effect of the exact function reaching an application expression at run time. Consequently, λ-tagging can find parallelism that eludes static parallelization techniques.

Chapter 4

Dynamic Resolution

The functional subcomputations in an imperative program are a fruitful source of parallelism. However, the program's imperative subcomputations may contain plentiful parallelism as well. One source of such parallelism is imperative expressions that traverse and modify dynamically-allocated data structures. When an expression's imperative subexpressions access and modify disjoint portions of a data structure, safe parallel evaluation of the subexpressions is often possible. This chapter develops a technique, called *dynamic resolution* (**dr**), which automatically parallelizes some functions that manipulate a single dynamic data structure. The **dr** technique dynamically detects and dynamically schedules conflicting expressions; it *resolves* conflicts at run time.

Parallel evaluation of expressions that modify (`get` and `set`) shared data—data that multiple expressions may concurrently access—must prevent **read/write** and **write/write** conflicts from violating the sequential semantics of the program. Detecting and synchronizing accesses to dynamic shared data is difficult for compilers (*e.g.*, [44, 68]) and programmers alike since sharing appears (and disappears) dynamically. Furthermore, a program's data-sharing characteristics depend on the program's data structures, which are often dependent on the program's input. At run time, however, shared data can be detected and access to it correctly coordinated.

For example, a compiler may statically deduce that a list l of mutable reference values *could* contain the same element a more than once (thereby sharing a with itself). This forces the compiler to perform operations on individual elements of l sequentially because, at compile time, it is not known when (at run time) or where (in l) such a shared element exists. For a given dynamic instance of l, however, l's elements may be disjoint so concurrent access and modification is safe. Furthermore, even if *some* elements of l are identical (shared), others can be modified concurrently if sharing detection and expression scheduling are dynamic.

The `incnode` function of Figure 4.1 illustrates the problem and serves as an example of automatic parallelization using dynamic resolution. The `incnode` function operates on dynamic data of the `tree` datatype. In particular, its single parameter is of type `int ref tree`; that is, leaf nodes contain integer reference values. When supplied a leaf node, the `incnode` function increments the integer reference value in that leaf. Otherwise, when supplied an internal node, `incnode` recursively descends into the `left` and `right` portions of the node's structure to increment their respective leaves.

The sequential semantics of the language requires that all modification (with `set`) of a reference

```
datatype α tree = Leaf of α | Node of (α tree * α tree)

fun incnode (Leaf x) = (set x (1 + (get x)))
  | incnode (Node(left,right)) = (incnode left ; incnode right)
```

Figure 4.1: The incnode function. Dynamic resolution can safely evaluate expressions (incnode left) and (incnode right) in parallel since it detects conflicts, due to sharing in incnode's argument, dynamically.

Figure 4.2: Possible tree and DAG arguments to incnode.

value r by the expression (incnode left) occur before (incnode right) accesses r. Similarly, the expression (incnode right) may not set r until (incnode left) completes its last access of r. Parallel evaluation of (incnode left) and (incnode right) is, however, safe when (incnode left) and (incnode right) access disjoint sets of reference values; *i.e.*, when the dynamic data bound to left and right do not share structure. Static detection of this parallelism requires the compiler to ascertain whether (and where) sharing exists in incnode's argument. This, however, is difficult because the tree datatype can be used to construct directed cyclic and acyclic graphs (DAGs) as well as trees;[1] *e.g.*,

```
let val n = Node(n1,n2)
in
    Node(n,n)
end
```

creates a DAG with sharing using the tree datatype. Figure 4.2 depicts valid arguments[2] to incnode with and without sharing: a tree and a DAG (the one constructed in the above let expression). A naive parallel version of incnode that simply evaluates the expressions (incnode left) and (incnode right) concurrently without coordinating leaf-node accesses cannot ensure correct result values for leaf nodes. This is because concurrent get and set operations to shared structure may produce indeterminate values in leaves. With naive parallel evaluation, for example, incnode applied to the O node in the DAG

[1] Although it may be the programmer's intention to only construct trees with the tree datatype, a compiler must consider *all* structures that a datatype can produce. Hendren [47] and Hummel *et al.* [55] describe programmer annotations for dynamic datatype definitions that express such intent to the compiler.

[2] Cyclic structures cannot be arguments to incnode since it is impossible to introduce a cycle into a structure of type int ref tree.

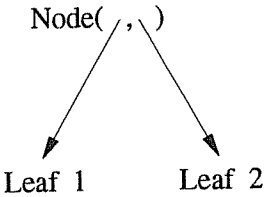
of Figure 4.2 may produce indeterminate results since expressions can concurrently modify the same reference values—the reference values in the leaves accessible from the ● node.

Even when a data structure contains sharing, it is still possible to (dynamically) discover and utilize parallelism in expressions that access portions of the structure that are not shared; *e.g.*, `incnode` can safely modify the leaves of disjoint trees that are subgraphs *within* a DAG (such as the structure below the ● node in Figure 4.2). Since static methods that approximate the structure of a program's dynamic data can, in general, only do so imprecisely, it is possible to design a program using `incnode` that a given static technique cannot parallelize: `incnode` applied to a DAG whose size and shape (*i.e.*, connectivity) exceeds the static technique's limit of precise approximation (see §4.9). As another example of such a program, consider the tree and DAG of Figure 4.2 both reaching an application of `incnode` via a conditional whose predicate is statically unknown—in this case, static techniques forego parallelism in `incnode` since they must conservatively approximate `incnode`'s argument as always containing sharing. By the definition of *necessity* (§1.1.2), therefore, dynamic methods such as dynamic resolution are necessary for the parallelization of imperative expressions that modify shared dynamic data.

This chapter develops dynamic resolution for the extended λ_v-S language. After preliminary definitions and notation (§4.1), I describe the underlying idea (§4.2.2). I then describe **dr**'s static (§4.3) and dynamic (§4.4) components, present extensions (§4.5) and an example of **dr** operation (§4.6), and discuss the implementation (§4.7) and empirical results (§4.8).

4.1 Preliminaries

Datatype constructors build *dynamic values*; that is, the reference values, tuples, and recursive data structures created with (non-nullary) data constructors are values that reside in dynamically-allocated storage in the program's *heap*. Denote the heap as $\mathcal{H}$. Implementations represent a program's dynamic values as *nodes* in $\mathcal{H}$. A node $h \in \mathcal{H}$, representing a dynamically-allocated value, contains basic values directly (*e.g.*, integers) and *links* to other nodes in $\mathcal{H}$. For example, the expression `Node(Leaf 1,Leaf 2)` of the `tree` datatype (Figure 4.1) creates the structure



in $\mathcal{H}$ that consists of three nodes and two links. The heap $\mathcal{H}$ is a directed graph with nodes as its vertices and links as its edges. A node h's in-degree, *in-degree*(h), is the number of links incident on h.

Definition 1 (Simple Node) *A node $h \in \mathcal{H}$ is a* simple *node if in-degree(h) ≤ 1.*

Definition 2 (Join Node) *A node $h \in \mathcal{H}$ is a* join *node if in-degree(h) > 1.*

Join nodes, as we shall see, serve as indicators of potentially-shared dynamic data.

Definition 3 (Path) *A* path *of length n in $\mathcal{H}$ is a sequence of nodes, $\langle h_1, \ldots, h_n \rangle \in \mathcal{H}$ where $n \geq 1$, such that $\forall i$, $1 \leq i < n$, there exists a link from h_i to h_{i+1}.*

Denote the existence of a path from $h \in \mathcal{H}$ to $h' \in \mathcal{H}$ as $h \Longrightarrow h'$. The nonexistence of a path from h to h' is noted $h \not\Longrightarrow h'$. If $h \Longrightarrow h'$, then node h is said to *reach* node h'.

Definition 4 (Simple Path) *A* simple path *of length n in $\mathcal{H}$ is a sequence of nodes, $\langle h_1, \ldots, h_n \rangle \in \mathcal{H}$ where $n \geq 1$, such that $\forall i$, $1 \leq i < n$, there exists a link from h_i to h_{i+1}, and $\forall i$, $1 \leq i \leq n$, node h_i is simple.*

Denote the existence of an simple path from $h \in \mathcal{H}$ to $h' \in \mathcal{H}$ as $h \longrightarrow h'$. The notation $h \not\longrightarrow h'$ denotes that no such path exists. If $h \longrightarrow h'$, then node h is said to *simply reach* node h'.

The relations $\Longrightarrow$, $\not\Longrightarrow$, $\longrightarrow$, and $\not\longrightarrow$ collectively comprise the *reaching relations* for nodes.

Definition 5 (Acyclic Node) *An* acyclic node *is a node $h \in \mathcal{H}$ such that all paths from h to h are of length* 1.

That is, h is acyclic when it does not lie on a cycle in $\mathcal{H}$. Dynamic resolution's static component determines when a dynamic value is always represented by an acyclic node.

Identification of the free variables (§2.2.1) of an expression that can bind dynamic values or functions will also be necessary. The *free dynamic variables* of an expression e are:

$$\mathrm{FDV}(e) = \{\, x \in \mathrm{FV}(e) \mid x \text{ can bind a dynamic value} \,\}$$

An identifier's type indicates whether it can bind dynamic values. The *free function variables* of an expression e are:

$$\mathrm{FFV}(e) = \{\, f \in \mathrm{FV}(e) \mid f \text{ can have type } \tau \rightarrow \tau' \}$$

That is, a free variable f in e is a free function variable if it can be used as a function (*i.e.*, can be applied). Finally, characterize a function f as *true* if the dynamic values accessible in f are either created in f or are parameters to f. Otherwise, f is *untrue*.

Definition 6 (True Function) *A recursive function $f \equiv (\lambda x.e)$ is a* true *function if $FDV(f) = \emptyset$ and if $\forall g \in FFV(f) \backslash \{f\}$ the function g is a true function.*

That is, f is a true function when f does not contain free dynamic variables and does not apply free functions that contain free dynamic variables. For example, in the function definition

```
fun f (x::xs) =
    let fun g y = (y+1)::xs
    in
        g x
    end
```

`f` is a true function because $\mathrm{FDV}(f) = \emptyset$ and $\mathrm{FFV}(f) = \{::, +\}$, where f denotes the λ-abstraction bound to `f`. The infix list constructor (`::`) and integer addition (`+`) are true functions. The function `g` is an untrue function since it accesses the dynamic value bound to `xs` (*i.e.*, $\mathrm{FDV}(g) = \{\texttt{xs}\}$, where g is the λ-abstraction bound to `g`). Other examples of true functions are `incnode` (Figure 4.1), `map` (Figure 3.1), `compose` (Figure 3.6), and `qs` (Figure 3.12).

4.2 Overview

In this section I briefly state the idea and the property that underlie dynamic resolution.

4.2.1 The Idea

To safely evaluate two expressions that update a dynamic data structure (*e.g.*, a DAG) in parallel, it is necessary to identify the dynamic data that is potentially reachable by both expressions, and to correctly coordinate the accesses to this data. Initially, evaluation of the two expressions can proceed in parallel. Upon detection of an access to shared data, however, all further evaluation occurs sequentially; *i.e.*, one expression must suspend on an access to shared data and may not restart until the other completes. Suspending one expression on access to shared data is a means of preserving the language's sequential semantics. Note that in the absence of shared data, dynamic resolution will completely evaluate both expressions in parallel. The detection of shared data and the coordination of the accesses to this data (*i.e.*, deciding which expression to suspend) occurs dynamically. A dynamic-resolution compiler automatically inserts code into the program text that detects potential sharing at run time; the **dr** run-time system decides which expressions may access shared data. Static analysis is used to select, for parallel evaluation, expressions whose shared reachable data can always be detected at run time. This analysis relies on the following property.

4.2.2 Dynamic Resolution Property

This section states the property that forms the basis of dynamic resolution. The property concerns paths and nodes, and enables the static selection of program expressions for which all shared data can be detected dynamically.

Property 1 *Let h, h' be nodes in the heap $\mathcal{H}$. If $h \not\rightarrow h'$ and $h' \not\rightarrow h$, then for all nodes $h'' \in \mathcal{H}$ such that $h \Longrightarrow h''$ and $h' \Longrightarrow h''$, the following relations hold: $h \not\rightarrow h''$ and $h' \not\rightarrow h''$.*

That is, if all paths from h to h' and from h' to h contain a join node, then all paths from h or h' to any shared node h'' (accessible from both h and h') must contain a join node.

Figure 4.3 illustrates the above property. If it is known that node h cannot simply reach h' (and vice versa), then all shared structure reachable from h and h' is always delimited by a join node (node a in the diagram). Note that simple nodes (*e.g*, node b) as well as join nodes may be shared; however, evaluation of an expression will always traverse a join node before encountering a shared simple node, thereby providing a means for detecting sharing dynamically.

Statically, dynamic resolution locates program identifiers that always bind nodes $h, h' \in \mathcal{H}$ such that the above property ($h \not\rightarrow h' \wedge h' \not\rightarrow h$) holds. Suppose that the only dynamic values accessible to expression e are those reachable from h. Similarly, suppose that the only dynamic values accessible to expression e' are those reachable from h'. Furthermore, assume e and e' are candidates for parallel evaluation, but potentially conflict (read/write or write/write conflicts). If the sequential semantics requires evaluation of e before e', then e and e' may be safely evaluated in parallel with the following restriction: e' may not access any join node until e completes (e, however, may access all—join or simple—nodes that it can reach). When e and e' do not share structure (*e.g.*, $\nexists h'' \in \mathcal{H}$ such that $h \Longrightarrow h'' \wedge h' \Longrightarrow h''$)

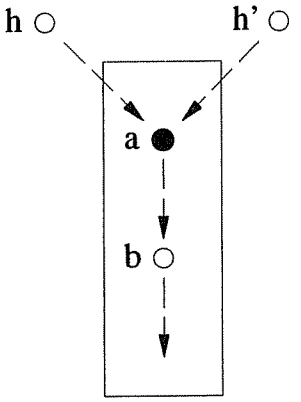


Figure 4.3: The nonexistence of simple paths from nodes h to h' and from h' to h imply that the shared structure reachable from h and h' (boxed region) is always guarded by a join node (node a). Dynamic resolution detects potentially sharing at run time by detecting join nodes.

then it is possible for e and e' to be evaluated in parallel with dynamic resolution. Otherwise, evaluation of e' must suspend upon access to a join node—a node potentially shared with e—until e's evaluation completes. Note that in the presence of sharing, only *some* of the evaluation of e' may be concurrent with that of e.

Dynamic resolution's static component identifies program identifiers that satisfy the conditions of the above property, and uses this information to select expressions for parallel evaluation. The dynamic component detects join nodes and dynamically schedules (suspends and restarts) expressions as necessary.

4.3 Static Component

Informally, the goal of dynamic resolution's static component is to find two expressions e and e' whose safe parallel evaluation is impeded by `set` operations to dynamic data potentially shared by both expressions. The static component ensures that all shared nodes reachable by e and e' can be detected dynamically. That is, it infers when the nodes bound to the free dynamic variables of e cannot simply reach the nodes bound to the free dynamic variables of e'. The **dr** property (§4.2.2) now holds. For such expressions, access to shared data can be detected and correctly coordinated at run time.

Static **dr** parallelization occurs at the function level. For a function f, the static component first identifies the data constructors in f's patterns[3] that always (dynamically) bind acyclic nodes (§4.1). Static classification of a datatype constructor as acyclic (*i.e.*, it only matches acyclic nodes) enables—in turn—static inference of the reaching relations among a pattern's variables. In particular, static classification of a data constructor as acyclic allows the static inference (§4.3.2) of strong (*i.e.*, $\not\rightarrow$) reaching relations among the constructor's variables. Such reaching relations permit **dr** parallelization

[3]Patterns (§2.2.3) match dynamic values against datatype constructors, constants, and variables. A pattern gives information about the reaching relations among its variables; it is a *positional* notation that notates the position of a pattern's variable with respect to the pattern's other variables and constructors.

because shared structure accessible from these variables can be dynamically detected by **dr**'s dynamic component (§4.4). Given such reaching relations, expressions are statically selected and restructured (§4.3.3) for concurrent **dr** evaluation. Finally, the static component places checks into the program that examine a node's status (join or simple) in expressions that can access its contents (§4.3.4).[4]

I first describe how to statically determine whether a data constructor in a pattern matches only acyclic nodes, and then how to use this information to infer the reaching relations among a function's variables. Lastly, I describe how to select candidate **dr** expressions and where, in the program text, to place the checks that sharing.

4.3.1 Data-Constructor Classification

A **dr** compiler must statically classify data constructors in patterns as cyclic or acyclic depending on whether the nodes that the constructor dynamically matches can lie on cyclic structures in the heap. Acyclic constructors admit **dr** parallelization; cyclic constructors inhibit **dr** parallelization because the shared structure reachable from a cyclic constructor's variables can not always be dynamically detected. For simplicity, I assume all patterns in the program contain at most one data constructor—this restriction is relaxed below (§4.3.2). The form of such a pattern is

$$p \equiv C(x_1, \ldots, x_n)$$

where C is a data constructor and the x_i, $0 < i \leq n$, are variables[5] that are bound when p is matched. For example, the pattern `Node(left,right)` of the `tree` datatype (Figure 4.1) contains the data constructor `Node` and variables `left` and `right`.

For a pattern p of the form above, **dr**'s static component classifies p's constructor C as cyclic or acyclic. I describe two possible methods of attaining this classification: from static type information and with programmer-supplied assertions.

From Static Type Information

Identification of a datatype constructor in pattern p as acyclic is often possible from p's type. In a call-by-value language, cyclic data structures arise only from the re-assignment of a reference value that resides in a dynamic data structure. Furthermore, to introduce a cycle, the contents of this reference value must be a dynamic value; *i.e.*, the reference value must have a dynamic-value type. A pattern's type, therefore, indicates whether the data it can match contains reference values. Hence, type information can identify a pattern's constructors that always match acyclic nodes.

For example, the pattern $p \equiv$ `(Node(left,right))` in the `incnode` function (Figure 4.1) has type `int ref tree`. Pattern p's variables (`left` and `right`) also have type `int ref tree`. This type information insures that p always dynamically matches an acyclic node in the heap (*i.e.*, p is acyclic) since the reference values in a structure of p's type can only contain integers.

---

[4]The contents of a node can be accessed only by matching (*deconstructing*) it in a pattern.

[5]Since the language's constants (*e.g.*, integer and Boolean constants) are not dynamic values, they cannot reach shared data and hence require no special treatment. Therefore, they need not be explicitly discussed here.

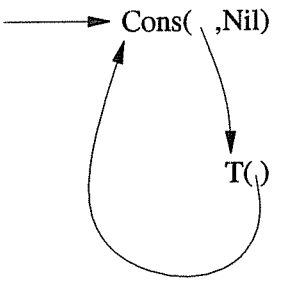


Figure 4.4: A cyclic list constructed with the conventional Cons constructor.

From Programmer-supplied Assertions

In a language with polymorphic datatypes (as here), static determination of whether a constructor only builds acyclic nodes is not always possible. Constructors in patterns that cannot be classified as acyclic inhibit parallelization with dynamic resolution because the compiler will not be able to infer strong (*i.e.*, $\not\rightarrow$) reaching relations for the variables of cyclic constructors (§4.3.2 below). For example, the Cons constructor of the conventional `list` datatype declaration

```
datatype α list = Nil | Cons of (α * α list)
```

can create cyclic nodes. The program

```
datatype t = T of t list ref | S

let val x = T (ref [S])
    val (T y) = x
in
    set y [x];
    get y
end
```

returns a list l whose single element (of type `t`) contains a reference value with contents l (*e.g.*, the list in Figure 4.4). A compiler cannot generally infer that the list Cons constructor matches acyclic nodes. For example, l is a valid argument to the `map` function (Figure 3.1)—accordingly, `map`'s pattern does not contain acyclic constructors, and dynamic resolution cannot parallelize the `map` function.

A programmer-supplied assertion can be used to identify acyclic constructors in the presence of polymorphism. Programmers are typically aware of cyclic data since precautions must be taken when traversing it—lists, tuples, trees, and DAGs can often be identified as acyclic by the programmer. I introduce the `acyclic` datatype qualifier for programmer assertion that a datatype's constructors are used only to create acyclic nodes.

Declaration of the acyclic list datatype

```
acyclic datatype α list' = Nil' | Cons' of (α * α list')
```

states that the list nodes constructed with the (acyclic) Cons' constructor will not lie on a cycle in the heap. This restricts the spine of a list thus constructed from containing cycle nodes. Elements of an acyclic list, however, may be cyclic structures; elements may also share structure (Figure 4.5). The list

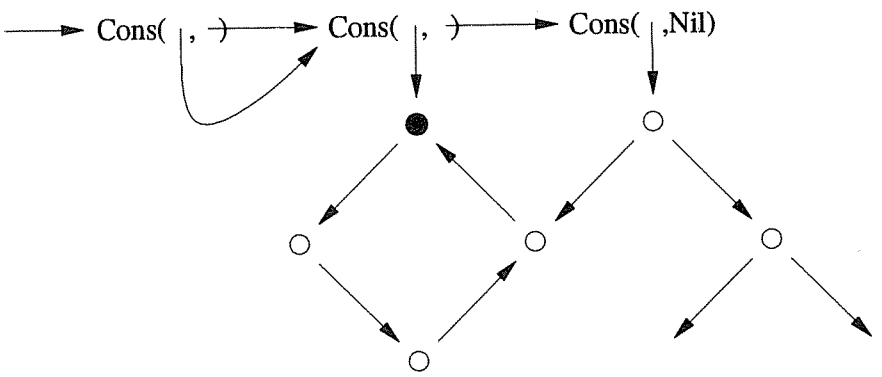


Figure 4.5: An acyclic list. An element of an acyclic list may reach tail elements; list elements may themselves be cyclic structures; and multiple list elements may reach shared structure.

```
acyclic datatype αlist' = Nil' | Cons' of (α * αlist')

fun map' f Nil' = Nil'
  | map' f (Cons'(x,xs)) = Cons'(f x,map' f xs)
```

Figure 4.6: The `map'` function for acyclic lists.

of Figure 4.4, however, is not a valid acyclic list since it violates the declaration of acyclic. Note that the compiler does not (statically or dynamically) detect such violations; incorrect usage of an `acyclic` datatype can cause indeterminate program behavior.

The function `map'` of Figure 4.6 is an acyclic version of `map` that may only be applied to lists of α`list'` type. The dynamic-resolution technique applies here because `Cons'` may only bind acyclic nodes. Hence, the compiler can infer strong reaching relations for its variables (`x` $\not\rightarrow$ `xs` and `x` $\not\rightarrow$ `xs`). Even if the higher-order parameter `f` performs imperative `get` and `set` operations (*cf.* §4.6), this can sometimes permit multiple expressions to execute in parallel.

4.3.2 Reaching-Relation Inference

Static classification of the data constructors in patterns as acyclic allows the automatic inference of reaching relations among a pattern's variables. When data constructor C in pattern p is acyclic, the nodes dynamically bound to C's variables x_i, $0 < i \leq n$, cannot reach one another via simple paths. That is, when C is acyclic, the compiler can safely infer that $x_j \not\rightarrow x_k$ for all pairs of C's variables x_j and x_k, where $0 < j,k \leq n$ and $j \neq k$. Proof of this follows. Let $h, h' \in \mathcal{H}$ denote the nodes bound to two of C's variables x_j and x_k (where $0 < j,k \leq n$ and $j \neq k$) when p matches dynamically. When h and h' are the same node ($h = h'$) then h (and h') are join nodes due to the two links from C's node. Alternately, when $h \neq h'$ a simple path cannot exist from h to h' ($h \not\rightarrow h'$). Suppose a simple path from h to h' exists. Node h' then has at least two links: one from C's node and one from the node preceding h' on the path from h to h' (this path cannot pass through C's node since C's node is acyclic; hence this path cannot use links from C's node). Since h' has at least two incident links, it must be a join

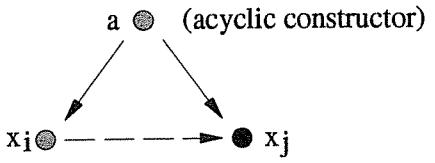


Figure 4.7: Node a is an acyclic data-constructor node. Nodes x_i and x_j are directly reachable—via a single link—from a. Any path from x_i to x_j is not simple because it always contains a join node (x_j). Such a path cannot use the link from a to x_j since a is acyclic. Black nodes are join nodes; gray nodes represent any (simple or join) node.

node. This, however, contradicts the supposition. Therefore, a simple path cannot exist from h to h'. Similarly, a simple path cannot exist from h' to h ($h' \not\rightarrow h$).

Figure 4.7 depicts the relationship between an acyclic node a (corresponding to an acyclic constructor) and the nodes x_i and x_j directly accessible from a. If x_i can reach x_j via any path, then that path must contain a join node (x_j). Since the constructor node a is acyclic, the path from x_i to x_j cannot pass through a and hence cannot include the link from a to x_j.

Reaching relations that assert the nonexistence of simple paths between pattern variables enable dynamic resolution—sharing in the structure bound to these variables can be detected at run time because a join node is always encountered before an expression reaches any shared structure.

In Section 4.3.1 the program's patterns were restricted to contain at most one data constructor. Relaxing this restriction is straightforward, and doing so admits nested data constructors in patterns. Without loss of generality, if the constructors C and C' in the pattern

$$p \equiv C(x_1, \ldots, x_n \text{ as } C'(y_1, \ldots, y_m))$$

are acyclic, the reaching relations

$$\begin{array}{ll} x_j \not\rightarrow x_k & 0 < j,k \leq n \ \wedge \ j \neq k \\ x_j \not\rightarrow y_k & 0 < j < n \ \wedge \ 0 < k \leq m \\ x_n \Longrightarrow y_k & 0 < k \leq m \end{array}$$

can be inferred. Any path from a variable x_j to a variable y_k cannot be simple because C is acyclic; however, a simple path can exist from variable x_n to a variable y_k because the nodes (dynamically) corresponding to the constructors C, C', and to variables y_k may all be simple. This occurs, for example, when p matches an unshared tree.

4.3.3 Expression Selection

Static analysis propagates the reaching relations induced by a pattern into the pattern's scope. Static selection of expressions for parallelization with dynamic resolution then commences.

Two expressions, e and e', whose safe parallel evaluation is constrained only by **read/write** or **write/write** conflicts (§2.3), are candidates for parallel evaluation using dynamic resolution if they meet three criteria:

```
fun incnodeDynamic (Leaf x) = (set x (1 + (get x)))
  | incnodeDynamic (Node(left,right)) = (incnodeDynamic left ;||dr
                                         incnodeDynamic right)
```

Figure 4.8: The incnode function after expression selection. The separator $;_{||_{dr}}$ indicates that (incnodeDynamic left) may evaluate concurrently with (incnodeDynamic right) provided that accesses to shared data are dynamically detected and coordinated.

1. $\forall x \in \text{FDV}(e), \forall x' \in \text{FDV}(e')$ the relations $x \not\rightarrow x'$ and $x' \not\rightarrow x$ hold.
2. $\forall f \in \text{FFV}(e)$, f is a true function; and $\forall f' \in \text{FFV}(e')$, f' is a true function.
3. $\forall x \in \text{FDV}(e)$, x does not contain untrue functions; and $\forall x' \in \text{FDV}(e')$, x' does not contain untrue functions.

The first criterion requires that all dynamic values bound to the free variables in e cannot reach, via a simple path, dynamic values bound to the free variables in e'. It thereby ensures that all shared data accessible to both e and e' can be detected dynamically (§4.2.2 and §4.4). The second criterion restricts the functions in e and e' to not have access, through their free variables, to dynamic values other than those available in e and e' (due to the first criterion). The last criterion requires e and e' to not apply untrue functions contained in their accessible dynamic data; it prohibits access to (arbitrary) dynamic values through the free variables of higher-order (untrue) functions stored in dynamic data. A free dynamic variable's type indicates whether structure bound to it can contain functions.

The `incnode` function contains two expressions that can safely evaluate concurrently using dynamic resolution: $e \equiv$ `incnode left` and $e' \equiv$ `incnode right`. The pattern $p \equiv$ `(Node(left,right))` in `incnode` induces the set {`left` $\not\rightarrow$ `right`, `right` $\not\rightarrow$ `left`} of reaching relations for p's corresponding function body. Thus, since $\text{FDV}(e) = \{\texttt{left}\}$ and $\text{FDV}(e') = \{\texttt{right}\}$, expressions e and e' meet the first criterion. Furthermore, since e and e' do not apply untrue functions (`incnode` is a true function) and do not have access to data containing untrue functions (`left` and `right` cannot contain functions), expressions e and e' also meet the second and third criteria. The `incnodeDynamic` function of Figure 4.8 reflects the selection of (`incnode left`) and (`incnode right`) for parallel evaluation *provided* that all shared data is dynamically detected and accesses to this data dynamically coordinated. This detection and coordination is performed by dynamic resolution's dynamic component (§4.4). The sequence separator $;_{||_{dr}}$ specifies concurrent evaluation, requiring sharing detection, of the expressions it separates.

4.3.4 Check Placement

The last responsibility of dynamic resolution's static component is the identification, in the program text, of all accesses to nodes so that sharing can be dynamically detected. In particular, a check to determine if a node is a join node (and hence potentially accessible to other concurrent expressions) is placed immediately before a datum is deconstructed when it matches a datatype constructor (both cyclic and acyclic) in a pattern. Recall that only patterns can deconstruct (access) a dynamic value's contents.

```
fun incnodeDynamic (Leaf x) = (set x (1 + (get x)))
  | incnodeDynamic (Node(left,right)) = (incnodeDynamic left ;||dr
                                         incnodeDynamic right)
```

Figure 4.9: The final incnodeDynamic function after identification of the data accesses in patterns that require sharing checks. An overlined constructor denotes that such a check occurs before accesses to the constructor's components may commence.

Placing a check on every dynamic-value access ensures that sharing (*i.e.*, join nodes) can be dynamically detected along any path in the dynamic data that the program follows. These checks examine the status (join or simple) of the node matching the constructor. Figure 4.9 contains the dynamically-parallel version of incnode with these constructors identified. The (de)constructor Leaf checks the status of the nodes it matches before it accesses x. The result (join or simple) of this check governs the program's subsequent behavior; the full dynamic operation of these checks is discussed below (4.4.3).

4.4 Dynamic Component

Dynamic resolution's dynamic component detects join nodes in the heap at run time. It also maintains a *total order* of all concurrently-evaluating expressions that reflects the evaluation order required by the language's sequential semantics. An expression is dynamically scheduled for concurrent evaluation as a *thread.* Before access to potentially-shared data, an expression examines its position in the total order of threads to determine whether it may access the data or must wait for the evaluation of other expressions (threads earlier in the order) to complete.

4.4.1 Join-Node Detection

Reference counts are used to dynamically distinguish join nodes from simple nodes. The reference count of a node h counts the number of links incident on h—thereby, reference counts reveal information about the heap's structure. A pointer to a node h (*e.g.*, a variable bound to h) is not included in h's reference count because it does not reveal information about the connectivity of the data structure in which h resides.[6] A node with a reference count of zero[7] or one is simple; a node with reference count > 1 is a join node. A join node is an indicator of potential sharing because concurrent threads may potentially access the same nodes from a join node. Therefore, coordination of accesses to join nodes is necessary to preserve the program's desired semantics.

If a thread has access to a simple node, no other thread has concurrent access to this node. Expression selection (§4.3.3), in cooperation with **dr**'s dynamic component, establishes this invariant. Recall that the static component selects expressions e and e' for parallel evaluation using dynamic resolution only

[6]The reference-counting scheme required by dynamic resolution is similar to that used for Deutsch-Bobrow *deferred reference counting* [28].

[7]A program can have access to a node with a reference count of zero through pointers (*e.g.*, from local variables) to that node since they are not included in the node's reference count.

when the evaluation of e and that of e' will always encounter a join node before reaching shared data accessible to either expression.

Building new data (*e.g.*, *consing* an element onto a list) increments reference counts. An assignment to a reference value increments the count of the (dynamic) value being assigned. Assignment decrements the count of the (dynamic) value being overwritten with the following proviso: reference counts are *sticky*—a reference count of two never changes. That is, a join node never becomes simple. Sticky reference counts circumvent the following problem: Suppose an expression e makes a local binding to the contents v of a dynamic reference value r and then reassigns r's value. This would violate the invariant that a simple node is accessible to at most one concurrent thread. This is because the thread evaluating e has access to v (through the local binding) and—if reference counts are not sticky—another thread may now also have (uncoordinated) access to v since the assignment to r removes a link to v and can therefore make v simple. For example, if reference counts do not stick, then in the expression

```
let val (ref y) = x
in
    set x z;
    y
end
```

the reference count on the node bound to `y` may drop to one (making it simple and accessible) since the link to `y` from the reference value bound to `x` is removed (`set x z`); yet the thread evaluating the `let` expression still has access to `y`. A concurrent thread, however, may encounter and access `y`'s simple node—this, in turn, may produce indeterminate behavior. Not decrementing reference counts that are > 1 prevents a thread from inadvertently granting a concurrent thread access to its simple nodes.

Atomicity during reference-count increment and decrement operations is *not* necessary. This is because of the invariant that simple nodes are not concurrently accessible. Therefore, incrementing or decrementing the reference count of a simple node needs no synchronization. Since the reference counts of join nodes are never decremented[8] (*i.e.*, join nodes never become simple), changing the reference count on a join node also requires no synchronization.

Section 4.5.3 describes a method for reconstituting a node's reference count that has become imprecise (stuck at two).

4.4.2 Parallel-Thread Linearization

Dynamic resolution's run-time system imposes a total order on the program's concurrently evaluating expressions (threads). A doubly-linked list of *thread descriptors* forms a *linearization* that implements this order on threads. This linearization dictates which thread may access join nodes and which threads must suspend on access to join nodes. The first thread in the linearization (the *head* thread) may access all nodes whereas threads later in the linearization must suspend on access to join nodes. In this manner, the parallel evaluation of expressions that perform imperative operations adheres to the language's sequential semantics.

[8] Note that a reference count that is *greater* than the actual number of links incident on a node is conservative—such a count may indicate sharing where none exists, but it cannot admit uncoordinated access to a join node.

```
thread_descriptor ≡ {
    inuse : lock
    head : bool
    done : bool
    suspended : bool
    suspension : continuation
    next : thread_descriptor pointer
    prev : thread_descriptor pointer
}
```

Figure 4.10: A thread descriptor. Descriptors form a doubly-linked list that is used to dynamically order shared-data accesses.

The run time associates a thread descriptor with every **dr** thread. Figure 4.10 contains the record fields of such a descriptor. The (single) head thread is identified by a *true* `head` field. For a given thread t in the order, threads later than t in the linearization are accessible through t's `next` field; prior threads are accessible via t's `prev` field. Threads are inserted into the linearization as follows. A thread t evaluating the expression $e \equiv (e_1 \;||_{dr}\; e_2)$ creates a new thread t_2 to evaluate e_2; thread t continues and evaluates e_1. Thread t_2's descriptor is inserted directly behind thread t's descriptor in the linearization. Upon creation, a thread descriptor's `done` and `suspended` fields are initialized to *false*.

The linearization is a concurrent structure—insertions and deletions of (non-adjacent) thread descriptors occur in parallel. As such, the linearization does not sequentialize the program. A descriptor t's `inuse` field is a spin lock that must be held by a thread wishing to inspect or change t's `next` and `prev` fields (*e.g.*, for inserting a child thread into the linearization). The `head`, `done`, `suspended`, and `suspension` fields are used for dynamically scheduling a thread.

4.4.3 Expression Scheduling

The head thread in the linearization may freely access any node (join or simple) that it can reach. Non-head threads later in the linearization, however, must suspend on access to a join node since it—and all nodes accessible from it—are potentially shared with other concurrent threads. A thread t may not access a join node until t is the head thread; *i.e.*, until all prior threads have completed. To suspend itself, a thread stores the *continuation* (*cf.* [117]) of its computation in the descriptor's `suspension` field and sets the `suspended` field to *true*.[9] A non-head thread that completes without accessing a join node (*i.e.*, without suspending) sets its descriptor's `done` field to *true*. When the head thread completes, the next uncompleted (`done` = *false*) thread in the linearization becomes the head thread. If this thread is suspended, it is restarted and may now access any join node it can reach—if it is computing, it continues to do so. Since the head thread always makes progress, deadlock cannot occur.

[9]The processor that suspends a thread proceeds to evaluate the expressions of other (non-suspended) threads.

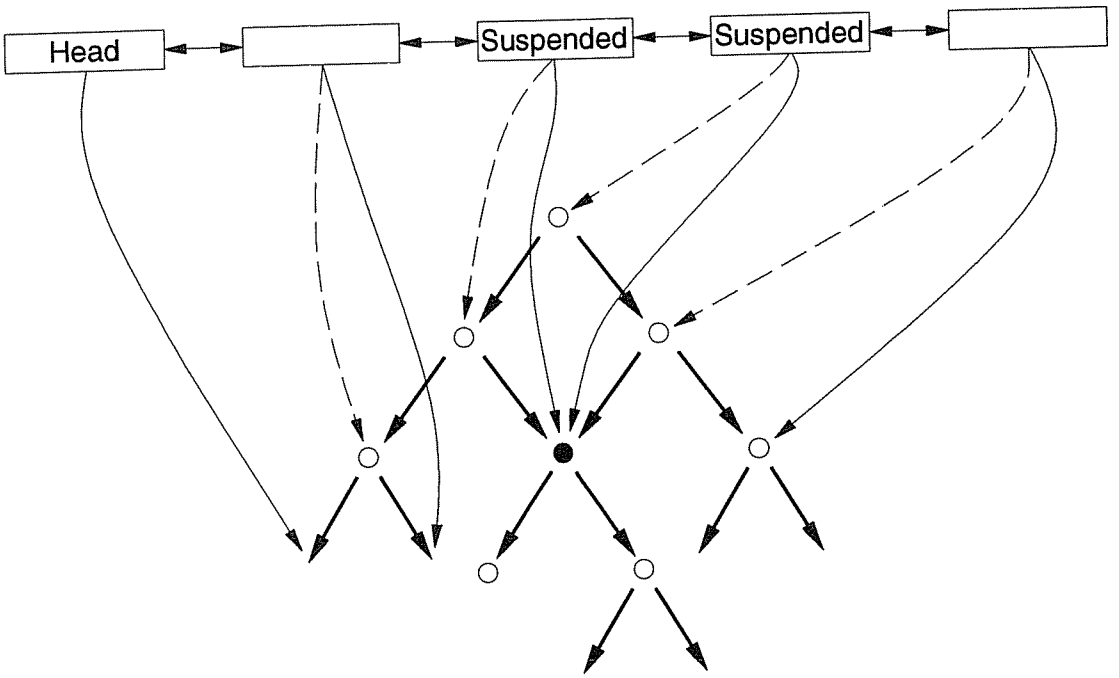


Figure 4.11: Operation of the thread linearization during resolution of incnode applied to a DAG.

This scheduling scheme preserves the language's sequential semantics because, in an expression $e \equiv (e_1 \,||_{dr}\, e_2)$, it allows e_1 (and threads created by e_1) to access all data potentially shared with e_2 (and threads created by e_2) *before* it allows e_2 access to this data. In the absence of sharing, e_1 and e_2 completely evaluate concurrently under dynamic resolution.

Figure 4.11 depicts dynamic resolution of an application of the `incnode` function (Figure 4.9) applied to a DAG. Straight uni-directional arrows (⟶), join nodes (●), and simple nodes (○) constitute `incnode`'s DAG argument. The boxes are thread descriptors in the linearization. Labels in thread descriptors indicate the head thread and suspended threads. In the linearization, bi-directional arrows (⟷) represent the `next-prev` link between adjacent descriptors. Curved solid arrows (⌒⟶) emanating from descriptors point to the node which the thread's expression is accessing. Curved dashed arrows (⌒⟶) emanating from descriptors designate the node at which the descriptor's thread was created.

In Figure 4.11, the thread currently at the head of the linearization is the thread that initially applied `incnode`. Therefore, no arrow to its point of creation is shown. Note that the head thread created threads at all nodes on the path from the DAG's root to its current evaluation point; *e.g.*, the computing thread immediately to the right of the head thread was created (and its descriptor inserted into the linearization) when the head thread encountered the DAG's left-most simple node. Two threads are suspended: the thread accessing the DAG's (only) join node from the left and the thread accessing this join node from the right. They will be restarted when all the threads before them in the linearization complete. Finally, note that the last thread t in the linearization will imminently create a new thread

for the evaluation of incnode applied to the right child of t's current node—the thread descriptor for this new thread will be inserted at the tail of the linearization.

4.5 Extensions

Several extensions to dynamic resolution can potentially improve its performance and precision (*i.e.*, the amount of parallelism it finds).

4.5.1 Specialized Function Versions

With dynamic resolution, deconstructing a pattern p upon its successful match to a dynamic value (node) incurs the additional cost of examining the matched node to determine whether it is a join or a simple node. As described, dynamic resolution always incurs this cost even when no **dr** parallelism exists. To curtail this expense, generate two versions[10] of a program function f: f_{seq} and f_{dr}. Version f_{seq} is the conventional sequential version of f. Version f_{dr} is dynamically parallel and contains checks to nodes as required by dynamic resolution. Functions applied by the f_{dr} version of f must themselves be dynamically parallel. Only when parallel **dr** threads are present need the dynamic versions of functions be used.

4.5.2 Head-Thread Optimization

Given sequential and dynamically-parallel function versions, an additional optimization is possible. Since the head thread in the linearization may unconditionally access any node, it never needs to check whether a node is a join node or a simple node. Therefore, the head thread may safely use the sequential code that does not examine node reference counts—it is important that the head thread evaluate quickly since suspended threads in the linearization are awaiting its completion. Non-head threads, however, must still check reference counts and suspend their evaluation on access to join nodes.

4.5.3 Reconstitution of Reference Counts

Reference counts on nodes become inaccurate for two reasons: (1) when a node becomes a join node it remains a join node (reference counts stick at two), although the actual count of the node's incident links may be less than two; and (2) much of the program's dynamic data is temporary and quickly becomes inaccessible to the program, but links from this inaccessible data are still reflected in the reference counts of accessible data. Imprecise reference counts restrict parallelization with dynamic resolution because they can (falsely) indicate sharing where none exists.

It is possible to periodically *reconstitute* a node's reference count to its actual value (*cf.* [120]). A language implementation's *garbage collector* ([34, 60]) reclaims and recycles the program's discarded data; it is a natural place in an implementation for performing reference-count reconstitution. Here, I assume a *copying* collector (*e.g.*, [14, 33, 9] and Chapter 6) that, when the program exhausts its heap storage, makes a copy of all the live data accessible to the program. When complete, this copy replaces

[10]The PARCEL system also creates multiple, specialized function versions [45].

the program's heap. Storage occupied by the original data—both accessible and inaccessible—is now reclaimed for reuse.

Reference count reconstitution works as follows. A *pointer* (*e.g.*, a node bound to a program variable as opposed to a link in the heap) held by the program to an uncopied node causes the node to be copied. This copy is given a reference count of zero. A *link* to an uncopied node also causes the node to be copied; however, the initial reference count of a copy initiated by a link is one. The reference count in this copy is one due to the single link that initially caused it to be copied (other links to the node have not yet been encountered; if a link had previously been encountered, a copy of the node would already exist). When a link to a previously-copied node is encountered, the reference count in the node's copy is incremented. Since the reference counts required by dynamic resolution are sticky, reconstitution need not increment reference counts past two.

Note that this method of reference-count reconstitution is valid only during a sequential phase in the program; *i.e.*, when no parallel **dr** threads exist. This restriction is necessary because of the problem described in Section 4.4.1: a thread may not make a node h simple if it has a binding (pointer) to h since bindings to h are not reflected in h's reference count—reconstitution during parallel **dr** evaluation can (incorrectly) make a node, with active pointers to it, simple.

4.6 Example

Figure 4.12 provides a further example of **dr**'s operation. The `mqs` function sorts a list of elements using the quicksort algorithm. Unlike a functional quicksort (*e.g.*, Figure 3.12), it performs the sort in place; that is, the links of the argument list's run-time representation are modified during the sort. The programmer has declared two acyclic datatypes: α`pair` and α`mlist`. The acyclic α`pair` datatype constructs binary tuples of identically-typed values. The α`mlist` datatype constructs *mutable* lists with elements of type α—the lists are mutable because their link fields (to the next list element) are reference values. With the `acyclic` declaration the programmer indicates that the `mCons` constructor is used only to create acyclic lists. The `mqs` function has type:

$$\texttt{mqs} : (\alpha \rightarrow \alpha \rightarrow \texttt{bool}) \rightarrow \alpha\ \texttt{mlist} \rightarrow \alpha\ \texttt{mlist}$$

The function's first parameter is a boolean predicate that compares elements of `mqs`'s second parameter, the mutable list to be sorted.

Dynamic resolution's static component identifies all constructors in the program's patterns (overlined in Figure 4.12). When dynamically matched, these constructors require a run-time check to the underlying node representing the datum before any access to the datum's components. This check determines whether the node is a join or simple node. Note that functions applied by `mqs` (*e.g.*, the `mAppend` function that destructively appends two mutable lists) also perform these checks.

In `mqs`, dynamic resolution finds parallelism in the concurrent evaluation of the recursive applications of `mqs` that sort the sublists produced by the auxiliary `split` function:

```
val Pair(l',g') = Pair(mqs p l,mqs p g)||dr
```

The $||_{dr}$ annotation indicates that the tuple's expressions, (`mqs p l`) and (`mqs p g`), may evaluate in parallel with dynamic resolution. By the criteria of Section 4.3.3, these expressions are candidates for **dr** evaluation: `mqs` is a true function, `l` can only reach `g` via paths that always contain a join node, and `g` can only reach `l` via paths that always contain a join node. Note that the predicate `p` must also be a

```
acyclic datatype αpair = Pair of (α * α)

acyclic datatype αmlist = mNil | mCons of (α * αmlist ref)


fun mAppend mNil y = y
  | mAppend x mNil = x
  | mAppend x y =
        let fun aux (mCons(_,r as ref mNil)) = r := y
            | aux (mCons(_,ref s)) = aux s
        in
              aux x;
              x
        end


fun mqs p mNil = mNil
  | mqs p (mCons(x,xs as ref xs')) =
        let fun split pivot l =
                let fun split' mNil less greater = Pair(less,greater)
                      | split' (l as mCons(y,ys as ref ys')) less greater =
                            if p pivot y then
                                (ys := less;
                                 split' ys' l greater)
                            else
                                (ys := greater;
                                 split' ys' less l)
                in
                    split' l mNil mNil
                end
            val _ = xs := mNil
            val Pair(l,g) = split x xs'
            val Pair(l',g') = Pair(mqs p l,mqs p g)||dr
        in
            mAppend l' (mCons(x,ref g'))
        end
```

Figure 4.12: Imperative quicksort (mqs) restructured for dynamic resolution.

true function. If it is statically unknown whether p is true, this can be determined dynamically; *e.g.*, a λ-tag (Chapter 3) can be used to carry a function's status (either true or untrue). In Figure 4.12, it is assumed that p is (statically or dynamically) known to be a true function.

Since `mqs` is polymorphic, it can sort (mutable) lists of many types, including lists whose elements are, perhaps cyclic, dynamic structures. Any sharing between elements is detected during application of the predicate p, which—as all functions in the program—detects access to shared data. If `mqs`'s list argument does not contain sharing, the recursive applications of `mqs` evaluate concurrently without suspending threads. To accomplish this, however, a further optimization is required. In `mqs`, the `Pair` constructor is used only to build temporary data—data that is inaccessible outside of `mqs`. This construction of temporary pairs, therefore, generates reference counts that (falsely) indicate sharing. Since `mqs` does not place `Pairs` in data structures, and `Pairs` are not accessible outside of `mqs`, it is safe to decrement[11] the reference counts on a pair's components upon its deconstruction. For example, in the expression

```
val Pair(l,g) = split x xs'
```

the reference counts on the nodes bound to `l` and `g` can be safely decremented after `Pair` matches. This optimization prevents temporary dynamic structures from obscuring safe parallelism.

4.7 Implementation

Dynamic resolution's static component (§4.3) and extensions (§4.5) have not been implemented. Although amenable to compiler automation, their tasks—identification of pattern constructors, inference of reaching relations, expression selection, and check placement—were performed manually for the programs tested. Dynamic resolution's dynamic component has been implemented in the SML/NJ optimizing ML compiler [11, 9].

An implementation of dynamic resolution requires reference counting of links to nodes (§4.4.1), a linearization of active parallel threads (§4.4.2), and a compiler primitive to inspect a node's reference count and, thereupon, to suspend evaluation if necessary (§4.4.3). Heap nodes in SML/NJ are created by the compiler's intermediate *record* form (*cf.* [9]). Heap nodes correspond directly to records. The compiler was modified to allocate reference-count fields on all dynamic-data nodes. The compiler's *sml2c* [110] back end was modified to increment a node h's reference count when a link to h is stored into a node and to decrement node h's reference count upon re-assignment of a link to it. The linearization of threads required to establish the sequential semantics was written in ML and implemented in SML/NJ's parallel run-time system (MP [82, 24]) as a doubly-linked list of thread descriptors. A primitive (written in C) was added to the SML/NJ compiler that creates a new thread descriptor at a given location in the linearization. This operation is time critical since it occurs every time a thread is created. In-line ML functions are used to inspect reference counts. Upon detection of access to a join node, a non-head thread t suspends by capturing its continuation (available through SML/NJ's non-standard `callcc`). This continuation is stored in t's thread descriptor and is invoked (thrown to) when t moves to the head of the linearization.

[11] As before, reference counts stick at two.

4.8 Results

The `incnode` function of Figure 4.1 contains abundant parallelism. However, dynamic resolution of `incnode`—as well as an explicitly parallel version of `incnode`—fail to reduce the function's execution time to less than that of its sequential execution time. Parallelism in `incnode` is of too fine a grain to effectively exploit on the Sequent Symmetry.[12] Therefore, I tested dynamic resolution on programs with coarser parallelism: imperative quicksort (`mqs`, Figure 4.12) and a topological sort applied to a list of trees (`topo`).

The `mqs` function was manually restructured to concurrently apply `mqs` recursively and to check reference counts on access to dynamic values (see §4.6). Figure 4.13 gives timings for sorting a list of 10000 random integers. The sequential version did not manipulate (allocate, increment, or decrement) reference counts. Dynamic resolution overhead stems from reference counting, linearizing threads, and checking for join nodes. The overhead for dynamic resolution in this program does not exceed 19% of the explicitly parallel time, an dynamic resolution already improves on sequential evaluation with only two processors.

Timings of the `topo` program sorting a forest of trees are in Figure 4.14. Programmer or compiler parallelization of this program is difficult because it is not possible to statically pinpoint where (and when) the trees share structure. The timing graph for this program therefore lacks a curve of explicit-parallel times. The program sorted 25 balanced trees of depth 13. The trees did not share structure. This program intensively accesses heap nodes, and dynamic resolution incurs significant overhead. Even so, dynamic resolution betters `topo`'s sequential execution time when more than 4 processors are available. This program was also restructured manually.

To measure the effect of sharing, the topological sort was applied to a forest of trees with the trees sharing a common leaf (at level 13). Dynamic resolution required 37.2 seconds to perform the sort with 8 processors. This is still an improvement over the sequential sort (40.5 seconds). With a shared node at the trees's sixth levels, however, the time required for the parallel sort increased to 65.0 seconds since much of the computation was sequential (due to sharing) *and* examined reference counts. These experiments indicate that dynamic resolution, in the presence of sharing, is viable only if its run-time overheads can be further reduced. Foremost, examination of reference counts must be moved to the back end of the compiler instead of being performed by an, (albeit in line) ML function. Optimizations, such as those described in Section 4.5, also stand to further improve dynamic resolution's performance.

It is interesting to note that dynamic-resolution overhead (from counting references, linearizing threads, and checking for shared data) is itself "parallel"; that is, its detrimental impact distributes over the available processors.

[12] Chapter 5 develops a technique that dynamically selects an expression for parallel evaluation only when its granularity concurs with that of the machine.

	Processors							
	1	2	3	4	5	6	7	8
Sequential	51.6	–	–	–	–	–	–	–
Explicit Parallel	59.5	37.2	30.3	24.9	21.8	20.9	19.2	18.6
Dynamic Resolution	68.6	43.1	33.1	26.5	24.2	22.3	21.8	20.4

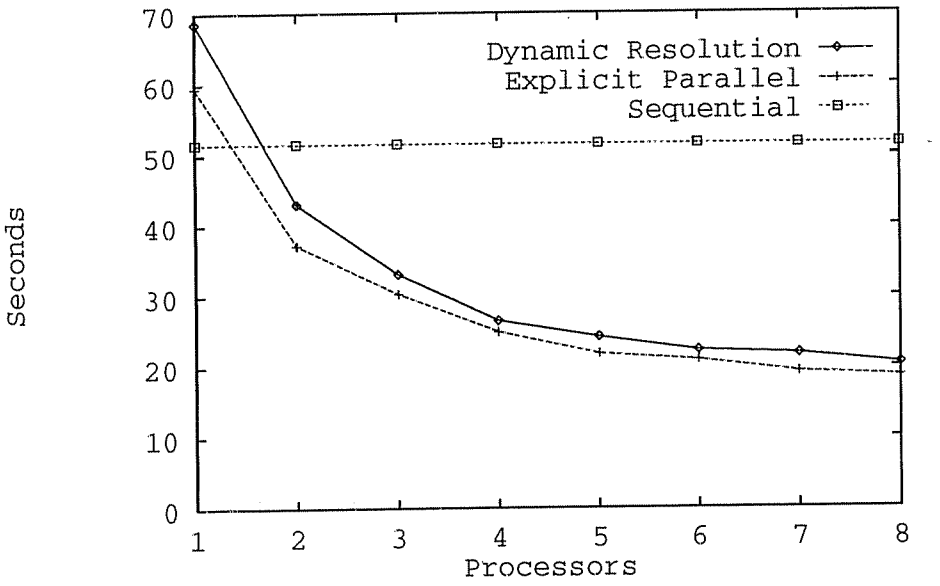


Figure 4.13: Timing results for dynamic resolution of destructive quicksort (`mqs`).

	Processors						
	1	2	4	6	8	12	16
Sequential	40.5	–	–	–	–	–	–
Dynamic Resolution	124.6	68.3	40.3	31.2	26.8	20.3	15.8

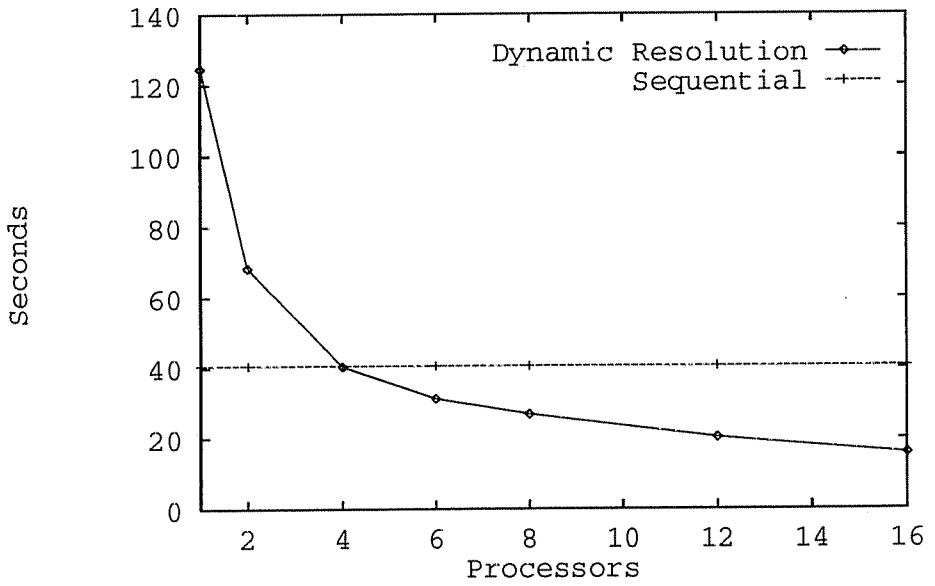


Figure 4.14: Timing results for dynamic resolution of topological sort (`topo`).

4.9 Notes

An early description of dynamic resolution appears in [53].

Lu [72], and Lu and Chen [73], describe run-time methods for parallelizing loops with indirect array accesses (in Fortran and C) and (restricted) pointer accesses (in C). Their methods pre-execute a loop nest at run time to find *data dependences* between program statements in the loop. The compiler, using static analysis, generates a *scheduler* for the loop's iterations. At run time, this scheduler dynamically records references to dynamic data and, using the reference patterns thus collected, allocates loop iterations to individual processors for parallel evaluation. Lu and Chen's method effectively parallelizes some simple loops. However, their method does not allow procedure calls in the loop's body. Furthermore, the loop may not modify data structure links and loop statements may not allocate new data; *i.e.*, the structure of all dynamic data must remain fixed for the duration of the loop. Dynamic resolution, on the other hand, parallelizes expressions that may apply functions and that can modify (and allocate) dynamic data. Lu and Chen's methods depend on extensive static pointer analysis (*e.g.*, [67]) that is expensive in practice [67] because it performs a data-flow analysis or an abstract interpretation on the entire program text. In contrast, the only interprocedural information the static analysis for dynamic resolution requires is whether a function is true (§4.1).[13]

Harrison's PARCEL system [45, 44] parallelizes sequential Scheme programs using a static analysis. His analysis is an abstract interpretation [25] that identifies interprocedural side effects that potentially interfere (conflict). In particular, this analysis statically approximates the possible (dynamic) configurations of the program's run-time stack. In this manner, PARCEL determines a dynamically-allocated object's *lifetime* (*cf.* [100, 99]). Lifetime analysis can establish that two program expressions cannot interfere and may safely evaluate in parallel. However, when mutable structures are long-lived, PARCEL cannot deduce when concurrent modification of them is safe. Furthermore, unlike dynamic techniques, PARCEL's analysis cannot exploit parallelism that is not always safe (*e.g.*, accesses to data that *might* be shared) since it is entirely static.

Larus's Curare uses abstract interpretation [25] to statically capture the structure of dynamic data in Scheme [95] programs. Approximation information thus obtained allows Curare to parallelize imperative expressions that cannot conflict due to shared-structure modifications. Specifically, Curare builds *alias graphs* that describe the potential aliases in a set of dynamic data structures. *Summary nodes*, nodes that conservatively approximate alias-graph nodes, are necessary to bound an alias graph's size. For large, irregular data, such bounded approximation leads to overly conservative parallelization—if sharing *can* occur dynamically, Curare assumes that this sharing *always* occurs. In the presence of sharing, therefore, some of the parallelism that dynamic resolution finds must elude Curare's analysis.

Hendren [47, 46] addresses the problem of parallelizing programs with recursive data structures with an algorithm for estimating the relationships between accessible nodes in a dynamic data structure. Relationships thus attained are then used to (statically) detect interference between program statements. Her analysis is defined for a first-order language with recursive procedures, and finds parallelism when it can statically determine that trees, rather than DAGs, always reach a given program point. That is,

[13] Interprocedural analysis for dynamic resolution can be avoided entirely by using a λ-tag (Chapter 3) to physically carry, with a function, an indication of whether it is true. Dynamic checks to λ-tags carrying this type of information can then select dynamic resolution when safe.

the analysis finds safe parallelism only in expressions statically known to manipulate trees. As such, this analysis cannot discover the type of parallelism that dynamic techniques can find (*e.g.*, parallelism in DAGs). Hendren's analysis can further detect when a set of *handles* (pointers) into a dynamic data structure cannot reach common structure. Relationships between handles are similar to the reaching relations that dynamic resolution obtains from pattern matching (§4.3.2)—Hendren's analysis can potentially perform the task of dynamic resolution's static component in languages that do not support patterns.

Many approaches to the general problem of static pointer analysis have been designed (*e.g.*, [58, 20, 51, 69, 119]). These approaches use a variety of bounded approximations (usually limited by some constant k) that attempt to statically describe the shape and connectivity of dynamic heap structures. Bounded approximation of these structures, however, limits the amount of the program's actual (dynamic) parallelism that these techniques can detect. These analyses also require expensive interprocedural analyses (*e.g.*, data flow or abstract interpretation) that discourage their practical use (*cf.* [67, 90]). Dynamic resolution's analysis is local to function definitions since interprocedural information (*i.e.*, sharing information) dynamically propagates into functions at run time.

Chapter 5

Dynamic Granularity Estimation

Functional languages, due to referential transparency, do not overly constrain a program's evaluation order with data dependences. This simplifies automatic parallelization: multiple arguments in a strict function application can evaluate in parallel, for example. For imperative dynamic languages, static and dynamic parallelization techniques can expose large amounts of parallelism (*cf.* [68, 44]; Chapters 3 and 4). Abundant parallelism, however, does not directly lead to effective parallel implementations. Efficient implementation of a dynamic language on a parallel architecture remains difficult in part because the creation of a parallel thread incurs considerable overhead costs [43, 81, 89, 80]. If an expression contains less computation than the cost of creating a thread for the expression, parallel evaluation slows program execution (Figure 5.1).

This chapter describes a new technique, *dynamic granularity estimation* (**dge**), that is based on the observation that a function's time complexity often depends on the size of the dynamic data with which it computes. For a program function f applied to a list parameter l, this technique conservatively determines the lengths of l for which the cost of computing the application $e \equiv (f\ l)$ always exceeds the overhead of creating a thread for e's concurrent evaluation.

As a dynamic technique, **dge** is a hybrid; it is composed of dynamic and static components. Static analysis identifies functions whose time complexity is dependent on the list data structures passed as parameters. The dynamic component approximates list lengths at run time. The compiler statically identifies program points at which the length of a list always influences the cost of an application expression. When evaluation reaches such a point, compiler-inserted code consults an approximation to the list's length (maintained dynamically) to determine whether it is beneficial to evaluate an application as a separate parallel thread.

The quicksort function (`qs`) of Figure 5.2 provides an example. In `qs`, the arguments to `append` can evaluate in parallel. Parallel evaluation of these expressions is advantageous if the costs of the recursive applications of `qs` exceed the cost of creating and scheduling these expressions as parallel threads. However, when the length of a sublist (`l` or `g`) is small (*e.g.*, zero), creating a parallel thread to sort the sublist is counterproductive. In this case, the arguments to `append` should evaluate sequentially. A static analysis based on abstract interpretation [25, 1] identifies list lengths for which the cost of applying `qs` to a list of that length is always greater than the overhead incurred in creating a new thread for the application's concurrent evaluation. At run time, the (approximate) lengths of the lists bound

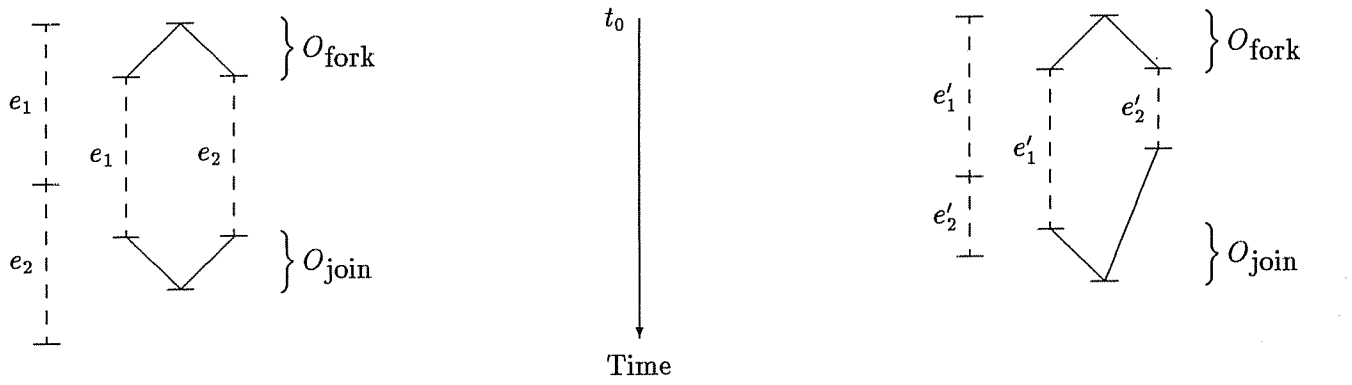


Figure 5.1: The impact of overhead. Time starts at t_0. The concurrent evaluation of e_1 and e_2, with overhead ($O = O_{\text{fork}} + O_{\text{join}}$) taken into account, completes before their sequential evaluation and is, therefore, beneficial. Concurrent evaluation of e'_1 and e'_2, however, slows the program's evaluation since e'_2 does not contain enough computation to offset scheduling overheads.

to the identifiers `l` and `g` in the `qs` function are available for making the final parallelization decision.

Dynamic techniques, like **dge**, that examine the size of dynamic data structures to conditionally select an expression's parallel evaluation are *necessary* (§1.1.2). This is evident from the `qs` example. When a statically-unknown list reaches `qs`, the sublist partition that `qs`'s auxiliary `split` function creates is also unknown. Therefore, the costs of the recursive applications of `qs` that sort the sublists cannot be known at compile time. In the absence of precise static information about `qs`'s list parameter, it is not possible to decide statically when it is advantageous to concurrently evaluate `qs`'s recursive applications.

In languages with explicit constructs for thread creation and synchronization, programmers typically use *cutoff* values to curb parallelism and to ensure that the program only creates large threads [81]. In the `qs` example, the programmer might explicitly check if the sublist being sorted contains $> k$ elements for some small k before creating parallel threads for `append`'s arguments. Dynamic granularity estimation deduces such cutoffs automatically. Code remains portable with **dge** since the language's implementation—not the programmer—is responsible for matching a cutoff to the underlying architecture. The granularity of parallel threads is less of a programming issue when thread sizes are determined automatically.

In the next section, I describe the language under consideration for dynamic granularity estimation and introduce terminology. I then describe **dge**'s static (§5.2) and dynamic (§5.3) components, present possible extensions to general data structures and mutable data (§5.4), illustrate **dge**'s operation with examples (§5.5), and discuss implementation results (§5.6 and §5.7).

```
fun qs p [] = []
  | qs p (x::xs) =
        let fun split l =
                let fun split' [] less greater = (less,greater)
                      | split' (y::ys) less greater =
                            if (p y x) then
                                split' ys (y::less) greater
                            else
                                split' ys less (y::greater)
                in
                    split' l [] []
                end
            val (l,g) = split xs
        in
            if (l̄ > cutoff) andalso (ḡ > cutoff) then
                append|| (qs p l) (x::(qs p g))
            else
                append (qs p l) (x::(qs p g))
        end
```

Figure 5.2: Functional quicksort automatically restructured by dynamic granularity estimation. Static analysis determines that the amounts of computation in the arguments to append depend on the lengths (denoted $\bar{l}$ and $\bar{g}$) of the sublists produced by `split`. The compiler inserts a run-time check (the conditional in qs's body) to examine the lengths of `l` and `r` (stored with the list representation). Based on these dynamic lengths, the check decides whether to create parallel threads. The compiler also deduces the `cutoff` value.

5.1 Preliminaries

The language under consideration for dynamic granularity estimation is functional,[1] call-by-value, higher-order, and untyped—it is the λ_v-calculus; *i.e.*, it is the λ_v-S calculus (§2.2) without imperative operations and reference values. For simplicity, I focus on the *list* as the dynamic structure for dynamic granularity estimation—a list's size is its length. Section 5.4 describes possible extension to general recursive datatypes that give rise to trees, for example.

Denote the time required to evaluate an expression e as $|e|$, the *cost* of e. The cost of a parallel thread is the cost of the expression that the thread evaluates in addition to the overhead, O, required to create and schedule a parallel thread.[2] Let $T \geq O$ be a machine-dependent cost *threshold* so that if $|e| > T$

[1] Restriction to a functional language allows efficient implementation of dge's dynamic component that approximates the sizes of dynamic data at run time. In a functional language, a datum d's size can only monotonically increase whereas, in a language with assignment to reference values, d's size can decrease and the efficient propagation of d's new (reduced) size estimate to other data that share d is difficult. Section 5.4 describes possible methods for estimating data sizes in imperative dynamic languages.

[2] It is assumed that the cost of creating and scheduling a thread is bounded and can be (empirically) determined for a

then expression e is a candidate for parallel evaluation (*cf.* Figures 5.1 and 5.2). Sizes are measured in integer *evaluation units* (e-units). An e-unit corresponds to—again for simplicity—the operational notion of function application [30]. For a given implementation, normalization of e-units is necessary since all function applications do not have identical costs (*e.g.*, in SML/NJ some functions are compiled in line whereas others require the construction of a closure [9]).

For λ_v, I assume that the evaluation of variables, constants and λ-abstractions incurs no cost (zero e-units) and that the evaluation of the other language terms costs one e-unit. Under these simplifying assumptions, for example, the application $(f\ (g\ l))$, where f and g are functions and l is a list, incurs a cost of at least two e-units (the applications of f and g each cost one), but complete evaluation of $(f\ (g\ l))$ may require many more e-units and may depend on the size (length) of l.

The length of list l is written as $\bar{l}$. When i is a natural number, $\bar{i}$ represents a list of length i.

5.2 Static Component

The idea is to evaluate an application $e \equiv (f\ l)$ statically while counting the number of e-units required. This static e-unit is conservative; that is, static estimation of e-units does not overestimate the number of e-units evaluation of an expression requires. For example, if static analysis of e indicates that $|e| = i$, then actual evaluation of e must require $\geq i$ e-units. Since the aim is to identify functions whose list parameters control their complexity, an abstract semantics that interprets a list l as its length, $\bar{l}$, is used. E-units are (conservatively) counted under this abstract semantics. I first give the standard semantics for the language and then the abstract semantics. To make the abstract semantics computable, it is also necessary to bound the number of abstract evaluation steps (§5.2.3). This bound corresponds to the threshold T (§5.1) at which parallel evaluation of a thread becomes beneficial (*i.e.*, exceeds scheduling overheads).

5.2.1 Standard Semantics $\mathcal{S}$

The dynamic objects of the standard semantics $\mathcal{S}$ are in Figure 5.3. Since the list is the dynamic structure of interest for granularity estimation, it is directly represented with dynamic objects—the constant *nil* and *cons* pairs $\langle v, l \rangle$, where v is a list element and l is the list's tail—rather than indirectly encoded in λ_v. Basic list-manipulation functions (*hd*, *tl*, and *isnull*) are also dynamic objects.

Figure 5.4 gives a standard semantics for the language. The rules for list functions and objects are in Figure 5.5. The operational style of the semantics is derived from Tofte's semantics [112]. The semantics given here, however, additionally contains *integer time annotations* that indicate the number of e-units that an expression's evaluation requires. The evaluation relation

$$E \vdash e \longrightarrow_i v$$

(where $E \in \textsc{Env}$, $e \in \textsc{Exp}$, $v \in \textsc{DVal}$, and $i \in \mathbf{Z}$) indicates that the evaluation of expression e to value v with respect to environment E requires i e-units. For example, the **app** rule states that if the evaluation of e_1 to v_1 requires a e-units, the evaluation of e_2 to v_2 requires b e-units, and the application of v_1 to v_2 requires c e-units, then the evaluation of the application $(e_1\ e_2)$ requires $1 + a + b + c$ evaluation units.

given language implementation and machine architecture.

$$
\begin{array}{rcl}
i & \in & \textsc{Int} = \{\ldots,-1,0,1,\ldots\} \\
b & \in & \textsc{Bool} = \{\mathit{true},\mathit{false}\} \\
f & \in & \textsc{Fns} = \{\mathit{hd},\mathit{tl},\mathit{isnull}\} \\
\langle v,l\rangle & \in & \textsc{Cons} = \textsc{DVal} \times \textsc{List} \\
l & \in & \textsc{List} = \{\mathit{nil}\} + \textsc{Cons} \\
[x,e,E] & \in & \textsc{Clos} = \textsc{Var} \times \textsc{Exp} \times \textsc{Env} \\
v & \in & \textsc{DVal} = \textsc{Int} + \textsc{Bool} + \textsc{Fns} + \textsc{List} + \textsc{Clos} \\
E & \in & \textsc{Env} = \textsc{Var} \xrightarrow{fin} \textsc{DVal}
\end{array}
$$

Figure 5.3: Dynamic objects of the standard semantics $\mathcal{S}$.

$$\frac{x \mapsto v \in E}{E \vdash x \longrightarrow_0 v} \qquad \textbf{(var)}$$

$$\frac{}{E \vdash (\lambda x.e) \longrightarrow_0 [x,e,E]} \qquad \textbf{(abs)}$$

$$\frac{\begin{array}{c} E \vdash e_1 \longrightarrow_a [x,e,E'] \\ E \vdash e_2 \longrightarrow_b v \\ E' \pm \{x \mapsto v\} \vdash e \longrightarrow_c v' \end{array}}{E \vdash (e_1\ e_2) \longrightarrow_{1+a+b+c} v'} \qquad \textbf{(app)}$$

$$\frac{E \vdash e_1 \longrightarrow_a \mathit{true} \quad E \vdash e_2 \longrightarrow_b v}{E \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \longrightarrow_{1+a+b} v} \qquad \textbf{(if-true)}$$

$$\frac{E \vdash e_1 \longrightarrow_a \mathit{false} \quad E \vdash e_3 \longrightarrow_b v}{E \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \longrightarrow_{1+a+b} v} \qquad \textbf{(if-false)}$$

Figure 5.4: Standard semantics $\mathcal{S}$ with time annotations.

$$\frac{}{\vdash \texttt{nil} \longrightarrow_0 \mathit{nil}} \quad \textbf{(nil)}$$

$$\frac{}{\vdash \texttt{hd} \longrightarrow_0 \mathit{hd}} \quad \textbf{(hd)}$$

$$\frac{}{\vdash \texttt{tl} \longrightarrow_0 \mathit{tl}} \quad \textbf{(tl)}$$

$$\frac{}{\vdash \texttt{isnull} \longrightarrow_0 \mathit{isnull}} \quad \textbf{(isnull)}$$

$$\frac{E \vdash e_1 \longrightarrow_a v \quad E \vdash e_2 \longrightarrow_b l}{E \vdash (\texttt{cons}\ e_1\ e_2) \longrightarrow_{1+a+b} \langle v, l \rangle} \quad \textbf{(cons)}$$

$$\frac{E \vdash e_1 \longrightarrow_a \mathit{hd} \quad E \vdash e_2 \longrightarrow_b \langle v, l \rangle}{E \vdash (e_1\ e_2) \longrightarrow_{1+a+b} v} \quad \textbf{(app-hd)}$$

$$\frac{E \vdash e_1 \longrightarrow_a \mathit{tl} \quad E \vdash e_2 \longrightarrow_b \langle v, l \rangle}{E \vdash (e_1\ e_2) \longrightarrow_{1+a+b} l} \quad \textbf{(app-tl)}$$

$$\frac{E \vdash e_1 \longrightarrow_a \mathit{isnull} \quad E \vdash e_2 \longrightarrow_b \langle v, l \rangle}{E \vdash (e_1\ e_2) \longrightarrow_{1+a+b} \mathit{false}} \quad \textbf{(app-isnull-false)}$$

$$\frac{E \vdash e_1 \longrightarrow_a \mathit{isnull} \quad E \vdash e_2 \longrightarrow_b \mathit{nil}}{E \vdash (e_1\ e_2) \longrightarrow_{1+a+b} \mathit{true}} \quad \textbf{(app-isnull-true)}$$

Figure 5.5: Standard semantics $\mathcal{S}$ (for list objects) with time annotations.

Similarly, conditional evaluation (**if** rule) counts e-units only in the evaluation of the branch expression selected by the conditional's predicate. Note that the evaluation of λ_v's value terms (*e.g.*, variables and λ-abstractions) requires zero e-units under this relation; a specific implementation would, however, use an e-unit measure and evaluation rules that reflect their concrete costs.

5.2.2 Abstract Semantics $\mathcal{A}$

A non-standard (abstract) semantics $\mathcal{A}$ that abstracts lists as their lengths is used for counting e-units for dynamic granularity estimation. This analysis determines whether an application $(f\ l)$ always requires at least i (where $i \geq 0$) e-units of evaluation for a given length of l. The dynamic objects of the abstract semantics are in Figure 5.6. A list of length k in the abstract semantics is represented by L_k; that is, by the set of all lists with at least k elements.[3] An environment ($\textsc{Env}^{\mathcal{A}}$) maps a program variable to either a concrete finite subset of values or to any such subset (denoted $\top^{\mathcal{A}}$).

[3]Note that L_0 describes all lists and that $L_i \supset L_{i+1}$, $i \geq 0$.

$$
\begin{array}{rcl}
i & \in & \textsc{Int}^{\mathcal{A}} = \{\ldots, -1, 0, 1, \ldots\} \\
b & \in & \textsc{Bool}^{\mathcal{A}} = \{\mathit{true}, \mathit{false}\} \\
f & \in & \textsc{Fns}^{\mathcal{A}} = \{\mathit{hd}, \mathit{tl}, \mathit{isnull}\} \\
L_k & \in & \textsc{List}^{\mathcal{A}} = \{L_0, L_1, \ldots\} \text{ where } L_k \text{ denotes all lists of length} \geq k \\
[x, e, E^{\mathcal{A}}] & \in & \textsc{Clos}^{\mathcal{A}} = \textsc{Var} \times \textsc{Exp} \times \textsc{Env}^{\mathcal{A}} \\
v & \in & \textsc{DVal}^{\mathcal{A}} = \textsc{Int}^{\mathcal{A}} + \textsc{Bool}^{\mathcal{A}} + \textsc{Fns}^{\mathcal{A}} + \textsc{List}^{\mathcal{A}} + \textsc{Clos}^{\mathcal{A}} \\
E^{\mathcal{A}} & \in & \textsc{Env}^{\mathcal{A}} = \textsc{Var} \xrightarrow{fin} (\mathrm{Fin}(\textsc{DVal}^{\mathcal{A}}) + \top^{\mathcal{A}})
\end{array}
$$

Figure 5.6: Dynamic objects of the abstract semantics $\mathcal{A}$.

The abstract evaluation relation

$$E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_i V$$

(where $E^{\mathcal{A}} \in \textsc{Env}^{\mathcal{A}}$, $e \in \textsc{Exp}$, $V \subseteq \textsc{DVal}^{\mathcal{A}}$, and $i \in \mathbf{Z}$) evaluates the expression e with respect to (abstract) environment $\mathrm{E}^{\mathcal{A}}$ to a *set* of values V. This relation is defined such that when $e \xrightarrow{\mathcal{A}}_i V$ and $e \longrightarrow_j v$ then $v \in V$ and $i \leq j$. That is, the set of values computed by the abstract relation always contains e's actual value (as produced by $\mathcal{S}$). Furthermore, the e-unit count produced by the abstract semantics is conservative; standard evaluation of e under $\mathcal{S}$ always requires at least i e-units when abstract evaluation of e under $\mathcal{A}$ requires i e-units.

Figures 5.7 and 5.8 give the operational rules for the abstract semantics using the $\xrightarrow{\mathcal{A}}_i$ evaluation relation. Foremost, note that the $\mathbf{any}^{\mathcal{A}}$ rule can always be applied. It simply evaluates an expression e to any value and incurs no e-unit cost. Therefore, it is a conservative estimate of values and e-units. The $\mathbf{var}^{\mathcal{A}}$ rule retrieves the mapping of a variable from an environment at zero cost. The $\mathbf{abs}^{\mathcal{A}}$ rule evaluates a λ-abstraction term to a singleton set containing its closure at zero cost.

Abstract evaluation of an application $(e\ e')$ with the $\mathbf{app}^{\mathcal{A}}$ rule first abstractly evaluates e and e'. When e produces a set F of closures, each $f \in F$ is applied to the value set V that e' produces. The e-unit cost of an application is one e-unit (for the application proper), the e-units required for (abstractly) evaluating e and e', and the minimum of the e-unit costs incurred in applying each $f \in F$ to V. This gives a conservative e-unit count because the cost of the least expensive function reaching the application is used. The set of values produced by $\mathbf{app}^{\mathcal{A}}$ is the union of the value sets produced by the applications of the closures F.

The conditional rules ($\mathbf{if\text{-}true}^{\mathcal{A}}$, $\mathbf{if\text{-}true}^{\mathcal{A}}$, $\mathbf{if}^{\mathcal{A}}$) conservatively approximate a conditional's behavior. If the predicate abstractly evaluates to a singleton set containing either *true* or *false*, the respective conditional branch is abstractly evaluated. However, when the predicate's abstract value set is not precisely known (*e.g.*, when it contains both *true* and *false*), both conditional branches are abstractly evaluated[4] and the minimum e-unit cost of these evaluations is incorporated into the conditional's cost—the set of values produced by the conditional is the union of the value sets produced by both conditional

[4]In this case, abstract evaluation may not terminate since computation in a conditional branch—protected by the predicate in the standard semantics—may diverge. Section 5.2.3 describes how this termination problem is avoided.

$$\frac{}{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_0 \top^{\mathcal{A}}} \quad (\mathbf{any}^{\mathcal{A}})$$

$$\frac{x \mapsto V \in E^{\mathcal{A}}}{E^{\mathcal{A}} \vdash x \xrightarrow{\mathcal{A}}_0 V} \quad (\mathbf{var}^{\mathcal{A}})$$

$$\frac{}{E^{\mathcal{A}} \vdash (\lambda x.e) \xrightarrow{\mathcal{A}}_0 \{[x, e, E^{\mathcal{A}}]\}} \quad (\mathbf{abs}^{\mathcal{A}})$$

$$\frac{\begin{array}{c} E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_a \{[x_1, e_1, E_1^{\mathcal{A}}], \ldots, [x_n, e_n, E_n^{\mathcal{A}}]\} \\ E^{\mathcal{A}} \vdash e' \xrightarrow{\mathcal{A}}_b V \\ E_i^{\mathcal{A}} \pm \{x_i \mapsto V\} \vdash e_i \xrightarrow{\mathcal{A}}_{c_i} V_i,\ 1 \leq i \leq n \end{array}}{E^{\mathcal{A}} \vdash (e\ e') \xrightarrow{\mathcal{A}}_{(1+a+b+\min(c_1,\ldots,c_n))} \bigcup_{i=1}^{n} V_i} \quad (\mathbf{app}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a \{true\} \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_b V}{E^{\mathcal{A}} \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \xrightarrow{\mathcal{A}}_{1+a+b} V} \quad (\mathbf{if\text{-}true}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a \{false\} \quad E^{\mathcal{A}} \vdash e_3 \xrightarrow{\mathcal{A}}_b V}{E^{\mathcal{A}} \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \xrightarrow{\mathcal{A}}_{1+a+b} V} \quad (\mathbf{if\text{-}false}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a V_1 \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_b V_2 \quad E^{\mathcal{A}} \vdash e_3 \xrightarrow{\mathcal{A}}_c V_3}{E^{\mathcal{A}} \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \xrightarrow{\mathcal{A}}_{1+a+\min(b,c)} V_2 \cup V_3} \quad (\mathbf{if}^{\mathcal{A}})$$

Figure 5.7: Abstract semantics $\mathcal{A}$ with time annotations.

$$\frac{}{\vdash \texttt{nil} \xrightarrow{\mathcal{A}}_0 \{L_0\}} \quad (\textbf{nil}^{\mathcal{A}})$$

$$\frac{}{\vdash \texttt{hd} \xrightarrow{\mathcal{A}}_0 \{hd\}} \quad (\textbf{hd}^{\mathcal{A}})$$

$$\frac{}{\vdash \texttt{tl} \xrightarrow{\mathcal{A}}_0 \{tl\}} \quad (\textbf{tl}^{\mathcal{A}})$$

$$\frac{}{\vdash \texttt{isnull} \xrightarrow{\mathcal{A}}_0 \{isnull\}} \quad (\textbf{isnull}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a V \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_b \{L_i\}}{E^{\mathcal{A}} \vdash (\texttt{cons}\ e_1\ e_2) \xrightarrow{\mathcal{A}}_{1+a+b} \{L_{i+1}\}} \quad (\textbf{cons}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a \{hd\} \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_b \{L_i\}}{E^{\mathcal{A}} \vdash (e_1\ e_2) \xrightarrow{\mathcal{A}}_{1+a+b} \top^{\mathcal{A}}} \quad (\textbf{app-hd}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a \{tl\} \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_b \{L_i\}}{E^{\mathcal{A}} \vdash (e_1\ e_2) \xrightarrow{\mathcal{A}}_{1+a+b} \{L_{i-1}\}} \quad (\textbf{app-tl}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a \{isnull\} \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_b \{L_i\} \quad i > 0}{E^{\mathcal{A}} \vdash (e_1\ e_2) \xrightarrow{\mathcal{A}}_{1+a+b} \{false\}} \quad (\textbf{app-isnull-false}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a \{isnull\} \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_b \{L_i\} \quad i \geq 0}{E^{\mathcal{A}} \vdash (e_1\ e_2) \xrightarrow{\mathcal{A}}_{1+a+b} \{true, false\}} \quad (\textbf{app-isnull}^{\mathcal{A}})$$

Figure 5.8: Abstract semantics $\mathcal{A}$ (for list objects) with time annotations.

branches.

The abstract evaluation rules of Figure 5.8 handle list functions and objects. The **nil**$^{\mathcal{A}}$ rule evaluates the syntactic constant `nil` to the set of all lists (L_0). The list functions `hd`, `tl`, and `isnull` evaluate to the singleton sets of their respective dynamic function objects. The abstract evaluation of these list constants incurs no e-unit cost.

A list's size (length) increases when an element is *consed* onto it. List creation with the special `cons` form (**cons**$^{\mathcal{A}}$ rule)—when the tail of the new list is in the set L_i; *i.e.*, it is a list of length i)—produces the set of lists of length $i+1$, L_{i+1}. The abstract e-unit cost for this operation is one plus the cost of evaluating the arguments to `cons`.

Selecting the head (**app-hd**$^{\mathcal{A}}$ rule) of any list returns any value ($\top^{\mathcal{A}}$) since a list's contents (its elements) are not maintained in the abstract semantics. Selecting the tail (**app-tl**$^{\mathcal{A}}$ rule) of a list of length i returns L_{i-1}, the set of lists of length $i-1$, since the list returned by the tail selector is always one less than the length of its argument list.

Testing for the empty list with the `isnull` predicate produces the set $\{false\}$ when `isnull`'s argument is a list of length ≥ 1 (**app-isnull-false**$^{\mathcal{A}}$ rule). Otherwise, this test conservatively returns $\{true, false\}$ under abstract evaluation (**app-isnull**$^{\mathcal{A}}$ rule).

5.2.3 Termination

Counting e-units in the abstract semantics $\mathcal{A}$ proceeds conservatively along both arms of a conditional whose abstract predicate value is imprecise (*i.e.*, neither $\{true\}$ nor $\{false\}$). This ensures that the cost of an expression is conservatively approximated as the cost of its least-costly execution path. Doing so, however, introduces the possibility of non-termination under abstract evaluation since abstract evaluation can now attempt to evaluate a term (conditional branch) that diverges under the standard semantics.

The termination problem is solved by bounding the number of abstract evaluation steps. Evaluation of an execution path under $\mathcal{A}$ terminates (along that path) when the accumulated e-units exceed the overhead threshold T (§5.1). In other words, when viewed as a deductive proof, the proof tree of an expression's abstract evaluation never exceeds a depth of T unit-cost deductions; *i.e.*, the **any**$^{\mathcal{A}}$ rule is applied upon reaching this bound. Halting abstract evaluation in this manner avoids the non-termination issue since we only evaluate for a bounded T e-units along any execution path and return the cost of the least-cost path.

5.2.4 Program Restructuring

A compiler can use dynamic granularity estimation to restructure the program as follows. The compiler wraps a conditional around an application expression, `(f l)`, that applies function `f` to a list `l`. The conditional's branches respectively contain code for the sequential and parallel evaluation of the application expression (see, for example, Figure 5.2). The predicate of the compiler-supplied conditional examines the length of `l` (available at run time) and compares it to a compiler-deduced cutoff value. When `l`'s length is at least equal to this cutoff, the conditional selects parallel evaluation for `(f l)`. The compiler deduces the cutoff value using abstract evaluation in the following manner. Suppose that **dge**'s dynamic component (described below) precisely maintains the lengths of all lists of length $< n$, and that all lists with lengths $\geq n$ are approximated as such. The compiler abstractly evaluates `(f` L_i`)` for $0 \leq i < n$.

When (f L_i) $\xrightarrow{\mathcal{A}}_x V$, it notes the least i such that the cost x of this application is always greater than the overhead threshold T. This least i, if it exists, represents a length cutoff for 1 at which the creation of a parallel thread for (f 1) is always beneficial. The value of this least i is the cutoff value in the conditional guarding the application.

In general, the compiler can use the abstract-evaluation semantics to determine a cost threshold for *any* expression e, not just for the application of functions to lists. To do so, it must first identify all lists in e; it then abstractly evaluates e for all list-length combinations and records the lengths at which parallel evaluation of e is viable. This list-length information is then used to construct a predicate to select e's parallel evaluation only when beneficial.

Section 5.5 provides a concrete example of the abstract evaluation a compiler must perform to use dynamic granularity estimation.

5.3 Dynamic Component

At run-time, **dge**'s dynamic component maintains an approximation to the length of a list l along with l's physical representation. An implementation that represents lists with *cons cells* in a heap is assumed. A fixed field of b bits encodes length information. This allows lists of length $< 2^b - 1$ to be exactly represented. Approximate lists have length ∞; that is, an approximate list is of length $\geq 2^b - 1$. When a new list is formed with the list constructor, as in $l \equiv$ (cons x l'), the length field on l is set to $\overline{l'} + 1$ if $\overline{l'}$ is not ∞. Otherwise, it is set to ∞.

An implementation of the dynamic component can store the b bits of length information either:

1. in a cons cell, or

2. in the pointers to a cons cell

Storing the approximation within the cell requires an additional memory access when forming a new cell since the length field pointed to by the new cell's tail pointer must be fetched. If the cons-cell representation does not contain b unused bits, additional storage must also be allocated in the cell under the first scheme. The second approach requires the pointer representation to contain b unused bits, but avoids an additional memory fetch since construction of a new cons cell always requires the pointer to the list that becomes the new cell's tail field. The first approach is significantly simpler to implement because it only requires modification to the portion of the compiler that generates the code for cons-cell creation (§5.7). The second approach requires modifications to the implementation's run-time system (*e.g.*, the garbage collector), the generation of special pointer dereferencing code, and (potentially) a revision of the memory layout.

The final concern in the design of the dynamic component is, how many bits, b, to allocate for the length field. A value for b is best selected by consulting the empirical results of applying **dge**'s static analysis (§5.2) to actual programs because, for a typical application (f l), where $|(f\ l)|$ depends on the length of l, it is likely that a threshold value for $\overline{l}$ exists at which parallel evaluation of (f l) is fruitful. The number of bits b should be large enough to delineate this threshold for most cases.

5.4 Extensions

In this section, I describe possible extensions to dynamic granularity estimation that admit general dynamic data structures and mutable data.

5.4.1 Other Data Structures

In addition to lists, general recursive structures (*e.g.*, trees) can be handled by defining the size of such a structure to be the sum of the sizes of its substructures. Physical representation of a structure's node then contains the sum of the sizes of the structures pointed to by the node. A node for a binary tree, for example, would carry the sum of the sizes of its left and right subtrees. A static analysis, similar to this chapter's analysis for lists, can determine the data sizes for which an expression e's concurrent evaluation is beneficial. However, upon deconstruction of a dynamic node of size n, the analysis must now consider all possible combinations for the substructure's sizes. For example, deconstructing a binary tree of size n with subtrees *left* and *right* requires abstract evaluation with all (n) size assignments such that $|left| + |right| = n - 1$. Enumerating and abstractly evaluating these combinations increases the static analysis' complexity. It is, however, plausible that static examination of all small structures is practical and suffices to delineate a viable size threshold for making thread-creation decisions.

Run-time examination of the size of an *array* can be used to dynamically determine the granularities of expressions in array-based languages (*e.g.*, Fortran and C). An array descriptor (*e.g.*, [35]) can be used to dynamically convey an array's size and bounds. Static analysis can then determine, for a program expression e manipulating array a, the sizes of a for which concurrent evaluation of e is beneficial.

5.4.2 Mutable Dynamic Data

In languages with imperative assignment to mutable dynamic data (*e.g.*, ML), it is difficult to dynamically maintain conservative size approximations for such data. This is because a mutable datum's size may decrease. Furthermore, as with immutable data, mutable data are often shared. To maintain conservative approximations, it may be necessary to propagate—upon assignment into a dynamic structure—a new size to many structures. Identification of structures that share a datum d is, however, difficult because d has no information about the pointers to it. A possible approach to extending **dge** to mutable data is to *not* propagate reductions in a mutable datum's size. Instead, its size estimate can be reconstituted periodically. Such size reconstitution can occur in the language implementation's garbage collector (see §4.5.3). Since this approach permits approximations that may overestimate a datum's size, it may—in some cases—select expressions for concurrent evaluation that do not contain enough computation to compensate for scheduling overheads. However, if a large percentage of the dynamic scheduling decisions are correct, dynamic granularity estimation in the presence of modifications to dynamic structures may be viable.

5.5 Examples

Here, I illustrate the operation of dynamic granularity resolution's static component (abstract evaluation) and show how the compiler can use the information thus obtained, along with run-time list lengths, to

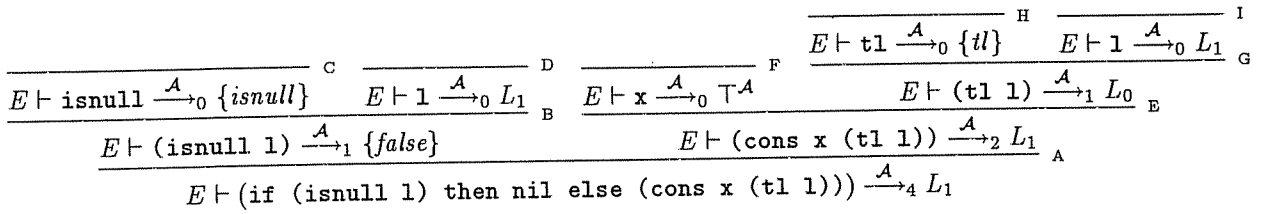


Figure 5.9: Example operation of **dge**'s static component. Environment E maps identifier `l` to all lists of length ≥ 1; *i.e.*, $E \equiv \{\texttt{l} \mapsto L_1\}$. Deduction A is the **if**$^{\mathcal{A}}$ rule, B is the **app-isnull-false**$^{\mathcal{A}}$ rule, C is the **isnull**$^{\mathcal{A}}$ rule, E is the **cons**$^{\mathcal{A}}$ rule, F is the **any**$^{\mathcal{A}}$ rule, G is the **app-tl**$^{\mathcal{A}}$ rule, and H is the **tl**$^{\mathcal{A}}$ rule. Deductions D and I are the **var**$^{\mathcal{A}}$ rule.

dynamically schedule concurrent expressions only when beneficial.

Figure 5.9 depicts the deductions that **dge** performs statically for the expression:

$$e \equiv \texttt{if (isnull l) then nil else (cons x (tl l))}$$

The compiler, upon encountering e in a program, can use **dge** to determine e's cost given the length of the list bound to identifier `l`. The figure abstractly evaluates e in an environment where `l` is bound to the set of lists of length ≥ 1 (*i.e.*, in the environment $\{\texttt{l} \mapsto L_1\}$). Abstract evaluation of e in this environment indicates that e's evaluation will produce a list in L_1 and will require at least four e-units(*i.e.*, $\{\texttt{l} \mapsto L_1\} \vdash e \xrightarrow{\mathcal{A}}_4 L_1$). Note that abstract evaluation of e in the environment $\{\texttt{l} \mapsto L_0\}$ produces a list in L_0 and requires two e-units (using the **if**$^{\mathcal{A}}$, **app-isnull**$^{\mathcal{A}}$, and **nil**$^{\mathcal{A}}$ rules).

As an example of how a compiler combines information from **dge**'s static and dynamic components, consider the function `f`:

```
fun f x = if (isnull x) then nil
          else f (tl x)
```

Abstract evaluation at compile time determines that `(f` L_0`)` requires three e-units, `(f` L_1`)` requires seven e-units, and `(f` L_2`)` requires eleven e-units. In general, abstract (and standard) evaluation of `(f` L_n`)` requires $3 + 4n$ e-units. However, a compiler need only abstractly evaluate `(f` L_i`)` for $0 \leq i < 2^b - 1$, where b is the number of bits of list-length information maintained by **dge**'s dynamic component (§5.3), since this encompasses the size information available at run time. The compiler then selects the least i such that $|\texttt{(f } L_i\texttt{)}| > T$ where T is the implementation-specific e-unit threshold (§5.1). Assuming the concrete values $b = 2$ and $T = 10$ for this example, using **dge**, the compiler can statically deduce that a concurrent thread for `(f l)` is beneficial when `l`'s length equals or exceeds two.

As a final example, dynamic granularity resolution statically determines that the time complexity of `qs` (Figure 5.2) depends on its list parameter. In particular, it detects that `split` always traverses the entire tail of this parameter. Therefore, the `qs` function's recursive applications—as well as external applications of `qs` in other parts of the program—warrant concurrent threads when `qs`'s list parameter is sufficiently[5] large.

[5]The length of `qs`'s parameter, at which parallel evaluation of an application of `qs` is beneficial, depends on the machine-dependent threshold T.

5.6 Implementation

The dynamic component of dynamic granularity estimation has been implemented in the Standard ML of New Jersey 0.73 optimizing compiler [11]. The compiler and run-time system were modified to incorporate one machine word (32 bits) of length information into the standard representation (three words) of every cons cell (*cf.* §5.3). The compiler's front end was modified to distinguish cons cells from all other types of dynamic objects. This modification identifies cons cells as such for the compiler's back end. The code generator was modified to produce code that computes list lengths upon cons-cell formation. Since a list's length is represented by a full machine word, code for approximating list lengths is unnecessary and is not generated. High-level primitives provide access to a list's length information. This integer length can be manipulated as an ML value and compared against threshold values determined empirically (§5.1). Low-level primitives, *i.e.* abstract machine instructions, would provide even better performance.

The static component for **dge** has not been implemented—abstract evaluation was performed manually.

5.7 Results

Figure 5.10 gives the results of dynamic granularity estimation applied to a quicksort (`qs`, Figure 5.2) sorting a list of 10000 random integers. The recursive applications of `qs` for sorting sublists were performed in parallel on 8 processors.[6] The graph plots list-length cutoffs versus execution time.[7] Execution, garbage collection, and total times are given for `qs` with and without **dge**. The graph's top two curves are the total time required with dynamic granularity estimation (dge) and with standard parallel evaluation (std) respectively. The x-axis is the cutoff values at which threads are retained for sequential evaluation. For the **dge** times, a length cutoff i indicates that the arguments to `append` in `qs` evaluate in parallel only when the lengths of the sublists bound to `l` and `g` both equal or exceed i. The graph's lower curves break the total time into execution (exec) and garbage collection (gc) times. Time spent in the operating system are included in the total times.

Dynamic granularity estimation improves `qs`'s performance at all cutoff values i, $0 \leq i \leq 10$. If thread creation is throttled when sublists are of length < 3, **dge** reduces the total time to execute the program by $\approx 23\%$. Figure 5.10 also reveals that garbage collection times slightly decrease as the cutoff length increases—fewer threads require fewer memory resources.

Two peculiarities in the timings of Figure 5.10 require further explanation. First, the non-monotonicity of the execution times arises because of a secondary effect: As the machine fills with threads, it becomes advantageous not to create new threads—even if these threads contain large amounts of computation relative to scheduling costs—since the machine is fully utilized. The input data to `qs` and the length cutoff (indirectly) influence the machine's load and cause this behavior. The second peculiarity is that the performance of **dge** at a cutoff of zero is better than that of the standard implementation. This is so

[6] Figure 3.15 gives times for `qs` as the number of processors varies.

[7] Here, I examine the effect of varying `qs`'s list-length cutoff value on the program's execution time. Parameters of a specific language implementation and machine architecture would enable **dge**'s static component to automatically select a concrete cutoff.

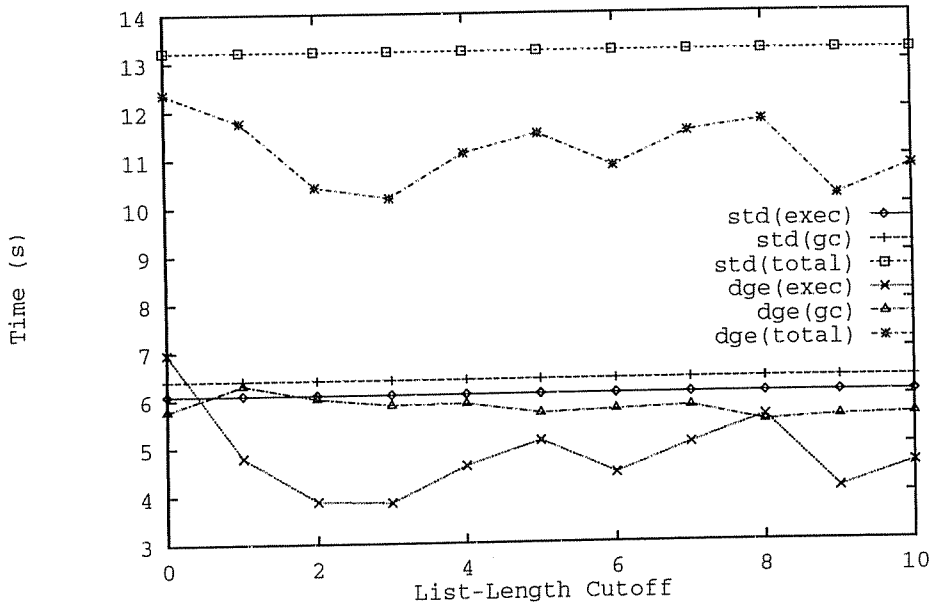


Figure 5.10: Effect of varying the list-length cutoff threshold in qs (Figure 5.2).

even though both versions create the same threads and the run-time system for **dge** incurs overhead; it allocates more data and performs more computation in maintaining list lengths than standard parallel evaluation. This occurs because the larger cons cells (four machine words versus three) of the **dge** run time improve processor data-cache performance.[8]

5.8 Notes

I do not know of previous approaches that estimate the amount of computation in an expression by examining dynamic information. Aside from simple heuristics [40], work related to (static) granularity estimation falls into one of two categories: load-balancing strategies that continually monitor the number of active threads in the machine to determine when it saturates, and systems that statically derive an algorithm's time complexity, if possible.

In Halstead's Multilisp [42, 43], the program ceases to create new parallel threads when the machine saturates with threads. When this occurs, processors evaluate the available threads to completion. Idle processors steal threads from busy processors in this load-based inlining scheme. Load-based inlining, in the presence of futures (§1.4), poses deadlock problems, but these can be avoided by Mohr *et al.*'s lazy task creation technique [81, 80]. Lazy task creation efficiently extracts computation from inlined threads

[8]This was verified by experiment. Setting cons-cell sizes to four machine words, improves the performance of some programs. Note that this phenomenon is, however, highly machine and implementation dependent.

when no runnable threads exist. Although lazy task creation increases the granularity of programs by coalescing threads, unlike **dge**, it does not prevent the production of fine-grain threads that are detrimental to the program's quick evaluation. WorkCrews [114] is a thread management package that performs lazy task creation, but requires programmer knowledge of the mechanism. Qlisp [36] provides primitives for performing load-based thread creation as well as automatic load-based inlining [89].

Dynamic granularity estimation is a *load-insensitive* technique that only creates parallel threads that are guaranteed to meet or exceed some granularity criterion. Therefore, **dge** is orthogonal—and complements—existing load-based inlining and task creation methods.

Harrison's parallel Lisp system, PARCEL [45], employs a non-standard list representation that dynamically maintains information about a list's length. PARCEL uses length information to implement lists contiguously in memory, but not for making parallelization or load-balancing decisions.

Static time-complexity analysis has been studied extensively; static algorithm and program analyzers have been built. Since the general problem of deducing a program's complexity is undecidable, these systems cannot always deduce a program's complexity. In many cases, however, the analyzers do correctly deduce the complexity of a program. METRIC [118] transforms Lisp programs into a set of mutually recursive equations and then seeks their solution to yield the program's complexity. Le Métayer's ACE complexity evaluator [70] matches list-based functional programs against a predefined library of function definitions to map programs to their time complexities. Sands extended this approach to higher-order lazy languages [103]. Dornic *et al.* [30] describe a practical *time system* that statically infers a function's complexity from its local definition; *i.e.*, their analysis does not require interprocedural information. Their time system, however, is conservative since it approximates recursive functions as always being expensive to evaluate. In contrast to dynamic granularity estimation, static time-complexity analyses cannot accurately predict an expression's cost when dynamic data sizes are not known at compile time.

Dynamic granularity estimation's static analysis is a form of abstract interpretation [25, 1, 56]. It differs from conventional abstract interpretation in two respects: it assumes the availability of dynamic information, and it does not abstract to finite domains—instead, the threshold that governs thread creation is used to terminate **dge**'s analysis which makes the domain finite in height. Wadler gives an abstract interpretation for a non-flat domain (lists) [115] and addresses the difficulties of static time analysis in (lazy) functional languages [116].

Chapter 6

Concurrent Garbage Collection

Programs written in dynamic languages *implicitly* allocate vast amounts of data—both in the construction of the program's dynamic data structures and in building auxiliary structures required by the language's implementation. Since allocation is implicit, the language implementation is responsible for the reclamation of discarded data. Therefore, a language parallelization system must—in addition to finding useful parallelism in the program—address the issue of parallel storage reclamation lest it become a performance bottleneck. This chapter describes the design and implementation of an algorithm that dedicates a processor, running concurrently with the program proper, to the task of reclaiming an ML [78, 79] program's[1] discarded storage for subsequent reuse.

Automatic recycling of storage is known as *garbage collection*. A garbage collector retains a program's data that are in-use (*live*) and reclaims data that are unreachable (*garbage*) by the program. Garbage collection is necessary since even large (virtual) memories are finite. In addition to reclaiming storage, a garbage collector can also restore *locality* to fragmented data by dynamically compacting the live data, thereby improving the performance of memory systems and reducing storage requirements [21, 33]. Automatic storage reclamation, however, introduces disruptive pauses into interactive programming environments and slows program execution.

The concurrent garbage collector described here is a step toward distributing the storage-reclamation task among parallel processors. The design uses a single processor, *the collector*, to reclaim storage for a single program, *the mutator*. Unlike other concurrent-collector designs [10, 106, 41], it requires neither special hardware nor non-standard operating-system support. This concurrent approach potentially reduces the collection overhead for sequential programs since the collector continually recycles the storage that a program, running on another processor, discards. Hence, existing sequential programs benefit directly from this approach. To collect the garbage in parallel systems, I describe an extension to this design that supports multiple concurrent mutators. With this design, program execution—sequential or parallel—will not be disrupted by garbage collection if the concurrent collector reclaims garbage at least as quickly as the program creates it. This is an attractive proposition for interactive environments.

Since a concurrent compacting collector relocates the mutator's dynamic data and allows concurrent

[1] Whereas the techniques of the previous chapters addressed subsets of ML, the concurrent algorithm of this chapter reclaims storage for a complete Standard ML implementation (SML/NJ [11, 9]).

mutator access to these data, the cost of mutator data access and allocation operations must be close to their cost with efficient sequential collection. To this end, the collector design presented here exploits the compile-time distinction between mutable and immutable data that some languages, such as ML, make. Pure functional languages (Pure Lisp [76], Haskell [52], *etc.*) can also be concurrently collected by this approach since they only manipulate immutable data.

After stating the assumptions that underlie the design of the concurrent collector, I briefly describe the sequential algorithm upon which the concurrent algorithm is based (§6.2). I then give the concurrent-collection algorithm (§6.3), its implementation in SML/NJ (§6.4), and performance measurements (§6.5). Extension to multiple mutators and further efficiency improvements are in Section 6.6. Previous work related to concurrent garbage collection is in Section 6.7.

6.1 Assumptions

The concurrent collector design assumes that the mutator and collector share a common address space and that this common memory is *processor consistent* [37].[2] This allows ordered memory operations to implicitly perform fine-grain synchronization without explicit synchronization primitives (*cf.* lock-free synchronization [48]).

The language or compiler must also statically distinguish the program's mutable-data accesses from its immutable-data accesses. This assumption enables the compiler to generate code for mutable accesses that implicitly synchronizes with the concurrent collector. Run-time separation of mutable and immutable data is not necessary.

The collector design assumes that mutator allocation is entirely heap based [5]; *i.e.*, the implementation of the language does not use a run-time stack. This simplifies the exposition of the collection algorithm since it elides the details of stack management during collection.

Critical to the success of this design is the assumption that most mutator data accesses are to immutable data. This is often valid for programs written in dynamic languages (*e.g.*, ML) and certainly valid for purely-functional programs.

6.2 Sequential Copying Collection

This section describes the algorithm for sequential copying collection that forms the basis for the concurrent collector. Further description of copying collection can be found in many places (*e.g.*, [60, 34, 9]).

A basic sequential copying collector [21, 33] requires two memory spaces of equal size. Denote these spaces as *From-Space* and *To-Space*. Figures 6.1 and 6.2 illustrate these spaces and the pointers that manage them.

Mutator allocation of heap objects (dynamic values) always occurs in *From-Space* at the next unused location (*New*). If allocation crosses a predetermined *Threshold*, *From-Space* is deemed full and garbage

[2] Processor consistency guarantees that writes from a processor p appear to all other processors as occurring in the sequence in which p executes them. For example, if processor p_0 executes the sequence (x ← 0; x ← 1; x ← 2) and p_1 reads the value 1 for x then p_1 cannot subsequently read the value 0 for x. Processor consistency is weaker than the sequential consistency that many shared-memory machines provide.

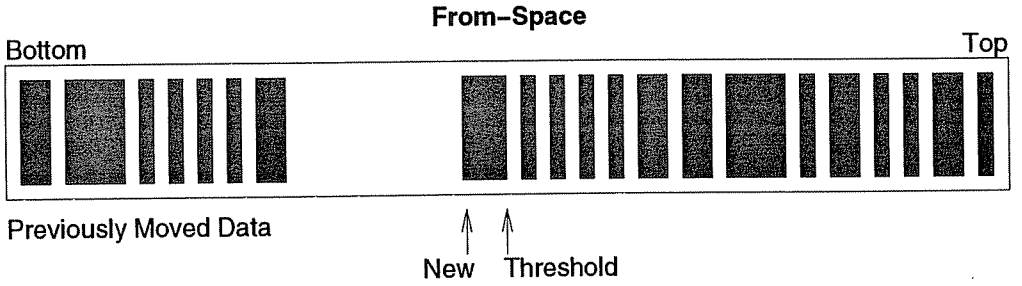


Figure 6.1: Sequential *From-Space*.

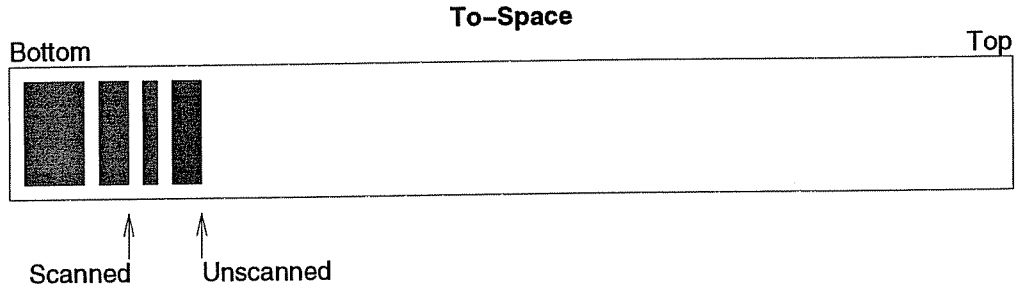


Figure 6.2: Sequential *To-Space*

collection occurs. The *From-Space* (Figure 6.1) is full and requires collection.

At the beginning of a collection, *Scanned* and *Unscanned* point to the bottom of (empty) *To-Space*. *Scanned/ Unscanned* form a queue for a breadth-first traversal of the live data in *From-Space*. *Scanned* points to the head of the queue and *Unscanned* to the tail. This queue is initialized with the objects pointed to by the *root set* of the mutator. Any pointer into the active heap (*From-Space*) that the mutator has access to (*e.g.*, registers) is a *root* and all such pointers constitute the mutator's root set. All live data available to the program are reachable via pointers from this root set. After an object's relocation in *To-Space*, the collector overwrites (*forwards*) the original object in *From-Space* with the object's new location in *To-Space* (its *forwarding* pointer).

Objects in the initialized *Scanned/ Unscanned* queue are scanned (from *Scanned* to *Unscanned*) for pointers into *From-Space*. An uncopied *From-Space* object is inserted into the queue and forwarded. A pointer to a previously copied object is translated to point to the copy using the forwarding pointer. All live data reside in *To-Space* when the queue of unscanned objects empties; that is, when *Scanned* = *Unscanned*. A *flip* operation then reverses the roles of the two spaces: *To-Space* becomes *From-Space* and *From-Space* becomes *To-Space*. The collection is now complete and the collector supplies an updated root set to the mutator. Mutator allocation proceeds anew from the top of the (empty) *From-Space*.

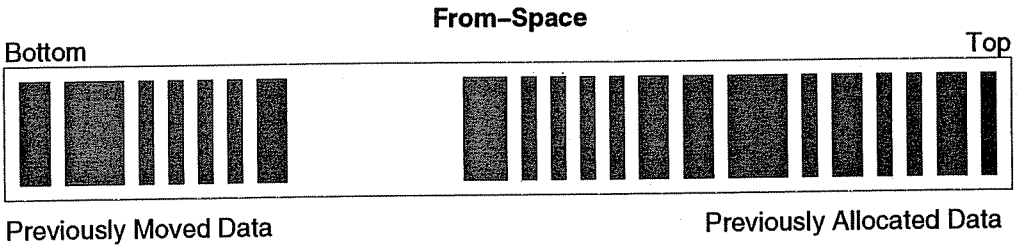


Figure 6.3: Concurrent *From-Space*.

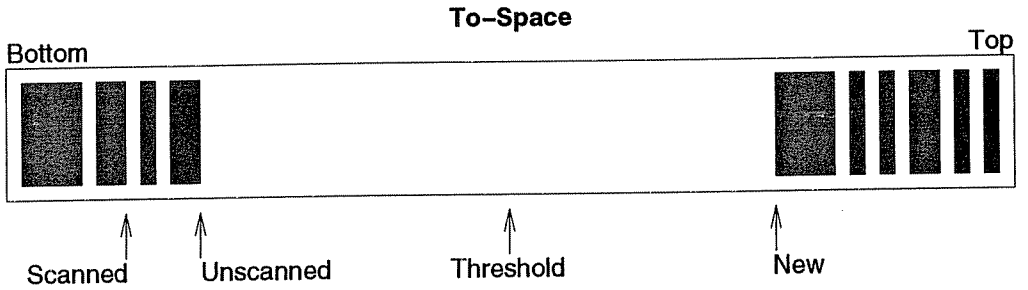


Figure 6.4: Concurrent *To-Space*. This *To-Space* is similar to the sequential *To-Space* (Figure 6.2), but allows the mutator to allocate new data at *New* while the collector concurrently scans and inserts objects into the *Scanned/Unscanned* queue. The collector is also responsible for translating *From-Space* pointers in freshly-allocated data (from *New* to *Top*) to their *To-Space* equivalents.

6.3 Concurrent Copying Collection

The concurrent collector design of this chapter is based on the sequential copying algorithm. It also copies live data between regions, but does not suspend mutator computation during the copying phase. Two modifications to the underlying data structures are required: the layout of the memory spaces differs, and all dynamically-allocated objects contain an additional field for a forwarding pointer. The first modification is to the heap's *To-Space*. Figures 6.3 and 6.4 illustrate the layouts of the concurrent *From-Space* and *To-Space*, respectively.[3] Note that mutator allocation now occurs in *To-Space* and not in *From-Space* as in the sequential algorithm. This is necessary to properly identify and collect all live data while permitting concurrent mutator allocation.[4]

The second modification required by the concurrent collector is that every object r created by either the mutator or collector must contain an additional field f for a forwarding pointer. The field f is distinct from r's data fields. This is in contrast to implementations of the sequential algorithm that are free to overwrite r's data fields with forwarding information. A separate field for f is required since the contents of r and the contents of its copy are simultaneously accessible to the mutator. Let A denote

[3] In all further exposition, "*From-Space*" and "*To-Space*" refer to the concurrent spaces.

[4] If the mutator were to allocate in *From-Space*, it would be difficult to determine when all live data have been copied out of *From-Space* since the mutator's root set continually changes.

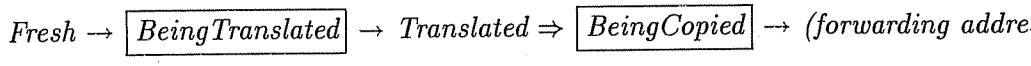


Figure 6.5: Possible transitions an object can undergo. The ⇒ indicates that a flip occurs during this transition. A *Boxed* state is one in which an object is inaccessible to the mutator.

all address values for f. Distinguish four values in A: *Fresh, BeingTranslated, Translated, BeingCopied.* Partition A into mutator *inaccessible* and *accessible* state sets:

$$inaccessible = \{BeingTranslated, BeingCopied\}$$

$$accessible = A \backslash inaccessible$$

An object r is always created with its forwarding pointer set to *Fresh.* Figure 6.5 shows the possible transitions for r's forwarding pointer. The mutator may access a mutable object only when it is in an accessible state. If the state of the object changes during the access, the mutator must redo the access. A change in state is easily detectable since the transitions can be implemented using a monotonic representation (*i.e.*, with reserved address values for the states). An access is performed in the forwarded copy when the state of the object is a true forwarding address. This synchronization is fast—only two fetches from an immutable object are required in the expected case of no mutator-collector conflict. Figure 6.7 contains pseudo-code for mutator read and write operations to mutable data that synchronize in this manner.

Figure 6.6 contains the concurrent collection algorithm. It runs on a dedicated processor. The predicate *Forwarded?*(p) is true if the object at p has been forwarded. *Forward*(p,q) forwards the object at p to location q; that is, it installs q as p's forwarding pointer. *SetState*(p,s) sets the state of the object at p to s. *GetState*(p) returns the state of the object at p. *Translate*(p) returns the forwarding address of the object at p if one exists, p otherwise. *SizeOf*(p) returns the number of fields in the object at p. *TranslateAllocArena* is described below.

As the **repeat** loop in *Collect* indicates, the collector runs continually. The algorithm divides naturally into three parts: flipping (**a**–**e**), copying live data (**f**–**m**), and translating old pointers into *From-Space* to their *To-Space* equivalents (**n**–**p**). I describe these parts in turn.

Lines **a**–**c** prepare the collector for interchanging *From-Space* and *To-Space*. Only during this part of the algorithm is the mutator inactive. Therefore, this section must execute quickly. At line **a**, copies of all live data reside in *To-Space* (newly-allocated data are placed in *To-Space* at *New*). Pointers to *From-Space* may, however, still exist in the new allocation arena (from *New* to *To-Space.Top*). Since all live data have been copied out of *From-Space*, it is only necessary to translate these *From-Space* pointers in the allocation arena to their *To-Space* equivalents. This is accomplished at line **c**. Since the new-data allocation arena may be large, it may not be possible to translate it in its entirety without pausing the mutator for an exorbitant amount of time—overhead we are attempting to avoid in the first place. For this reason, most of the allocation arena is translated concurrently (lines **n**–**p**) with mutator allocation; I discuss this portion of the algorithm below. *TranslateAllocArena* at line **c** only translates a small piece of this arena, namely the data not translated after the last iteration of the loop at line **n**. Line **d** performs the flip and repositions the pointers that govern the semi-spaces.

Lines **f**–**m** perform the actual copying and forwarding of live data from *From-Space* to *To-Space*. The

```
Collect()
    volatile Roots: set of pointers
    repeat
        begin
(a)         suspend mutator
(b)             Roots ←mutator's root set
(c)             Roots ←TranslateAllocArena(Roots)
(d)             Flip(From-Space,To-Space)
(e)         release mutator
(f)         Scanned ←To-Space.Bottom
(g)         Unscanned ←Scanned
(h)         for all r ∈ Roots do
(i)             r ←Copy&Forward(r)
(j)         while Scanned ≠ Unscanned do
                let object = object at Scanned
                    ptrs = set of pointers in object
                in
(k)                 SetState(object,BeingTranslated)
                    for all p ∈ ptrs do
                        p ←Copy&Forward(p)
                    Scanned ←Scanned + SizeOf(object)
(l)                 SetState(object,Translated)
(m)             end
(n)         while mutator still allocating do
(o)             Roots ←mutator's root set
(p)             TranslateAllocArena(Roots)
        end

Copy&Forward(p : object pointer) : object pointer
(q) if not Forwarded?(p) then
        begin
(r)         SetState(p,BeingCopied)
(s)         copy object at p to Unscanned
(t)         SetState(Unscanned,Fresh)
(u)         Forward(p,Unscanned)
(v)         Unscanned ←Unscanned + SizeOf(Unscanned)
(w)         return Unscanned − SizeOf(Unscanned)
        end
(x) else return Translate(p)
```

Figure 6.6: Concurrent collection algorithm.

```
Read(r : object pointer,i : integer) : value
    if Inaccessible(r) ∨ Forwarded?(r) then
        Read(Translate(r),i)
    else
        let  s = GetState(r)
             v = field i of r
        in
            if s ≠ GetState(r) then
                Read(r,i)
            else v
        end
```

```
Write(r : object pointer,i : integer,v : value)
    if Inaccessible(r) ∨ Forwarded?(r) then
        Write(Translate(r),i,v)
    else
        let  s = GetState(r)
        in
            set field i of r to v
            if s ≠ GetState(r) then
                Write(r,i,v)
        end
```

Figure 6.7: Mutator mutable-data access operations to *Read* (or *Write*) field i of an object r. If the concurrent collector has forwarded r, the read (or write) is performed in the copy. If the collector copies or updates r while the mutable access is underway, the access is redone in the copy.

Copy&Forward routine provides implicit synchronization information to the mutator (*cf.* Figure 6.7) by setting the object's state to *BeingCopied* before the actual copy operation ensues (**r**) and makes it accessible after the copy is complete (line **u**). Similar synchronization is also performed during the scan of an object (lines **k** and **l**).

All live data resides in *To-Space* when control reaches line **n**; that is, all live data from *From-Space* have been copied and forwarded into *To-Space*, and all data concurrently allocated by the mutator since the last flip are located in *To-Space*. However, pointers into *From-Space* may still exist in the fresh allocation arena (*New* to *To-Space.Top*) and possibly in the copied data (*To-Space.Bottom* to *To-Space.Unscanned*). This latter case occurs when the mutator updates a mutable location in this region *after* it was scanned and translated by the collector. Such *From-Space* pointers in the copied area are identified by requiring the mutator to maintain a list of mutable locations into which it stores pointers.[5]

TranslateAllocArena is an incremental depth-first marking algorithm.[6] *TranslateAllocArena* is given a set of pointers to translate. It recursively translates all *From-Space* pointers reachable from its parameter into their *To-Space* equivalent and returns a translated version of its parameter. *TranslateAllocArena* uses the *BeingTranslated* and *Translated* states for synchronization in a manner similar to their use in the scanning and copying portions of the collector. The algorithm is incremental in the sense that successive invocations of *TranslateAllocArena* do not reexamine previously-translated objects. This insures only short pauses when translating (mutator) reachable, yet untranslated, objects immediately before a flip (line **d**).

6.4 Implementation

Implementation of the concurrent collector is in the SML/NJ optimizing ML compiler [11]. The SML/NJ implementation is normally collected by an efficient, sequential generational stop-and-copy collector [7].

[5] *Store lists* are commonly used in generational collectors to detect pointers from older generations to younger ones [113, 7].

[6] The stack required for depth-first marking can be allocated explicitly or *threaded* through garbage objects in *From-Space*.

Knuth-Bendix (16MB)			
	Stop-&-Copy	Generational	Concurrent
Exec	200.7s	200.4s	270.9s
GC	117.1	5.2	0.04
Sys	3.6	2.7	7.0
Elapsed	320.9	208.2	277.9
Max Pause	7380ms	420ms	10ms
Avg Pause	2251	199	1

Quicksort (1MB)			
	Stop-&-Copy	Generational	Concurrent
Exec	32.3s	32.4s	40.0s
GC	16.9	4.1	1.4
Sys	0.5	0.3	0.4
Elapsed	49.7	36.8	41.8
Max Pause	220ms	180ms	20ms
Avg Pause	98	31	13

Table 6.1: Performance comparison of the concurrent collector to sequential collectors.

The *sml2c* [110] back end to SML/NJ generates portable C code that can be compiled on many platforms using native C compilers. The concurrent implementation of this chapter runs on a Sequent Symmetry shared-memory computer. Only two of the Sequent's 20 processors were used—one to execute the program (the mutator) and one to collect the program's garbage (the collector).

The compiler was modified to allocate an extra word as the forwarding pointer with all objects. Upon object allocation, this pointer is initialized to *Fresh*. The compiler's code generator was modified to include synchronization instructions on access to mutable data and to indirect through forwarding pointers when two object pointers are tested for equality. Since the compiler's *sml2c* code generator emits C code, these additions were implemented as C functions.

The concurrent collector is written in C. The SML/NJ run-time system, the concurrent collector, and the compiled ML program (produced by *sml2c*) are linked into an executable image. Upon initiation, this image creates two parallel processes that concurrently execute the collector and mutator threads. Communication among the mutator and the collector occurs through shared memory. The collector acquires the mutator's roots by setting a single location. The mutator periodically (*i.e.*, during its checks for sufficient heap space) polls this location. Upon receiving such a request for its roots, the mutator copies its roots into a shared vector and communicates to the collector that the roots are in place. The same vector is used by the collector to communicate updated root sets to the mutator.

6.5 Results

This section presents performance measurements of the concurrent collector for two benchmark programs.[7] I compare the absolute performance of the concurrent collector to that of two sequential collectors: a simple stop-and-copy collector and a fast copying collector [7]. The simple stop-and-copy collector allows informative comparison; it performs the same amount of work as the concurrent collector. The fast sequential collector, on the other hand, performs further optimization by partitioning data into two generations (see §6.6.2 below). Comparison of the concurrent collector to the generational collector serves only to establish the performance that concurrent collection must attain to surpass contemporary garbage-collection capabilities.

For both benchmarks, the heap size was fixed for the entire execution of the program and was set to the smallest size in which the concurrent collector always completed its copying phase (lines **f–m**) before the mutator exhausted its allocation arena. This permitted normal execution of the concurrent collector while also introducing frequent flips of the memory spaces. Since flips introduce pauses, this setting for heap sizes reveals, in an empirical sense, the worst-case pause times.

The first benchmark implements the Knuth-Bendix completion algorithm processing some axioms of geometry (*cf.* [9]). It performs side effects in the form of IO. IO buffers are dynamically allocated heap objects in SML/NJ and subject to collection; hence, they require mutator synchronization when read or written. Table 6.1 gives measurements for garbage collection with the concurrent and sequential collectors. Foremost, note that the concurrent collector only creates low latency disruptions—average

[7] When compiling the concurrent collector, the compiler was instructed to treat all variables as *volatile* since the Sequent's processor-consistent memory is used to synchronize mutator-collector interaction. Volatile variables restrict many compiler optimizations.

pause times are almost two orders of magnitude smaller than the pauses exhibited by the Appel-Ellis-Li collector [10]. It also reduces the time the mutator spends in the garbage collector (less than 1% of the stop-and-copy or generational times).

A comparison of the total elapsed times shows that the concurrent design (Concurrent) improves on the simple collector (Stop-&-Copy) by 13%. Furthermore, with concurrent collection, the program spends very little time in the garbage collector proper (0.04 seconds versus 117.1 seconds for Stop-&-Copy). Note that the sequential generational scheme (Generational) is almost 34% faster than the concurrent scheme even though it uses only one processor. This is because all generational collectors focus their collection efforts on the data that contain the most garbage (*i.e.*, on recently allocated data). The difference in total elapsed times stems entirely from the difference in execution times. The Appel-Ellis-Li collector [10], using non-standard operating system support, attained an 18% improvement over a simple stop-and-copy collector on a single benchmark, but was not compared to a generational collector.

The second program measured is a quicksort applied to a list of 1000 random integers (Table 6.1). Here concurrent collection again improves on its sequential counterpart (Stop-&-Copy). Relative to the sequential generational collector, it is 12% slower. This again emphasizes the need for concurrent generational collection (§6.6.2). As with Knuth-Bendix, concurrent collection of quicksort exhibits low pause latencies: worst-case pause latency is 9 times lower than that of the generational collector and average pause latency is less than half as long. Compared to both sequential collectors, the time the mutator spends in the collector is again small for the concurrent collector.

6.6 Extensions

This section describes possible improvements to increase the concurrent collector's efficiency, and extends the design to data generations and to multiple parallel mutators.

6.6.1 Efficiency Improvements

Inefficiencies in the concurrent collector are because of two factors: overhead incurred by the mutator in correctly cooperating with the collector, and decreased locality inherent in distributing work among parallel processors. An inefficiency in the concurrent algorithm is mutator overhead due to forwarding pointer creation and manipulation. The additional forwarding pointer on every heap object increases the amount of mutator computation on data allocation; it also decreases the amount of heap space available for program data.[8] Depending on the language's implementation, the forwarding pointer can be integrated with the object's usual fields. For example, in SML/NJ all objects carry a word of tag information. It is possible to merge the forwarding pointer with this tag (*e.g.*, [86]); this can reduce allocation costs and improve memory utilization.

Performance degradation due to disrupted locality is a more subtle point. For example, the concurrent collector disturbs mutator locality when it installs a forwarding pointer in an object that resides

[8]Objects in SML/NJ occupy approximately 2.9 words in the heap on average [6]. Adding an additional field to each object therefore reduces heap space by approximately one quarter.

in the mutator's data cache. Furthermore, all data constantly circulate between the mutator's and collector's data caches. I suspect that this effect accounts for a large portion of the concurrent collector's inefficiencies.

6.6.2 Generations

Incorporating generational techniques (*e.g.*, [71, 113]) into this concurrent collector design is paramount to improving its efficiency (*cf.* [10, 29]). This is because the collector continually examines and moves *all* data. As the experiments show, sequential generational collectors still provide better performance than the concurrent collector. These sequential collectors frequently reclaim the garbage in recently allocated data (young generations), and only infrequently reclaim the garbage from long-lived data (older generations). This proves to be effective in practice; old data has been empirically shown to outlive young data. Generations must be incorporated into the concurrent collector so that it, too, can focus its efforts on the data in which most space can be reclaimed for reuse. I do not present an explicit design for generational concurrent collection here.

6.6.3 Parallel Mutators

The concurrent collector can be extended to reclaim the discarded storage of p parallel mutators as follows. On a flip, the collector partitions the new allocation arena (Figure 6.4) into m blocks of size k such that $m \gg p$. Each mutator is given a block in which to allocate. When a mutator exhausts the storage in its current block, it requests an additional block from the collector. In addition to dispensing heap blocks, the collector concurrently copies and translates live data.[9] The collector successively transmits requests for root sets to the p mutators. After dispensing m blocks, the collector halts the mutators and completes the translation of a p_i's untranslated reachable data. Note that a processor must always complete an access to a heap object that is in progress before it may suspend (or be suspended)—this is necessary to prevent a flip from occurring while an object in *From-Space* is still being accessed. Finally, the *From-Space* storage is reclaimed, the flip occurs, and the mutators are released.

For a large number of parallel mutators, the collector should also be distributed among multiple processors, but it is premature to propose such a design here.

6.7 Notes

The concurrent garbage collector of this chapter has been previously published [54].

Various designs for concurrent garbage collectors have been proposed. Most require special hardware or operating system support; few have been implemented on parallel machines.

Sequential copying collectors [21, 33] are an attractive base for concurrent-collector design since they operate in time proportional to the amount of data in use by the system. They also remove fragmentation and restore locality by compacting data as they are copied. Copying collectors are more efficient than reference-counting and mark-and-sweep collectors in theory and practice [14, 5]. Zorn provides analyses and simulations of mark-and-sweep and stop-and-copy collectors [124].

[9] In the same manner as for the case $p = 1$ described in §6.3.

Sequential implementations of dynamic languages provide fast object allocation and access [8, 9, 5]. To retain efficiency, concurrent collectors must not substantially increase the costs of object allocation and access. Central to this goal is Brooks's idea that all dynamic heap objects contain an extra field for a forwarding pointer [19]. His approach sets the forwarding pointer of an object a to point to itself (*i.e.*, to a) if the object has not been copied. When the collector copies the object to a new object a', a's forwarding pointer is set to point to a'. Accesses to a simply indirect through a's forwarding pointer. The collector of this chapter implements Brooks's forwarding pointer, but indirects through it only on access to mutable data. This is possible due to the assumption that the language (or compiler) separates mutable data from immutable data at compile-time. For programs with infrequent access to mutable data, indirection is rarely necessary. The Pegasus system's garbage collector [88] also employs Brooks's technique, but does not provide a parallel implementation and the outlined parallel design relies on the explicit locking of objects.

Real-time response is highly desirable in garbage-collected systems. Baker's algorithm [14] addresses this issue by interleaving allocation with collection, but his algorithm requires special hardware to run efficiently. Halstead [41] extended Baker's algorithm to shared-memory multiprocessors by using fine-grain synchronization on individual objects. Efficient implementation of Halstead's algorithm relies heavily on hardware support. A concurrent lock-free version of Halstead's algorithm is given by Herlihy and Moss [49]. Their algorithm never performs global synchronization. This, however, is achieved only at the cost of expensive object access and allocation. Again no implementation is provided.

Appel, Ellis, and Li implemented the first concurrent copying collector on a stock shared-memory multiprocessor [10]. It directly addresses the efficiency and response-time issues. The Appel-Ellis-Li collector uses operating-system memory-protection devices to detect mutator references to heap objects that have not been collected. Their collector is independent of the language and compiler, but requires modification of the operating system. Response latency of the Appel-Ellis-Li collector is on the order of hundreds of milliseconds—but, as noted by the authors, this is still too slow for some interactive applications. Efficiency of their concurrent collector is measured only in terms of a simple sequential stop-and-copy collector, not relative to common generational collectors [71, 113] that provide efficient sequential collection.

Doligez and Leroy [29] implemented a hybrid concurrent collector for ML. A copying collector reclaims storage in a processor's local memory, and a mark-and-sweep algorithm collects shared global memory. Since mark-and-sweep collection fragments storage, the shared-memory objects in this design must eventually be relocated—the authors do not provide a concurrent relocation algorithm. Worst-case pause latencies of their collector, not including shared-data relocation costs, are on the order of hundreds of milliseconds. This thesis's concurrent collector eliminates such perceptible pauses.

Nettles and O'Toole designed and implemented a *replication-based* sequential collector that incrementally reclaims discarded storage [86]. Of relevance here, is their use of store lists to log updates to mutable data since a similar approach was taken in design presented in this chapter. Nettles and O'Toole also outline the design, but do not provide an implementation, of a concurrent version of their sequential collector.

Chapter 7

Conclusions and Future Work

Dynamic language parallelization is feasible. Programs written in sequential languages with higher-order functions, imperative operators, and implicit storage reclamation can exploit the performance advantages of parallel machines. Although dynamic parallelization incurs run-time costs in maintaining and examining dynamic information about the program's actual data and computation structures, its potentially precise information uncovers parallelism that offsets these overheads and speeds program execution.

The benefits of dynamic language parallelization, as described in this thesis, are threefold. As compared to completely static techniques, dynamic techniques can:

1. better *detect*, and hence, utilize more of a program's parallelism,
2. better *select*, for parallel evaluation, program expressions whose computation granularity corresponds to the granularity of the underlying parallel machine,
3. substantially *reduce* the cost and complexity of the compile-time analyses.

Better parallelism detection and selection improves program performance. A reduction in the cost of static analyses enables separate compilation of program modules and interactive programming environments. In light of these points, I summarize the contributions of this thesis (§7.1). I close with possible directions for future work in dynamic language parallelization (§7.2).

7.1 Contributions

The general contribution of this thesis is the idea of using dynamic information to automatically parallelize imperative programming languages with expressive features (*e.g.*, higher-order functions and dynamic data structures). Foremost, the contents of tags that dynamically propagate with a function at run time is not limited to side-effect information—they can generally carry other function properties (*e.g.*, strictness and load-balancing information). Such tags are an efficient means of obtaining information about a higher-order function. This thesis also posits that shared structure can be effectively detected at run time by maintaining information with the physical nodes that represent this structure. Size information can also be automatically and incrementally constructed for dynamic data structures—this enables

dynamic optimization of functions that traverse these structures. Augmenting conventional static analyses (*e.g*, data-flow formulations and abstract interpretations) with dynamic information leads to new analysis methods; *e.g.*, for some analysis problems (such as load balancing), it is possible to perform an abstract interpretation over infinite, rather than over conventional, finite domains.

The specific contributions of this thesis are the design and implementation of several parallelization techniques. These techniques provide a "proof of concept" for dynamic language parallelization. The specific designs are for higher-order imperative languages with implicit storage management. The specific prototype implementations are for the SML/NJ optimizing ML compiler on a shared-memory Sequent Symmetry computer.

λ-Tagging demonstrates that the dynamic detection of functional parallelism in programs with imperative higher-order functions is feasible and effective. λ-tagging's costs are small; its static analysis is tractable in practice. Measurements of the implementation indicate that λ-tagging's run-time overheads are readily offset by the additional parallelism λ-tagging finds.

Dynamic Resolution demonstrates that sharing in dynamic data structures can be dynamically detected, and that accesses to shared data can be correctly coordinated at run time. Dynamic resolution's static analysis requires only basic interprocedural information (*i.e.*, symbol-table entries) and is therefore inexpensive. Measurements of the implementation indicate that dynamic resolution incurs substantial run-time costs. Nevertheless, dynamic resolution is able to profitably parallelize some non-trivial functions that elude static parallelization.

Dynamic Granularity Estimation demonstrates the viability of dynamically approximating the sizes of data structures for the purpose of dynamically selecting expressions for parallel evaluation based on the amount of computation they require. The static analysis for dynamic granularity estimation in a list-based functional language is inexpensive. Although interprocedural, its cost depends on the constant (implementation and machine dependent) overhead that demarcates when an expression's parallelization becomes profitable. Implementation measurements suggest that two prerequisites for effective dynamic granularity estimation hold: the dynamic cost of maintaining list-length estimates is negligible and the run-time use of this information can substantially improve a program's performance.

Concurrent Garbage Collection is addressed with the design and implementation of a new concurrent collector. The collector uses static type information to reduce the frequency of (expensive) program-collector synchronization. It is the first concurrent copying collector that does not require special hardware or operating systems support. The implementation uses a dedicated processor to collect a sequential program's spent storage. For the programs tested to date, it successfully removes all perceptible garbage-collection pauses.

7.2 Directions for Future Work

I sketch several directions for future research in dynamic language parallelization:

New Dynamic Techniques and Architectures This thesis reports the performance of several dynamic parallelization techniques. These results are for a specific language (ML), compiler (SML/NJ), and computer (Sequent Symmetry). As the architectures of parallel machines evolve, their resources (*e.g.*, computation and communication) change. Architectural advances will alter the effectiveness of dynamic parallelization techniques. Architectural evolution will *guide* the design of dynamic parallelization techniques. The need to parallelize programs with other expressive language features—*e.g.*, arrays, general (cyclic) data structures, and flexible control flow such as continuations—will *prompt* the further design of new dynamic parallelization techniques.

New Languages for Dynamic Parallelization Languages, or language extensions, designed with regard to dynamic parallelization promise to extend the class of programs that can be automatically parallelized. For example, it is possible to envision a language whose static type system completely (and safely) infers whether a dynamic data structure is acyclic (*i.e.*, suitable for dynamic resolution). Furthermore, a language's syntax (and, perhaps its semantics) can be designed to encourage programming styles amenable to dynamic parallelization.

Theoretical Aspects Of theoretical interest is the amount of information a dynamic parallelization technique gathers at run time in proportion to the amount of information it requires at compile time. A technique that, in some sense, requires little information at run time will incur less dynamic overhead and can be expected to outperform techniques that require much dynamic information. Furthermore, I speculate that techniques requiring little static information require extensive dynamic support, and vice versa. For example, dynamic resolution requires relatively little static analysis and a lot of dynamic information, whereas λ-tagging and dynamic granularity estimation require (relatively) more static analyses and only little dynamic information. It would be interesting to characterize—in some manner—the class of language parallelization (and optimization) problems that have effective dynamic solutions; problems that have ineffective compile-time solutions seem to be a good starting point.

Dynamic language parallelization is a fruitful area for further research. It holds the promise of solving problems facing the automatic parallelization of programs—problems for which adequate static solutions are elusive, difficult, or unattainable.

Bibliography

[1] S. Abramsky and C. L. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Ltd., Chichester, West Sussex, England, 1987.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, 1988.

[4] J. R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[5] A. W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25:275–279, 1987.

[6] A. W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[7] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice & Experience*, 19(2):171–183, February 1989.

[8] A. W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3:343–380, 1990.

[9] A. W. Appel. *Compiling with Continuations.* University of Cambridge Press, 1992.

[10] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Conference on Programming Language Design and Implementation*, pages 11–20. Association for Computing Machinery, June 1988.

[11] A. W. Appel and D. B. MacQueen. A Standard ML compiler. *Functional Programming Languages and Computer Architecture*, 274:301–324, 1987.

[12] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[13] J. Backus. Can programming be liberated from the von Neumann style? *Communications of the ACM*, 21(8):613–641, August 1978.

[14] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[15] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Functional Programming Languages and Computer Architecture*, pages 538–568. Association for Computing Machinery, August 1991.

[16] D. Berry, R. Milner, and D. Turner. A semantics for ML concurrency primitives. In *Symposium on Principles of Programming Languages*, pages 119–129. Association for Computing Machinery, January 1992.

[17] G. M. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA BEGIN*. Auerbach Publishers Inc., Piladelphia, Pa., 1973.

[18] J. M. Boyle, K. W. Dritz, M. N. Muralidharan, and R. J. Taylor. Deriving sequential and parallel programs from pure LISP specifications by program transformation. In *Working Conference on Programme Specification and Transformation*. IFIP, North-Holland Publishing Company, 1986.

[19] R. A. Brooks. Trading data space for reduced time and code space in real time garbage collection on stock hardware. In *Lisp and Functional Programming*, pages 256–262. Association for Computing Machinery, August 1984.

[20] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation*, pages 296–310. Association for Computing Machinery, June 1990.

[21] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[22] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, 1981.

[23] E. C. Cooper and R. P. Draves. C threads. Technical Report CMU-CS-88-54, Carnegie Mellon University, February 1988.

[24] E. C. Cooper and J. G. Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[25] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252. Association for Computing Machinery, 1977.

[26] L. Damas and R. Milner. Principle types for functional programs. In *Symposium on Principles of Programming Languages*, pages 207–212. Association for Computing Machinery, January 1982.

[27] J. B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.

[28] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(7):522–526, July 1976.

[29] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Symposium on Principles of Programming Languages*, pages 113–123. Association for Computing Machinery, 1993.

[30] V. Dornic, P. Jouvelot, and D. K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, 1(1):33–45, March 1992.

[31] M. Felleisen. *The Calculi of Lambda-v-CS Conversion in Imperative Higher-order Programming Languages*. PhD thesis, Indiana University, Computer Science Department, 1987.

[32] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts - III*, pages 193–219. North-Holland, New York, N. Y., 1986.

[33] R. R. Fenichel and J. C. Yochelson. A Lisp garbage-collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[34] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[35] C. N. Fischer. *Crafting a Compiler.* Benjamin-Cummings, 1988.

[36] R. P. Gabriel and J. McCarthy. Queue-based multi-processing Lisp. In *Lisp and Functional Programming*, pages 25–44. Association for Computing Machinery, August 1984.

[37] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 15–26. Association for Computing Machinery, May 1990.

[38] D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, September 1987.

[39] R. Goldman and R. P. Gabriel. Preliminary results with the initial implementation of Qlisp. In *Lisp and Functional Programming*, pages 143–152. Association for Computing Machinery, July 1988.

[40] S. L. Gray. Using futures to exploit parallelism in Lisp. Master's thesis, MIT, February 1986.

[41] R. H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Lisp and Functional Programming*, pages 9–17. Association for Computing Machinery, August 1984.

[42] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[43] R. H. Halstead, Jr. An assessment of Multilisp: Lessons from experience. *International Journal of Parallel Programming*, 15(6):459–501, 1986.

[44] W. L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.

[45] W. L. Harrison and D. A. Padua. PARCEL: Project for the automatic restructuring and concurrent evaluation of lisp. In *International Conference on Supercomputing*, pages 527–538, July 1988.

[46] L. J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, August 1990.

[47] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[48] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, January 1991.

[49] M. P. Herlihy and J. E. B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992.

[50] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[51] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Conference on Programming Language Design and Implementation.* Association for Computing Machinery, June 1989.

[52] P. Hudak, S. P. Jones, P. Wadler, et al. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[53] L. Huelsbergen and J. R. Larus. Dynamic program parallelization. In *Lisp and Functional Programming*, pages 311–323. Association for Computing Machinery, June 1992.

[54] L. Huelsbergen and J. R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Principles and Practice of Parallel Programming*, pages 73–82. Association for Computing Machinery, May 1993.

[55] J. Hummel, L. J. Hendren, and A. Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3):243–260, September 1992.

[56] L. S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 1991.

[57] S. Jagannathan and J. Philbin. A customizable substrate for concurrent languages. In *Conference on Programming Language Design and Implementation*, pages 55–67. Association for Computing Machinery, July 1992.

[58] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Symposium on Principles of Programming Languages*, pages 244–256. Association for Computing Machinery, January 1979.

[59] S. L. P. Jones and P. Wadler. Imperative functional programming. In *Symposium on Principles of Programming Languages*, pages 71–83. Association for Computing Machinery, January 1993.

[60] S. P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[61] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Symposium on Principles of Programming Languages*, pages 303–310. Association for Computing Machinery, January 1991.

[62] M. Katz. ParaTran: A transparent, transaction based runtime mechanism for parallel execution of Scheme. Technical Report LCS/TR-454, MIT, July 1989.

[63] D. Keppel, S. J. Eggers, and R. R. Henry. A case for runtime code generation. Technical Report UWCSE 91–11–04, University of Washington, Department of Computer Science and Engineering, November 1991.

[64] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices*, 21(7):219–233, July 1986. Proceedings of the 1986 Symposium on Compiler Construction.

[65] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T, a high-performance parallel Lisp. In *Conference on Programming Language Design and Implementation*, pages 81–90. Association for Computing Machinery, June 1989.

[66] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Symposium on Principles of Programming Languages*, pages 207–218. Association for Computing Machinery, January 1981.

[67] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California, Berkeley, Computer Science Division, May 1989.

[68] J. R. Larus. Compiling Lisp programs for parallel execution. *Lisp and Symbolic Computation*, 4:29–99, 1991.

[69] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Conference on Programming Language Design and Implementation*, pages 21–34. Association for Computing Machinery, June 1988.

[70] D. Le Métayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[71] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[72] L. Lu. *Loop Transformations for Massive Parallelism.* PhD thesis, Yale University, November 1992.

[73] L. Lu and M. C. Chen. Parallelizing loops with indirect array references or pointers. In *Preliminary Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[74] J. M. Lucassen. *Towards the Integration of Functional and Imperative Programming.* PhD thesis, MIT Laboratory for Computer Science, August 1987.

[75] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Symposium on Principles of Programming Languages*, pages 47–57. Association for Computing Machinery, January 1988.

[76] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, pages 184–195, April 1960.

[77] J. Miller. *MultiScheme: A Parallel Processing System based on MIT Scheme.* PhD thesis, MIT, EECS Dept., September 1987.

[78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[79] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[80] E. Mohr. *Dynamic Partitioning of Parallel Lisp Programs.* PhD thesis, Yale University, August 1991.

[81] E. Mohr, D. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Lisp and Functional Programming*, pages 185–197. Association for Computing Machinery, June 1990.

[82] J. G. Morrisett and A. Tolmach. Procs and locks: A portable multiprocessing platform for Standard ML of New Jersey. In *Principles and Practice of Parallel Programming*, pages 198–207. Association for Computing Machinery, May 1993.

[83] Z. G. Mou and P. Hudak. An algebraic model for divide-and-conquer and its parallelism. *Journal of Supercomputing*, 2:257–278, 1988.

[84] P. Naur. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[85] A. Neirynck. *Static Analysis and Side Effects in Higher-Order Languages.* PhD thesis, Cornell University, February 1988.

[86] S. Nettles and J. O'Toole. Replication-based real-time garbage collection. In *Conference on Programming Language Design and Implementation.* Association for Computing Machinery, June 1993.

[87] R. S. Nikhil. Id (version 90.0) reference manual. CSG Memo 284–1, MIT Laboratory for Computer Science, July 1990.

[88] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 113–133. Springer-Verlag, 1987.

[89] J. D. Pehoushek and J. S. Weening. Low-cost process creation and dynamic partioning in Qlisp. In *US/Japan Workshop on Parallel Lisp*, pages 183–199. Lecture Notes in Computer Science, June 1989.

[90] P. E. Pfeiffer. *Dependence-Based Representations for Programs with Reference Variables.* PhD thesis, University of Wisconsin-Madison, 1991.

[91] G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[92] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987.

[93] C. Ponder, P. McGeer, and A. Ng. Are applicative languages inefficient? *SIGPLAN Notices*, 23(6):135–139, June 1988.

[94] N. Ramsey. Concurrent programming in ML. Technical Report CS-TR-262-90, Princeton University, Department of Computer Science, April 1990.

[95] J. Rees and W. Clinger (eds.). Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[96] J. H. Reppy. CML: A higher-order concurrent language. In *Conference on Programming Language Design and Implementation*, pages 293–305. Association for Computing Machinery, June 1991.

[97] J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Cornell University, June 1992.

[98] J. C. Reynolds. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, 13(5):308–319, 1970.

[99] C. Ruggieri. *Dynamic Memory Allocation Techniques Based on the Lifetime of Objects*. PhD thesis, Purdue University, August 1987.

[100] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Symposium on Principles of Programming Languages*, pages 285–293. Association for Computing Machinery, January 1988.

[101] J. H. Saltz, H. Berryman, and J. Wu. Runtime compilation for multiprocessors. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.

[102] J. H. Saltz and R. Mirchandaney. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[103] D. Sands. Complexity analysis for a lazy higher-order language. In *ESOP*, pages 361–376. Lecture Notes in Computer Science, May 1990.

[104] O. Shivers. Control flow analysis in Scheme. In *Conference on Programming Language Design and Implementation*, pages 164–174. Association for Computing Machinery, June 1988.

[105] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, May 1991.

[106] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[107] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[108] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(4), 1992.

[109] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Symposium on Logic in Computer Science*, pages 162–173. IEEE, 1992.

[110] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

[111] P. Tinker and M. Katz. Parallel execution of sequential Scheme with ParaTran. In *Lisp and Functional Programming*, pages 28–39, July 1988.

[112] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, Department of Computer Science, May 1988.

[113] D. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press, 1987.

[114] M. T. Vandevoorde and E. S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, 1988.

[115] P. L. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In S. Abramsky and C.L. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis Horwood Ltd., Chichester, West Sussex, England, 1987.

[116] P. L. Wadler. Strictness analysis aids time analysis. In *Symposium on Principles of Programming Languages*, pages 119–132. Association for Computing Machinery, January 1988.

[117] M. Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 LISP Conference*, pages 19–28, August 1980.

[118] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, September 1975.

[119] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Symposium on Principles of Programming Languages*, pages 83–94. Association for Computing Machinery, January 1980.

[120] D. S. Wise. Stop-and-copy and one-bit reference counting. Technical Report 360, Indiana University, 1992.

[121] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, 1989.

[122] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University, Department of Computer Science, April 1991.

[123] C. A. Zhu and P.C. Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Computers*, 6(36):726–739, June 1987.

[124] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Lisp and Functional Programming*, pages 87–98, June 1990.