*International Journal of Parallel Programming, Vol. 25, No. 5, 1997*

# Dynamic Resolution: A Runtime Technique for the Parallelization of Modifications to Directed Acyclic Graphs

Lorenz Huelsbergen[1]

Static program analysis limits the performance improvements possible from compile-time parallelization. *Dynamic parallelization* shifts a portion of the analysis from compile-time to runtime, thereby enabling optimizations whose static detection is overly expensive or impossible. *Dynamic resolution* is a dynamic-parallelization technique for finding loop and nonloop parallelism in imperative, sequential programs that destructively manipulate dynamic directed acyclic graphs (DAGs). Dynamic resolution uses runtime reference counts on heap data, a runtime linearization of threads, and a simple static analysis to dynamically detect potential heap aliases and to correctly coordinate parallel access to shared structures. We describe dynamic resolution in the context of two imperative procedures: DAG rewrite and destructive quicksort. The description is couched in the pointer-safe language ML; with some programmer assertions and custom macros for pointer and memory manipulation, dynamic resolution is applicable to pointer-unsafe languages (C extended with threads) as well. Furthermore, with programmer identification of cyclic structure, dynamic resolution can be used to find parallelism in programs that manipulate cyclic structures. Shared-memory implementations of dynamic resolution for ML and C have attained parallel speedup for nontrivial sequential procedures such as destructive quicksort; empirical speedup results obtained on fast contemporary machines are given.

**KEY WORDS:** Dynamic language parallelization; runtime pointer analysis; medium-grain parallelism; automatic compiler parallelization.

[1] Bell Laboratories, Lucent Technologies, 700 Mountain Ave., Murray Hill, New Jersey, 07974. E-mail: lorenz@research.bell-labs.com.



0885-7458/97/1000-0385$12.50/0 

## 1. INTRODUCTION

Parallelization of irregular computations involving mutable dynamic data structures is difficult for programmers and compilers alike. Programmers must reason about shared substructures in a program's dynamic data to correctly synchronize parallel access to shared state. Parallelizing compilers, moreover, must statically infer structure sharing and produce a safe evaluation and synchronization schedule.[8, 22] The difficulty lies in the dynamic nature of irregular computations—shared structure appears and disappears dynamically. Hence, the available parallelism, and its attendant synchronization requirements, necessarily vary during program execution.

This article describes the design of a *dynamic-parallelization technique*[3, 4] called *dynamic resolution* (**DR**). In pointer-safe languages (such as ML[5]) dynamic resolution can automatically parallelize program procedures that destructively manipulate directed acyclic graphs (DAGs). For programs that manipulate cyclic structures, dynamic resolution—in conjunction with programmer identification of cyclic and acyclic structures—can also find parallelism in the program's computations on acyclic data.

Dynamic resolution dynamically detects and dynamically schedules potentially conflicting DAG modifications; it preserves the program's sequential semantics by *resolving* conflicts at runtime. Dynamic resolution is interprocedural and higher order in that it finds expression-level parallelism across procedure calls. Coupled with another dynamic-parallelization technique called $\lambda$-tagging,[3, 4] **DR** can furthermore find such parallelism in the presence of higher-order functions ($\lambda$-tagging is a general technique for dynamically propagating properties of a function as a tag on its runtime closure.) In pointer-unsafe languages (such as C[6] with user-level threads), **DR**, along with programmer assertions on data types and custom macros for memory and pointer manipulation, can be used to dynamically detect and exploit parallelism.

Static pointer analyses (e.g., Refs. 1, 2, 7–11) cannot provide precise parallelization information in the presence of dynamic structure sharing—such analyses must conservatively assume that *if* sharing can occur, then it *always* occurs. Hence dynamic approaches are needed to parallelize programs that may statically share structure when in fact they do not dynamically share structure.

Dynamic resolution is a hybrid: a *static component* computes inexpensive, but partial, information about the program at compile time; a *dynamic component* gathers and maintains information about the program's heap structure at runtime. The combination of static and dynamic information is then used to find and utilize parallelism necessarily obscured by

compile-time approaches. **DR**'s hybrid structure can discover and effectively utilize parallelism in nontrivial imperative procedures such as destructive quicksort.[12]

Implementations of **DR** have been built for the Standard ML of New Jersey[13] ML compiler and for C with user-level threads. Details of the ML implementation—and the experiments conducted therein—have been previously reported.[3,4] This article serves to first codify **DR** and its extensions in an implementation-independent manner; and then to present measurements of C parallelized via **DR** annotations. In particular, we present empirical measurements that compare C with **DR** to hand-parallelized C (unsafe parallelism) and to sequential C on a contemporaneously fast shared-memory multiprocessor. On the programs we examine, we find that **DR**—although it incurs runtime overheads—already "breaks even" with only a few processors; that is, parallelism discovered by **DR** readily offsets **DR**'s runtime costs and enables it to outperform its sequential counterparts. Furthermore, we show that dynamic resolution is competitive with explicitly parallel versions of the C programs.

This article also studies the impact of shared structure on the performance of dynamic resolution. Concurrent access to shared structure dynamically selects sequential execution under the dynamic-resolution evaluation model. Dynamic sharing can therefore inhibit **DR** parallelization. In contrast, completely static techniques must conclude that if sharing *can* occur, then it *always* occurs. **DR** instead adapts to the actual sharing present at runtime. It is our assumption that sharing occurs in programs, but in many programs does so only infrequently. Empirical evidence suggests that **DR** can tolerate some sharing and still outperform a sequential implementation.

The next section is an overview of the problem dynamic resolution addresses; it provides the main example used throughout the paper. Section 3 provides definitions and notation. Section 4 explains the idea underlying dynamic resolution. Sections 5 and 6 respectively describe **DR**'s static and dynamic components. Extensions and optimizations are in Section 7. Section 8 contains an example of **DR** applied to quicksort. Section 9 briefly describes application of **DR** in pointer-unsafe languages. Implementation and measurements of C-based dynamic resolution are described and reported in Section 10. Related work is in Section 11 and Section 12 concludes.

## 2. OVERVIEW

Parallel evaluation of program expressions that read (get) and modify (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 language. A program's data-sharing characteristics, however, depend on the program's dynamic data structures which often depend on the program's input. Not surprisingly, dynamic data structures are difficult to precisely analyze at compile time.[7, 10, 11, 14–17]

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

The incnode function of Fig. 1 illustrates the problem and will serve as the example of automatic parallelization using dynamic resolution. The incnode function operates on dynamic data of the tree datatype. Function incnode's single parameter has type int tree; that is, internal nodes contain integer reference values and two subtrees. When supplied a leaf node, the incnode function does nothing. Otherwise, when supplied an internal node, incnode first increments the integer reference value at that node and then recursively descends into the node's left and right subtrees.

The sequential semantics of ML requires that all modification (with set) of a reference value $r$ by the expression (incnode left) occur before expression (incnode right) accesses $r$. Similarly (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, however, requires the compiler to ascertain whether (and where) sharing exists in incnode's argument.

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

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

Fig. 1. The tree datatype and the incnode function. A tree is a Leaf or a Node. Leaf is a nullary constructor; Node is a ternary constructor. A Node contains a mutable reference of type $\alpha$ and two subtrees. Note that DAGs can also be constructed from this datatype. Dynamic resolution can safely evaluate expressions (incnode left) and (incnode right) in parallel since it detects conflicts, due to potential sharing in incnode's argument (of type int tree), dynamically.

Static extraction of parallelism from incnode is difficult because the tree datatype can be used to both construct DAGs as well as trees. For example, the expression

```
let val n = Node(ref 0,n1,n2)
in
    Node(ref 0,n,n)
end
```

creates a DAG with sharing using the tree datatype. Figure 2 depicts valid arguments to incnode with and without sharing: a tree and a DAG (the one constructed in this let expression). [Note: cyclic structures cannot be arguments to incnode since ML's type system prohibits the introduction of a cycle into a structure of type int tree.] A naive parallel version of incnode that simply evaluates expressions (incnode left) and (incnode right) concurrently without coordinating internal-node accesses cannot ensure correct results in the references. Because of race conditions, concurrent get and set operations to shared structure may produce indeterminate values. With naive parallel evaluation, for example, incnode applied to the ● node in the DAG of Fig. 2 may produce indeterminate results since expressions can concurrently access the same reference values—references in and below 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 nodes of disjoint trees that are subgraphs *within* a DAG (such as in the structure below the ● node in Fig. 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

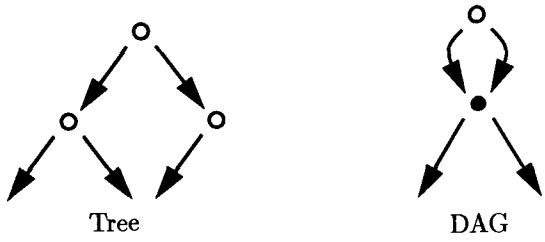



Fig. 2. Possible structures of type α tree. **DR** is able to find parallelism in DAGs with sharing.

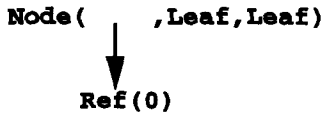
of precise approximation (see Section 1). As another example of such a program, consider the tree and DAG of Fig. 2 both reaching an application of incnode via a conditional whose predicate is statically unknown—in this case, static techniques forgo parallelism in incnode since they must conservatively approximate incnode's argument as always containing shared nodes.

The dynamic-resolution technique described herein can automatically extract parallelism from incnode.

## 3. PRELIMINARIES

In this section we provide definitions for describing heap structure. They are used in the presentations of dynamic resolution's static and dynamic components in the subsequent sections.

ML datatype constructors build *dynamic values*—values that reside in dynamically-allocated storage in the program's *heap*. Dynamic values are references, tuples, and recursive data structures created with (non-nullary) data constructors. We denote the heap as $\mathscr{H}$. Implementations represent a program's dynamic values as *nodes* in $\mathscr{H}$. A node $h \in \mathscr{H}$, representing a dynamically-allocated value, contains basic values directly (e.g., integers and nullary constructors) and *links* to other nodes in $\mathscr{H}$. For example, the expression Node(ref 0,Leaf,Leaf), using the tree datatype of Fig. 1, creates the structure




in $\mathscr{H}$ that consists of two heap nodes and one link. The heap $\mathscr{H}$ 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). Node $h \in \mathscr{H}$ is a simple *node if* $\textit{in-degree}(h) \leqslant 1$.

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

Join nodes will serve as indicators of potentially shared dynamic data.

**Definition 3** (Path). A path of length $n$ in $\mathscr{H}$ is a sequence of nodes, $\langle h_1, \ldots, h_n \rangle \in \mathscr{H}$ where $n > 1$, such that $\forall i, 1 \leqslant i < n$, 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 $h \Rightarrow h'$. The nonexistence of a path from $h$ to $h'$ is noted $h \not\Rightarrow h'$. If $h \Rightarrow h'$, 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, $\langle h_1, ..., h_n \rangle \in \mathcal{H}$ 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 a simple path from $h \in \mathcal{H}$ to $h' \in \mathcal{H}$ as $h \rightarrow h'$. The notation $h \not\rightarrow h'$ denotes that no such path exists. If $h \rightarrow h'$, then node $h$ is said to *simply* reach node $h'$.

The relations $\Rightarrow$, $\not\Rightarrow$, $\rightarrow$, and $\not\rightarrow$ collectively comprise the *reaching relations* for heap nodes.

**Definition 5** (Acyclic Node). Node $h \in \mathcal{H}$ is an acyclic node if all paths from $h$ to $h$ have 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 of an expression that can bind dynamic values or functions will also be necessary. As usual, let FV($e$) be the set of free variables in $e$. The *free dynamic variables* of an expression $e$ are:

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

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

$$\text{FFV}(e) = \{f \in \text{FV}(e) \mid f \text{ has type } \tau \rightarrow \tau'\}$$

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 all dynamic values accessible in $f$ are either created in $f$ or are parameters to $f$. Otherwise, $f$ is said to be *untrue*.

**Definition 6** (True Function). Function $f$ is a true function if $\text{FDV}(f) = \varnothing$ and if $\forall g \in \text{FFV}(f) \backslash \{f\}$ 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 (Cons(x,xs)) =
    let fun g y = Cons(y+1,xs)
    in
        g x
    end
```

f is true since FDV(f) = ∅ and FFV(f) = {Cons, +}. (The list constructor (Cons) and integer addition (+) are true functions.) Function g is an untrue function since it accesses the dynamic value bound to xs; i.e., FDV(g) = {xs}.

## 4. DYNAMIC RESOLUTION PROPERTY

In this section we describe the basic idea underlying dynamic resolution.

To safely evaluate two expressions $e$ and $e'$ 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 accesses to this data. Initially, evaluation of the two expressions can proceed in parallel with $e$ having priority over $e'$ in the following sense. Upon detection of an access to any shared data by $e'$, all further evaluation occurs sequentially; i.e., $e'$ must suspend on an access to shared data and may not resume until $e$ completes. When a processor suspends an expression's evaluation, it need (and must) not idle but should rather evaluate other available expressions. Suspending $e'$ 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 evaluate both expressions completely in parallel.

The detection of shared data and the coordination of accesses to this data (i.e., deciding which expression to suspend) occurs dynamically. A dynamic-resolution compiler can automatically insert code into the program text to detect potential sharing at runtime; and, the **DR** runtime system can govern 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 runtime. This analysis relies on the following property concerning paths and nodes.

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

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. This property enables the static selection of program expressions for which all shared data can be detected dynamically.

Figure 3 illustrates this 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 $h, h' \in \mathscr{H}$ such that this property ($h \not\mapsto h' \wedge h' \not\mapsto h$) 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 (due to 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., $\nexists h'' \in \mathscr{H}$ such that $h \Rightarrow h'' \wedge h' \Rightarrow h''$) then it is possible for $e$ and $e'$ to completely evaluate in

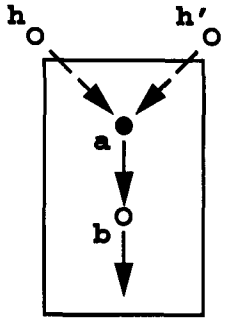



Fig. 3. The nonexistence of simple paths from nodes $h$ to $h'$ and from $h'$ to $h$ imply that the shared structure reachable from $h$ and $h'$ (boxed region) is always guarded by a join node (node $a$). Dynamic resolution detects potential sharing by checking for join nodes at runtime.

parallel with dynamic resolution. Otherwise, **DR** evaluation of $e'$ will 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, *some* evaluation of $e'$ may still be concurrent with that of $e$. Dynamically, the program detects accesses to join nodes and correctly schedules $e$ and $e'$ (Section 6).

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

## 5. STATIC COMPONENT

Here we first informally describe **DR**'s static component. Sections 5.1–5.4 supply the technical detail.

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 if the **DR** property holds. For such expressions, access to shared data can be detected and correctly coordinated at runtime.

Static **DR** parallelization occurs at the function level. For a function $f$, the static component first identifies the data constructors in $f$'s patterns that always (dynamically) bind acyclic nodes (Section 3). [Note: Patterns (see e.g., Ref. 18) 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 reveals the positions of a pattern's variable relative to the pattern's other variables and constructors.] Static classification of a datatype constructor as acyclic (i.e., it only matches acyclic nodes) in turn enables 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 (Section 5.2) of strong (i.e., $\not\rightarrow$) reaching relations among the constructor's variables. Such reaching relations permit **DR** parallelization because shared structure accessible from these variables can be dynamically detected by **DR**'s dynamic component (Section 6). Given such reaching relations, expressions are statically selected and restructured (Section 5.3) 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 (Section 5.4). [Note: without loss of generality, we assume that

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

We 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, we describe how to select candidate **DR** expressions and where, in the program text, to place the checks that detect sharing.

## 5.1. 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 cannot always be dynamically detected with join nodes. For simplicity, we first assume all patterns in the program contain at most one data constructor—this restriction is relaxed in Section 5.2. The form of such a pattern is

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

where $C$ is a data constructor and the $x_i$, $0 < i \leqslant n$, are variables that are bound when $p$ is matched. [Note: the language's constants (e.g., integers) may also appear in pattern. However, since they are not dynamic values they cannot reach shared data and hence require no special treatment.] For example, the pattern Node(x,left,right) of the tree datatype (Fig. 1) contains the data constructor Node and variables x, left and right.

For a pattern $p$ of therefore above, **DR**'s static component classifies $p$'s constructor $C$ as cyclic or acyclic. We describe two possible methods of attaining this classification: from static type information (inferred automatically in ML) or from programmer-supplied assertions.

### *5.1.1. Classification from Static Types*

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 at compile time. A pattern's type, therefore, indicates whether the data it can match contains reference values. Hence,

type information can identify pattern constructors that always match acyclic nodes.

For example, the pattern $p \equiv$ (Node(x,left,right)) in the incnode function (Fig. 1) has type int tree since the contents of x is used in an integer addition. Pattern $p$'s dynamic variables (left and right) also have type int tree. This type information insures that $p$ always dynamically matches an acyclic node in the heap since the reference values in a structure of $p$'s type can only contain integers.

### *5.1.2. Classification from Assertions*

In a language with polymorphic datatypes (ML), static determination of whether a constructor only builds acyclic nodes is not always possible. In a language without strong static typing, this is furthermore often impossible. Since dynamic resolution is applicable only to acyclic data, we describe an assertion mechanism that allows a programmer to declare a recursive ML data structure as acyclic. With assertions for acyclic data, **DR** may be applied to programs that compute with cyclic structures—parallelism discovered via **DR** is restricted to that in functions traversing only acyclic data.

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 (see Section 5.2 next). 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 Fig. 4). A compiler cannot generally infer that the list-constructor Cons matches acyclic nodes. For example, $l$ is a valid argument to the standard map function (see e. g., Ref. 18 for its

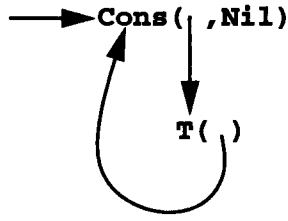



Fig. 4. Cyclic list constructed with conventional Cons.

definition)—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. We introduce the acyclic qualifier for programmer assertion that a datatype's constructors are used only to create acyclic nodes.

For example, the acyclic modifier can be used to declare a polymorphic acyclic-list type:

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

The programmer must ensure that list nodes constructed with the acyclic Cons' constructor never lie on a cycle in the heap. Note that this restriction concerns only the spine of a list thus constructed. Elements of an acyclic list, however, may be cyclic structures; elements may also share structure (Fig. 5). The list of Fig. 4, however, is not a valid acyclic list since it violates the declaration of acyclic. Note that the compiler cannot in general detect such violations; incorrect usage of the acyclic declarator can cause indeterminate program behavior.

The function map' of Fig. 6 is an acyclic version of map that can only be applied to lists of type α list'. The dynamic-resolution technique can be applied here because Cons' may only bind acyclic nodes. Hence, the compiler can infer strong reaching relations for its variables ($x \not\rightarrow xs$ and $xs \not\rightarrow x$). Even if the higher-order parameter f performs imperative get and set operations it may still be possible to evaluate the expressions of map' in parallel; static effect inference[19] may ascertain at compile time that f's effects admit parallelization or a $\lambda$-tag[3] may–convey this information at runtime. $\lambda$-tagging is general technique for propagating properties of a function $f$ with $f$'s closures at runtime. In this situation, the $\lambda$-tag carries

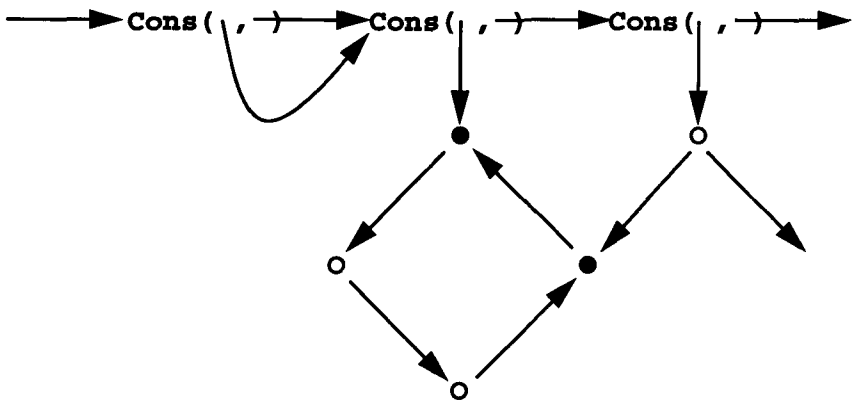



Fig. 5. An acyclic list suitable for dynamic resolution. An element of an acyclic list may reach tail elements, list elements may themselves be cyclic structures, and multiple list elements may reach shared structure.

a boolean value indicating that the higher-order function admits dynamic resolution. Compiler-generated code checks the $\lambda$-tags of higher-order function parameters ($f$) to function $g$ on entry to $g$ and selects parallel **DR** evaluation only when $f$ is amenable to dynamic resolution. $\lambda$-tags are assigned statically, when possible; otherwise, they are computed (from the $\lambda$-tags of other functions) when a closure for $f$ is created (cf. Section 8).

## 5.2. 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 $x_i$, $0 < x \leqslant n$, 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 $0 < j, k \leqslant n$ and $j \neq k$. Proof of this follows. Let $h, h' \in \mathscr{H}$ denote the nodes bound to two of $C$'s variables $x_j$ and $x_k$ (where $0 < j, k \leqslant n$ 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

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

Fig. 6. The map' function for acyclic lists.

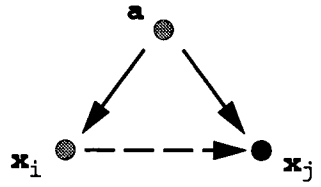



Fig. 7. Node $a$ is an acyclic data-constructor node. Nodes $x_i$ and $x_j$ are directly reachable—via a single link–from $a$. Any path from $x_i$ to $x_j$ is not simple because it always contains a join node ($x_j$). Such a path cannot use the link from $a$ to $x_j$ since $a$ is acyclic. Black nodes are join nodes; gray nodes represent any (simple or join) node.

simple path cannot exist from $h$ to $h'$ (i.e., $h \not\rightarrow h'$). 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 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 $h$ (i.e., $h' \not\rightarrow h$).

Figure 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 runtime because a join node is always encountered before an expression reaches any shared structure.

In Section 5.1, 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. If the constructors $C$ and $C'$ in the general pattern

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

are acyclic, the reaching relations

$$x_j \not\rightarrow x_k \qquad 0 < j, k \leqslant n \wedge j \neq k$$

$$x_j \not\rightarrow y_k \qquad 0 < j < n \wedge 0 < k \leqslant m$$

$$x_n \Rightarrow y_k \qquad 0 < k \leqslant m$$

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.

## 5.3. 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 as follows.

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

1. $\forall x \in \mathrm{FDV}(e)$ and $\forall x' \in \mathrm{FDV}(e')$ the relations $x \not\rightarrow x'$ and $x' \not\rightarrow x$ hold.
2. $\forall x \in \mathrm{FFV}(e)$, $f$ is a true function; and, $\forall f' \in \mathrm{FFV}(e')$, $f'$ is a true function.
3. $\forall x \in \mathrm{FDV}(e)$, $x$ does not contain untrue functions; and, $\forall x' \in \mathrm{FDV}(e')$, $x'$ 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 (Section 6). 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 as free variables in $e$ and $e'$. Since **DR** admits higher-order functions, the last criterion requires $e$ and $e'$ to not apply untrue functions contained in their accessible dynamic data; it thereby prohibits access to (arbitrary) dynamic values through the free variables of higher-order untrue functions stored in dynamic data. [Note: the type of a free dynamic variable indicates whether any structure is may bind can contain functions.]

The example 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(x,left,right)) in incnode induces the set $\{x \nrightarrow y \mid x, y \in \{\text{x,left,right}\} \wedge x \neq y\}$ of relations for $p$'s corresponding function body. Thus, since $\text{FDV}(e) = \{\text{left}\}$ and $\text{FDV}(e') = \{\text{right}\}$, 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 $e'$ meet the second and third criteria. Figure 8 reflects the selection of (incnode left) and (incnode right) for parallel evaluation *provided* that all shared data is dynamically detected and access to this data dynamically coordinated. This detection and coordination is performed by dynamic resolution's dynamic component, described later. We use the sequence separator $;_{\|dr}$ to specify parallel evaluation with sharing detection of the expressions it separates.

## 5.4. Check Placement

The last responsibility of dynamic resolution's static component is the identification, in the program text, of all heap-node accesses 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 (either cyclic or acyclic) in a pattern. Placing a check on every dynamic-value access ensures that sharing (i.e., a join node) is dynamically detected along any path in the dynamic data that the program traverses. These checks examine the status (join or simple) of the node matching the constructor. In Fig. 8, the (de)constructor $\overline{\text{Node}}$ must check the status of the nodes it matches before it accesses any of their fields. The result of this check (join or simple) governs the program's subsequent behavior; the full dynamic operation of these checks is discussed in Section 6.3

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

Fig. 8. The incnode function with annotations for dynamic resolution. The expressions (incnodeDR left) and (incnodeDR right) are evaluated in parallel using dynamic resolution. An overlined constructor requires a check for sharing of the matched heap node before any access of its components.

## 6. DYNAMIC COMPONENT

Dynamic resolution's runtime component does two things: it detects when an expression is about to access potentially shared data and it controls which expression may access such data. Shared data is detected by checking, upon a heap-node access, whether the node being accessed is a simple node or a join node. To control access, **DR** maintains a *total order* of all concurrently-evaluating *threads* which reflects the evaluation order—with respect to side-effects—required by the language's sequential semantics. (An expression is dynamically scheduled for concurrent evaluation as a *thread.*) This *linearization* governs only the order in which threads side-effect potentially shared structure; it does not restrict the parallel evaluation of expressions that perform no side effects or that alter only unshared state. Before access to potentially shared data, an expression examines its position in the thread linearization to determine whether it may access the data or must wait for the evaluation of other expressions (threads earlier in the order) to complete.

Before describing the details of **DR**'s dynamic component, we note that sharing detection and thread scheduling constitute the runtime overhead of dynamic resolution. Additional parallelism discovered by **DR** must offset its cost for **DR** to be effective. Note that **DR** overhead is itself "parallel;" its cost distributes over available processors. Experimental measurements of **DR** implementations exhibit this behavior (Section 10 and Refs. 3, 4).

### 6.1. 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 from other nodes incident on $h$—thereby, reference counts reveal information about the heap's structure. A nonlink pointer to a node $h$ (e.g., a local variable pointing 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. A node with a reference count $\leqslant 1$ 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 semantics. [Note: 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 nonlink pointers are not included in the node's reference count.]

If a thread has access to a simple node, no other thread has concurrent access to this node. Expression selection (Section 5.3), 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 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. Assignment to a reference value increments the count of the (dynamic) value being assigned; assignment also decrements the count of the (dynamic) value being overwritten with the following proviso: reference counts are *sticky*—counts of two never change. Therefore, a join node can never become 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. If reference counts are not sticky, this would violate the invariant that a simple node is accessible to at most one concurrent thread because the thread evaluating $e$ has access to $v$ (through the local binding). 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 were to 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. It is possible to use garbage collection to reconstitute reference counts that have become imprecise; Section 7.3 describes such an approach.

Atomicity is *not* necessary for the reference-count increment and decrement operations. This is because of the invariant that simple nodes are not concurrently accessible. Since the reference counts of join nodes are never decremented, changing the reference count on a join node also requires no synchronization. [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 incorrect uncoordinated access to a join node.]

## 6.2. Parallel-Thread Linearization

Dynamic resolution's runtime system imposes a total order on the program's concurrently evaluating expressions (threads) with respect to side-effects. A 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. It serves solely to order side-effects to shared data—it does not constrain parallel evaluation of threads that are side-effect free or that mutate unshared data.

A thread descriptor has three fields: the thread, the thread's run state, and a pointer to the next thread descriptor. A thread can be in one of three run states: *active*, *suspended*, or *finished*. The **DR** runtime system also maintains a single global pointer to the head of the linearization (the *head* thread).

Threads are inserted into the linearization as follows. A thread $t$ evaluating the expression $(e;_{\|dr} e')$ creates a new thread $t'$ to evaluate $e'$. Thread $t$ continues with the evaluation of $e$. A descriptor is created for $t'$ that is inserted into the list directly behind the descriptor of $t$ in the linearization. This will force $t$ to complete before it may access mutable shared state. Upon creation, a thread descriptor's run state is set to active.

When the head thread finishes, the head is moved to the next thread descriptor in the linearization that has not finished. If the associated thread is in the suspended state it is restarted. The computation is complete when the head reaches the end of the linearization.

It is important to note that the linearization is a concurrent data structure—insertions of thread descriptors by different threads occur in parallel without synchronization. As such, the linearization does not sequentialize the program.

## 6.3. Expression Scheduling

The head thread in the linearization may freely access any node (join or simple) that it can reach. Nonhead 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 it is the head thread; i.e., until all prior threads have completed. On access to a join node, a nonhead thread sets the run state in its thread descriptor to *suspended* and then

suspends itself. [The processor that suspends a thread proceeds to evaluate the (nonsuspended) threads.] A nonhead thread that completes without accessing a join node sets its descriptor's run state to *finished.* When the head thread completes, the next uncompleted (run state $\neq$ *finished*) 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.

This scheduling scheme preserves the language's sequential semantics because, in an expression $(e;_{\|dr} e')$, $e$ (and threads created by $e$) may access all data potentially shared with $e'$ (and threads created by $e'$) before $e'$ is given access to this data. In the absence of sharing, $e$ and $e'$ evaluate concurrently without synchronization under dynamic resolution.

Figure 9 depicts dynamic resolution of an application of the example incnode function (Fig. 8) applied to a DAG. Straight uni-directional arrows ( $\rightarrow$ ), join nodes ( $\bullet$ ), and simple nodes ( $\circ$ ) constitute incnode's DAG argument. The boxes are thread descriptors in the linearization. Text labels indicate a thread's run state: "A" denotes *active* and "S" denotes *suspended.* In the linearization, bi-directional arrows ( $\leftrightarrow$ ) represent the next-prev link between adjacent descriptors. Curved solid arrows ( $\rightsquigarrow$ ) emanating from descriptors point to the heap node which the thread's

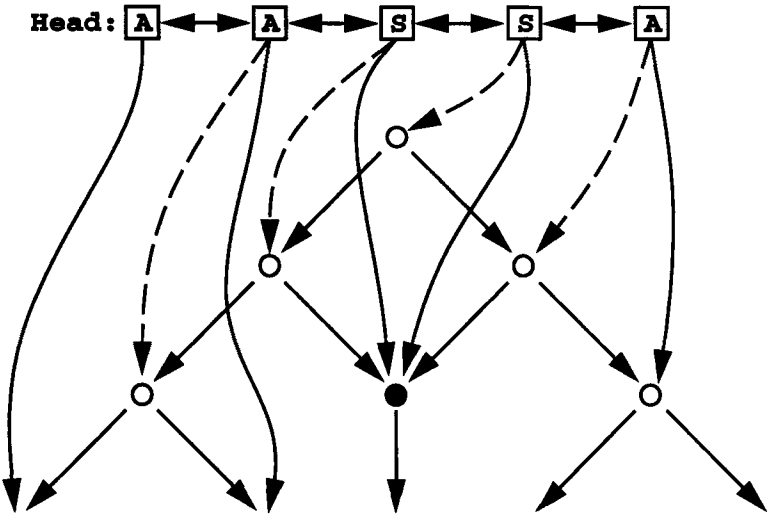



Fig. 9. Operation of the thread linearization during dynamic resolution of incnode applied to a DAG. Boxes denote active (A) and suspended (S) thread descriptors; circles and straight uni-directional arrows comprise the argument DAG.

expression is accessing. Curved dashed arrows (⌒⌒➤) emanating from descriptors designate the node at which the descriptor's thread was created.

In Fig. 9, 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.

## 7. EXTENSIONS

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

### 7.1. 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, a compiler can generate two versions 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. [Note: the PARCEL system similarly creates multiple, specialized function versions.][1]

### 7.2. 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. Nonhead threads, however, must still check reference counts and suspend their evaluation on access to join nodes.

## 7.3. 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) some of the program's dynamic data is temporary and may quickly become 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. A language implementation's *garbage collector*[20, 21] reclaims and recycles the program's discarded data; it is a natural place in an implementation for performing reference-count reconstitution. Reference-count reconstitution requires a collection algorithm that traverses all live links in the heap. During traversal, the collector counts and records (as the reconstituted reference count) the number of links incident on a live node. In a mark-&-sweep collector, for example, the mark phase visits all links and can thus restore accurate reference counts during its traversal. Here, we fully describe reference-count reconstitution for a *copying* collector (e.g., Refs. 22 and 23).

Reference-count reconstitution, in a copying collector, 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 6.1: 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.

## 8. EXAMPLE

The ML program in Fig. 10 provides a further example of **DR**'s operation. The mqs function destructively sorts a list of elements using the quicksort algorithm.[12] It performs the sort in place; that is, the links of the argument list's runtime representation are modified during the sort. The programmer has declared two acyclic datatypes: $\alpha$ pair and $\alpha$ mlist. The acyclic $\alpha$ pair datatype constructs binary tuples of identically-typed values. The $\alpha$ list datatype constructs *mutable* lists with elements of type $\alpha$—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

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

and works as follows. The first parameter is a binary boolean predicate that compares elements of the second parameter, the mutable list to be sorted. The function returns a sorted mutable list. If the list to sort is empty (mNil), the empty list is returned. Otherwise, the argument list is decomposed—via pattern match—to the list's head (x) and tail (xs'). The auxiliary function split partitions its argument list (l) with respect to a pivot element into two lists: the elements that satisfy predicate p (the "<" relation on integers, for example) and the elements that do not satisfy p. In mqs proper, the tail of the list to be sorted (xs') is partitioned using the head (x) as the pivot. Quicksort is then recursively applied to the two lists returned from split. Sorted sublists are reattached with the call to mAppend to form mqs's result.

Dynamic resolution's static component (Section 5.4) identifies the constructors in the program's patterns (overlined in Fig. 10). When dynamically matched, these constructors require a runtime 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 (Section 6.1). Note that functions applied by mqs (e.g., the mAppend

```
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
```

Fig. 10. Imperative quicksort (mqs) with annotations for dynamic resolution.

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 5.3, 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 true function. If it is statically unknown whether p is true, this can be determined dynamically; for example, a $\lambda$-tag[3,4] (see Section 5.1) can be used to carry a function's status (either true or untrue) during runtime. Here we assume that p is (statically or dynamically) known to be a true function.

A further optimization of mqs is required for **DR** to find parallelism. The Pair constructor is used only to build temporary data—data that is inaccessible outside of mqs. This construction of temporary pairs, however, 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 [as before, reference counts stick at two] 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.

If the elements of mqs's argument list are not shared, sublists—created with split—will also not contain sharing. This permits dynamic resolution to evaluate the recursive calls to mqs in parallel without suspending threads. On the other hand, if the argument list does contain elements with reference-counts greater than one, this will curb some (but not necessarily all) parallelization of quicksort. 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.

## 9. DYNAMIC RESOLUTION IN POINTER-UNSAFE LANGUAGES

In this section we briefly describe how dynamic resolution may be applied to programs written in pointer-unsafe languages. In particular, we consider **DR** in the context of C[6] with user-level threads. A framework for **DR** in C programs has been built (Section 10 and Ref. 24) that demonstrates parallel speedup for the incnode (Section 2) and destructive quicksort (Section 8) functions. Application of **DR** in a C program requires programmer assistance (for the static component) and the programmer's

use of custom macros for thread creation, memory allocation, and pointer manipulation.

In particular, the C programmer must assert that the data structures accessed by a C function using dynamic resolution have the *acyclic* property as required by **DR**'s static component (Section 5). Additionally, the programmer must identify the statements that should evaluate in parallel using dynamic resolution. For a C version of the incnode function of Section 2, for example, the programmer must assert that the data structure passed to the function is acyclic and must explicitly fork the recursive calls to incnode as separate threads. Improvements in the design and static analyses of pointer-unsafe languages (e.g., Refs. 17 and 25) may help shift the analysis required by **DR**'s static component from the programmer to the compiler.

Since **DR** requires heap nodes to hold reference counts, a custom macro for memory allocation must be used that allocates storage for—and properly initializes—the reference counts. Furthermore, since heap pointers to heap nodes must be reflected in reference counts, macros that manipulate reference counts must be used in heap-node pointer assignments. In the example of the C version of incnode, reference counts must be incremented as new nodes are inserted into the argument DAG during its construction. Access to a node must be checked for join or simple status (Section 5.4). Again, this can be accomplished by a set of macros for accessing components of a C structure; such macros perform the checks (and requisite actions) before accessing a node's component fields.

Finally, the thread system must be extended to provide a thread-fork operation that maintains a linearization of threads (Section 6.2) as required by dynamic resolution.

## 10. IMPLEMENTATION AND RESULTS

Dynamic resolution was previously implemented[3, 4] for an early version of the SML/NJ optimizing ML compiler[13, 22] on a shared-memory Sequent multiprocessor. Although this implementation achieved speedup over sequential ML, it did not improve on optimized sequential C; that is, ML, even with parallelism due to dynamic resolution, did not outperform sequential C. Here we report measurements of a C-based implementation of dynamic resolution. By using C in our experiments, we can make direct comparisons with explicitly parallel and sequential C programs.

C-based dynamic resolution was implemented for the two example programs presented in this article: for DAG rewrite (incnode) and for destructive list quicksort (mqs). Programs were compiled with optimization enabled (-O) and used assembly-language synchronization routines.

All measurements were taken on an unloaded bus-based shared-memory SGI Challenge machine with one gigabyte of shared memory and eight 150 Mhz MIPS R4400 processors.

We compare three versions of each program: a sequential version, an unsafe-parallel version that operates correctly only on data guaranteed not to contain shared structure, and a dynamic-resolution version that correctly handles sharing. For the sequential programs, recursive and non-recursive implementations were compared—the results of the faster implementation are reported here.

The unsafe-parallel and the dynamic-resolution versions both use a simple work-queue scheme (e.g., Refs. 26 and 27) to distribute work. Each processor has a thread queue into which it inserts the new threads that it creates. When a processor exhausts the work in its queue, it steals work from other processors if possible. A C macro is used to demarcate C statements that are to evaluate in parallel; it inserts newly created threads into **DR**'s linearization.

The dynamic-resolution versions of the programs differ from the unsafe-parallel versions in three respects. The **DR** versions maintain reference counts on dynamic data, they linearize all threads using a linked-list of thread descriptors, and, when necessary, they conditionally suspend threads on access to potentially shared data (i.e., they check reference counts on node access).

A collection of **DR** C macros was used for manipulating dynamic data as described in Section 9. In particular, a node-creation macro allocates storage for the node along with a reference count. Pointer-linkage macros for installing and removing pointers update reference counts as necessary. Node-access macros check for join nodes and, on access to such a node, consult the linearization to either continue or suspend the current thread's execution. In safe languages, the actions performed by the macros can be built into the compiler—dynamic resolution was implemented in the SML/NJ compiler in this manner.[3]

The first program measured is incnode (Figs. 1 and 8) applied to a balanced tree with $2^{21}$ unshared internal nodes. The absolute execution times in Table I show that **DR** already outperforms the sequential version with only two processors. With four processors **DR** achieves a speedup of 2.3. Furthermore, the additional overhead with respect to unsafe-parallel is only 13%. Surprisingly, with more than five processors dynamic resolution outperforms unsafe-parallel. This is because termination detection with **DR** is trivial. A **DR** computation is complete when the head of the linearization becomes empty. Unsafe-parallel, on the other hand, has no thread linearization so it must reach agreement that all processors have no more work—the time required to reach agreement seems to be a function of the number of processors.

**Table I. Timing Results for DAG Rewrite (incnode)[a]**

| | incnode (Balanced Tree of Height $2^{22}$) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SC[b] | 1.36 | — | — | — | — | — | — | — |
| UP[b] | 1.96 | 0.97 | 0.67 | 0.52 | 0.48 | 0.42 | 0.39 | 0.36 |
| DR[b] | 2.27 | 1.16 | 0.77 | 0.59 | 0.48 | 0.40 | 0.35 | 0.33 |
| Processors | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

[a] Values in seconds.
[b] SC is sequential C; UP is unsafe-parallel C; and DR is C with dynamic-resolution annotations.

The second program we measured is a destructive (in place) quicksort for lists. This is another program for which dynamic resolution automatically finds expression-level parallelism. The program sorted lists of $10^6$ random integers. List elements were not shared within the list or with other data structures. The timing results for the three versions (sequential, unsafe-parallel, dynamic-resolution) are in Table II. Again, dynamic resolution outperforms the sequential program with two processors. At four processors, **DR** incurs overhead of 21% relative to the unsafe-parallel version; this overhead drops to 17% at eight processors.

To ascertain the parallelism-inhibiting effects of sharing, we introduced sharing into the argument DAG to **incnode**. We simulated sharing by selecting nodes of $d$ at random and setting their reference count to two. Table III contains the absolute execution times on six processors for the dynamic-resolution version of **incnode** applied to a DAG of $2^{21}$ internal nodes with varying degrees of sharing. Since the choice of which nodes to share can greatly influence performance, the reported numbers are the

**Table II. Timing Results for Destructive Quicksort (mqs)[a]**

| | mqs (List of $10^6$ Random Integers) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SC[b] | 26.2 | — | — | — | — | — | — | — |
| UP[b] | 29.6 | 18.5 | 18.2 | 15.1 | 13.6 | 11.6 | 11.1 | 10.7 |
| DR[b] | 3.37 | 23.2 | 23.3 | 18.4 | 16.7 | 14.8 | 13.6 | 12.6 |
| Processors | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

[a] Values in seconds.
[b] SC is sequential C; UP is unsafe-parallel C; and DR is C with dynamic-resolution annotations.

**Table III. Timing Results for DAG Rewrite (incnode) with Varying Degrees of Sharing in the Argument DAG on 6 Processors[a]**

| Percentage of Shared Nodes | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0.5% | 1.0% | 1.5% | 2.0% | 2.5% | 3.0% | 3.5% | 4.0% |
| 0.50 | 0.64 | 0.80 | 0.96 | 1.05 | 1.22 | 1.27 | 1.38 |

mean of ten trials, each selecting a different set of random nodes. For this problem, **DR** can tolerate some sharing (by performing some subcomputations sequentially) and still outperform its sequential counterpart.

## 11. RELATED WORK

Dynamic resolution was suggested by Huelsbergen and Larus;[4] subsequently, it was further developed by Huelsbergen as part of a thesis[3] and into an implementation for pointer-unsafe languages.[24]

Work related to the parallelization of languages with dynamic data structure falls into two classes: static-dynamic techniques and solely static techniques.

Tinker and Katz propose to model a Scheme implementation as a database in their *ParaTran* system.[28, 29] Concurrent reads and writes are concurrent transactions under this model. Evaluation in *Paratran* proceeds optimistically. Upon dynamic detection of a conflict, the computation must be *rolled back* to a point where the linear access order is intact. Reversing large computations is expensive. By contrast, dynamic resolution suspends a potentially conflicting expression before the conflict and can immediately process pending work. The amount of runtime information required by dynamic resolution is also small (reference counts, thread descriptors) in comparison to the complex time-stamps Paratran maintains for heap objects. *Paratran* has not demonstrated effective speed-up.

Lu,[30] and Lu and Chen,[31] describe runtime 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 runtime to find *data dependences* between program statements in the loop. The compiler, using static analysis, generates a *scheduler* for the loop's iterations. At runtime, this scheduler dynamically records references to dynamic data and, using the reference patterns thus collected, allocates loop iterations to individual processors for parallel evaluation. Unlike dynamic resolution, Lu and Chen's method does not handle procedure calls, modification of existing data structure links, or the allocation of new data.

Their method also depends on extensive pointer analysis that has been shown to be expensive in practice.[2]

Harrison's PARCEL system[1, 32] and Larus's Curare[2] seek parallelism in sequential Scheme programs using static analyses. Both systems compute bounded approximation information intended to allow parallelization of non-interfering imperative expressions: For large, irregular data, such bounded approximation leads to overly conservative parallelization—if sharing can occur dynamically, PARCEL and Curare assume that it *always* occurs. In the presence of sharing some of the parallelism that dynamic resolution can find must therefore elude these systems.

Many approaches to static pointer analysis have been described (e.g., Refs. 7, 9–11, and 14–16). These approaches usually use a form of bounded approximations (of the heap itself, of the store of heap variables, or a combination of the two). Bounded approximations are conservative—they must account for all possible configurations of the program's dynamic data. In contrast, dynamic techniques can adapt to individual instances of individual dynamic structures at runtime. Static pointer analyses also require expensive interprocedural analyses that curtail their practical use (cf. Refs. 2 and 33). With dynamic resolution, interprocedural information (i.e., sharing information) dynamically propagates into functions at runtime.

Hendren[9, 25] 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 finds parallelism when it can statically determine that trees, rather than DAGs, always reach a given program point. This analysis, therefore, cannot discover parallelism in DAGs—the type of parallelism that dynamic techniques can find. Hendren's analysis also detects 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 (Section 5.2). Hendren's analysis can potentially perform the task of dynamic resolution's static component in languages that do not support patterns.

## 12. CONCLUSIONS

We have described a runtime parallelization technique called dynamic resolution. Dynamic resolution extracts parallelism from program procedures that destructively manipulate DAGs; that is, for procedures that modify a DAGs edges or update fields stored in its vertices. Dynamic

resolution can find parallelism inaccessible to solely static approaches. Dynamic resolution is the first parallelization technique that can automatically and effectively parallelize the destructive DAG rewrite and destructive quicksort problems.

In the context of pointer-safe languages (such as ML) and programs without cyclic structures, dynamic resolution can find parallelism automatically while preserving the language's sequential semantics. In pointer-unsafe languages such as C with threads, some programmer assistance is required to realize dynamic resolution.

Dynamic parallelization in general, and dynamic resolution in particular, are viable approaches for the parallelization of imperative languages and programs.

## ACKNOWLEDGMENTS

Jim Larus supervised this research in its early stages and helped shape the ideas therein. As members of the author's thesis committee, Charles Fischer, Susan Horwitz, and Tom Reps suggested many improvements. Tom Ball and Phil Pfeiffer spurred refinements in content and presentation. Anonymous referees provided numerous valuable suggestions.

## REFERENCES

1. W. L. Harrison and D. A. Padua, PARCEL: Project for the Automatic Restructuring and Concurrent Evaluation of Lisp, *Int'l. Conf. Supercomputing*, pp. 527–538 (July 1988).
2. J. R. Larus, Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors, Ph.D. Thesis, University of California, Berkeley, Computer Science Division (May 1989).
3. L. Huelsbergen, Dynamic Language Parallelization, Ph.D. Thesis, University of Wisconsin-Madison (August 1993).
4. L. Huelsbergen and J. R. Larus, Dynamic Program Parallelization. Association for Computing Machinery. *Proc. Conf. Lisp and Functional Progr.*, pp. 311–323 (June 1992).
5. R. Milner, A Theory of Type Polymorphism in Programming, *J. Comput. Syst. Sci.* 17:348–375 (1978).
6. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Second Edition (1988).
7. A. Deutsch, Interprocedural May-Alias Analysis for Pointers: Beyond *k*-limiting, *Conf. Progr. Lang. Design and Implementation*, Association for Computing Machinery, pp. 230–241 (June 1994).
8. J. Hummel, L. J. Hendren, and A. Nicolau, A General Data Dependence Test for Dynamic, Pointer-Based Data Structures. Association for Computing Machinery, *Conf. Progr. Lang. Design and Implementation*, pp. 218–229 (June 1994).
9. L. J. Hendren and A. Nicolau, Parallelizing Programs with Recursive Data Structures, *IEEE Trans. Parallel Distrib. Syst.* **1**(1):35–47 (January 1990).
10. W. E. Weihl, Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables. Association for Computing Machinery, *Symp. Principles Progr. Lang.* pp. 83–94 (January 1980).

11. N. D. Jones and S. S. Muchnick, Flow Analysis and Optimization of Lisp-Like Structures. Association for Computing Machinery, *Symp. Principles of Progr. Lang.*, pp. 244–225 (January 1979).
12. C. A. R. Hoare, Quicksort, *Computer Journal* **5**(1):xx–xx (1962).
13. A. W. Appel and D. B. MacQueen, A Standard ML Compiler, *Functional Programming Languages and Computer Architecture* **274**:301–324 (1987).
14. D. R. Chase, M. Wegman, and F. K. Zadeck, Analysis of Pointers and Structures. Association for Computing Machinery, *Conf. Progr. Lang. Design and Implementation*, pp. 296–310 (June 1990).
15. J. R. Larus and P. N. Hilfinger, Detecting Conflicts Between Structure Accesses. Association for Computing Machinery, *Conf. Progr. Lang. Design and Implementation*, pp. 21–34 (June 1988).
16. S. Horwitz, P. Pfeiffer, and T. Reps. Dependence Analysis for Pointer Variables, *Conf. Progr. Lang. Design and Implementation.* Association for Computing Machinery (June 1989).
17. 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 Lett. Progr. Lang. and Syst.* **1**(3):243–260 (September 1992).
18. R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML.* MIT Press (1990).
19. J. M. Lucassen and D. K. Gifford, Polymorphic Effect Systems. Association for Computing Machinary, *Symp. Principles Progr. Lang.*, pp. 47–57 (January 1988).
20. A. J. Field and P. G. Harrison, *Functional Programming*, Addison-Wesley (1988).
21. S. P. Jones, *The Implementation of Functional Programming Languages*, Prentice Hall (1987).
22. A. W. Appel, *Compiling with Continuations*, Cambridge University Press (1992).
23 R. R. Fenichel and J. C. Yochelson, A Lisp Garbage-Collector for Virtual Memory Computer Systems, *Comm. ACM* **12**(11):611–612 (November 1969).
24. L. Huelsbergen, Dynamic Parallelization of Modifications to Directed Acyclic Graphs, *Proc. Conf. Parallel Architectures and Compilation Techniques*, pp. 186–197 (October 1996).
25. L. J. Hendren, Parallelizing Programs with Recursive Data Structures, Ph.D. Thesis, Cornell University (August 1990).
26. E. Mohr, D. Kranz, and R. H. Halstead, Jr., Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. Association for Computing Machinery, *Proc. Conf. on Lisp and Functional Progr.*, pp. 185–197 (June 1990).
27. M. T. Vandevoorde and E. S. Roberts, WorkCrews: An Abstraction for Controlling Parallelism. *IJPP* **17**(4):347–366 (1988).
28. P. Tinker and M. Katz, Parallel Execution of Sequential Scheme with ParaTran, *Proc. Conf. on Lisp and Functional Progr.*, pp. 28–39 (July 1988).
29. M. Katz, ParaTran: A Transparent, Transaction Based Runtime Mechanism for Parallel Execution of Scheme. Technical Report LCS/TR-454, MIT (July 1989).
30. L. Lu, Loop Transformations for Massive Parallelism, Ph.D. Thesis, Yale University (November 1992).
31. L. Lu and M. C. Chen, Parallelizing Loops with Indirect Array References or Pointers, *Preliminary Proc. 4th Workshop on Languages and Compilers for Parallel Computing* (August 1991).
32. W. L. Harrison, The Interprocedural Analysis and Automatic Parallelization of Scheme Programs, *Lisp- and Symbolic Computation* **2**(3/4):179–396 (October 1989).
33. P. E. Pfeiffer, Dependence-Based Representations for Programs with Reference Variables, P.D. Thesis, University of Wisconsin-Madison (1991).