Static Dependent Costs for Estimating Execution Time

Brian Reistad David K. Gifford

Laboratory for Computer Science,
Massachusetts Institute of Technology,
Cambridge, Massachusetts 02139.
{reistad, gifford}@lcs.mit.edu

Abstract

We present the first system for estimating and using data-dependent expression execution times in a language with first-class procedures and imperative constructs. The presence of first-class procedures and imperative constructs makes cost estimation a global problem that can benefit from type information. We estimate expression costs with the aid of an algebraic type reconstruction system that assigns every procedure a type that includes a static dependent cost. A *static dependent cost* describes the execution time of a procedure in terms of its inputs. In particular, a procedure's static dependent cost can depend on the size of input data structures and the cost of input first-class procedures. Our cost system produces symbolic cost expressions that contain free variables describing the size and cost of the procedure's inputs. At run-time, a cost estimate is dynamically computed from the statically determined cost expression and run-time cost and size information. We present experimental results that validate our cost system on three compilers and architectures. We experimentally demonstrate the utility of cost estimates in making dynamic parallelization decisions. In our experience, dynamic parallelization meets or exceeds the parallel performance of any fixed number of processors.

1 Introduction

We present a new method for estimating program execution time that can be added to any statically typed programming language with polymorphism. Reliable static estimates of the execution time of program expressions have important applications such as optimization, documentation, automatic parallelization, and providing real-time performance guarantees. With reliable static estimates an optimizing compiler can focus its attention on the most important portion of a program and analyze which expressions might be profitably evaluated in parallel [G86, SH86, G88, MKH90]. We have developed a simple dynamic parallelization system which uses cost estimates to make parallelization decisions based on their profitability.

This research supported by DARPA/ONR Grant. No. DABT63-92-c-0012

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

A cost system estimates the cost of a program in terms of a desired metric such as time. A *micro analysis* system is a cost system in which costs are expressed using constants that describe the costs of common, elementary operations. Micro analysis was first presented by Knuth [K68] and more recently by Wegbreit [W75] and Cohen [C82]. For instance, the cost of the Scheme expression `(+ x 2)` would be the sum of:

- the cost of looking up the operator `+`,
- the cost of evaluating the arguments, which requires looking up the variable `x` and evaluating the number 2,
- the cost of calling the operator, and
- the cost of performing the operation.

This cost is expressed as

```
(sum C_var C_var C_num C_call C_+)
```

where the symbolic constants are execution target specific.

In the presence of first-class procedures it is impossible to syntactically determine the expected execution cost of a procedure call. In the expression `(f x 2)` the total cost includes the cost of performing the operation named by `f`; however, the cost of `f` is not syntactically apparent. The difficulty arises from the presence of an unknown procedure and is present even if procedures can only be passed as arguments or stored in data structures but not returned as values.

In a static dependent cost system each procedure type is automatically annotated with a latent cost description. A *latent cost* communicates the expected execution time of a procedure from the point of its definition to the point of its use. Thus in `(f x 2)`, `f` would have type $(T_1 \times \texttt{num}) \xrightarrow{C_{latent}} T_2$ where C_{latent} denotes the cost of performing the operation named by `f`. This type is written in S-expression syntax as `(->` C_{latent} `(`T_1 `num) `T_2`)`. The cost of the expression `(f x 2)` can be obtained by extracting the latent cost of `f` from its type; giving a total cost of `(sum C_var C_var C_num C_call `C_{latent}`)`.

Adding latent costs to procedure types provides a way to describe the cost of higher-order procedures. Assume the above expression `(f x 2)` is the body of a procedure: `(lambda (f x) (f x 2))`. The latent cost of the procedure defined by this lambda expression is the cost from above, giving the procedure the following type:

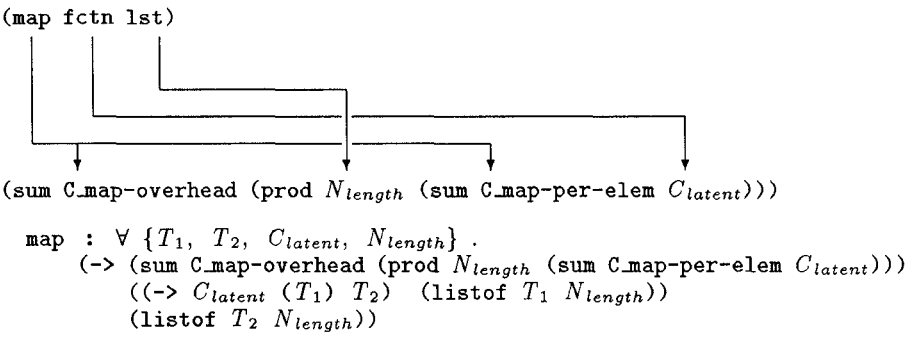

Figure 1: Factors in The Latent Cost of map

```
(-> (sum C_var C_var C_num C_call C_latent)
    ( (-> C_latent (T1 num) T2)
      T1)
    T2)
```

The latent cost of this procedure depends on the latent cost of its procedural argument f.

The latent cost of a procedure may also depend on the size of the procedure's actual arguments. Consider the expression (map fctn lst) in which the familiar map operator is used to apply the procedure fctn to each element of the list lst. Following the above analysis, this expression would have cost (sum C_var C_var C_var C_call C_{map}) where C_{map} is the latent cost of map. The latent cost of map depends on both the latent cost of the input procedure and the length of the input list. Without any information about the size of the list, there is no finite value that can be chosen for C_{map} to give an upper bound for lists of all sizes.

A *static dependent cost* is a latent cost that depends on the size of the procedure's arguments. In our cost system, data structure types are annotated with a size description that provides an upper bound on the run-time size of the data structure. A size description communicates the data structure's size from its point of definition to its point of use in much the same way that latent costs communicate the expected execution cost of a procedure.

Figure 1 shows how a static dependent cost is used to describe the execution time of map. The arrows show how the definition of map and the values of its arguments all contribute to the cost of executing its body for the expression (map fctn lst). The latent cost of fctn is denoted C_{latent} and is provided the type of fctn. N_{length} is an upper bound on the length of the input list lst and is provided by the type of lst. Our system uses static dependent costs to provide accurate estimates for a complete set of data parallel operators in a manner similar to that shown here for map.

Polymorphism is central to our static dependent cost system. Figure 1 shows the polymorphic type of map. Cost parameters such as the latent costs of higher-order procedures and the size of data structures are made fully polymorphic in our cost system. Cost and size polymorphism provides the key descriptive power required to provide reliable cost estimates for non-trivial programs.

Our static dependent cost system contains dependent costs for a complete set of data parallel operators. A large number of interesting programs can be written using only first-class procedures and a complete set of data parallel operators [B78]. The example programs in this paper were all written without general recursion, including Sussman's n-body simulation [ADGHSS85].

Our cost system does not currently provide estimates for user-defined recursive procedures, but still provides cost estimates for non-recursive subexpressions. Predicting costs for recursive procedures requires solving recurrence equations which is not always possible. This problem is not addressed as it is beyond the scope of this paper and has been the subject of previous work [W75, L88, R89, S90]. Our cost system could potentially be augmented with a pre-defined database of recursion equations and their closed forms.

We show experimental results that demonstrate that the estimates produced by our cost system are accurate to within a factor of three of the actual cost incurred in the context of multiple applications and architectures. Thus our cost estimates are sufficiently accurate to be of use to programmers and optimizing compilers.

We have used our cost estimates to make dynamic parallelization decisions based on a profitability analysis. An expression can be profitably evaluated in parallel if the cost of evaluating the expression exceeds the overhead of performing the parallelization. Our system computes cost expressions for polymorphic procedures that contain free cost and size variables. Parallelization decisions cannot be made at compile time because the values of these free cost and size variables are not known. By providing this information at run-time, parallelization decisions can be made dynamically. The decisions are low cost because the majority of the cost analysis is done statically and the dynamic decision only involves evaluating a cost expression. In our experience, dynamic parallelization meets or exceeds the parallel performance of any fixed number of processors.

In this paper we present previous work (Section 2), our cost analysis (Section 3), an algebraic cost and size reconstruction algorithm (Section 4), experimental results demonstrating our cost system predicting execution times on various target architectures (Section 5), and results of using our cost system to make dynamic processor allocation decisions (Section 6).

2 Previous Work

Our work is related to previous work on profiling, effect systems, cost systems, cost estimation, automatic complexity analysis, and range analysis.

Profiling Profiling is a dynamic alternative to obtaining cost information about a program's execution time. The general approach is to run a program once, gather statistics about where time is spent, and feed this cost information back into the compiler to re-compile the program. Profile data has been used for various optimization efforts including partitioning and scheduling parallel programs [SH86]. Unfortunately, the profile data from one run is not always a good predictor of subsequent runs with different input data [W91]. The static cost estimates produced by our system do not suffer from this disadvantage because our system produces cost estimates that depend on the size of the input data and thus generalize to previously unseen data sizes. Static analysis can be done without choosing "typical" input data that is required to gather profile statistics.

Effect systems Effect systems originated the idea of annotating procedure types with static information about how a program computes and form the basis for cost systems. Lucassen and Gifford [LG88] first proposed effect systems to analyze side effects. Jouvelot and Gifford [JG91] present an algebraic reconstruction algorithm to infer effect descriptions and provide let-polymorphism using algebraic type schemes. Talpin and Jouvelot [TJ92] extend reconstruction to regions describing memory locations and show how to include the notion of subeffecting.

Cost systems Cost systems are an extension of effect systems to analyze program execution time, but to date they have not captured dependence on data structure size. Dornic *et al.* [DJG92] propose a cost system that parallels an effect system but it requires explicit typing. Dornic [D92] presents the first cost reconstruction algorithm and includes a notion of subcosting, but he does not describe how to handle polymorphism. Dornic [D93] presents a refinement that labels recursive calls, thus identifying the sources of recursion. These systems cannot provide cost estimates for the examples given in this paper because they are not powerful enough to handle size dependencies and thus cannot describe any form of iteration.

Cost Estimation Skillicorn and Cai [SC93] present a cost calculus for a parallel functional programming language that can be used in a program development system. They use sizes and costs in a similar manner to our system in describing costs for data parallel operators; however, they do not deal with first-class procedures that may have dependent costs.

Huelsbergen *et al.* [HLA94] present an automatic parallelization system that statically estimates costs and makes dynamic parallelization decisions. They use abstract evaluation to compute a lower bound on the cost of evaluating an expression for various sizes of an input list. Their system does not compute cost expressions that depend on the size of the list. Instead Huelsbergen *et al.* determine the list size at which evaluation of the expression will execute longer than the parallelization threshold. Parallelization decisions are made dynamically by comparing this statically determined cutoff with run-time estimates of the list's size. Presumably, the abstract evaluation is run on increasing input sizes until the abstract cost estimate exceeds the parallelization threshold; however, they did not implement the static portion of their system and do not specify how the cutoff would be automatically determined. Abstract evaluation seems be expensive for first-class procedures. If multiple procedures can possibly reach a point in the program (the value set for the operator contains multiple procedures), the application of each procedure to the argument value set must be abstractly evaluated. On the other hand, our system uses a single type description for all the procedures that can reach that point. Abstract evaluation may diverge for recursive procedures, so the parallelization threshold is used during static analysis to ensure that it halts. Lastly, their costs are in terms of e-units that estimate the number of procedure applications and conditionals in the evaluation. This implies that iterators such as `map` are not primitives or their cost would be greatly underestimated.

$$
\begin{array}{l}
E \in \textit{Expression} \\
E ::= \textit{Identifier} \mid \textit{Primitive Operator} \\
\quad \mid \textit{Symbol} \mid \textit{Boolean} \mid \textit{Float} \mid \textit{Natural Number} \\
\quad \mid \texttt{(lambda } (I)\ E\texttt{)} \mid \texttt{(rec } (I_f\ I_x)\ E\texttt{)} \\
\quad \mid (E\ E) \\
\quad \mid \texttt{(let } (I\ E)\ E\texttt{)} \mid \texttt{(if } E\ E\ E\texttt{)} \mid \texttt{(begin } E^*\texttt{)}
\end{array}
$$

Figure 2: μFX/SDC Syntax

Complexity Analysis Automatic complexity analysis attempts to provide closed form costs by analyzing recurrence equations, but has been developed only for languages without first-class procedures and mutation. Wegbreit [W75] presents one of the earliest automatic complexity analysis tools METRIC to analyze simple Lisp programs. Le Métayer [L88] presents the ACE system for analyzing FP programs by rewriting to a time-complexity function and matching against a database. Rosendahl [R89] presents a similar system for a first-order subset of Lisp based on abstract interpretation. Sands [S88, S90] presents a mechanism to produce time-complexity expressions for a language with first-class procedures (but not mutation) to extend Le Métayer's ACE system. It seems that his approach still exposes first-class procedures to the ACE system which would require a powerful deduction system to solve recurrences containing first-class procedures.

Interval/Range Analysis Range analysis is relevant because our system includes estimates of data structure sizes that allow us to describe dependent costs. Chatterjee *et al.* [CBF91] analyze a data parallel program graph to discover which vectors have the same run-time sizes. They observe that their algorithm is similar to type inference. Harrison [H77] presents a mechanism for determining the value ranges of variables in the context of loops.

3 Cost System Semantics

In this section we define our language and introduce our cost system.

3.1 Language Definition

The experimental work described in this paper has been carried out in a subset of the FX programming language [GJSO92], called μFX/SDC. μFX/SDC is statically typed with first-class procedures and mutation. μFX/SDC has been used to write a number of programs, including matrix multiplication, the game of life, and n-body simulation.

The syntax of our language is shown in Figure 2. μFX/SDC has symbolic, boolean, float, and natural number literals, means of declaring both regular and recursive procedures, procedure application, `let` bindings, a conditional

expression and a sequence expression. The rec expression defines a recursive procedure in which I_f is the name of the recursive procedure, I_x is the name of the argument, and E is the body. We restrict our discussion to single argument procedures for simplicity, but our implementation actually handles multiple argument procedures. The set of primitives includes a full set of data parallel operators as well as imperative primitives for allocating, reading and writing mutable cells.

Values include first-class procedures, floats, natural numbers, symbols, booleans, pairs, lists, vectors, permutations and mutable cells. Only natural number arithmetic can be used to compute values that are used to size data structures; see Section 4.2.

The call-by-value dynamic semantics is shown in Figure 3. The dynamic semantics of μFX/SDC includes a counter to measure execution cost. Evaluation rules have the following form: $\sigma,\ Env \vdash E \rightarrow v,\ c,\ \sigma'$ which is read "given store σ and environment Env, the expression E evaluates to value v incurring cost c and producing the modified store σ'." Each rule is assigned a symbolic cost constant allowing us to calculate a micro measure of the steps required to evaluate an expression.

Recursive bindings are implemented by unwinding the recursive binding in the fourth component of the closure. This binding is unwound once by the *Rec* function during application and appended (::) with the current environment. Dynamic semantics are not given for the arithmetic, mutable cell or data parallel operators.

3.2 Static Semantics

We begin by presenting value descriptions that include cost and size annotations and then we discuss the static semantics for deducing descriptions.

Value expressions are described by *descriptions*. As shown in Figure 4, legal descriptions are types, costs and sizes:

- Types include base types (unit, symbol, boolean, float), pairs, reference types (mutable cells), procedure types and data structure types. Procedure types include a latent cost that describes the procedure's execution cost. Data structure types include a size that is a static upper bound on the run-time size of the data structure. The numof type includes an upper bound on the value of natural numbers that is used to predict execution times of primitive iterators, such as **make-vector**, that require a numeric argument.

  Polymorphism is expressed with *type schemes* that abstract a type T over a set of description variables D_{v_i}: $\forall\ \{D_{v_i}\}.T$.

  In the remainder of this paper we will not discuss the following types: base types (**unit**, **sym**, **bool**, and **float**), **pairof** (analogous to **refof**), **vectorof** and **permutationof** (analogous to **listof** and **numof**).

- Costs are upper bounds on the execution time of expressions. Costs are described by expressions that include symbolic constants, the constant **long** denoting an unbounded estimate, and the sum or maximum of two cost estimates. Cost can also be proportional to the size of a data structure. We interpret **sum**, **prod** and **max** as the usual algebraic operators with associativity and commutativity.

$$
\begin{array}{rcl}
v & \in & Value = BaseValue + Closure + DataStructure \\
 & & BaseValue = Boolean + Symbol \\
 & & \quad + Float + \{unit\} + Ref + Pair \\
 & & Closure = \langle Id, Expression, Env, Env \rangle \\
 & & DataStructure = \\
 & & \quad Natural + List + Vector + Permutation \\
Env & \in & Environment = Identifier \rightarrow Value \\
\sigma & \in & Store = Location \rightarrow Value \\
c & \in & Cost
\end{array}
$$

(num)

$$\sigma,\ Env \vdash Nat \rightarrow Nat,\ \texttt{C\_num},\ \sigma$$

(var)

$$\frac{[I \mapsto v] \in Env}{\sigma,\ Env \vdash I \rightarrow v,\ \texttt{C\_var},\ \sigma}$$

(lambda)

$$\sigma,\ Env \vdash (\texttt{lambda}\ (I)\ E_b) \rightarrow \langle I,\ E_b,\ Env,\ [\,]\ \rangle,\ \texttt{C\_lambda},\ \sigma$$

(rec)

$$\sigma,\ Env \vdash (\texttt{rec}\ (I_f\ I_x)\ E_b) \rightarrow \langle I_x, E_b, Env, [I_f \mapsto \langle I_x, E_b, Env, [\,]\rangle\,]\rangle,\ \texttt{C\_rec},\ \sigma$$

(call)

$$\frac{\begin{array}{l}\sigma, Env \vdash E_{op} \rightarrow \langle I,\ E_b,\ Env',\ Env''\rangle,\ c_{op},\ \sigma_{op} \\ \sigma_{op}, Env \vdash E_{arg} \rightarrow v_{arg},\ c_{arg},\ \sigma_{arg} \\ \sigma_{arg}, Env'[I \mapsto v_{arg}] :: Rec(Env'') \vdash E_b \rightarrow v,\ c,\ \sigma'\end{array}}{\sigma,\ Env \vdash (E_{op}\ E_{arg}) \rightarrow v,\ (\texttt{sum C\_call}\ c_{op}\ c_{arg}\ c),\ \sigma'}$$

(let)

$$\frac{\begin{array}{rcl}\sigma,\ Env & \vdash & E \rightarrow v_1,\ c_1,\ \sigma_1 \\ \sigma_1,\ Env[I \mapsto v_1] & \vdash & E_b \rightarrow v,\ c,\ \sigma'\end{array}}{\sigma,\ Env \vdash (\texttt{let}\ (I\ E)\ E_b) \rightarrow v,\ (\texttt{sum C\_let}\ c_1\ c),\ \sigma'}$$

(if-true)

$$\frac{\begin{array}{rcl}\sigma,\ Env & \vdash & E_{test} \rightarrow true,\ c_{test},\ \sigma' \\ \sigma',\ Env & \vdash & E_{con} \rightarrow v,\ c,\ \sigma''\end{array}}{\sigma,\ Env \vdash (\texttt{if}\ E_{test}\ E_{con}\ E_{alt}) \rightarrow v,\ (\texttt{sum C\_if}\ c_{test}\ c),\ \sigma''}$$

(if-false)

$$\frac{\begin{array}{rcl}\sigma,\ Env & \vdash & E_{test} \rightarrow false,\ c_{test},\ \sigma' \\ \sigma',\ Env & \vdash & E_{alt} \rightarrow v,\ c,\ \sigma''\end{array}}{\sigma,\ Env \vdash (\texttt{if}\ E_{test}\ E_{con}\ E_{alt}) \rightarrow v,\ (\texttt{sum C\_if}\ c_{test}\ c),\ \sigma''}$$

$$
\begin{array}{l}
Rec([I_f \mapsto \langle I_x, E, Env, [\,]\rangle]) = \\
\qquad [I_f \mapsto \langle I_x, E, Env, [I_f \mapsto \langle I_x, E, Env, [\,]\rangle]\rangle] \\
Rec([\,]) = [\,]
\end{array}
$$

Figure 3: μFX/SDC Dynamic Semantics

$D \in Desc ::= T \mid C \mid N$

$T \in Type$
$T ::= I \mid \texttt{unit} \mid \texttt{sym} \mid \texttt{bool} \mid \texttt{float}$
$\quad \mid \texttt{(refof } T\texttt{)} \mid \texttt{(pairof } T\ T\texttt{)}$
$\quad \mid \texttt{(->} \ C\ \texttt{(}T\texttt{)}\ T\texttt{)}$
$\quad \mid \texttt{(numof } N\texttt{)} \mid \texttt{(listof } T\ N\texttt{)}$
$\quad \mid \texttt{(vectorof } T\ N\texttt{)} \mid \texttt{(permutationof } N\texttt{)}$

$C \in Cost$
$C ::= I \mid$ *Symbolic Constant* (*eg.* `C_call`) $\mid$ `long`
$\quad \mid \texttt{(sum } C\ C\texttt{)} \mid \texttt{(prod } N\ C\texttt{)} \mid \texttt{(max } C\ C\texttt{)}$

$N \in Size ::= I \mid Nat \mid$ `long`
$\quad \mid \texttt{(sum } N\ N\texttt{)} \mid \texttt{(prod } N\ N\texttt{)} \mid \texttt{(max } N\ N\texttt{)}$

Figure 4: μFX/SDC Descriptions

- Sizes are upper bounds on the run-time size of data structures and have an algebra similar to that for costs. Size in this context refers to the dimension of the data structure over which the primitive iterators work: the length of vectors and lists and the magnitude of numbers.

The static semantics for μFX/SDC is shown in Figure 5. Judgments in the static semantics are of the form: $A \vdash E : T \ \$ \ C$ which is read "in type environment A, the expression E has type T and maximum cost C." The type environment A binds identifiers to types or type schemes; types are just type schemes with no free description variables.

The static semantics contains rules for each language construct. The rule for each language construct computes the cost of the subexpressions and adds a symbolic constant representing the overhead of that construct. For example, the cost computed by the **let** rule includes the overhead `C_let`, the cost of the named expression, and the cost of the body. The **if** rule uses **max** to choose the larger cost of the consequent and alternative subexpressions.

The latent cost of a procedure is communicated from its point of definition to its point of use by the **lambda**, **rec**, and **call** rules. Procedure types are annotated with latent costs by the **lambda** and **rec** rules. The rule for procedure application extracts the latent cost of the procedure from the operator type. The cost of the application includes the overhead of calling the procedure, the cost of the subexpressions, and the latent cost of the operator. We have not distinguished between primitives and general procedures as has been done in other cost analysis systems such as [W75, S90]. To be conservative, we must assume that every application incurs the overhead of general procedure call.

The static semantics includes some flexibility in deducing size and cost descriptions. For example, the **num** rule allows us to report a **numof** type with any size at least as large as the literal. Without this subsizing flexibility, the expression `(if #t 1 2)` would not type-check because the types `(numof 1)` and `(numof 2)` would not be equivalent. The **num** rule allows us to claim that `1` has type `(numof 2)` and thus the entire expression also has type `(numof 2)`. We could also claim that the expression has an even larger type, but doing so will produce overly conservative estimates.

(num)

$$\frac{Nat \sqsubseteq_{size} N}{A \vdash Nat : \texttt{(numof } N\texttt{)} \ \$ \ \texttt{C\_num}}$$

(var)

$$\frac{[I : \forall\ \{D_{v_i}\}.T] \in A \qquad [D'_i/D_{v_i}]T \sqsubseteq_{type} T'}{A \vdash I : T' \ \$ \ \texttt{C\_var}}$$

(lambda)

$$\frac{A[I : T_{arg}] \vdash E_b : T_{return} \ \$ \ C \qquad C \sqsubseteq_{cost} C_{latent}}{A \vdash \texttt{(lambda (}I\texttt{)}\ E_b\texttt{)} : \texttt{(->} \ C_{latent}\ \texttt{(}T_{arg}\texttt{)}\ T_{return}\texttt{)} \ \$ \ \texttt{C\_lambda}}$$

(rec)

$$\frac{A[I_f : \texttt{(->} \ C_{latent}\ \texttt{(}T_{arg}\texttt{)}\ T_{return}\texttt{)},\ I_x : T_{arg}] \vdash E : T_{return} \ \$ \ C \qquad C \sqsubseteq_{cost} C_{latent}}{A \vdash \texttt{(rec (}I_f\ I_x\texttt{)}\ E\texttt{)} : \texttt{(->} \ C_{latent}\ \texttt{(}T_{arg}\texttt{)}\ T_{return}\texttt{)} \ \$ \ \texttt{C\_rec}}$$

(call)

$$\frac{A \vdash E_{op} : \texttt{(->} \ C_{latent}\ \texttt{(}T_{arg}\texttt{)}\ T_{ret}\texttt{)} \ \$ \ C_{op} \qquad A \vdash E_{arg} : T_{arg} \ \$ \ C_{arg} \qquad T_{ret} \sqsubseteq_{type} T'_{ret}}{A \vdash \texttt{(}E_{op}\ E_{arg}\texttt{)} : T'_{ret} \ \$ \ \texttt{(sum C\_call } C_{op}\ C_{arg}\ C_{latent}\texttt{)}}$$

(let)

$$\frac{E \text{ non-expansive} \qquad A \vdash E : T \ \$ \ C \qquad A[I : Gen(T, A)] \vdash E_b : T_b \ \$ \ C_b}{A \vdash \texttt{(let (}I\ E\texttt{)}\ E_b\texttt{)} : T_b \ \$ \ \texttt{(sum C\_let } C\ C_b\texttt{)}}$$

(if)

$$\frac{A \vdash E_{test} : \texttt{bool} \ \$ \ C_{test} \qquad A \vdash E_{con} : T \ \$ \ C_{con} \qquad A \vdash E_{alt} : T \ \$ \ C_{alt}}{A \vdash \texttt{(if } E_{test}\ E_{con}\ E_{alt}\texttt{)} : T \ \$ \ \texttt{(sum C\_if } C_{test}\ \texttt{(max } C_{con}\ C_{alt}\texttt{))}}$$

$Gen(T, A) = \forall\ \{D_{v_i}\}\ .\ T$
$\quad where\ \{D_{v_i}\} = FV(T) \setminus FV(A)$

Figure 5: μFX/SDC Static Semantics

(subtype-numof)

$$\frac{N \sqsubseteq_{size} N'}{(\texttt{numof}\ N) \sqsubseteq_{type} (\texttt{numof}\ N')}$$

(subtype-listof)

$$\frac{N \sqsubseteq_{size} N'}{(\texttt{listof}\ T\ N) \sqsubseteq_{type} (\texttt{listof}\ T\ N')}$$

(subtype-arrow)

$$\frac{\begin{array}{c} C \sqsubseteq_{cost} C' \\ T'_{arg} \sqsubseteq_{type} T_{arg} \\ T_{ret} \sqsubseteq_{type} T'_{ret} \end{array}}{(\texttt{->}\ C\ (T_{arg})\ T_{ret}) \sqsubseteq_{type} (\texttt{->}\ C'\ (T'_{arg})\ T'_{ret})}$$

Figure 6: Subtyping Relation

The **lambda** and **rec** rules report a latent cost that is larger than the cost deduced for the body expression. This subcosting flexibility allows us to report the larger of two latent costs when two procedures are constrained to have the same type. These two rules provide the same functionality as Dornic's subcosting rule [D92].

The **var** rule must incorporate the same subcosting and subsizing flexibility as the **num** and **lambda** rules because the identifier may be bound to a natural number or a procedure. This flexibility is expressed by the subtyping relation shown in Figure 6. No subtyping is provided on mutable data types such as the types in `refof` and `listof` types.

Notice the latent cost determined by the **rec** rule may be unbounded if there is a recursive call in the body. The latent cost of the procedure C_{latent} must be greater than or equal to the cost of the body C, but this cost includes both the latent cost and the call overhead: $C_{latent} \geq C =$ (sum C_{latent} C_call), forcing $C_{latent} =$ `long`.

Let-polymorphism is provided for *non-expansive* [T87] expressions by the use of type schemes. (The rule for expansive expressions is straightforward and omitted.) Previous effect reconstruction systems have used substitution to provide let-polymorphism, but substituting the let-bound expression may artificially increase the cost estimate of the body. The *Gen* function generalizes the type by abstracting over the description variables that occur free in T but are not bound in A (Figure 5), where FV computes the set of free description variables in a type or type environment. The type scheme is instantiated by the **var** rule.

Type schemes abstract over costs and sizes as well as types. A procedure has a static dependent cost if a description variable denoting the size of one of its arguments occurs free in the procedure's latent cost. The initial type environment contains type schemes for the primitive operators. Below we give the static dependent costs for some of the data parallel primitives that we use to "bootstrap" the system.

Primitive operator types

The set of primitives includes floating point and natural number arithmetic, mutable cell operators, and a complete set of data parallel operators from FX including list, vector, and permutation iterators. We give the types of a few selected primitive operators in Figure 7 to demonstrate how size estimates are computed and used in costs.

```
+ : ∀ {n1, n2} . (-> C_+ ((numof n1) (numof n2))
                       (numof (sum n1 n2)))
* : ∀ {n1, n2} . (-> C_* ((numof n1) (numof n2))
                       (numof (prod n1 n2)))
- : ∀ {n1, n2} . (-> C_- ((numof n1) (numof n2))
                       (numof n1))

map : ∀ {t1, t2, c, len} .
    (-> (sum C_map-overhead
             (prod len (sum C_map-per-elem c)))
        ((-> c (t1) t2) (listof t1 len))
        (listof t2 len))

reduce : ∀ {t1, t2, c, len} .
    (-> (sum C_reduce-overhead
             (prod len (sum C_reduce-per-elem c)))
        ((-> c (t1 t2) t2) (listof t1 len) t2)
        t2)

make-vector : ∀ {t, n} .
    (-> (sum C_make-vector-overhead
             (prod n C_make-vector-per-elem))
        ((numof n) t)
        (vectorof t n))
```

Figure 7: Primitive Operator Types

The size contained in a procedure's return type may depend on the size of the procedure's arguments. The return size for + is the sum of its input sizes. The return size for - is the same as the size of its first argument because our size algebra does not have a subtraction operator; see Section 4.2.

Primitive iterators such as `map` and `reduce` have a static dependent cost. These primitives have an execution time proportional to the size of the data structures over which they iterate. Our system assumes that the latent cost of the procedural argument is the same for all applications so the cost of the iterator can be written in closed form.

3.3 Correctness Issues

The static semantics is consistent with the dynamic semantics if the static semantics computes a valid typing for the resultant value and reports a cost estimate that is an upper bound on the actual cost. An inspection of Figures 3 and 5 shows that the static semantics closely mirrors the dynamic semantics. The main difference being that the procedure body is dynamically evaluated when the procedure is used (third antecedent in the **call** rule of Figure 3), but it is statically analyzed at the procedure's point of definition (first antecedents of the **lambda** and **rec** rules of Figure 5). A formal proof of consistency must deal with the complication that an expression can be assigned more than one type (see the discussion of the **num** rule above). We further discuss correctness issues in Section 4.3.

4 Cost Reconstruction

Cost analysis is most useful when cost estimates can be automatically generated. This frees the programmer from the burden of cost declarations and allows the compiler to identify expensive expressions for possible automatic parallelization. We provide an algebraic cost reconstruction algorithm for our cost system that computes the type and cost of an expression. It reconstructs size bounds on data structure types and uses let-polymorphism to compute static dependent costs. Algebraic reconstruction has three major components:

- The reconstruction algorithm R walks over the expression calculating the type and cost along with a set of constraints on cost and size variables.
- The unification algorithm U produces a substitution on description variables that makes types equivalent.
- The constraint solver CS generates minimal assignments to cost and size variables that are consistent with the constraints discovered by the reconstruction and unification algorithms.

Given an expression and a type environment, the reconstruction algorithm returns the expression's type and cost along with a substitution on description variables and a constraint set on cost and size variables:

R : *Type environment* $\times$ *Exp*
$\rightarrow$ *Type* $\times$ *Cost* $\times$ *Substitution* $\times$ *Constraint set*

The algorithm is shown in Figure 8. A substitution is represented as $[D/D_v]$ where D is substituted for the description variable D_v. Substitutions can be applied to a type, cost, size, or constraint set. The constraint set is denoted K and each constraint is represented as a pair $(C_v,\ C)$ where C_v must be greater than or equal to C.

The reconstruction algorithm maintains two invariants. First, the resultant type, cost, and constraint set have had the substitution applied to them. Second, and more importantly, cost and size descriptions within types are always variables. This allows unification of size and costs to be trivial because constraints are recorded in the constraint set. The type schemes for the primitive operators must be converted to algebraic type schemes to insure only size and cost variables appear in types. An *algebraic type scheme* is a pair of a type and constraint set that are abstracted over a set of description variables [JG91].

The reconstruction algorithm directly implements the static semantics. For compound expressions, R is applied to the subexpressions and the results are combined appropriately. For example, the case for `if` applies R to each subexpression, unifies the type of the predicate with `bool`, unifies the consequent and alternate types, and returns an appropriate cost and a merged constraint set.

The subtyping flexibility contained in the static semantics is implemented by the constraint set. For example, the case for natural number literals creates the constraint $\{(N_v,\ Nat)\}$. This constraint ensures that the size of the reported type is at least as large as the literal itself. Similarly the cases for `lambda` and `rec` include constraints for the reported latent cost.

The subtyping algorithm shown in Figure 9 implements the subtyping relation of the static semantics (Figure 6). The subtyping algorithm consists of two algorithms: *lift-type*

$R(A,\ E)$ = case E in
$Nat \rightarrow N_v$ fresh,
return ((**numof** N_v), C_num, [ ], $\{(N_v,\ Nat)\}$)
$I \rightarrow$ if $[I : \forall\ \{D_{v_i}\}.(T,\ K)] \in A$ then
let $S = [D'_{v_i}/D_{v_i}]$, D'_{v_i} fresh
let $(T',\ K') = \mathit{lift\text{-}type}(ST)$
return $(T',$ C_var, [ ], $SK \cup K')$
else *fail*
(**lambda** $(I : T_s)\ E$)
$\rightarrow C_v$ fresh
let $T' = \mathit{newtype}(T_s)$
let $(T,C,S,K) = R(A[I : T'],\ E)$
return ((-> C_v (ST') $T)$, C_lambda, $S, K \cup \{(C_v,\ C)\}$)
(**rec** $(I_f\ I_x : T_x)\ E : T_{ret}$)
$\rightarrow C_v$ *fresh*
let $T'_x = \mathit{newtype}(T_x)$
let $T'_{ret} = \mathit{newtype}(T_{ret})$
let $A' = A[I_f : (\text{-> } C_v\ (T'_x)\ T'_{ret}),\ I_x : T'_x]$
let $(T_b, C_b, S_b, K_b) = R(A',\ E)$
let $S = U(T_b,\ S_b T'_{ret})$
return $S(S_b(\text{-> } C_v\ (T'_x)\ T'_{ret}),$ C_rec,
$S_b,\ K_b \cup \{(S_b C_v,\ C_b)\})$
$(E_{op}\ E_{arg})$
$\rightarrow T_v, C_v$ fresh
let $(T_{op}, C_{op}, S_{op}, K_{op}) = R(A,\ E_{op})$
let $(T_{arg}, C_{arg}, S_{arg}, K_{arg}) = R(S_{op}A,\ E_{arg})$
let $S = U(S_{arg}T_{op},\ (\text{-> } C_v\ (T_{arg})\ T_v))$
let $(T', K') = \mathit{lift\text{-}type}(ST_v)$
return $(T',\ S($**sum** C_call $S_{arg}C_{op}\ \ C_{arg}\ \ C_v),$
$SS_{arg}S_{op},\ SS_{arg}K_{op} \cup SK_{arg} \cup K')$
(**let** $(I\ E)\ E_b$) $\rightarrow$
if E **expansive** then
let $(T,C,S,K) = R(A,\ E)$
let $(T_b, C_b, S_b, K_b) = R(SA[I : T],\ E_b)$
return $(T_b,$ (**sum** C_let $S_b C\ C_b),\ S_b S,\ S_b K \cup K_b)$
else (E **non-expansive**)
let $(T,C,S,K) = R(A,\ E)$
let $\{D_{v_1} \ldots D_{v_n}\} = (FV(T) \cup FV(K)) \setminus FV(SA)$
let $(T_b, C_b, S_b, K_b) = R(SA[I : \forall\ \{D_{v_i}\}.(T,\ K)],\ E_b)$
return $(T_b,$ (**sum** C_let $S_b C\ C_b),\ S_b S,\ S_b K \cup K_b)$
(**if** $E_1\ E_2\ E_3$)
$\rightarrow$ let $(T_1, C_1, S_1, K_1) = R(A,\ E_1)$
let $(T_2, C_2, S_2, K_2) = R(S_1 A,\ E_2)$
let $(T_3, C_3, S_3, K_3) = R(S_2 S_1 A,\ E_3)$
let $S = U(S_3 S_2 T_1,$ **bool**$)$
let $S' = U(SS_3 T_2,\ ST_3)$
return $S'S(T_3,$ (**sum** C_if $S_3 S_2 C_1$ (**max** $S_3 C_2\ \ C_3)),$
$S_3 S_2 S_1,\ S_3 S_2 K_1 \cup S_3 K_2 \cup K_3)$
else *fail*

Figure 8: μFX/SDC Reconstruction Algorithm

```
lift-type(T) = case T in
   (-> C (T_arg) T_ret)
    → C_v fresh,
       let (T'_arg, K_arg) = sink-type(T_arg)
       let (T'_ret, K_ret) = lift-type(T_ret)
       return ((-> C_v (T'_arg) T'_ret),
               {(C_v, C)} ∪ K_arg ∪ K_ret)
   (numof N)
    → N_v fresh, return ((numof N_v), {(N_v, N)})
   (listof T N)
    → N_v fresh, return ((listof T N_v), {(N_v, N)})
   else return (T, ∅)

sink-type(T) = case T in
   (-> C (T_arg) T_ret)
    → C_v fresh,
       let (T'_arg, K_arg) = lift-type(T_arg)
       let (T'_ret, K_ret) = sink-type(T_ret)
       return ((-> C_v (T'_arg) T'_ret),
               {(C, C_v)} ∪ K_arg ∪ K_ret)
   (numof N)
    → N_v fresh, return ((numof N_v), {(N, N_v)})
   (listof T N)
    → N_v fresh, return ((listof T N_v), {(N, N_v)})
   else return (T, ∅)
```

Figure 9: Subtyping algorithm

to compute a larger type and *sink-type* to compute a smaller type. Recall, all cost and size annotations on types are variables, so the *sink-type* algorithm is well defined.

Subtyping is applied to the types of variable references and the result type of procedure calls in both the static semantics and the reconstruction algorithm; however, the application of *lift-type* requires that the type be known. In most type reconstruction algorithms, the `lambda` case introduces a fresh type variable allowing the type of the variable to be determined by the context. This implies that the type of the variable is initially unknown and thus some uses of the variable may be given the type variable as a type before the determining context is reached. Such an approach is not sufficient to implement the static semantics of the previous section because subtyping cannot be correctly applied to the type variable. Thus we assume that the program is explicitly annotated with type skeletons that provide information about the bound variables in `lambda` and `rec` expressions. Type skeletons are types without cost or size annotations. In our implementation, type skeletons are computed by a reconstruction algorithm similar to Tofte's [T87]. The type skeletons are converted to annotated types by the *newtype* algorithm (Figure 10) which inserts fresh cost and size variables. Thus the type of the variable will be known for all references and *lift-type* can be correctly applied.

In the case for `rec`, the latent cost of the recursive procedure C_v is forced to be greater than or equal to the cost of the procedure body C_b. If the recursive procedure is ever called, then the cost of the body is at least the latent cost of the recursive procedure plus the call overhead. Thus, $C_v \geq C_b \geq$ (`sum C_call` C_v) which implies $C_v =$ `long`.

```
newtype(T_s) = case T_s in
   (-> _ (T_arg) T_ret)
    → C_v fresh,
       return (-> C_v (newtype(T_arg)) newtype(T_ret)))
   (numof _)
    → N_v fresh, return (numof N_v)
   (listof T _)
    → N_v fresh, return (listof newtype(T) N_v)
   (refof T) → return (refof newtype(T))
   else return T
```

Figure 10: *newtype*

```
U(T_1, T_2) = case (T_1, T_2) of
      (base type, base type) or (T_v, T_v) → [ ]
      ((refof T), (refof T')) → U(T,T')
      (T_v, T) or (T, T_v) → if T_v ∈ FV(T) then fail
                             else [T/T_v]
      ((-> C_v (T) T_r), (-> C'_v (T') T'_r))
          → let S_0 = U(T, T')
            let S_1 = U(S_0 T_r, S_0 T'_r)
            return [C_v/C'_v]S_1 S_0
      ((numof N_v), (numof N'_v)) → [N_v/ N'_v]
      ((listof T N_v), (listof T' N'_v))
              → let S= U(T, T')
                return [N_v/N'_v]S
      else fail
```

Figure 11: μFX/SDC Unification Algorithm

4.1 Unification Algorithm

The unification algorithm in Figure 11 is straightforward and in the spirit of Robinson [R65]. The unification algorithm works on types:

$$U : (Type \times Type) \rightarrow Substitution$$

Unification of procedure types not only requires unifying the input and return types, but also unifying their latent costs. Unification of data structure types such as natural numbers and lists requires unification of the size estimates. Unification of costs and sizes, however, is straightforward because types only include cost and size variables.

4.2 Constraint Solving

The final phase of type and cost reconstruction involves solving the deduced constraints. Each constraint describes a lower bound on a cost or size variable. Since costs depend on size estimates, we must solve the constraints on size variables first. Since sizes and costs have the same algebra, we can solve them with a single algorithm. We will use C to denote both costs and sizes in the following discussion. Multiple constraints on a single variable are merged by taking the `max` of the lower bounds.

The constraint set is always solvable by assigning all C_{v_i} to `long` since `long` is greater than any size or cost. However, `long` does not provide us with any useful information, so we would like a minimal assignment to C_{v_i} that satisfies the constraints. This is referred to as the least pre-fixpoint

of the constraint equations [A90]. The *least pre-fixpoint* is the smallest solution of a set of inequalities. Because the operators **sum**, **prod**, and **max** are monotonically increasing and continuous, the least pre-fixpoint can be calculated with the least fixpoint. The *least fixpoint* is the smallest solution to a set of equalities. The least fixpoint can be calculated by assigning the variables to 0 and counting up, but this will not halt if the least fixpoint of some variable is **long**.

The following algorithm *CS* computes the least fixpoint by counting up but recognizes when it has entered a loop. Thus it can halt and assign **long** to the required variables. Assume there are n bounds in the constraint set. Let V_i be the constrained variables and F_i the associated lower bounds.

Least Fixpoint Algorithm, *CS*:

[Step 1] $\forall\ i$, set $V_i = 0$.

[Step 2] Repeat n times:
$\forall\ i$, set $V_i = F_i(V_1, ..., V_n)$.

[Step 3] $\forall\ i$, let $V'_i = V_i$.
Repeat n more times:
$\forall\ i$, set $V_i = F_i(V_1, ..., V_n)$.

[Step 4] $\forall\ i$, if $V_i \neq V'_i$ then set $V_i =$ **long**.

On each iteration, the value for the variable is updated based on the values of all variables from the previous iteration. If the least fixpoint of a variable is **long**, then the variable's value will change at least every n iterations because the circular dependency involves at most n constraints. If the least fixpoint of a variable is finite, then it will be computed in the first n iterations and will not change on subsequent iterations.

This algorithm is quadratic in the number of constraints and can be improved in two ways. The easiest improvement is to group the constraints into strongly connected components so that the algorithm is only run on a set of constraints that includes circular dependencies. The more important improvement is to eliminate unnecessary constraints. When type checking a **lambda** expression, the only item of interest is how the output costs and sizes depend upon the input costs and sizes and the costs and sizes in the type environment. However, the reconstruction algorithm inserts constraints in every place that subtyping, subsizing or subcosting can be used in the static semantics. Each constraint behaves as a "rubber band" by allowing the cost or size to be increased as needed during constraint solving. Thus the dependence between the inputs and outputs of a procedure is expressed by a chain of "rubber bands," but flexibility within this chain is no longer needed and it can be replaced with a single "rubber band." In our implementation, this is done by running the constraint solving algorithm on a subset of the constraint set that does not include constraints on the inputs and outputs. Reducing the size of the constraint set is particularly important if the **lambda** expression occurs in a let binding and is generalized over because the constraint set will be copied each time the variable is referenced and its type scheme is instantiated.

Our cost algebra does not contain a subtraction operator because it complicates constraint solving. The major problem with subtraction is that a minimal solution to the constraint set is meaningless. Consider the constraint set $\{C_{v1} \geq 5,\ C_{v2} \geq$ (**sub** 10 C_{v1})$\}$. Minimizing C_{v1} makes C_{v2} larger and vice versa. An alternative approach could possibly provide better size information by using interval arithmetic [H77]. One could allow a subtraction operator and compute a reduced constraint set, but what should be done with the reduced constraint set remains an open issue.

4.3 Correctness Issues

The reconstruction algorithm is sound if the type and cost it computes are a valid solution to the static semantics. An inspection of Figures 5 and 8 shows that the reconstruction algorithm directly implements the static semantics. Each subtyping, subcosting, or subsizing clause in the static semantics in mirrored by the use of constraint sets in the reconstruction algorithm. For example, the subsizing in the **num** rule (Figure 5) is implemented by the constraint $\{(N_v, Nat)\}$ in the case for natural number literals of R (Figure 8). The subtyping relation of Figure 6 is implemented by the subtyping algorithm of Figure 9. The reconstruction algorithm would be sound even if it reported **long** for all costs and sizes. It would only be unsound if it failed to merge constraints to reflect the requirement that two types be equivalent; however, this is implemented by the unification algorithm (Figure 11) which is straightforward and invoked in the procedure application and **if** cases. Thus we believe the reconstruction algorithm is sound even though we have not given a formal proof.

The reconstruction algorithm is complete if the type and cost it computes are the best solution to the static semantics. Completeness depends upon two things: the reconstruction algorithm must generate all the appropriate constraints and the constraint solver must compute the optimal solution. The first requirement seems to be met as discussed above for soundness; however, while overly conservative constraints are not a problem for soundness, they do affect completeness. The second requirement that the constraint solver compute the optimal solution depends on how that solution is expressed. Types are directly annotated with cost and size expressions in the static semantics while the reconstruction algorithm annotates types with cost and size variables and records information about the actual costs and sizes in the constraint set. An argument for completeness must compare costs and sizes from the static semantics and the reconstruction algorithm; however, if the completeness argument is to be inductive, the constraint solver cannot replace the cost and size variables in the resultant type with actual cost and size expressions (because this violates one of the assumptions of the reconstruction algorithm). Constraint solving must proceed by reducing the size of the constraint set while maintaining the invariants of the reconstruction algorithm. Demonstrating that the constraint solver meets these criterion requires establishing two things: the reconstruction algorithm must create constraints which admit a minimal solution and the constraint solver must compute this solution. A solution is minimal if all other solutions to the constraint set can be expressed in terms of the the minimal solution. Thus completeness remains an open issue, but as long as the reconstruction algorithm computes good solutions it will be useful pragmatically.

5 Using Our Cost System to Predict Execution Times

Our cost system successfully predicted execution costs within a factor of three for various programs run on different compilers and architectures. We have conducted experiments on the three different compilers and target architectures described briefly below:

	μFX/DLX	Mul-T no caching	SML/NJ
	# instrs	# cycles	μsecs
C_call	39	21	1.50
C_lambda	10	3	0.50
C_var	10	1	1.00
C_let	50	24	2.00
C_if	5	4	0.00
C_+	1	1	3.00
C_*	5	15	4.00
C_cons	8	12	1.67
C_map-overhead	118	45	3.01
C_map-per-elem	101	56	14.06
C_map2-overhead	133	18	2.27
C_map2-per-elem	118	76	10.15
C_reduce-overhead	126	27	4.81
C_reduce-per-elem	98	42	5.15

Table 1: Values for Symbolic Constants

- The μFX/DLX compiler is a very simple compiler used for instructional purposes at MIT. As such, it emphasizes readability over performance optimizations. It compiles μFX to Hennessy and Patterson's DLX architecture [HP90]. We measured the actual number of DLX instructions executed.
- The Mul-T compiler compiles a parallel version of T to the Alewife machine. Our experiments were run with ASIM, a cycle-by-cycle simulator for the Alewife machine [L92]. We measured the actual number of cycles executed in a configuration of one processor.
- We used the SML/NJ compiler (version 0.93) on a Sparc IPX to run experiments after a simple syntactic translation from μFX/SDC to ML. We measured actual execution time with the SML/NJ profile tool [SML/NJ93].

We have implemented the reconstruction algorithm discussed in the previous section. Our implementation computes the cost of program expressions in terms of symbolic constants that describe the cost of basic language components. We experimentally determined values for these constants in the above systems. Since these constants form the basis of our system's cost predictions, it is important to determine accurate values while still providing conservative upper bounds. Table 1 summarizes some of the values.

The values in Table 1 are the best conservative bounds we could experimentally determine. Some values were determined by inspecting actual machine code, but most were picked by running small experiments and choosing an upper bound on the actual cost over a number of trials. Some of the constants were difficult to estimate because they could not be independently estimated. For instance, the cost of C_if was too small to measure for SML/NJ. The per element costs for primitive iterators were influenced by caching effects as discussed below.

5.1 Experimental Results

We ran three test programs on a variety of inputs to examine the accuracy of the static cost estimates. The cost estimates were always within a factor of three of the actual cost and correctly captured how the costs depended on problem size. Table 2 presents the ratio of estimated cost to actual cost. Not all programs were run on every system because some systems did not provide sufficient facilities. We describe each program below and discuss the results in detail for matrix multiply.

	μFX/DLX	Mul-T no caching	SML/NJ
Matrix Multiply	1.39 – 1.11	1.23 – 1.09	1.72
Game of Life		1.24	1.20
N-body Simulation			1.51

Table 2: Ratio of Estimated Cost to Actual Cost

Matrix multiply We implemented matrix multiplication using data parallel operators as shown in Figure 12. The procedure takes a matrix x and the transpose of a matrix y and uses reduce and map2 to calculate the dot product of a row from x and a column from y. In this expression the cost of a matrix multiply is: (the number of rows in x) × (the number of columns in y) × (the cost to calculate the dot product).

Figure 12 also includes the type of the procedure as inferred by the system. The system has deduced that multiplying an $N_m \times N_n$ matrix by an $N'_n \times N_k$ matrix yields an $N_m \times N_k$ matrix. The system does not force N_n and N'_n to be equal because it allows subsizing flexibility for each input matrix. Most importantly, the system infers a latent cost proportional to the dimensions of the input matrices. Lastly, the size of the elements in the resultant matrix are unbounded because reduce is passed +, forcing the following types to be equivalent:

(-> C_+ ((numof N_1) (numof N_2)) (numof (sum N_1 N_2))
(-> C (T_1 T_2) T_2)

This forces $N_2 =$ (sum N_1 N_2) which has solution long.

Figure 13 plots predicted and actual cost in each system for multiplying two square matrices of increasing size. The predictions closely matched the actual cost for μFX/DLX and Mul-T without caching. The ratios shown in Table 2 are for dimensions from 0 to 6 with the worst prediction for dimension 0.

Our cost system does not model the effects of the cache because it assumes primitive iterators incur the same overhead for each element. Figure 13 contains two plots for Mul-T: one with caching and one without. Without caching the simulator assumes memory accesses are satisfied in one cycle, so the actual cost incurred with caching is slightly larger than the cost without caching. The predictions for the system with caching are significantly larger because we must assume worst case cache performance. As expected, the predictions were less accurate for SML/NJ because of caching effects.

To analyze the automatically generated cost expression we fit a polynomial to the observed actual execution time. The algorithm is $O(n^3)$ for square matrices, so we fit the experimental data to a degree-3 polynomial. We compared the experimental polynomial term by term with the static cost expression. For large matrices, the actual and estimated cost are dominated by the highest order term. Thus, the amount of overestimation is the ratio of the coefficient for the third

```
E = (let ((dot-product (lambda (r c) (reduce + (map2 * r c) 0))))
       (lambda (x y-transpose)
           (map (lambda (row)
                    (map (lambda (col) (dot-product row col))
                         y-transpose))
                x))

E : (-> (sum C_call C_lambda C_map-overhead (* 2 C_var)
            (prod N_m (sum C_map-per-elem C_call C_lambda C_map-overhead (* 2 C_var)
                           (prod N_k (sum C_map-per-elem C_map2-overhead C_num
                                          C_reduce-overhead (* 3 C_call) (* 9 C_var)
                                          (prod (max N_n N'_n)
                                                (sum C_map2-per-elem   C_*
                                                     C_reduce-per-elem C_+ ))))))))

      ( (listof (listof (numof N_x) N_n) N_m)
        (listof (listof (numof N_y) N'_n) N_k))
      (listof (listof (numof long) N_k) N_m))
```

Figure 12: Matrix Multiply Code and Inferred Type

degree term. For Mul-T with caching, the coefficient was overestimated by a factor of three while for SML/NJ, the coefficient was overestimated by less than a factor of two. The results in Table 2 show that for the SML/NJ experiments and the other experiments except for Mul-T with caching, the cost estimates were within a factor of two of the actual cost. Caching is a large source of cost overestimation, but we were surprised its effects were not worse.

Game of Life We implemented the game of life using FX's permutation operators and our system is able to predict the cost of computing a single generation. We ran experiments for the game of life on a five by five grid.

N-body simulation We translated Sussman's code for the n-body problem [ADGHSS85] as found in [M87] to μFX/SDC. This code simulates the movement of the solar system by using differential systems. Our cost system is able to assign a finite cost to a single step of the simulation which depends on the number of bodies being simulated.

5.2 Sources of Overestimation

As shown above, our cost system successfully predicted execution times for data parallel programs within a factor of three. Variances between the static cost estimate and the actual cost incurred arose for a variety of reasons:

- Our cost analysis does not distinguish between primitives such as + that can be in-lined and procedures that require general procedure call overhead. Thus, we must assume that every application incurs this overhead.

  Other optimizations such as common subexpression elimination and constant folding can also contribute to overestimation.

- We overestimate sizes and costs where necessary to avoid reporting a type error because of conflicting size or cost descriptions. This approximation results in overestimation when the smaller size or cost is dynamically incurred. The same holds true for the two branches of an `if` expression.

  We must also make conservative approximations on the cost of first-class procedures. This allows us to express the cost of higher-order procedures in closed form, but also gives rise to overestimation. For example, the cost of `map` assumes its procedural argument has the same cost for all elements of the list.

- As discussed above, our system does not model the effects of caching. Since we are guaranteeing upper bound cost estimates, we must assume worst case cache performance.

Our cost system also does not model the effects of garbage collection which can cause the system to underestimate the actual cost. The cost of garbage collection is an important factor, but it is difficult to predict when garbage collection will be initiated and how to account for its distributed costs. Thus, the measurements in this section do not include the cost of garbage collection.

6 Dynamic Parallelization

Efforts in automatic parallelization have been primarily concerned with identifying expressions that can be *safely* executed in parallel [HG88, JG89, TJ93, HL92]. However, a static cost system provides information about what expressions can be *profitably* evaluated in parallel. It is only worthwhile to evaluate two expressions in parallel if the time saved is greater than the cost of setting up the parallel computation. Gray [G86] introduced a system for inserting futures that estimates costs based on a local, syntactic method.

Our cost system provides static cost estimates that can be used to make parallelization decisions at compile time. If the cost expression contains no free cost or size variables, then a dynamic parallelization decision can be made at compile time. If there is a definite benefit to be gained by parallel execution, the appropriate code can be inserted. If cost analysis shows there is no benefit, then the code is left unchanged.

If the cost estimate contains free cost or size variables, then the parallelization decision must be made dynamically. This can occur if the cost of a procedure is polymorphic in the cost or size of some of its inputs. When such a polymorphic procedure is called, dynamic information must be passed to convey free costs and sizes. Then a decision can be

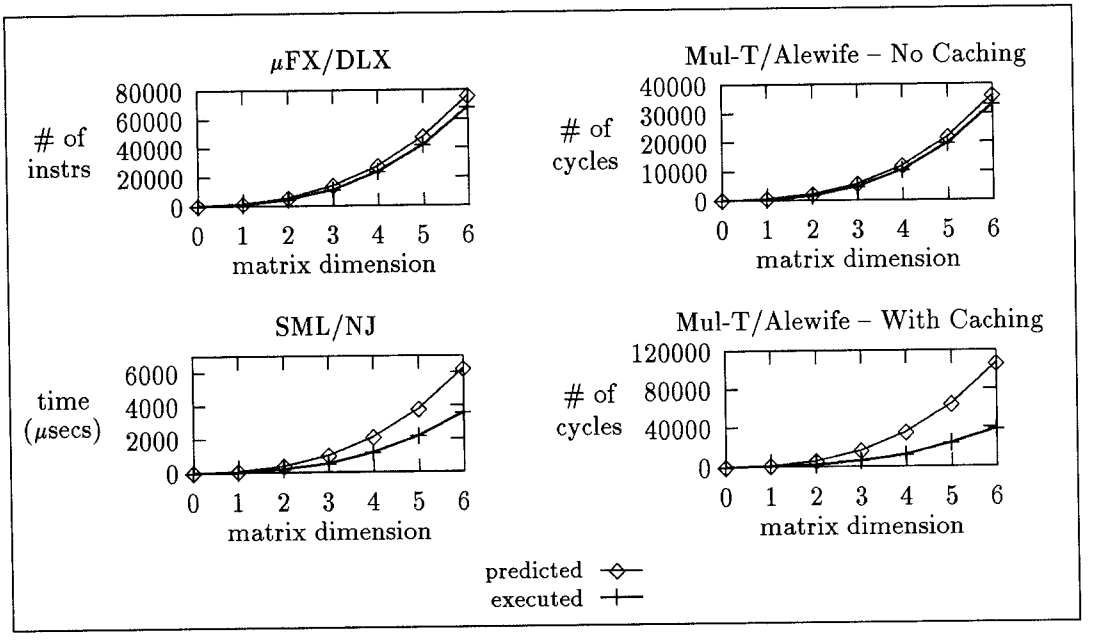

Figure 13: Matrix Multiply – Predictions and Actual Execution

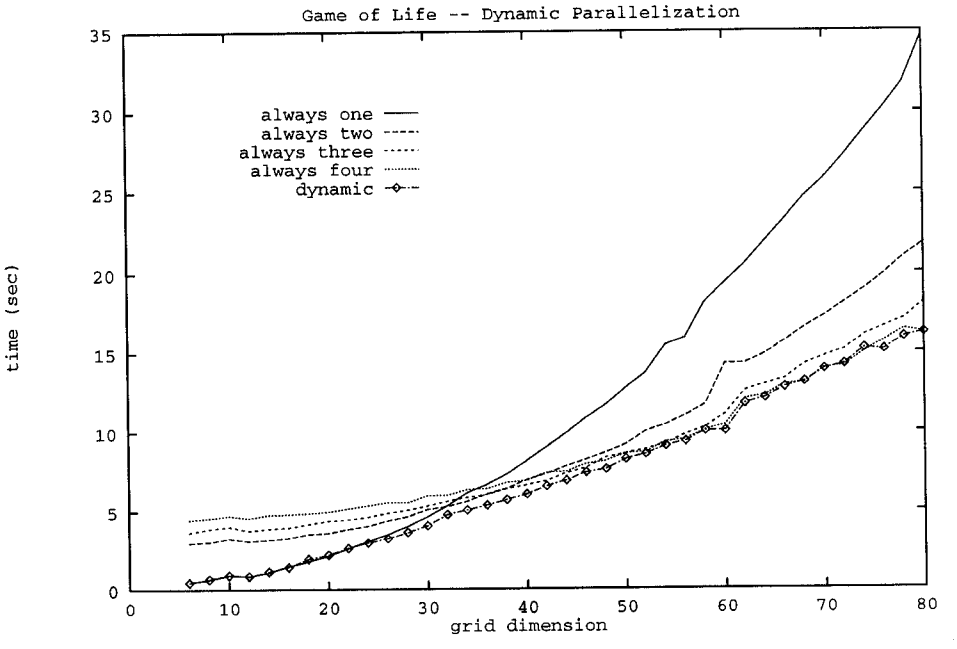

Figure 14: Dynamic Parallelization of the Game of Life

made dynamically on which parts of the body can be profitably evaluated in parallel. Thus, once type and cost reconstruction is complete, the program can be annotated with dynamic information where appropriate. In our simple dynamic parallelization system, this annotation was performed by hand. There is no fundamental obstacle to automating this annotation; a similar mechanism has been used with effect systems [TJ93, HL92].

We use maximum cost estimates for parallelization decisions even though they cannot guarantee speed up. If the maximum cost is much larger than the actual cost then we may parallelize when it is not profitable. On the other hand, if the maximum cost is in fairly close correspondence with the actual cost then parallelization will have been successful. As shown in Section 5, our cost estimates were within a factor of two for the majority of the experiments, so we correspondingly adjusted our parallelization threshold by a factor two. Minimum cost estimates could ensure speed up, but may overlook opportunities for parallelism if the minimum estimate is too low.

We built a simple dynamic parallelization system on an SGI IRIX computer with four processors using SML/NJ with a multiprocessor interface [MT92]. Our system exploits data parallelism by performing vector map operations in parallel. Since there are a limited number of processors, the vector is broken into segments and each processor performs the vector map on a given segment. The extra processors busy wait until there is work for them to do.

Adding multiprocessor support to our program slowed its single-processor performance by 40%. This overhead was traced in part to the SML/NJ multiprocessing support software and possibly could be eliminated. The overhead imposed by dynamic cost and size information is not noticeable.

We tested our dynamic parallelization system with the game of life. The program was manually annotated with the cost and size information computed by our reconstruction algorithm. We experimentally determined values for the symbolic constants and the fork cost for SML/NJ on the SGI IRIX computer as explained in Section 5. Then we ran trials in which the program decided dynamically how many processors to use for each vector map. Five strategies were considered:

- Always use a constant number of processors for all vector maps, either 1, 2, 3 or 4.
- Dynamically choose how many to use based on the latent cost of the procedure being mapped and the cost of forking off a new thread.

Figure 14 shows these tests for the game of life with the grid size ranging from 10 to 80. To reduce the effects of garbage collection, a major garbage collection was forced before each trial. Trials for grid sizes above 60 incurred one major garbage collection while those bellow 60 ran without requiring a major garbage collection. To reduce other system noise each trial was repeated ten times and the number reported is the average of the smallest five.

The dynamic strategy chooses the optimal number of processors for small and large grid sizes. For grid sizes less than 25, using a single processor is clearly the best strategy, while for large grids (above 60 or so) using all four processors is best. At these extremes, the dynamic strategy is choosing to use the optimum number of processors.

The dynamic strategy performed better than all other strategies for grid sizes in the range from 25 to 50. This may seem impossible at first as the dynamic strategy is supposed to simply pick the optimum number of processors for a particular grid size. In fact, the dynamic strategy is able to make multiprocessing decisions independently for different subcomputations. Thus the superior performance in this range is the result of the dynamic strategy using a different number of processors on different parts of the computation. For instance when the grid size is 42, the dynamic strategy chooses to use two processors to compute the number of neighbors for each cell, three processors to apply the liveness criterion to each cell, and four processors to shift grids in preparation for the neighbor calculation.

Conclusion

We introduced the notion of static dependent costs to describe the execution times of expressions that depend on the size of data structures. Our cost system uses static types to provide bounds on the size of data structures and to provide information about latent costs. Our system includes static dependent costs for primitive data parallel operators which allow us to predict costs for a number of programs. We successfully predicted the execution cost of these programs within a factor of three on a variety of target systems. We demonstrated the utility of static cost estimates in a simple dynamic parallelization system that was able to selectively choose how many processors to use based on cost information.

Our cost system could be improved by integrating our ideas for handling first-class procedures and mutation with previous work in automatic complexity analysis. Our cost analysis could also produce estimates of storage costs or communication costs by re-interpreting the symbolic constants. Our system could also benefit from minimum estimates on the size of data structures.

Acknowledgments

We would like to thank Forest Baskett for providing us with the SGI processor boards that allowed us to perform the multiprocessing experiments. Additional thanks to Pierre Jouvelot and Vincent Dornic for their previous work and discussions relating to our extensions. Thanks also to our readers: Michael Blair, Jim O'Toole, Jonathan Rees, Guillermo Rozas, Mark Sheldon, and Franklyn Turbak.

References

[ADGHSS85] Applegate, J.F., Douglas, M.R., Gürsel, Y., Hunter, P., Seitz, C., and Sussman, G.J. A Digital Orrery. *IEEE Computer*, September 1985.

[A90] Apt, K.R. Logic Programming. *Handbook of TCS, Vol B, Formal Models and Semantics*, Jan Van Leeuwen (editor), Elsevier, 1990, 491-574.

[B78] Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *CACM*, 21(8), 1978, 613-641.

[CBF91] Chatterjee, S., Blelloch, G.E., and Fisher, A.L. Size and Access Inference for Data-Parallel Programs. *PLDI 1991*, 130-144.

[C82] Cohen, J. Computer-Assisted Microanalysis of Programs. *CACM*, 25(10), 1982, 724-733.

[D92] Dornic, V. *Analyse de Complexité des Programmes: Vérification et Inférence.* Ph.D. Thesis, Ecole des Mines, Paris, France, CRI/A/212, June 1992.

[D93] Dornic, V. Ordering Times. Yale University, Research Report YALEU/DCS/RR-956, April 1993.

[DJG92] Dornic, V., Jouvelot, P., and Gifford, D.K. Polymorphic Time Systems for Estimating Program Complexity. *LoPLaS*, 1(1), 1992, 33-45.

[GJSO92] Gifford, D.K., Jouvelot, P., Sheldon, M.A., and O'Toole, J.W. Report on the FX-91 Programming Language. MIT/LCS/TR-531, February 1992.

[G88] Goldberg, B. Multiprocessor Execution of Functional Programs. *International Journal of Parallel Programming*, 17(5), 1988, 425-473.

[G86] Gray, S.L. *Using Futures to Exploit Parallelism in Lisp.* M.S. Thesis, MIT, February 1986.

[GSO92] Grundman, D., Stata, R., and O'Toole, J. μFX/DLX – A Pedagogic Compiler. MIT/LCS/TR-538, March 1992.

[HG88] Hammel, R.T., and Gifford, D.K. FX-87 Performance Measurements: Dataflow Implementation. MIT/LCS/TR-421, November 1988.

[H77] Harrison, W.H. Compiler Analysis of the Value Ranges of Variables. *IEEE Transactions on Software Engineering*, SE-3(3), 1977, 243-250.

[HP90] Hennessy, J.L., and Patterson, D.A. *Computer Architecture – A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[HL92] Huelsbergen, L., and Larus, J. Dynamic Program Parallelization. *LFP 1992*, 311-323.

[HLA94] Huelsbergen, L., Larus, J., and Aiken, A. Using the Run-Time Sizes of Data Structures to Guide Parallel-Thread Creation. *LFP 1994.*

[JG89] Jouvelot, P., and Gifford, D.K. Parallel Functional Programming: The FX Project. *Parallel and Distributed Algorithms*, M. Cosnard *et al.* (editors), Elsevier Science Publishers B.V. (North-Holland), 1989, 257-267.

[JG91] Jouvelot, P., and Gifford, D.K. Algebraic Reconstruction of Types and Effects. *POPL 1991*, 303-310.

[K68] Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, 1968.

[L88] Le Métayer, D. ACE: An Automatic Complexity Evaluator. *ToPLaS*, 10(2), 1988, 248-266.

[L92] Lim, B. Instructions for Obtaining and Installing ASIM. MIT/LCS, Alewife Systems Memo #30, September 1992.

[LG88] Lucassen, J.M., and Gifford, D.K. Polymorphic Effect Systems. *POPL 1988*, 47-57.

[M87] Miller, J.S. *MultiScheme: A Parallel Processing System Based on MIT Scheme.* Ph.D. Thesis, MIT/LCS/TR-402, September 1987.

[MKH90] Mohr, E., Kranz, D.A., and Halstead, R.H. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *LFP 1990*, 197-185.

[MT92] Morrisett, J.G., and Tolmach, A. A Portable Multiprocessor Interface for Standard ML of New Jersey. Carnegie Mellon University, CMU-CS-92-155, June 1992.

[R65] Robinson, J.A. A Machine Oriented Logic Based on the Resolution Principle. *JACM*, 12(1), 1965, 23-41.

[R89] Rosendahl, M. Automatic Complexity Analysis. *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, 1989, 144-156.

[S88] Sands, D. Complexity Analysis for Higher Order Languages. Imperial College, London, Research Report DOC 88/14, October 1988.

[S90] Sands, D. *Calculi for Time Analysis of Functional Programs.* Ph.D. Thesis, University of London, September 1990.

[SH86] Sarkar, V., and Hennessy, J. Compile-Time Partitioning and Scheduling of Parallel Programs. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, 1986, 17-26.

[SC93] Skillicorn, D.B., and Cai, W. A Cost Calculus for Parallel Functional Programming. Queen's University, Kingston, Canada, ISSN-0836-0227-93-348, March 1993.

[SML/NJ93] *Standard ML of New Jersey Reference Manual (Version 0.93).* February 1993.

[T93] Talpin, J.P. *Aspects Théoriques et Pratiques de l'Inférence de Type et d'Effets.* Ph.D. Thesis, Paris University VI, May 1993.

[TJ92] Talpin, J., and Jouvelot, P. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(3), 1992, 245-271.

[TJ93] Talpin, J., and Jouvelot, P. Compiling FX on the CM-2. *Proceedings of the Third Workshop on Static Analysis*, LNCS 724, September 1993, 87-98.

[T87] Tofte, M. *Operational Semantics and Polymorphic Type Inference.* Ph.D. Thesis, University of Edinburgh, 1987.

[W91] Wall, D. Predicting Program Behavior Using Real or Estimated Profiles. *PLDI 1991*, 59-70.

[W75] Wegbreit, B. Mechanical Program Analysis. *CACM*, 18(9), 1975, 528-539.