Building Incremental Programs
Using Partial Evaluation

R.S. Sundaresh
Research Report YALEU/DCS/RR-828
November, 1990

This work is supported by an IBM graduate fellowship.

Building Incremental Programs using Partial Evaluation

R. S. Sundaresh *
Yale University
Department of Computer Science
Box 2158 Yale Station
New Haven, CT 06520
sundaresh@cs.yale.edu
YALEU/DCS/RR-828

November 29, 1990

Abstract

This paper describes an implementation of a high level framework for constructing *incremental programs* – programs which can efficiently update the result of a computation when the input changes only slightly. This framework was described in [SH91] and uses *partial evaluation*. The implementation is based on the partial evaluator Schism [Con90b]. As an example of the use of the framework, the construction of an incremental data flow analyzer is described in detail. The framework can be expressed in the form of an *incremental interpreter*. Efficient "compiled" incremental programs are obtained by specializing this incremental interpreter. Thus, partial evaluation is used for two purposes: in the framework to describe incremental programs, and to implement the framework efficiently. Performance results for interesting problems show sizeable speedups of the incremental program over the corresponding non-incremental one.

1 Introduction

Incremental tools – i.e. ones which can efficiently update the result of a computation when the input changes only slightly – are increasingly playing an important role in programming environments. It is very common to apply a program (e.g. interpreters, compilers, text formatters etc.) to a series of similar inputs. However, despite the preponderance of work on incremental algorithms and programs, formal and general treatments of the problem are rare. Historically the approach has been to hand-craft incremental algorithms for many important problems, and as a result common elements of the designs are often obscured. Indeed, looking at various extant incremental algorithms, one

*Supported by an IBM graduate fellowship.

might be led to believe that there is no common element at all! There seems to be some consensus that incremental algorithms are hard to derive, debug and maintain [Pug88, YS89, FT90], and as we attempt to create incremental programs for larger tasks, this problem will only get worse.

This problem is addressed in [SH91] where a *framework* for incremental computation based on *partial evaluation* [BEN88] is presented. This framework makes use of *residual functions* (the results of partial evaluation) to provide a problem-independent way of storing intermediate computations. In addition to providing a precise definition of the term "incremental program", this framework offers a methodology to generate an incremental program from its non-incremental counterpart plus a specification of a partition of the input object. (This reduces the designer's primary task to determining the partition of the input domain, which controls the "granularity" of the incrementality as well as overall efficiency.) It also provides an algebraic basis for reasoning about the correctness of the incremental programs thus generated. While there have been other attempts to provide a general framework for the construction of incremental programs, this framework is the only one based on the notion of partial evaluation.

In this paper we describe an implementation of the framework described in [SH91] and evaluate its performance on some interesting problems. We begin by reviewing the salient features of our framework for incremental computation. Each stage of the implementation is described using the example of incremental data flow analysis. First the problems of transforming the data flow analyzer to a form amenable to partial evaluation are discussed. Then an implementation of the algorithm to compute the least upper bound of residual functions is described. Finally these results are combined to construct an *incremental interpreter* which executes a specification of an incremental program efficiently. We describe the methods used to avoid the overhead of interpretation and to generate "compiled incremental programs". We also present performance results for two problems: incremental data flow analysis and incremental attribute grammar evaluation. These results sizeable speedups of the incremental program over the non-incremental (batch) program.

Partial evaluation is used for two distinct purposes: it forms the basis of the framework to describe incremental programs. It is also used to *implement* this framework efficiently (by specializing the incremental interpreter). Another interesting feature is the reliance on partial evaluators based on binding time analysis: as we will see, the implementation depends crucially on binding time information of the program.

The implementation is based on Schism, which is a partial evaluator for a side-effect free dialect of Scheme [Con90b, Con90a]. The source programs are written in pure Scheme: a dynamically typed, applicative order implementation of lambda-calculus. Schism handles *higher order functions* as well as the *data structures* manipulated by the source programs, even when they are only *partially known*. Schism is written in pure Scheme and is *self-applicable*. Schism also provides mechanisms to control the specialization process by means of user annotations called *filters* [Con88].

2 Incremental Computation and Partial Evaluation

2.1 Partial Evaluation

This section describes the notation we will use for *partial evaluation*. We use Launchbury's [Lau88] definition of partial evaluation using *projections*.

Definition 2.1 *A* projection *on a domain $\mathcal{D}$ is a continuous mapping $p : \mathcal{D} \to \mathcal{D}$ such that:*

- $p \sqsubseteq ID$ *(no information addition)*
- $p \circ p = p$ *(idempotence)*

Note that *ID* (the identity function) is the greatest projection and *ABSENT* (the constant function with value $\perp$) the least (under the standard information ordering on functions).

Definition 2.2 *If p and q are projections and $p \sqcup q = ID$, then q is a* complement *of p.*

Note that by the above definition the complement of a projection may not be unique (for example, *ID* is a complement of *every* projection). This problem is avoided by choosing an appropriate domain of projections where complement is defined in terms of a difference operator. (for details, see [SH91]). We write $\bar{p}$ to denote the unique complement of p.

Definition 2.3 *A* partial evaluator *$\mathcal{PE}$ is a function which takes representations of a function f, a projection p, and a value a, and produces a representation of the residual function, f_{pa}, defined as follows:*

$$\mathcal{PE}\ f\ p\ (apply\ p\ a)\ =\ f_{pa}$$

such that

$$apply\ f_{pa}\ (apply\ \bar{p}\ a) = apply\ f\ a$$

where apply takes the representation of a function and its argument and produces a representation of the result.

The idea here is to use projections to capture the known parts of the input. When there is no ambiguity we use r_p to denote $\mathcal{PE}\ f\ p\ a$. Given this notation, note that $r_{ID} = apply\ f\ a$.

In [Lau90] Launchbury has extended this framework by showing how the *type* of the residual function depends on the static value used to produce it. Using the notion of dependent sum and dependent product domain constructions, he shows how the residual function can be made to take as little information as possible.

2.2 Incremental Computation

Returning now to the problem of incremental computation, we can summarize the situation as in Figure 1. Here the function f (which may be a compiler, text formatter, etc.) is being applied to a structured argument to give the result. If only part of the argument changes, such as part **b**, we would like to compute the new result without having to redo the entire computation; in other words, we would like to avoid having f reprocess parts **a**, **c**, and **d**.

Now, here's the connection to partial evaluation, and the basis of our framework: The partitioning of the input domain can be described using a set of *projections* as defined in the previous

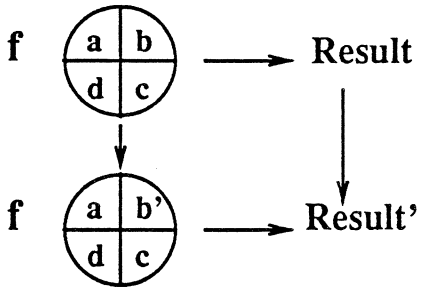


Figure 1: Incremental computation

section; let's call them p_a, p_b, p_c, and p_d for the example in Figure 1. If we then compute the residual functions r_{p_a}, r_{p_b}, r_{p_c}, and r_{p_d}, we have essentially "cached" those portions of the computation that depend only on parts **a**, **b**, **c**, and **d** of the input, respectively.

Recalling that $r_{ID} = \mathit{apply}\ f\ a$, all we need now to compute the final result is a (presumably efficient) way to construct r_{ID} from the set of residual functions — for now, let's assume that such a technique exists. If part of the input were to change, say **b** changes to **b'**, then all we have to do is replace r_{p_b} with $r_{p_{b'}}$; computation of r_{ID} then takes place with this new residual function in place.

An alternative way to describe this process is as follows: At the point when **b** changes to **b'**, suppose we had by some means already computed $r_{\bar{p_b}}$ — then all we need to do to compute the new result is to apply $r_{\bar{p_b}}$ to **b'**. We can thus view the problem as an attempt to find (at least a conservative approximation to) $r_{\bar{p_b}}$ by combining existing residual functions.

We can define all this more formally as follows:

Definition 2.4 *A* partition *P of a domain $\mathcal{D}$ is a set of projections $\{p_i\}$ on $\mathcal{D}$ such that $\sqcup\{p_i\} = ID$.*

Definition 2.5 *An* incremental program specification *is a pair $\langle f, P\rangle$ where $f : \mathcal{D} \rightarrow \mathcal{E}$ is the function to be incrementalized and P is a partition of $\mathcal{D}$.*

We now describe an "incremental interpreter," denoted $\mathcal{I}$, which captures the methodology described earlier. $\mathcal{I}$ has functionality:

$$\mathcal{I} : \langle f, P\rangle \rightarrow a_0 \rightarrow \langle \delta_0, \delta_1, \ldots\rangle \rightarrow \langle b_0, b_1, \ldots\rangle$$

$\langle f, P\rangle$ is the incremental program specification, and a_0 is the initial argument. The δs are functions capturing "small" changes to the input, and the bs are the successive output results.

Algorithm $\mathcal{I}$:

- **Setup:** Compute $r_{p_i} = \mathcal{PE}\ f\ p_i\ a$ for each p_i in the partition P.
- **Reestablish:** If a changes to a', recompute all r_{p_i} for which $p_i\ a \neq p_i\ a'$.

- **Combine:** The new result r_{ID} is obtained from $\{r_{p_i}\}$ using appropriate combining operations.

The main purpose of $\mathcal{I}$ is to maintain the invariant: $r_{p_i} = \mathcal{PE}\ f\ p_i\ a$ for all $p_i \in P$, and in so doing satisfies the following correctness criterion:

$$b_i = f\ a_i \text{ where } a_i = \delta_{i-1}\ a_{i-1}$$

This forms the basis for our approach. In [SH91] we show how the notion of "combining" residual functions can be formalized as a *least upper bound* of residual functions in an appropriate domain. An algorithm for this operation (which forms the basis for the implementation) is also discussed. Other questions addressed include the choice of the partition of the input data object. In the sections which follow we will discuss each stage of the incremental interpreter by means of an example.

3 Reaching Definitions

As an example of compiler data flow analysis, consider the problem of determining the set of "reaching definitions" at every program point. The example and the choice of the partition is inspired by an algorithm for incremental data flow analysis from [MR90]. The algorithm we sythesize from our framework has the same characteristics as the the one in [MR90]. A definition of a variable is said to "reach" a program point if there exists a path from the definition to the program point which does not pass through a redefinition of the same variable. For example, consider the flow graph in Figure 2. Each of the circles denotes a basic block, where a label "$x =$" means that x is assigned a value in that block. Arcs are labelled with sets L_i of definitions which reach that arc. For example, the set $L_0 = \{(x, e1), (y, e2)\}$ means that the definition of x at program point $e1$ and y at $e2$ reach the arc labelled L_0. We can then write the following equations ("?" refers to a wildcard):

$$\begin{aligned} L_0 &= \{(x, e1), (y, e2)\} \\ L_1 &= L_0 \cup L_3 \\ L_2 &= L_1 - \{(x, ?)\} \cup \{(x, B)\} \\ L_3 &= (L_1 \cup L_2) - \{(y, ?)\} \cup \{(y, C)\} \end{aligned}$$

The solution to these set equations is defined by a least fixpoint construction, yielding:

$$\begin{aligned} L_0 &= \{(x, e1), (y, e2)\} \\ L_1 &= \{(x, e1), (y, e2), (x, B), (y, C)\} \\ L_2 &= \{(y, e2), (y, C), (x, B)\} \\ L_3 &= \{(x, e1), (x, B), (y, C)\} \end{aligned}$$

Recall that an incremental algorithm specification consists of a non-incremental program plus a partition of the input domain. Therefore we first need to describe a non-incremental algorithm. We assume the input to the algorithm to be a list of strongly connected components of the set of data flow equations in topological order (which can be produced by a standard dependency analysis). The solution is then defined by the following Scheme program:

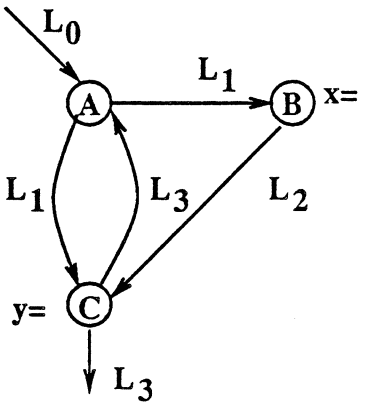


Figure 2: Example flow graph

```
(define (dfa graph env0)
  (foldl (lambda (env scc) (solve scc env)) env0 graph))

(define (solve scc env)
  (let ((env' (next-approx scc env)))
       (if (env-eq? env env')
           env
           (solve scc env'))))
```

`solve` is a function which takes an initial (empty) environment (mapping arc labels to sets) and a set of mutually recursive equations. It then computes the least fixpoint of the equations using the initial environment as the first approximation. Function `next-approx` recomputes the identifiers defined by the equations `eqns` using the old environment `env` to produce a new environment `env'`. To complete the incremental program specification we specify the following input partition:

$P = \{p_i\}$ such that
$p_1\ [] = \bot, p_1\ (x : xs) = (x : \bot)$ and
$p_i\ [] = \bot, p_i\ (x : xs) = (\bot : (p_{i-1}\ xs))$.

This means that each strongly connected component is in a separate element of the partition. This partition enables the fixpoint computation to go to completion on each element of the partition (as will be seen in the following sections).

In the sections which follow, we will show how an incremental program can be extracted from this incremental program specification. In the rest of the paper we will use the terms "known" and `STATIC` interchangeably, and similarly the terms "unknown" and `DYNAMIC`.

4 Partial Evaluation

In the **Setup** and **Reestablish** phases of the incremental interpreter, the computation of r_{p_i} is the main activity. In this section we will see how to do this. Before we use the function defined in the previous section we need to transform the definition into a form amenable to partial evaluation. The projections which form the partition describe partially static input. To make sure that the binding times for arguments to the main function are not partially static, we carry out a process similar to *arity raising*. This implies fixing the number of connected components *a priori*. If the graph grows to exceed this limit, we may either redo this arity raising with a higher number of arguments or we may club more connected components into a single argument. The latter solution ends up using a "coarser" partition of the input and may yield poorer incremental performance. In practice one may choose a sufficiently large arity to avoid this problem. For the example in this paper we choose to convert the list of components into three arguments: (The principles involved for a larger number of arguments remain the same)

```
(define (dfa scc1 scc2 scc3 env0)
  (solve scc3
         (solve scc2
                (solve scc1 env0))))
```

During specialization, if the first argument to the function `solve` is `STATIC`, we know that the solve function can be completely unrolled. This is because the termination of the fixpoint computation does not depend on the environment argument. To convey this information to Schism we make use of the *filter* mechanism. Filters in Schism provide the user with control over the specialization process. First the environment is split into two components: a static and dynamic component. This is done since filters can only contain tests for either `STATIC` or `DYNAMIC`. The fixpoint iteration is always started with the static environment empty. The dynamic environment is the solution to the problem flowing in to the component. Thus the termination of the unfolding process is made to depend only on static information since the dynamic environment does not change during iteration within this component. Given this knowledge, the filter specifies that `solve` can be unfolded only if both `scc` and `env_s` are known:

```
(define (solve scc env_s env_d)
  (filter (if (and (stat? scc) (stat? env_s)) UNFOLD SPECIALIZE)
          (list scc env_s env_d))
  (let ((env_s' (next-approx scc_s env_s))
        (env_d' (next-approx scc_d env_d)))
       (if (env-eq? env_s env_s')
           (env-union env_d env_s)
           (solve scc env_s' env_d'))))
```

The binding time analysis of Schism is *monovariant*, i.e. each function can only have one binding time signature[1]. This loss of information is unacceptable in our case where we wish to do binding time analysis with one of the components known and the rest being unknown. In this case the `solve` function would never be unfolded because each call to it would have all its arguments mapped to `DYNAMIC`. To overcome this problem, each call to `solve` is made to a different version. This entails duplication of code, but can be done in a straightforward manner. The function applied to the `i`th component will be named `solvei`.

The result of specializing `dfa` with `scc2` and `env0` as `STATIC` is shown below. The value used for `scc2` is the graph shown in Figure 2 and the value used for `env0` is the empty environment. Note that `solve2` has been completely unrolled. The static part of the result is completely computed. The dynamic environment (flowing in from the result of the first component) is processed before being combined with the static environment to yield the environment to be passed to the next component in the topological order:

```
(lambda (scc1 scc3)
  (solve3 scc3 (map31 (car scc3))
    (env-union2
      (next-approx scc2
        (next-approx scc2
          (next-approx scc2
            (solve1 scc1 (map11 (car scc1))
                    '((l0) (l1) (l2) (l3))))))
    '((l0 (x . e1) (y . e2))
      (l1 (x . e1) (y . e2) (x . b) (y . c))
      (l2 (y . e2) (y . c) (x . b))
      (l3 (x . e1) (x . b) (y . c))))))
```

The residual function corresponding to `scc1` and `env0` being `STATIC` is shown below: (Note that in this case `solve1` is completely unfolded since all its arguments were static, the values used are the same as in the last case.)

```
(lambda (scc2 scc3)
  (solve3 scc3 (map31 (car scc3))
          (solve2 scc2 (map21 (car scc2))
                  '((l0 (x . e1) (y . e2))
                    (l1 (x . e1) (y . e2) (x . b) (y . c))
                    (l2 (y . e2) (y . c) (x . b))
                    (l3 (x . e1) (x . b) (y . c))))))
```

[1]A binding time signature is a description of the binding times for the arguments and the result of a function

During the **Setup** phase of the incremental interpreter, residual functions (including the ones shown above) are computed corresponding to each element of the partition. The **Reestablish** phase simply entails recomputing those residual functions affected by changes in the input.

5 Combining Residual Functions

The last step of the incremental interpreter is the **Combine** step which computes the least upper bound of two residual functions r_p and r_q, namely $r_{p \sqcup q}$. This section will describe the implementation of an algorithm to combine residual functions r_p and r_q to produce their least upper bound. The algorithm avoids redoing any of the reductions already done in the computation of either r_p and r_q. Indeed if the incremental interpreter is to achieve good performance, this is essential. The algorithm is driven by actions, an interpretation of the binding time information, computed for the two residual functions. Schism computes actions for every node of the source program from the binding time information. It produces four kinds of actions [CD90]:

- **Reduce:** An action which says process the children of the syntax tree according to the action subtrees rooted at this node and then **Reduce** the node.
- **Rebuild:** An action which says process the children of the syntax tree according to the action subtrees rooted at this node and then **Rebuild** the node.
- **Eval:** Evaluate (using the standard semantics of the language) the subtree rooted at this node.
- **Id:** Freeze the subtree rooted at this node.

The algorithm constructs $r_{p \sqcup q}$ from subexpressions of r_p and r_q. Which subexpression is chosen depends on the binding times. The algorithm follows the general rule: "choose the subexpression which is more evaluated". The function `lub` takes as arguments the bodies of the two residual functions: `r1` and `r2`, the source code corresponding to the two bodies: `c1` and `c2` (which should be the same always, and so could be replaced by a single argument), the action trees which produced the two residual functions: `at1` and `at2`. In addition two kinds of environments are used, whose purpose will soon become clear. `ana1` and `ana2` are the results of binding time analysis for the two residual functions. The algorithm operates on a first order subset of Scheme.

The entire code for the algorithm is presented in Appendix A. In this section we will examine some salient features of the algorithm. We use `fst`, `snd`, `trd`, `frt` to mean the obvious list selector functions. The algorithm works by comparing the two residual functions and their action trees. The three main cases to consider are the node reduced being an application, a variable and a conditional. In the case of the node under consideration being an application, the cases of interest are: both residual functions have reduced the application, one has reduced it while the other rebuilt it, and both have rebuilt it. Of these three cases consider the case when one application has been reduced and the other rebuilt. The algorithm is recursively called on the body of the function. In addition the environment of the rebuilt application must be enhanced with the residual expressions corresponding to its arguments. This is essential as the arguments of the rebuilt may incorporate reductions not in the arguments of the reduced function. The code for this case is shown below:

```
((and (reduce? at1) (rebuild? at2))
 (let ((v1 (sel ana1 (fn-name c1)))
       (v2 (sel ana2 (fn-name c2))))

 (inc:lub r1 (extract-code v1) (extract-at v1)
      (update-env env1 (arg-names v1)
                       (dup (inc:length (arg-names v1)) '?)
                       (arg-bodies c1)
                       (arg-ats at1))
      cenv1 ana1
      (extract-code v2) (extract-code v2)
      (extract-at v2)
      (update-env env2 (arg-names v2)
                       (arg-bodies r2)
                       (arg-bodies c2)
                       (arg-ats at2))
      cenv2 ana2)))
```

When the node under consideration is a variable, a case analysis needs to be done on the binding times. We will look at one of the branches of the case: namely when one variable is reduced and the other rebuilt. `lub` is recursively called with the reduced variable expression and the lookup of the variable in the environment of the rebuilt variable. New action trees and environments are also obtained from each environment. The code for this case is shown below:

```
((and (reduce? at1) (rebuild? at2))
 (let ((v1 (inc:lookup-env env1 (var-name c1)))
       (v2 (inc:lookup-env env2 (var-name c2))))
      (inc:lub r1       (snd v1) (trd v1) (frt v1) cenv1 ana1
               (fst v2) (snd v2) (trd v2) (frt v2) cenv2 ana2)))
```

The case of the conditional explains the use of the two environments `cenv1` and `cenv2`. When faced with a reduced conditional, the binding time information does not tell us which expression – consequent or alternate – was chosen. This information is needed for the recursive call to `lub` to be made. We assume that the specializer records a trace of how each conditional in the program was resolved. This information is presented in the form of an environment mapping program points to traces. In the following case branch, we consider the possibility of one conditional being reduced and the other being left residual. The `lub` algorithm is called recursively with arguments depending on which way the conditional was reduced (as shown below).

```
((and (rebuild? at1) (reduce? at2))
 (let ((v (lookup-cenv cenv2 (cond-name c2))))
      (inc:lub (choose-b (car v) r1) (choose-b (car v) c1)
           (choose-a (car v) at1) env1 cenv1 ana1
           r2 (choose-b (car v) c2)
           (choose-a (car v) at2) env2 (cdr v) ana2)))
```

6 The Incremental Interpreter

Shown below is an *incremental interpreter*, i.e. it takes a specification of an incremental program (a non-incremental program and a partition of the input), an initial argument and a series of small changes to the input and "incrementally" produces a series of answers. The function `setup` computes the residual functions correponding to `prog` on each element of the partition. `reestablish` takes a set of residual functions, a small change to the argument of the program and recomputes only those residuals affected by the change. These functions are built out of the components discussed in the previous sections.

```
(define (Inc prog part init_arg deltas)
  (I (setup prog part init_arg) deltas))

(define (I residuals deltas)
  (cond
    ((null? deltas)
     (cons (eval (ast->t (combine residuals)) (repl-env)) '()))
    (else
      (let* ((delta (car deltas)))
            (cond
              (let ((residuals' (reestablish residuals delta)))
                   (cons (eval (ast->t (combine residuals')) (repl-env))
                         (I residuals' (cdr deltas)))))))))
```

7 Compiled Incremental Programs

That the use of an interpreter entails a performance overhead is well known. Partial evaluation has been used to capture the essence of compilation by specializing an interpreter wrt a program yielding a compiled program. We can use this idea profitably to specialize the incremental interpreter (`Inc`)

wrt a incremental program specification (**prog** and **part**). This yields a "compiled incremental program" which can yield substantial performance benefits over the interpreted version. We have actually carried out such a specialization for many problems (Section 8). This can be described precisely as follows: (f is the function to be incrementalized, P is the input partition, $\mathcal{I}$ is the incremental interpreter, f_{inc} is the "compiled" incremental program and $[\![f]\!]$ denotes the function corresponding to the program f)

$$[\![\mathcal{PE}]\!]\ \mathcal{I}\ \langle f, P\rangle = f_{inc}$$

Since the incremental interpreter itself makes liberal use of partial evaluation (in the **Setup** and **Reestablish** phases), the abovementioned specialization entails self application of the partial evaluator. How exactly is increased performance achieved by this specialization? First, binding time analysis of **prog** on each element of **part** can be carried out at specialization time. Also, the specializer can be specialized with each result of the binding time analyses to yield specialized specializers (the second Futamura projection [Fut71]). The **Combine** phase does not benefit from this specialization because the conditional environment arguments to the `lub` function (`cenv1` and `cenv2`) are both `DYNAMIC`. Thus the `lub` function has to be left residual.

8 Performance

In this section we describe the performance results of two problems: the data flow analysis problem discussed above and an evaluator for non-circular attribute grammars. The evaluator reattributes the attribute tree after a subtree replacement. For more details of the incremental program for the attribute grammar example, see [SH91]. Other examples we have worked on include constraint solving, strictness analysis and type inference [Sun91].

The implementation runs under the T system (a dialect of Scheme) using the partial evaluator Schism. In each case we will compare the performance of the "compiled incremental program" (as described in the last section) with the non-incremental (batch) version. The choice of input data has been done as follows. It is possible to show very good performance by making the changing element of the partition very small and the rest of the input arbitrarily large. Correspondingly it is possible to make an incremental program perform poorly by making the changes very large, so that the overhead of recomputing the solution becomes comparable to the batch version. To present a more accurate picture, we have chosen input data where the elements of the partition are of equal size. Thus if there are n elements in the partition, one would expect the incremental version to show an n-fold speedup over the batch version for changes which only affect one element of the partition. In practice we observe that the overhead associated with the methodology gives us speedups less than n (because of the overheads associated with the **Combine** step).

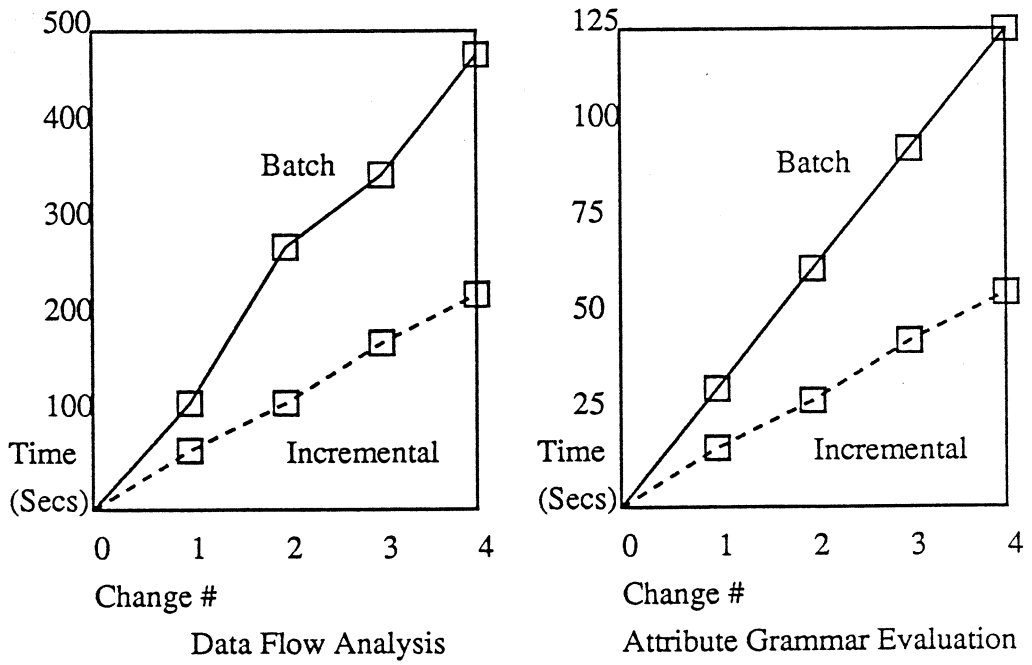
Shown in Figures 3 and 4 are the performance figures for the two examples. The tables show the costs of recomputing the result in a non-incremental (batch) mode and using the incremental program. Each row of the tables represents a small change to the input in the previous row. The graphs in Figure 8 plot the data with the x-axis representing consecutive changes to the input and the y-axis representing cost of recomputation. The cost of the **Combine** step grows slower than the cost of the **Reestablish** step. Thus as the input partition size gets larger (more components in the data flow analysis example), the speedup will also increase.

Batch	*Reestablish*	*Combine*	*Total*	*Speedup*
115.32	31.24	26.39	57.63	2.00
116.97	29.26	24.48	53.74	2.18
118.35	29.57	24.90	54.47	2.17
117.26	27.74	26.30	54.04	2.17

Figure 3: Data Flow Analysis: Reaching Definitions. A 120 node graph with 3 connected components of approximately equal size.

Batch	*Reestablish*	*Combine*	*Total*	*Speedup*
31.63	6.03	8.71	14.74	2.15
30.43	6.29	7.51	13.80	2.20
30.17	5.92	8.20	14.12	2.14
32.79	4.85	7.53	12.38	2.65

Figure 4: Attribute Evaluation: $L \rightarrow L\ L$, $L \rightarrow a$. A 100 node tree with changes made to a subtree of approximately half the size.



9 Conclusions and Future Work

The performance results and the relative ease of construction suggests that partial evaluation can be a useful tool to build incremental programs. An important reason for the good performance of the programs is the *self-applicable* nature of the partial evaluator. Given that most successfully self-applicable partial evaluators have been based on *binding-time analysis* [JSS89], we can infer that binding time analysis is crucial to the approach. This can also be seen from the fact that the least upper bound algorithm is driven by an interpretation of the binding time information of the two residual functions. The experience gained during implementation suggests some avenues to improve the framework:

Storage Costs of Residual Functions. The intermediate results maintained by the interpreter are residual functions. As the size of the partition grows, the number of residual functions also grows. This may lead to excessive storage consumption. However, note that usually large portions of any two residual functions are usually the same. This is because portions of the computation not affected by either member of the partition are essentially the same. This observation can be exploited to come up with an efficient storage scheme for residual functions.

Restrictions on the use of Residual Functions. The framework does not encourage reuse of residual functions in the following sense. If two data flow graphs share a strongly connected component, it should be possible to use the residual function from one incremental session in the other session. But this may not be possible since the two strongly connected components, although they are the same, may occupy different positions in the list of strongly connected components. We are investigating ways of overcoming such restrictions.

Higher order functions. The system as currently implemented only handles first order functions. Schism computes an extended set of actions for higher order functions. We are currently reformulating the `lub` algorithm to make use of these actions.

Acknowledgements Thanks to Paul Hudak for many discussions and to Charles Consel for making Schism available and answering innumerable questions.

References

[BEN88] D. Bjørner, A.P. Ershov, and N.D.Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[CD90] C. Consel and O. Danvy. From interpreting to compiling binding times. In *Proceedings of the 3rd European Symposium on Programming, Lecture Notes in Computer Science, Vol. 432.* Springer-Verlag, May 1990.

[Con88] C. Consel. New insights into partial evaluation: the schism experiment. In *ESOP'88, 2nd European Symposium on Programming, Nancy, France*, volume 300 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1988.

[Con90a] C. Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Languages*, June 1990.

[Con90b] C. Consel. *The Schism Manual.* Yale University, Department of Computer Science, November 1990.

[FT90] J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

[Fut71] Y. Futamura. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5), 1971.

[JSS89] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1), 1989.

[Lau88] J. Launchbury. Projections for specialisation. In A.P.Ershov D.Bjørner and N.D.Jones, editors, *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[Lau90] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, University of Glasgow, January 1990.

[MR90] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.

[Pug88] W. W. Pugh, Jr. *Incremental Computation and the Incremental Evaluation of Functional Programs.* PhD thesis, Cornell University, August 1988.

[SH91] R. S. Sundaresh and Paul Hudak. Incremental computation via partial evaluation. In *Conference Record of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1991.

[Sun91] R. S. Sundaresh. *Incremental Computation and Partial Evaluation.* PhD thesis, Yale University, (Forthcoming) 1991.

[YS89] D. Yellin and R. Strom. INC: A language for incremental computation. Technical report, IBM, RC 14375(#64375) 1989.

A Algorithm LUB

```
(define (inc:lub r1 c1 at1 env1 cenv1 ana1 r2 c2 at2 env2 cenv2 ana2)
  (cond
    ((inc:application? c1)
     (cond
       ((eval? at1) r1)
       ((eval? at2) r2)
       ((id? at1) r2)
       ((id? at2) r1)

       ((and (reduce? at1) (reduce? at2))
        (let ((v1 (sel ana1 (fn-name c1)))
              (v2 (sel ana2 (fn-name c2))))

        (inc:lub r1 (extract-code v1) (extract-at v1)
             (update-env env1 (arg-names v1)
                              (dup (inc:length (arg-names v1)) '?)
                              (arg-bodies c1)
                              (arg-ats at1))
             cenv1 ana1
             r2 (extract-code v2) (extract-at v2)
             (update-env env2 (arg-names v2)
                              (dup (inc:length (arg-names v2)) '?)
                              (arg-bodies c2)
                              (arg-ats at2))
             cenv2 ana2)))
       ((and (reduce? at1) (rebuild? at2))
        (let ((v1 (sel ana1 (fn-name c1)))
              (v2 (sel ana2 (fn-name c2))))

        (inc:lub r1 (extract-code v1) (extract-at v1)
             (update-env env1 (arg-names v1)
                              (dup (inc:length (arg-names v1)) '?)
                              (arg-bodies c1)
                              (arg-ats at1))
             cenv1 ana1
             (extract-code v2) (extract-code v2)
             (extract-at v2)
             (update-env env2 (arg-names v2)
                              (arg-bodies r2)
                              (arg-bodies c2)
                              (arg-ats at2))
             cenv2 ana2)))

       ((and (rebuild? at1) (reduce? at2))
        (... ANALOGOUS TO THE LAST CASE ...))

       ((and (rebuild? at1) (rebuild? at2))
        (mk-application (fn-name c1)
                        (map6 (lambda (x y z u v w)
                                (inc:lub x y z env1 cenv1 ana1 u v w env2 cenv2 ana2))
                              (arg-bodies r1) (arg-bodies c1) (arg-ats at1)
                              (arg-bodies r2) (arg-bodies c2) (arg-ats at2))))))
```

```
((inc:variable? c1)
 (cond
   ((eval? at1) r1)
   ((eval? at2) r2)

   ((and (rebuild? at1) (fail? (inc:lookup-env env1 (var-name c1))))
    r2)

   ((and (rebuild? at2) (fail? (inc:lookup-env env2 (var-name c2))))
    r1)

   ((and (reduce? at1) (reduce? at2))
    (let ((v1 (inc:lookup-env env1 (var-name c1)))
          (v2 (inc:lookup-env env2 (var-name c2))))
    (inc:lub r1 (snd v1) (trd v1) (frt v1) cenv1 ana1
         r2 (snd v2) (trd v2) (frt v2) cenv2 ana2)
    ))

   ((and (rebuild? at1) (rebuild? at2))
    (let ((v1 (inc:lookup-env env1 (var-name c1)))
          (v2 (inc:lookup-env env2 (var-name c2))))
         (inc:lub (fst v1) (snd v1) (trd v1) (frt v1) cenv1 ana1
              (fst v2) (snd v2) (trd v2) (frt v2) cenv2 ana2)))

   ((and (reduce? at1) (rebuild? at2))
    (let ((v1 (inc:lookup-env env1 (var-name c1)))
          (v2 (inc:lookup-env env2 (var-name c2))))
         (inc:lub r1       (snd v1) (trd v1) (frt v1) cenv1 ana1
              (fst v2) (snd v2) (trd v2) (frt v2) cenv2 ana2)))

   ((and (rebuild? at1) (reduce? at2))
    (... ANALOGOUS TO THE LAST CASE ...))))

((inc:conditional? c1)
 (cond
   ((eval? at1) r1)
   ((eval? at2) r2)
   ((id? at1) r2)
   ((id? at2) r1)

   ((and (rebuild? at1) (rebuild? at2))
    (mk-if
      (inc:lub (test-body r1) (test-body c1) (test-at at1) env1 cenv1
               ana1
               (test-body r2) (test-body c2) (test-at at2) env2 cenv2
                ana2)
      (inc:lub (cons-body r1) (cons-body c1) (cons-at at1) env1 cenv1
               ana1
               (cons-body r2) (cons-body c2) (cons-at at2) env2 cenv2
               ana2)
      (inc:lub (alt-body r1) (alt-body c1) (alt-at at1) env1 cenv1 ana1
               (alt-body r2) (alt-body c2) (alt-at at2) env2 cenv2 ana2
   )))
```

```
((and (rebuild? at1) (reduce? at2))
 (let ((v (lookup-cenv cenv2 (cond-name c2))))
      (inc:lub (choose-b (car v) r1) (choose-b (car v) c1)
           (choose-a (car v) at1) env1 cenv1 ana1
           r2 (choose-b (car v) c2)
           (choose-a (car v) at2) env2 (cdr v) ana2)))

((and (reduce? at1) (rebuild? at2))
 (... ANALOGOUS TO THE LAST CASE ...))

((and (reduce? at1) (reduce? at2))
 (let  ((v1 (lookup-cenv cenv1 (cond-name c1)))
        (v2 (lookup-cenv cenv2 (cond-name c2))))
       (inc:lub r1 (choose-b (car v1) c1) (choose-a (car v1) at1)
            env1 (cdr v1) ana1
            r2 (choose-b (car v2) c2) (choose-a (car v2) at2)
            env2 (cdr v2) ana2)))))))
```