Incremental Computation via Partial Evaluation

R.S. Sundaresh and Paul Hudak
Research Report YALEU/DCS/RR-770
Revised November, 1990

This work is supported by the National Science Foundation CCR-8809919.

Incremental Computation via Partial Evaluation

R.S. Sundaresh* Paul Hudak†

Yale University
Department of Computer Science
Box 2158 Yale Station
New Haven, CT 06520
{sundaresh,hudak}@cs.yale.edu

1 Introduction

It is a common occurrence in a programming environment to apply a software tool to a series of similar inputs. Examples include compilers, interpreters, text formatters, etc., whose inputs are usually incrementally modifed text files. Thus programming environment researchers have recognized the importance of building *incremental* versions of these tools — i.e. ones which can efficiently update the result of a computation when the input changes only slightly.

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 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.

Thus there is an increasing need for a *framework* for incremental computation which will help us understand existing incremental algorithms and facilitate (if not automate) the construction of new ones. We have developed such a framework based on the recently popular notion of *partial evaluation*. Besides 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 domain. (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.)
- An algebraic basis for *reasoning* about the correctness of the incremental programs thus generated. (The framework relies partially on the notion of a *Brouwerian algebra*.)
- A comparative basis for better understanding existing incremental algorithms. In particular, we have re-cast many existing incremental programs into our framework.
- To overcome the overhead of "incremental interpretation," a method to generate "compiled" incremental programs using *Futamura projections* is described.

2 Incremental Computation and Partial Evaluation

To understand our framework one must first have a good understanding of *partial evaluation*, and thus we begin with some basic definitions (for a good survey of the field see [JSS89]).

2.1 Partial Evaluation

Partial evaluation is a program transformation technique for specializing a function with respect to some known (i.e. "static") part of its input. The result is called a *residual function*, and has the property that when applied to the remaining part of the input, will yield the desired result. Following Launchbury [Lau88], we give a precise 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:*

*Supported in part by an IBM graduate fellowship and DARPA grant N00014-88-K-0573.

†Supported in part by DARPA grant N00014-88-K-0573.

- $p \sqsubseteq ID$ *(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). We will tighten this definition in Section 3 to achieve uniqueness by choosing the "least" of these projections. Indeed, a major goal of that section is to define domains of projections where such a construction always exists. We write $\bar{p}$ to denote the unique (to be defined later) 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.[1]

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 $r_{ID} = apply\ f\ a$.

Although the partial evaluator really takes *representations* of its arguments and not actual *values*, hereafter we will treat it as taking values as arguments (primarily to avoid having to propagate *apply* everywhere). On the other hand, the algorithm for "combining" residual functions in Section 5 depends crucially on manipulating the representations.

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.

[1] This can be seen extensionally as a restatement of Kleene's S_n^m theorem from recursive function theory.

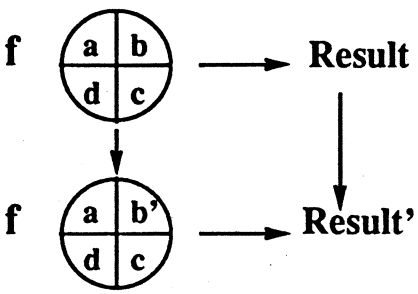

Figure 1: Incremental computation

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 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 $r_{ID} = apply\ f\ a$, 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:

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

The above forms the basis for our approach. But we have so far made many assumptions that require fleshing out. In particular:

1. What is the basis for choosing a good partition of the input domain? (If it is too coarse, even small changes will trigger massive recomputation; if too fine, the stored residual functions will each capture very little computation and excessive work will be done in the combining phase.)
2. How do we combine residual functions to get "larger" ones? (Does the construction even exist? If so, is it unique? Can it be done efficiently?)
3. How does one determine which residual functions need to be recomputed? (I.e., how does one determine the set of projections that "see" the changes to the input?)

These and other technical questions are answered in the next 3 sections. For examples of the *application* of our framework, the reader may wish to jump to Section 6 and return to the technical sections after having developed more intuition for the methodology.

3 Projection Algebras

The most critical aspect of our methodology is the ability to combine residual functions — correctly and efficiently. We first deal with correctness, which requires the construction of domains in which suitable combining operators are well-defined.

We begin with a set of domains and domain formers that are adequate in capturing most of the domains found in conventional programming languages.

t ::=	1, Nat,...	base domains
|	τ_i	type parameter
|	$t_1 + t_2$	separated sum
|	$t_1 \times t_2$	non-strict product
|	$\mu\tau_1.\ t$	recursive domain

The structure described above can be easily generalized to more than one type parameter. In what follows we assume that standard domains such as lists, pairs, and natural numbers have been pre-defined, and we use conventional notation (1, 2, Nil, $Cons$, etc.) when referring to elements of the domains. For example, polymorphic lists and pairs can be defined by:

$$List(\tau_1) = \mu\tau_2.\ 1 + (\tau_1 \times \tau_2)$$
$$Pair(\tau_1, \tau_2) = \tau_1 \times \tau_2$$

3.1 Projection Domains

Consider a domain of projections under the standard information ordering of functions — this domain is not closed with respect to greatest lower bound. This is because $p_1 \sqcap p_2 = \lambda x.\ (p_1 x) \sqcap (p_2 x)$ is not necessarily idempotent, and therefore may not be a projection. A simple example should convince the reader of this:

Example 3.1 *Consider projections on the domain of pairs of natural numbers. Let p_1 and p_2 be defined as follows:*

$$p_1\ (x,y) = \textit{if } x = 2 \textit{ then } (\bot, y) \textit{ else } (x,y)$$
$$p_2\ (x,y) = \textit{if } x = \bot \textit{ then } (x, \bot) \textit{ else } (x,y)$$

Let p be $\lambda x.(p_1 x) \sqcap (p_2 x)$. It is easy to verify that $p\ (2,3) = (\bot, 3)$ which is not the same as $p \circ p\ (2,3) = (\bot, \bot)$. Thus p is not idempotent, and is therefore not a projection.

This problem arises because the domain of all projections is too large. We are interested (for purposes of partial evaluation) in projections which only depend on the *structure* of the object they are manipulating and not on the *values* of its components. Launchbury [Lau88] describes a smaller (finite) domain of projections, but his domain does not serve our purpose because it does not contain useful projections such as the following one on the domain of lists:

$$p\ Nil = Nil$$
$$p\ (Cons\ x\ xs) = Cons\ \bot\ xs$$

Here is a first attempt at constructing a domain of projections small enough to possess properties of interest to us yet large enough to contain examples such as the above.

Definition 3.1 *A* polymorphic projection *on a domain $F(\tau)$ is a collection of instances $f_A : F(A) \to F(A)$, such that for any strict function $\alpha : A \to B$, the diagram in Figure 2 commutes; i.e. $f_B \circ mapF(\alpha) = mapF(\alpha) \circ f_A$. By $mapF$ we mean the appropriate map function for the datatype F.*

Example 3.2 *Define two projections over the domain of pairs: $p\ (\bot, b) = (\bot, \bot)$, $p\ (a,b) = (a,b)$; $LEFT\ (x,y) = (x, \bot)$. p is not polymorphic , but $LEFT$ is.*

Unfortunately, the domain of polymorphic projections does not exactly capture the intuition of depending only on the structure of the input. Consider g: $g\ Nil = Nil$, $g\ (x : \bot) = \bot : \bot$, $g\ x = x$. g is a polymorphic projection but still depends on the values of subcomponents of its input (in this case $\bot$ as the tail). To restrict away projections like g, we define a domain of projections as follows:

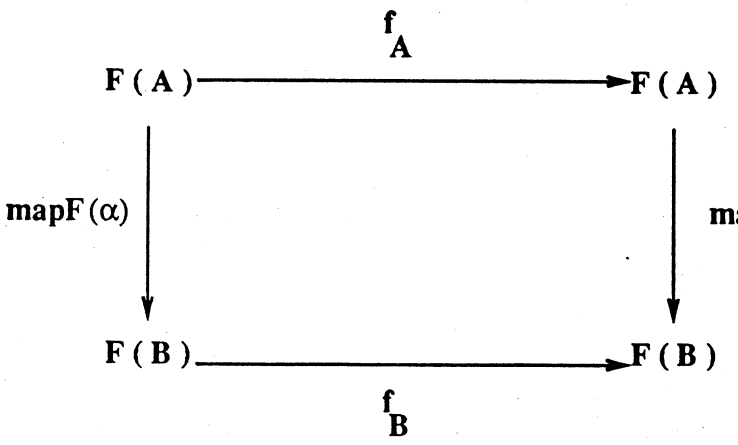

Figure 2: Polymorphic projections

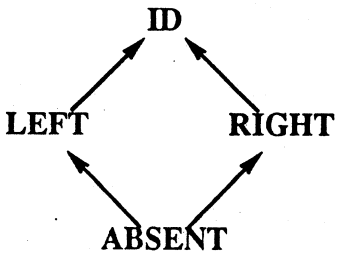

Figure 3: Projections on the domain of pairs

$$
\begin{array}{rcl}
\mathcal{P}\ (\mathrm{d}) & = & \{\ ID_d,\ ABSENT_d\ \}(\mathrm{d\ is\ a\ base\ domain}) \\
\mathcal{P}\ (\tau) & = & \{\ ID_\tau,\ ABSENT_\tau\ \} \\
\mathcal{P}\ (d_1 + d_2) & = & \{\ p_1 + p_2 \mid p_1 \in \mathcal{P}(d_1),\ p_2 \in \mathcal{P}(d_2)\ \} \\
\mathcal{P}\ (d_1 \times d_2) & = & \{\ p_1 \times p_2 \mid p_1 \in \mathcal{P}(d_1),\ p_2 \in \mathcal{P}(d_2)\ \} \\
\mathcal{P}\ (\mu\tau.T(\tau)) & = & \mathcal{P}(T(\mu\tau.T(\tau))) \cup \{\ ABSENT_{\mu\tau.T(\tau)}\ \}
\end{array}
$$

The subscript to ID or $ABSENT$ refers to the domain of definition. $+$ and $\times$ are defined for functions as follows: $(f \times g)(a,b) = (fa, gb)$ and $(f+g)\bot = \bot$, $(f+g)(inl\ a) = inl(f\ a)$, $(f+g)(inr\ b) = inr(g\ b)$. Note the case of the recursive domain where the $ABSENT$ projection can be invoked on any tail of the list. We use $\mathcal{D} \xrightarrow{proj} \mathcal{D}$ to denote the domain defined above, under the standard information ordering. Note that any element of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$ is guaranteed to be polymorphic.

Example 3.3 *Consider projections on the domain of pairs. $Pair(\tau_1,\tau_2) \xrightarrow{proj} Pair(\tau_1,\tau_2)$ is shown in Figure 3, where $LEFT(x,y) = (x,\bot)$ and $RIGHT(x,y) = (\bot,y)$.*

3.2 Properties of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$

Definition 3.2 *A* commutative *domain is one whose elements commute wrt function composition.*

Lemma 3.1 *$\mathcal{D} \xrightarrow{proj} \mathcal{D}$ is a commutative domain.*
Proof: The proof is by induction on the structure of domains. The base domains are commutative (because the only projections defined on them are ID and ABSENT). Assume that $d_1 \xrightarrow{proj} d_1$ and $d_2 \xrightarrow{proj} d_2$ are commutative domains, $p_1, p_3 \in d_1 \xrightarrow{proj} d_1$, $p_2, p_4 \in d_2 \xrightarrow{proj} d_2$.

- $(p_1 \times p_2) \circ (p_3 \times p_4) = (p_1 \circ p_3) \times (p_2 \circ p_4) = (p_3 \circ p_1) \times (p_4 \circ p_2) = (p_3 \times p_4) \circ (p_1 \times p2)$. Thus $d_1 \times d_2 \xrightarrow{proj} d_1 \times d_2$ is commutative.
- $(p_1 + p_2) \circ (p_3 + p_4) = (p_1 \circ p_3) + (p_2 \circ p_4) = (p_3 \circ p_1) + (p_4 \circ p_2) = (p_3 + p_4) \circ (p_1 + p2)$. Thus $d_1 + d_2 \xrightarrow{proj} d_1 + d_2$ is commutative.

In the case of the recursive domain, if one of the projections chosen is ABSENT commutativity holds since $p \circ q$ is $ABSENT$ if either p or q is $ABSENT$. If both of the projections are in $\mathcal{P}(T(\tau))$, then whether or not the projections commute depend on whether the elements of $\mathcal{P}(\mu\tau.T(\tau))$ chosen in place of τ commute. This follows from the fact that since T is constructed from $+$ and $\times$, $T(\tau)$ commutes whenever τ does. Repeating the earlier argument, if $ABSENT$ is chosen for either of the projections, we are done. By repeating this argument, as long as $ABSENT$ is chosen at some level of recursion, the projections will commute. The other case is when $ABSENT$ is never chosen. This represents the single projection $\mu p.T(p)$. In this case since both the projections have to be the same, they commute. □

We prove the following properties (Lemmas 3.2 through 3.4) for any p, q from a commutative projection domain.

Lemma 3.2 *$p \circ q$ is a projection, if p and q are members of a commutative domain.*
Proof: No information addition: $p \sqsubseteq ID$ and $q \sqsubseteq ID \Rightarrow p \circ q \sqsubseteq ID$. Idempotence: $(p \circ q) \circ (p \circ q) = p \circ (q \circ p) \circ q = p \circ (p \circ q) \circ q = (p \circ p) \circ (q \circ q) = p \circ q$. □

Lemma 3.3 *In a commutative projection domain, the greatest lower bound (glb) of p and q exists and is $p \circ q$.*
Proof: Note that since $p \sqsubseteq ID$ and $q \sqsubseteq ID$, $p \circ q \sqsubseteq p$ and $p \circ q \sqsubseteq q$. Let r be any projection such that $r \sqsubseteq p$ and $r \sqsubseteq q$. Then by montonicity, $r \circ r \sqsubseteq p \circ q$ and by the definition of projection, $r \sqsubseteq p \circ q$. □

Lemma 3.4 *In a commutative projection domain, the least upper bound (lub) exists.*
Proof: (By contradiction.) Suppose that there exists p and q such that $p \sqcup q$ does not exist. Since ID is the top element, this implies that there exist two incomparable upper bounds l_1 and l_2, and that there is no upper bound less than both l_1 and l_2. (This is because there are no infinite descending chains in $\mathcal{D} \xrightarrow{proj} \mathcal{D}$.) But since glbs exist, $l_1 \circ l_2$ is an upper bound of p and q which is (by definition of glb) less than both l_1 and l_2. Contradiction. □

Commutative domains with the additional property of distributivity are of special interest.

Definition 3.3 *A domain is said to be* distributive *iff for all elements p, q, r of the domain, $p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r)$ and $p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r)$.*

Lemma 3.5 *$\mathcal{D} \xrightarrow{proj} \mathcal{D}$ is a distributive domain.*
Proof: The proof is by induction on the domain structure. We detail the proof of $p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r)$; the proof of $p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r)$ is similar. The distributivity properties of the base domains are easy to verify. Assume that $d_1 \xrightarrow{proj} d_1$ and $d_2 \xrightarrow{proj} d_2$ are distributive, and $p_1, q_1, r_1 \in d_1$ and $p_2, q_2, r_2 \in d_2$. In what follows we write (f, g) for $f \times g$.

- $(p_1, p_2) \sqcup ((q_1, q2) \sqcap (r_1, r_2)) = (p_1, p_2) \sqcup (q_1 \sqcup r_1, q_2 \sqcup r_2) = (p_1 \sqcup (q_1 \circ r_1), p_2 \sqcup (q_2 \circ r_2)) = ((p_1 \sqcup q_1) \sqcap (p_1 \sqcup r_1), (p_2 \sqcup q_2) \sqcap (p_2 \sqcup r_2)) = (p_1 \sqcup q_1, p_2 \sqcup q_2) \sqcap (p_1 \sqcup r_1, p_2 \sqcup r_2) = ((p_1, p2) \sqcup (q_1, q_2)) \sqcap ((p_1, p_2) \sqcup (r_1, r_2))$. Thus $d_1 \times d_2 \xrightarrow{proj} d_1 \times d_2$ is distributive.
- The proof for $d_1 + d_2$ works similarly.
- In the case of the recursive domain construction, first note that if any of p, q or r is chosen as ABSENT, the distributivity property holds. If none of them are $ABSENT$, then each of them must be in $\mathcal{P}(T(r))$ where r is in $\mathcal{P}(\mu r.T(r))$. Now by the above arguments, we know that $T(r)$ commutes whenever r does. Thus if any of the rs are chosen as $ABSENT$ then we are done. We can repeat this argument to show that as long as $ABSENT$ is chosen at some level of recursion, commutativity holds. The other case (when $ABSENT$ is never chosen) represents the single projection $\mu p.T(p)$. In this case since the three projections are the same, distributivity holds.

□

Lemma 3.6 *For a commutative distributive domain, there exists a least r such that $p \sqsubseteq q \sqcup r$.*
Proof: Clearly there is always at least one r which satisfies the definition (take $r = ID$). Assume we have two incomparable elements r_1 and r_2 such that $p \sqsubseteq q \sqcup r_1$ and $p \sqsubseteq q \sqcup r_2$. Also assume that there is no element smaller than both r_1 and r_2 satisfying the difference condition. Then $p \sqsubseteq q \sqcup r_1$ and $p \sqsubseteq q \sqcup r_2$, implies (from definition of glb) $p \sqsubseteq (q \sqcup r_1) \sqcap (q \sqcup r_2)$ and by distributivity: $p \sqsubseteq q \sqcup (r_1 \sqcap r_2)$. But $r_1 \sqcap r_2$ is less than r_1 and r_2 and it satisfys the difference equation. Contradiction. (In case there are an infinite number of r_i satisfying the above condition, the existence of the infinite glb needs to be shown. The infinite glb exists because we know that the domain has no infinite decreasing chains (this can be seen from the definition of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$) and r_1, $r_1 \sqcap r_2$, $r_1 \sqcap r_2 \sqcap r_3$,... is a decreasing chain. It is also easy to see that lub distributes over the infinite glb.) □

For domains which are distributive in addition to being commutative, we define the difference operation as follows:

Definition 3.4 *If p and q are elements of a commutative distributive domain, the* difference *of p and q (written $p - q$) is the least r such that $p \sqsubseteq q \sqcup r$.*

Lemma 3.6 ensures that the difference is uniquely and well defined. This leads us, as promised, to a unique definition of complement:

Definition 3.5 *The* complement *$\bar{p}$ of an element p in a commutative distributive domain is $ID - p$.*

3.3 Algebraic Properties of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$

The properties that we have thus far defined were motivated by our application to incremental computation. Interestingly, they form what is known as a *Brouwerian algebra.*

Definition 3.6 *A* Brouwerian algebra *is an algebra $\langle L, \sqcup, \sqcap, \dot{-}, \top \rangle$ where $\langle L, \sqcup, \sqcap \rangle$ is a lattice with greatest element $\top$, L is closed under $\dot{-}$, and $a \dot{-} b \sqsubseteq c$ iff $a \sqsubseteq b \sqcup c$.*

A Brouwerian algebra can be seen as a generalization of a Boolean algebra where the following equation need not hold: $ID - (ID - p) = p$.

Theorem 3.1 *$\langle \mathcal{D} \xrightarrow{proj} \mathcal{D}, \sqcup, \sqcap, -, ID \rangle$ is a Brouwerian algebra.*
Proof: Follows directly from lemmas 3.3 through 3.6. □

Knowing that we are dealing with a Brouwerian algebra allows us to use known properties of such algebras for reasoning about our projection domains. For example, when specifying an incremental program, the partition we are interested in may only be a subset of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$, and thus we may need to extend the domain to make it Brouwerian; it is a known theorem that such "completions" always exist. The following theorem partially addresses this problem, quoted without proof from [MT46]:

Theorem 3.2 *If $\langle \mathcal{L}, \sqcup, \sqcap, -, \top \rangle$ is a Brouwerian algebra, and $\mathcal{M}$ is a finite subset of $\mathcal{L}$ containing n elements, then there exists a subset $\mathcal{L}'$ of $\mathcal{L}$ and an operation $-'$ with the following properties:*

- *$\langle \mathcal{L}', \sqcup, \sqcap, -', \top \rangle$ is a Brouwerian algebra.*
- *$\mathcal{L}'$ contains at most 2^{2^n} elements.*
- *$\mathcal{M}$ is a subset of $\mathcal{L}'$.*
- *If x, y and $x - y$ are in $\mathcal{L}'$, then $x -' y = x - y$.*

One possible $\mathcal{L}'$ is the set of all elements of $\mathcal{L}$ which are expressible as $\sqcup$s and $\sqcap$s of elements of $\mathcal{M}$.

Example 3.4 *Consider projections on the domain of lists of pairs. $\mathcal{D} = List(Pair(a, b))$. A finite subdomain of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$ is shown in Figure 4. LEFT, RIGHT and ABSENT are projections on pairs. Here $\mathcal{L} = \mathcal{D} \xrightarrow{proj} \mathcal{D}$ and $\mathcal{L}'$ and $\mathcal{M}$ are the subdomain shown in the figure. However, if ID were not one of the projections, then the algebra would not be Brouwerian.*

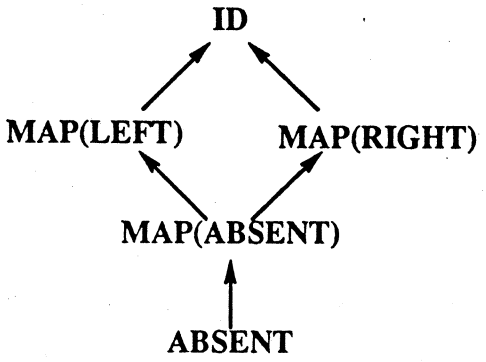

Figure 4: Projections on the domain of lists of pairs

An interesting and much more extensive application of Brouwerian algebras, the modelling of *program integration*, may be found in [Rep90].

4 Residual Function Algebras

We now return to partial evaluation. We demonstrate that the domain of projections in the last section induces an isomorphic domain of residual functions. This will make precise the notion of "combining" residual functions which was alluded to earlier.

Definition 4.1 *The domain $\mathcal{R}$ of residual functions for a function f, its argument a, and a commutative, distributive domain of projections $\mathcal{P}$ is defined as $\mathcal{R} = \{r \mid r = \mathcal{PE}\ f\ p\ a,\ p \in \mathcal{P}\}$. The ordering relation on $\mathcal{R}$ is defined as follows: $r_1 \sqsubseteq r_2$ iff $p \sqsubseteq q$ where $r_1 = \mathcal{PE}\ f\ p\ a$ and $r_2 = \mathcal{PE}\ f\ q\ a$ for some f, p, q and a. We define $\sqcup$, $\sqcap$ and $-$ for the domain $\mathcal{R}$ as follows:*

$$r_p \sqcup r_q \stackrel{def}{=} r_{p \sqcup q},$$
$$r_p \sqcap r_q \stackrel{def}{=} r_{p \sqcap q} \text{ and}$$
$$r_p - r_q \stackrel{def}{=} r_{p-q}$$

It is easy to verify that the ordering on residuals is a partial order. This ordering is intimately related to the standard information ordering. If the type of f is $A \to B$, the type of a residual function for a fixed argument a is $A' \to B$ where A' is the subdomain of A of elements $\sqsubseteq a$. The monotonicity of the $\mathcal{PE}$ implies:

$$p \sqsubseteq q \Rightarrow r_p \sqsubseteq r_q$$

Theorem 4.1 *Given a commutative, distributive projection domain $\mathcal{P}$ and its corresponding domain of residual functions $\mathcal{R}$, $\lambda p.\ \mathcal{PE}\ f\ p\ a$ is a homomorphism from $\langle \mathcal{P}, \sqcup, \sqcap, -, ID \rangle$ to $\langle \mathcal{R}, \sqcup, \sqcap, -, r_{ID} \rangle$.*

Proof: Clearly $\mathcal{PE}$ preserves the identities of $\sqcup$, $\sqcap$ and $-$. It follows directly from definition 4.2 that $\mathcal{PE}$ satisfies:

$$\mathcal{PE}\ f\ (p \sqcup q)\ a = (\mathcal{PE}\ f\ p\ a) \sqcup (\mathcal{PE}\ f\ q\ a)$$
$$\mathcal{PE}\ f\ (p \sqcap q)\ a = (\mathcal{PE}\ f\ p\ a) \sqcap (\mathcal{PE}\ f\ q\ a)$$
$$\mathcal{PE}\ f\ (p - q)\ a = (\mathcal{PE}\ f\ p\ a) - (\mathcal{PE}\ f\ q\ a)$$

□

Corollary 4.1 *$\sqcup$, $\sqcap$ and $-$ on the domain $\mathcal{R}$ are the lub, glb and difference operations, respectively.*

We quote the following theorem from [MT46]:

Theorem 4.2 *Any homomorphic image of a Brouwerian algebra is a Brouwerian algebra.*

Since the domain of residuals is a homomorphic image of the projection domain, we can state that:

Corollary 4.2 *$\langle \mathcal{R}, \sqcup, \sqcap, -, r_{ID} \rangle$ is a Brouwerian algebra.*

5 Algorithm for Least Upper Bound of Residual Functions

The last section described three binary operations on residual functions: $\sqcup$, $\sqcap$ and $-$, but did not describe algorithms for them. If we had an efficient algorithm for $-$, the whole problem of incremental computation would be solved! But this is a difficult operation to compute since it involves "backing up" of computation. Our methods can be seen as trying to approximate $-$ by using the other two operations. In this paper (and in all applications that we have investigated to date) we only use $\sqcup$, and thus in this section we develop an efficient algorithm for it. Since we know that $\mathcal{PE}$ is a homomorphism from the domain of projections to the domain of residual functions, the following equality holds:

$$(\mathcal{PE}\ f\ p\ a) \sqcup (\mathcal{PE}\ f\ q\ a) = \mathcal{PE}\ f\ (p \sqcup q)\ a$$

While this gives us a simple method to compute the lub, it is obviously inefficient, since it ignores the work already done in computing r_p and r_q. A good algorithm will avoid redoing any reductions already done to compute r_p and r_q. Indeed if our incremental interpreter is to achieve good performance, this is essential. In what follows, we assume that the partial evaluator is implemented using *binding time analysis*. This technique has been shown to be crucial in achieving self-application of partial evaluators [JSS89].

Binding time analysis. The algorithm to compute the lub uses *binding time information* of the two residual functions in the form of *action trees* [CD90]. Binding time analysis computes the binding time (static or dynamic) of each expression in the source program given the binding times of the argument. For purposes of partial evaluation it is common to determine the binding time information prior to actual specialization. The source program (a λ-term) is represented as a tree with three kinds of nodes: application, abstraction and variable. Given a source program and a description of the binding time of its input (in the form of a projection), the result of binding time analysis is an *action tree*, isomorphic to the source program, with the following nodes:

- **Reduce:** An action tree 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 tree which says process the children of the syntax tree according to the action subtrees rooted at this node and then **Rebuild** the node.

The specializer simply processes each node of the source program by executing the corresponding action.

Algorithm LUB. We describe the algorithm in the context of the λ-calculus; it can easily be adapted to languages with more syntactic sugar. Also, we assume that the functions have first been α-converted to avoid any name clashes.

A residual function is described by a triple $\langle r, a, e\rangle$ where r is the λ-term representing the residual function, a is the action tree which produced the residual function and e is an associated environment which is initially empty. The environment is meant to map variables to pairs of the form $\langle t, a\rangle$, where t is a λ-term and a is the associated action tree.

Algorithm LUB($\langle r_1, a_1, e_1\rangle$, $\langle r_2, a_2, e_2\rangle$ **)**

- Apply rewrite rules in Figure 5 to $\langle r_1, a_1, e_1\rangle \sqcup \langle r_2, a_2, e_2\rangle$ until all $\sqcup$ symbols are removed from the term.
- Reduce nodes all of whose children are reduced. This is needed to perform reductions not performed by either of the two residual functions but made possible by their combination.

To reduce the number of rules, symmetric cases have been omitted from Figure 5. The rules can be understood if we realize that the motivation is to avoid doing any reduction which has already been done in any one of the other residual functions. For example in rule 1, an application has been reduced in one residual function but has been left residual in another. The rule reduces this to taking the $\sqcup$ of the bodies of the abstractions, but also updates the environment of the second residual function so as not to lose reductions performed in the argument. The other rules are similar. There is a final post-processing step to be executed, whose purpose should be clear from the following example:

```
g (x,y) = if (x == 0) then f (x,y) else f (x,y) - 5
f (x,y) = x*x - y*y
```

Partial evaluating g with the projection *LEFT* and argument (0,2) gives:[2]

```
g1 (x,y) = 0 - y*y
```

Partial evaluating g with the projection *RIGHT* and the same argument (0,2) gives:

[2] Note that since residual functions have the same type as the function being partially evaluated, g1 still takes a pair as argument. It simply ignores the left element of the pair. A similar comment applies to g2.

1. $\langle r_1,$ **Reduce**$(a_{11})(a_{12}),$ $e_1\rangle \sqcup$
   $\langle (\lambda x.t_{21})(t_{22}),$ **Rebuild**$(a_{21})(a_{22}),$ $e_2\rangle$
   $\Longrightarrow \langle r_1,\ a_{11},\ e_1\rangle \sqcup$
   $\langle t_{21},\ a_{21},\ e_2[x/\langle t_{22}, a_{22}\rangle]\rangle$
2. $\langle r_1,$ **Reduce**$(a_{11})(a_{12}),$ $e_1[x/\langle t_1, a_1\rangle]\rangle \sqcup$
   $\langle x,$ **Rebuild**$()(),$ $e_2[x/\langle t_2, a_2\rangle]\rangle$
   $\Longrightarrow \langle r_1, a_1,\ e_1\rangle \sqcup$
   $\langle t_2,\ a_2,\ e_2\rangle$
3. $\langle r_1,$ **Reduce**$(a_{11})(a_{12}),$ $e_1\rangle \sqcup$
   $\langle r_2,$ **Reduce**$(a_{21})(a_{22}),$ $e_2\rangle$
   $\Longrightarrow \langle r_1,\ a_{11},\ e_1\rangle \sqcup$
   $\langle r_2,\ a_{21},\ e_2\rangle$ (If the node reduced was an application)
4. $\langle r_1,$ **Reduce**$(a_{11})(a_{12}),$ $e_1[x/\langle t_1, a_1\rangle]\rangle \sqcup$
   $\langle r_2,$ **Reduce**$(a_{21})(a_{22}),$ $e_2[x/\langle t_2, a_2\rangle]\rangle$
   $\Longrightarrow \langle r_1,\ a_1,\ e_1\rangle \sqcup$
   $\langle r_2,\ a_2,\ e_2\rangle$(If the node reduced was a variable x)
5. $\langle (\lambda x.t_{11})t_{12},$ **Rebuild**$(a_{11})(a_{12}),$ $e_1\rangle \sqcup$
   $\langle (\lambda x.t_{21})t_{22},$ **Rebuild**$(a_{21})(a_{22}),$ $e_2\rangle$
   $\Longrightarrow \lambda x.(\langle t_{11},\ a_{11},\ e_1\rangle \sqcup \langle t_{21},\ a_{21},\ e_2\rangle)$
   $(\langle t_{12},\ a_{12},\ e_1\rangle \sqcup \langle t_{22},\ a_{22},\ e_2\rangle)$
6. $\langle x,$ **Rebuild**$()(),$ $e_1[x/\langle t_1, a_1\rangle]\rangle \sqcup$
   $\langle x,$ **Rebuild**$()(),$ $e_2[x/\langle t_2, a_2\rangle]\rangle \Longrightarrow \langle t_1, a_1, e_1\rangle \sqcup$
   $\langle t_2, a_2, e_2\rangle$
7. $\langle x,$ **Rebuild**$()(),$ $e_1\rangle \sqcup$
   $\langle x,$ **Rebuild**$()(),$ $e_2\rangle \Longrightarrow x$ (if neither e_1 nor e_2 have bindings for x).

Figure 5: Rewrite rules for algorithm **LUB**

```
g2 (x,y) = if (x == 0) then x*x - 4 else x*x - 9
```

The result of using the above rewrite rules is:

```
g12 (x,y) = 0 - 4
```

Clearly this can be further reduced; i.e., there may be reductions in the least upper bound which are neither in g1 nor g2. These can be performed through another phase of partial evaluation.

Before we go on to examine properties of algorithm **LUB**, note that as long as there is still a $\sqcup$ in the term, one of the rules will apply. Termination of the first phase of the algorithm is not difficult to verify: each rule reduces the size of the λ-term under consideration, until the base case is reached. The second phase may not terminate when the source program contains non-terminating computations, but this is common among partial evaluators. First we prove the correctness of algorithm **LUB**. To do so let us first examine some properties of action trees. The domain of actions is a two point domain with **Rebuild** $\sqsubseteq$ **Reduce**. We now define an ordering on action trees. Note that since action trees are isomorphic to the source program, they are isomorphic to one another. Thus if at_1 and at_2 are two isomorphic

action trees for the same program,there is an obvious one-to-one onto mapping from the nodes of at_1 to the nodes of at_2.

Definition 5.1 *An action tree at_1 is $\sqsubseteq$ at_2 iff every action in at_1 is $\sqsubseteq$ its corresponding action in at_2. It is not difficult to see that least upper bounds exist under this ordering. An* enabling transformation *on an action tree is the replacement a* Rebuild *action node whose children are all* Reduce *action nodes by a* Reduce *action node. The* closure *of an action tree is the result of repeatedly applying enabling transformations to it until no opportunities to do so remain. We denote the closure of t by $\mathcal{C}(t)$.*

How is the action tree corresponding to $r_{p \sqcup q}$ related to r_p and r_q? We use the notation at_p to refer to the action tree corresponding to r_p.

Lemma 5.1 $at_{p \sqcup q} = \mathcal{C}(at_p \sqcup at_q)$
Proof: Any expression marked static in either r_p or r_q must be marked static in $at_{p \sqcup q}$. In addition if this makes all the children of a Rebuild node Reduce, then the Rebuild node must also be made into Reduce. □

Lemma 5.2 *Algorithm* **LUB** *outputs a lambda term whose action tree is $\mathcal{C}(at_p \sqcup at_q)$ where r_p and r_q are the inputs to the algorithm.*
Proof: By case analysis of the rewrite rules of the algorithm, it is not difficult to see that the result of the first phase has the following action tree: $at_p \sqcup at_q$. This is because each rule chooses Reduce over Rebuild. The post-processing stage simply applies as many enabling transformations as possible. I.e. it computes the closure of the action tree. □

Theorem 5.1 *Algorithm* **LUB** *correctly computes the least upper bound.*
Proof: From the last two lemmas we note that the output of algorithm **LUB** and $at_{p \sqcup q}$ have the same action tree. Since both are specializations of the same function, they have to be equal. □

Theorem 5.2 *Algorithm* **LUB** *does not re-perform any reduction already performed in the computation of either r_p or r_q.*
Proof: During the application of the rewrite rules, no reductions are done (all of them are done in the post-processing phase). At the end of this phase, the term has the action tree $at_p \sqcup at_q$. This means that all reductions in r_p and r_q are already incorporated. □

6 Applications

In this section we consider two well known problems — *data flow analysis* and *attribute grammar evaluation* — for which we have recast, and implemented, existing algorithms using our framework. Other problems for which we have constructed incremental programs include strictness analysis, Hindley-Milner type inference and solving simple systems of constraints [Sun91].

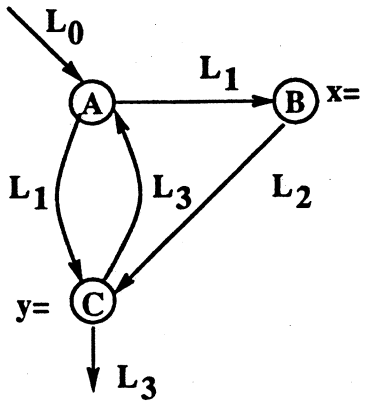

Figure 6: Example flow graph

6.1 Incremental Data Flow Analysis

As an example of compiler data flow analysis, consider the problem of determining the set of "reaching definitions" at every program point. 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 6 (taken from [MR90]). 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 in the obvious manner, 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. (In this and subsequent examples Haskell [HWe90] syntax is used to describe the algorithms.)

We assume the input to the algorithm to be a list eqns of strongly connected components of the set of data flow equations in topological order (which can be produced by a standard dependency analysis). The overall solution is defined by:

```
dfa eqns = foldl fix null-env eqns
```

where `fix` (defined below) is a function which takes an initial environment mapping arc labels to sets, and a set of mutually recursive equations, and computes the least fixpoint of the equations using the initial environment as the first approximation. The fixpoint is found by computing the least upper bound of the ascending Kleene chain; since the domain is finite, termination is guaranteed.

```
fix env eqns =  if env == env' then env
                else fix env' eqns
                where env' = eval eqns env
```

`eval` is a function which recomputes the identifiers defined by the equations `eqns` using the old environment `env` to produce a new environment `env'`.

The next step is to describe a partition of the input. The partition we use is quite simple:

```
p = [ pr i | i <- [1..] ]
where  pr 1  []     = bot
       pr 1 (x:xs) = x : bot
       pr i  []     = bot
       pr i (x:xs) = bot : pr (i-1) xs
```

Note that `pr i`, the ith projection in the partition, discards all information but the ith element of its input list (`bot` is the "unknown" marker for the partial evaluator). Also note that although the partition itself is infinite (the notation `[1..]` denotes the infinite list of positive integers), for a finite list only finitely many of these projections are needed; i.e., there are only a finite number of residual functions to store.

The incremental algorithm specification is thus $\langle$`dfa,p`$\rangle$.

Example 6.1 *We use the flowgraph in Figure 6 to detail the working of the algorithm. Assume that it represents the second connected component in the input list. During the* Setup *phase, the computation of the residual function corresponding to projection* `pr 2` *unrolls the fixpoint iteration for that component to yield: (this is sanitized Haskell code representing the actual output of our partial evaluator)*

```
dfa_pr2 (eqn1 : ( _ : rest_eqns)) =
  foldl fix env2 rest_eqns
  where env2 =
         eval [ (L1,(union,L0,[(x,B),(y,C)])),
                (L2,(union,(diff,L0,[(x,?)]),
                           [(x,B),(y,C)])),
                (L3,(union,(diff,L0,[(y,?)]),
                           [(x,B),(y,C)])) ]
              env1
        env1 = fix null-env eqn1
```

Note that `dfa_pr2` *ignores the second element of its input (the notation _ means don't care). A similar computation has been carried out for each element of the partition (a connected component). If any of the components change, then the* Reestablish *phase computes the residual for the affected members of the partition. During the third phase (*Combine*), the result is obtained by computing the lub of all the residual functions, which effectively propagates information in topological order between the components. For example, the predecessor (in topological order) of the component above will supply the value of L_0, which will enable this component to supply its topological successor the value of L_3, and so on. It is easy to see that there is no fixpoint iteration in this stage.*

Note that partitioning into strongly connected components plays an important role in enabling fixpoint iteration to proceed to completion even when only part of the input is known. This is essentially the intuition behind the algorithm in [MR90]. How can we be sure that this algorithm posseses the same time complexity as the one in [MR90]? The algorithm in [MR90] has two main steps (called steps 5 and 6 in the paper).[3] The first step corresponds to recomputing residual functions of projections which "see" changes in the input. The second corresponds to the compuation of the result via a lub construction.

During the **Setup** and **Reestablish** phases the partial evaluator has enough information to unroll the function `fix` for a given strongly connected component.

Lemma 6.1 *The cost of the* **Reestablish** *phase is proportional to the size of the affected strongly connected component.*

Proof: The cost of carrying fixpoint iteration (for this problem) to completion is well known to be $\mathcal{O}(B \times V)$ where B is the size of the component and V is the number of variables of interest. For a fixed number of variables the cost is $\mathcal{O}(B)$. □

The next step simply combines residual functions using the $\sqcup$ operation. No fixpoint iteration needs to be performed. The only task left to do is to transmit information among the various components. The cost of this is easy to see:

Lemma 6.2. *The* **Combine** *phase does no fixpoint iteration, but only propagates information in topological order. The cost of this step is proportional to the condensed flow graph.*

Proof: Follows from the fact that the information is propagated in topological order with a constant amount of work done at each component. □

6.2 Incremental Attribute Evaluation

The use of attribute grammars [?] to describe language specifications is well known. These specifications have been used as the starting point for the generation of language based programming environments [Rep84]. Programs in this model are represented as *attributed trees*, i.e. syntax trees

[3] The first four steps of the algorithm in [MR90] are implicitly present in our algorithm because of the input DataFlowAnalysis expects, namely strongly connected components of the flow graph in toplogical order.

$$
\begin{array}{lcll}
N \rightarrow SL & \{\ L.scale & = & 0 \\
 & N.val & = & \textit{if } S.neg \textit{ then } -L.val \\
 & & & \textit{else} L.val\ \} \\
S \rightarrow + & \{\ S.neg & = & false\ \} \\
S \rightarrow - & \{\ S.neg & = & true\ \} \\
L \rightarrow B & \{\ B.scale & = & L.scale \\
 & L.val & = & B.val\ \} \\
L_0 \rightarrow L_1 B & \{\ L_1.scale & = & L_0.scale + 1 \\
 & B.scale & = & L_0.scale \\
 & L_0.val & = & L_1.val + B.val\ \} \\
B \rightarrow 0 & \{\ B.val & = & 0\ \} \\
B \rightarrow 1 & \{\ B.val & = & 2^{B.scale}\ \}
\end{array}
$$

Figure 7: Attribute grammar for signed binary numerals

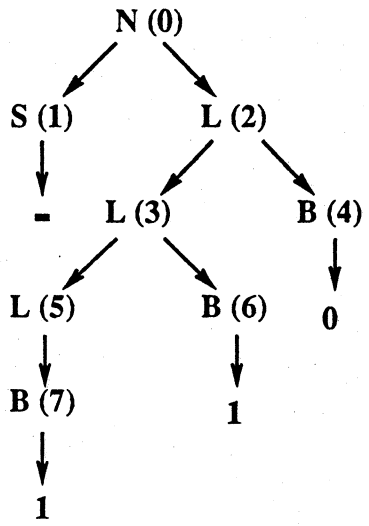

Figure 8: Parse tree for -110

with attributes carrying semantic values. The goal of incremental attribute evaluation is to efficiently produce a well attributed tree after each editing operation. In what follows, editing is modelled by subtree replacement, and we only consider *non-circular* attribute grammars.

As usual let us first describe a non-incremental algorithm, which we do by means of an example. Consider the attribute grammar in Figure 7, which describes the syntax for signed binary numerals, as well as the semantics: the decimal value that the numeral denotes. For example, the parse tree for -110 is shown in Figure 8 (we will refer to this example later).

Given an attribute grammar and a parse tree, the non-incremental algorithm for attribute evaluation is as follows:

- Generate the equations described by the parse tree. (For example, Figure 9 shows the equations for the parse tree in Figure 8; note the subscripting of the attributes by the node numbers.)
- Generate a dependency graph, where an attribute a_i depends on a_j iff the right hand side for the equation defining a_i contains a_j. (See Figure 10.)
- Perform a topological sort of the dependency graph, to get a safe order of evaluation of the attributes, and then perform the evaluation.[4]

$$
\begin{array}{rcl}
val_0 & = & if\ neg_1\ then\ -val_2\ else\ val_2 \\
scale_2 & = & 0 \\
neg_1 & = & true \\
val_2 & = & val_3 + val_4 \\
scale_3 & = & scale_2 + 1 \\
scale_4 & = & scale_2 \\
val_4 & = & 0 \\
val_3 & = & val_5 + val_6 \\
scale_5 & = & scale_3 + 1 \\
scale_6 & = & scale_3 \\
val_6 & = & 2^{scale_6} \\
val_5 & = & val_7 \\
scale_7 & = & scale_5 \\
val_7 & = & 2^{scale_7}
\end{array}
$$

Figure 9: Attribute equations for the example parse tree

To generate an incremental algorithm, we need only specify a partition of the input domain (in this case a parse tree). As a first attempt at such a partition, suppose p_i is the projection reflecting the fact that only the descendants of node i are known. Then $\overline{p_i}$ is the projection where every node *except* the descendants of i are known. A candidate partition is thus:

$$P = \{p_i\} \cup \{\overline{p_i}\}$$

Recall that the **Setup** phase computes the residual functions corresponding to each projection in the partition. Using the above partition for Figure 8, let us see what $r_{\overline{p_5}}$ looks like. The partial evaluator has knowledge of all nodes except 7. This means that it can construct the modified dependency graph shown in Figure 11, where the dotted portion is unknown. It is not difficult to see that the partial evaluation of a toplogical sort on this graph will result in the evaluation of $scale_2$, $scale_3$, $scale_4$, $scale_5$, $scale_6$, val_4, val_6, and neg_1, even though the subtree rooted at 5 is not known.

The **Reestablish** and **Combine** phases work as follows. If we are given a new subtree at (say) node 5, we compute r_{p_5} and take the lub of $r_{\overline{p_5}}$ and r_{p_5} to obtain the answer. But now the **Reestablish** phase can be unacceptably expensive. Changing a subtree rooted at node 5 causes residual functions far away from node 5 to be altered. This means that every time a subtree is changed, unacceptably many residual functions will have to be recomputed.

This problem leads us to seek a new partition. Consulting the literature, we find in [RTD83] an incremental

[4] Since we are considering only *noncircular* attribute grammars, the graph is guaranteed to be acyclic.

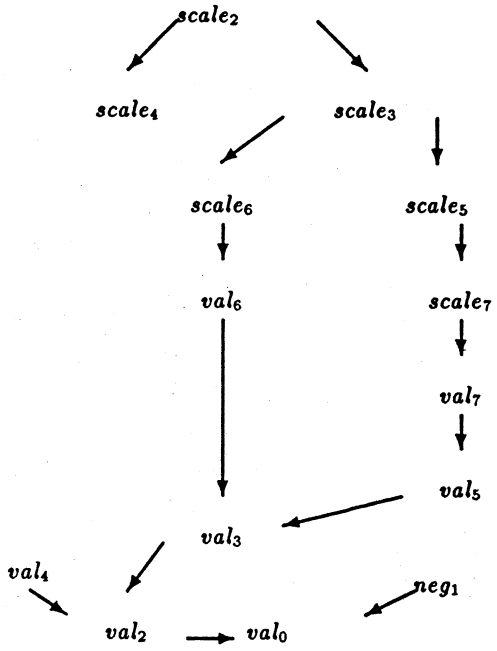

Figure 10: Dependency graph

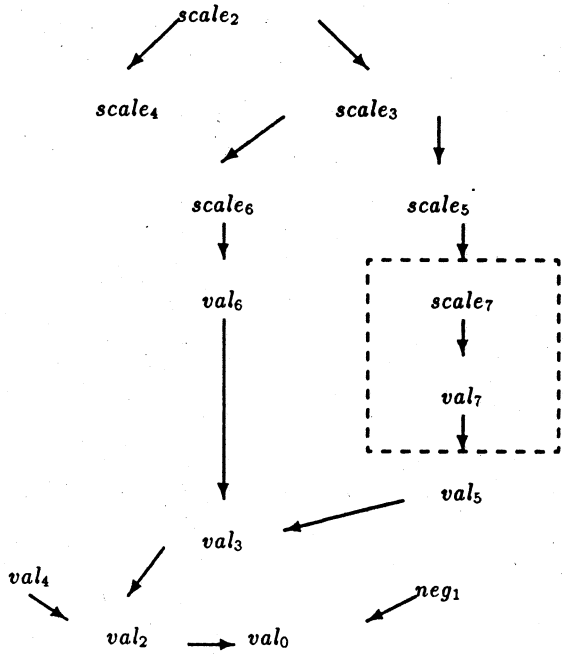

Figure 11: Partial dependency graph

algorithm for the same problem which reduces the amount of work done while updating the stored information. We can use exactly the same method, described below in our framework.

The key idea is to make use of a *restricted editing model*, in which *cursor* position is maintained:

- The cursor is at any given moment at one node in the parse tree, and can be moved in one of two ways: to the parent node, or to one of the children.
- The only edit operation permitted is subtree replacement.

Thus at any given moment the nodes can be partitioned into three disjoint sets: The first is the singleton set R consisting of the cursor position r. The second is the set S of nodes on the path from the root to the cursor position, including the root. The third is the set T of remaining nodes, which includes nodes below the cursor. Using this knowledge, the new input partition is as follows:

$$\mathrm{P} = \{p_i, \overline{p_i} \mid i \in R\} \cup \{p_i \mid i \in T\} \cup \{\overline{p_i} \mid i \in S\}$$

The **Setup** phase is similar to that described for the previous partition, only simpler. The **Reestablish** phase, however, is more interesting. Consider a node a with three children b, c and d. There can now be three kinds of changes.

- **Move To Parent.** The set T gets a new member (the old cursor position); but since the old R had both r_p and $r_{\overline{p}}$ computed, there is no work to do. The set S loses a member, so again their is no work to do. The set R gets a new member for which we need to compute r_p. Moving from child b (say) to parent a, the operation needed to **Reestablish** the invariant is:

$$r_{p_a} = r_{p_b} \sqcup r_{p_c} \sqcup r_{p_d}$$

- **Move to Child.** By similar reasoning, the operation to be performed for a move from parent a to child c is:

$$r_{\overline{p_c}} = r_{\overline{p_a}} \sqcup r_{p_b} \sqcup r_{p_d}$$

- **Replace Subtree.** Here we simply need to recompute r_p for the current cursor position. By virtue of the partition, nothing else needs to be changed (cf. the previous partition).

When a subtree is replaced at node p, the new result is obtained by computing $r_p \sqcup r_{\overline{p}}$. The first observation we make concerns the computation of $r_{\overline{p}}$: any attribute which does not depend on projection p of the tree gets a value during the computation of $r_{\overline{p}}$.

How much work is done when a subtree is replaced?

Theorem 6.1 *The work done by the* **Reestablish** *phase after a subtree replacement is proportional to the number of attributes potentially affected by the change.*

Proof: During the **Reestablish** stage, the work done is in computing those attributes in the modified subtree which do not depend on the rest of the tree. During the **Combine** phase the work done is in computing those attributes in the modified subtree which depend on the rest of the tree. Thus the total work done is exactly in recomputing those attributes which are potentially affected by the subtree modification. □

Compare this to the algorithm in [RTD83] which achieves a better time complexity, namely that an attribute will be re-evaluated only if the values of any of the attributes it depends upon change. The algorithm we outlined will re-evaluate all attributes which depend on the changed subtree. During cursor movement, the algorithm in [RTD83] achieves unit cost per move. The algorithm we have outlined takes time proportional to the number of attributes whose values are resolved as a result of the $\sqcup$ operation. We are investigating ways to improve our solution to match the efficiency of [RTD83].

7 Brief Comparison With Other Approaches

Readers familiar with the literature of partial evaluation will recall Lombardi and Raphael's pioneering paper [LR64], which coined the term "partial evaluation" in the context of doing "incremental computation." However, their notion of incremental computation was to monotonically add information about the input to a function, and to do as much computation as possible at each step. This is achieved by computing a series of residual functions each of which is the result of partially evaluating the previous one with the additional input. Our definition of incremental computation (and what is generally understood by the term today) is more general than this in that changes to the input need not always be in the form of adding information — information can change non-monotonically. Thus our work can be seen as a generalization of the work which originally introduced partial evaluation.

The goals of all the following approaches are very similar to ours, but they differ in methodology.

INC. Yellin and Strom [YS89] describe a restricted functional language for incremental computation. The main data structure in the language is a *bag*. Programs written in the language make use of certain combining forms which are guaranteed to have efficient incremental performance. The emphasis in this work has been to design efficient incremental algorithms for the various operations in the language. A main difference is that the target language for INC are objects called circuits. Our target language is the same as the one in which the non-incremental program is described.

Caching. Pugh [Pug88] tackles the problem of caching results of function calls to achieve incrementality. The scheme depends crucially on clever run-time support. In addition, programs have to be written using what Pugh calls *stable decompositions* of data structures to obtain good performance.

Incremental λ-Calculus Reduction. In [FT90] Field and Teitelbaum construct an incremental evaluator for the λ-calculus. It keeps track of exactly which reductions did not depend on the part of the λ-term which changed. For this purpose the parts of the term which can change have to be marked in advance. In contrast with all these approaches, our approach allows the possibility of generating a "compiled" incremental program via partial evaluation of the incremental interpreter itself.

8 Discussion

Compiled Incremental Programs. The use of the incremental interpreter $\mathcal{I}$ gives rise to inefficiency due to a level of interpretation. Inspired by the Futamura Projections [Fut71], we can achieve a process of "incrementalization" as follows:[5] (we use the notation $[\![f]\!]$ to denote the function corresponding to the program f, i.e. the semantic function of the language)

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

Running f_{inc} is more efficient than using the incremental interpreter, and has the advantage that it can be used independently of the incremental interpreter. Using the second Futamura projection, we can self apply the partial evaluator to generate an *incrementalizer*: a program which converts incremental program specifications into incremental programs.

$$[\![\mathcal{PE}]\!]\ \mathcal{PE}\ \mathcal{I} = \mathcal{INC}$$

Binding Time Analysis. Binding time analysis is essential for good performance of the incremental programs we have described. Since the partition is known before the actual data elements are available, binding time analysis can be carried out off-line. This means that the self-application described above can avoid the overhead of interpretation. Binding time analysis is usually achieved by an abstract interpretation of the program. To ensure termination, finite domains are usually used. But as we have seen our domain of projections is infinite (although the number of projections used in any given session is finite). One way to overcome this problem for implementation purposes is to place a fixed upper limit on the size of the input data. We are investigating other solutions.

Maintaining the Partition. We have not discussed the cost of maintaining the input partition. In some cases (the attribute grammar example) the cost of maintaining the partition is not very high. In the case of the data flow analysis we

[5] In what follows we use the more common notation for partial evaluation: a partial evaluator takes a function and *some* of its arguments to produce a residual function.

must employ a method to maintain the strongly connected components of the flow graph.

Restrictions on the use of Residual Functions. A drawback of our framework is that residual functions cannot be freely used 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.

9 Conclusions

We have presented a framework for constructing incremental programs from their non-incremental counterparts. We have used the notion of partial evaluation to provide a basis for our framework. This is particularly appropriate since binding-time analysis (an important phase of partial evaluation) is concerned with analysing dependencies in a program based on the known/unkown signature of the input. This approach offers a degree of automation in the construction of incremental programs. The framework presented in this paper (including the examples) has actually been implemented. For details of the implementation, the reader is referred to [Sun90].

Acknowledgements. Thanks to Charles Consel, Olivier Danvy, John Hughes, G. Ramalingam and Tom Reps for their detailed comments and suggestions. Thanks also to Juan C. Guzmán for help with LaTeX.

References

[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.

[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.

[HWe90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell. Technical Report YALEU/DCS/RR-777, Yale University, Department of Computer Science, April 1990.

[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.

[Knu68] D. Knuth. Semantics of context-free languages. *Math Systems Theory*, 2(2):127–145, February 1968.

[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.

[LR64] L. A. Lombardi and B. Raphael. Lisp as the language for an incremental computer. In *The Programming Language LISP: Its Operation and Applications*, pages 204–219. Information International Inc., The MIT Press, 1964.

[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.

[MT46] J. C. C. McKinsey and A. Tarski. On closed elements in closure algebras. *Annals of Mathematics*, 47(1), January 1946.

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

[Rep84] T. W. Reps. *Generating Language-Based Environments.* The MIT Press, 1984.

[Rep90] T. Reps. Algebraic properties of program integration. In *Proceedings of the 3rd European Symposium on Programming, Lecture Notes in Computer Science, Vol. 432.* Springer-Verlag, May 1990.

[RTD83] T. Reps, T. Teitelbaum, and A. Demers. Incremental context dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[Sun90] R. S. Sundaresh. Implementing incremental computaion via partial evaluation. Technical Report YALEU/DCS/RR828, Yale University, Department of Computer Science, November 1990.

[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.