Incremental Computation

3.1 Introduction
A computation performed on the same or nearly identical data inputs would be termed incremental if the results of the preceding computations are used constructively for the succeeding computations. In most interactive systems, computing from scratch may result in a considerably delayed response. Moreover, such systems fail to scale up. In other words, the time lag between the request and the response might be large enough to irritate and disinterest the user. Possible solutions to this problem are making algorithms even faster (which is, however, limited to its optimal performance), using faster hardware platforms and, finally making the algorithms incremental. In the following section, we describe various techniques and terminology used in incremental computation. Section 3 deals with the applications of incrementality. Finite differencing, graph-based approach, language-based approach and message interchange approach are looked into and analyzed in sections 4, 5, 6 and 7, respectively. Hybrid and emerging techniques are described in section 8 and conclusions drawn in section 9.

3.2 Techniques
Incremental Techniques, therefore, store and reuse the results of previous computations, thus minimizing both time and computational efforts unlike batch evaluators which start computations from scratch. Incremental Evaluators are used to evaluate incrementally an algorithm that computes from the input. On the other hand, Explicitly Incremental Algorithms specify both, how to compute output from an input and, how to effect updations in response to changes in the input.
It becomes quite apparent after understanding these environments, that the inherent aim of all these systems is to minimize the amount of computation at each stage of iteration by taking advantage of the fact that very similar computations have already been performed [Yel91]. Even though it is accepted that the principle underlying incremental systems is the same, no universal approach has been agreed upon.

Different approaches suggested by the implementors of such systems -- claimed to be optimal -- only reflect the applications for which they have been developed. They find little use outside their particular area of applications. Researchers have often suggested highly efficient solutions for specific problem domains, without any contribution towards the development of a Unified Incremental Theory. Quite a few of these techniques are based on well defined formalisms, but more often than not, ad-hoc techniques and heuristics are employed. In general, we can classify the various approaches under the following heads: 1) Finite Differencing; 2) Graph Propagation; 3) Language Based Approach; 4) Message Interchange Approach; 5) Function Caching; 6) Incremental Constraint Solving; 7) Hybrid and other techniques like Retraction, Function Inversion etc.

### 3.3 Applications

Incremental systems have been successfully developed and tested in various areas, the more prominent among them being the following.

#### 3.3.1 Software Development Environments

These are interactive environments, providing fast response times which report errors and anomalies in the programs, during the coding phase itself. Among the popular ones are Cornell Synthesizer Generator [Rep82, Rep83, Rep84], Gandalf, POEGEN etc. A slightly different kind of environment like GALAXY [Bee91] among others in its class, provides the programmer with incremental tools like Incremental Scanners, Incremental Parsers, Incremental Semantic Analysers and Incremental Linkers [Quo91]. Then there are
Generators for such systems like Incremental Parser generators \cite{Hee89, Hor88} etc. Optimisers and Anomaly Reporters and Incremental Data Flow Analyzers are used instead in the analysis and synthesis phases of the traditional compiler phases.

### 3.3.2 Spreadsheet Environment

Extensively used in the industry, these are quite useful for business calculations. They allow the user to interactively update and change data objects and view the results quickly by exploiting the fact that computations for most data objects have already been performed and retained. VisiCalc (Visible Calculator) was the first among the family of modern day Spreadsheets.

### 3.3.3 Modeling & Simulation Systems

Such systems allow the user to model circuits, architectures, real time systems, economic models, etc. and view the system response under various constraints and under different input data that are imposed on the system. The success of the idea depends on how quickly the results are presented to the user. Incrementality in such systems ensures that the user can experiment with the system, and yet carry on development at a fast pace, without getting inhibited by system responses. Graphics Systems and Editors follow a similar philosophy whereby the user changes a couple of parameters in a picture and wishes to visualize the resulting picture \cite{Ase87}.

### 3.4 Finite Differencing

#### 3.4.1 Introduction

This methodology tries to universalize the problem by expressing it in terms of a function, which in itself may contain a host of other functions. The concept of function transformation is used to convert the function into an incremental function by using certain predefined transformations. Finite Differencing, thus addresses the incrementality of a single function and may be termed as a fine grained approach \cite{Yel91}. A major drawback
of this approach is that it is unable to take care of situations where it is cheaper to compute from scratch.

### 3.4.2 Background

Exemplified by Paige [Pai82], this employs the technique of strength reduction generalizing, Cockes' [Coc77] technique to provide a flexible framework for efficient program optimization. Program Transformation in this technique is looked more through the optimization window, than as a specific technique towards achieving incrementality. The genesis of the technique lies in the finite difference method devised by the sixteenth century English mathematician Henry Briggs to generate a sequence of polynomial values. Cocke's optimization technique called the "reduction in operator strength" could speed up programs and it was later observed that Finite Differencing when applied in a set-theoretic milieu can considerably improve the asymptotic behavior of an algorithm. Keeping in tune with this result, Paige [Pai82, Pai86] has applied Finite Differencing to a programming language SETL that in addition to the usual constructs provides high-level set operations on maps, tuples and sets.

### 3.4.3 Verification

Since the emphasis is on program optimization, the need for program verification has led to the introduction of two additional statements:

\[
\begin{align*}
\text{assume} & \quad <\text{cond}> \\
\text{assert} & \quad <\text{cond}>
\end{align*}
\]

Where \(<\text{cond}>\) is any predicate allowed in the language.

The execution of a program is defined as valid if all the assume conditions are satisfied or the first unsatisfied assert condition is preceded by an unsatisfied assume condition. If all possible executions are valid, the program is defined as valid. Domain is a set of inputs for which every assume condition holds. A transformation of a program \(P\), as shown in Figure 3.1, can have two implications.
Validity preserving: A transformation preserves the validity if the assert conditions at the output are intact in the transformed program.

Semantics preserving: If the mapping $T$ is such that the domain of $P'$ is the same as that of $P$, in addition to being validity preserving, it is termed semantics preserving.

3.4.4 Approach

Let $V = f(x_1,\ldots,x_n)$ be an applicative expression where $x_1,\ldots,x_n$ are the arguments of the function $f$, and $V$ is the variable into which the final result, after computation is stored. All the arguments presented hereafter are for program blocks, a single entry, single exit region of code. If $f$ has been computed prior to entering the block and its value stored in $V$, then all the statements of the form $V = f(x_1,\ldots,x_n)$ within the block are redundant, if the arguments $x_1,\ldots,x_n$ are not changed. In such cases, all the occurrences of $f$ may be replaced by $V$ and same will be available at the exit. Should the arguments of $f$ change values inside the block, we must keep $V$ available anyway. Hence the basic premise under which finite differencing works is that the cost of keeping $V$ available within the region of interest must be less than the cost of calculating $f$ from scratch each time it is referenced.

Towards this end, some transformations are defined and an important transformation is:
Incremental Computation

\textit{achieve} \( E = f(x_1, \ldots, x_n) \)

which carries the same semantics as assignment statement.

This optimization is generally viewed as an extension of code motion, the difference being that in code motion, a loop invariant computation is placed before the loop while in strength reduction, the computation \( E := f(x_1, \ldots, x_n) \), in spite of changes to its parameters within a program region, is moved outside the program region and is kept available within the region by updating \( E \) every time one of its parameters undergoes a change in the value.

3.4.5 Transformations

Three semantics preserving transformations are used to implement the Cocke's scheme.

1. \textbf{Init Transformation} \textit{Init}(\( P \)) replaces contiguous sequence of achieve \( E := f(x_1, \ldots, x_n) \) statements by a code block which computes \( f \) and initializes \( E \).

2. \textbf{Diff Transformation} \( \partial J(R) \) inserts code in the program region \( R \) to keep \( J \) available within it, despite modifications to its parameters.

3. \textbf{Clean Transformation} to eliminate the dead code.

The Differential Program Transformation \( \partial J(R) \) depends on the choice of an appropriate computable derivative that allows the computations of the new value of \( E \), when its value is spoiled by a change in one of the variables on which \( E \) depends.

3.4.6 Computable Derivative

This is a code block pair \([B_1, B_2]\) such that:

1. Only variable modified by \( B_1 \) or \( B_2 \) are \( E \) and local variables of \( B_1 \) and \( B_2 \).
2. Code block

\[
\text{achieve } E = f(x_1, \ldots, x_n) \\
B_1 \\
\partial_j x_i \\
B_2 \\
\text{assert } E = f(x_1, \ldots, x_n)
\]

is semantics preserving with respect to \(dx;\) and contains only redundant uses of \(f.\)

\(B_1 = \partial^- E(dx;\)\) is the Prederivative and

\(B_2 = \partial^+ E(dx;\)\) is the Postderivative.

The derivatives rules are further collected as quadruples into a library of \textit{derivs}. The \textit{derivs} are of the form:

\[[\text{Applicative expression, } dx; , \partial^- E(dx;\) , \partial^+ E(dx;\) ]

Similarly, we define \(\partial E(B)\) where \(B\) is a code block such that the derivative code with respect to each assignment \(dx;\) occurring in \(B\) is included.

3.4.7 Example

We explain this with an example taken from [Pai 82].

Let \(E=\{x \in A \mid x \mod 2 = 0\}\), where \(A\) is a set of integers, be an applicative expression.

\(\partial^- E< A \text{ with } :=i >\) is if \(i \mod 2 = 0\) then \(E \text{ with } :=i;\) end if;

\(\partial^+ E< A \text{ with } :=i >\) is nil

\(\partial^- E< A := \{\} >\) is \(E:=\{\}\);

\(\partial^+ E< A := \{\} >\) is nil

Let \(B\) be the code block

\[a:=\{\};\]
\[(\text{while } \text{eof } = \text{false})\]
\[\text{read}(i);\]
\[a \text{ with } :=i;\]
\[\text{end while};\]
\begin{equation*}
\text{print} \{x \in a \mid x \text{ mod } 2 = 0\},
\end{equation*}

\begin{verbatim}
\partial E<B> --> \partial E:a:=\{}\>/* \partial^* E< > = \text{nil }*/
\text{a} = \{}\;
(\text{while eof } = \text{false})
\text{read}(i),
\partial E< a \text{ with } :=i> /* \partial^* E< > = \text{nil }*/
a \text{ with } :=i;
\text{end while};
\text{print} \{x \in a \mid x \text{ mod } 2 = 0\};
\end{verbatim}

By using assertions and propagating them, the semantic preserving property of the
transformation may be proved.

### 3.4.8 Differential Calculus

A whole calculus of computable derivatives may be proved from the basic operations and
some of the results from [Pai82] are enunciated here:

1. Differential Transformation is a linear operator with respect to sequential code blocks.

\begin{equation*}
\partial E<B_1,B_2> = \partial E<B_1> \partial E<B_2>
\end{equation*}

2. If \(\partial E_1<dx>\) represents the differential of \(E_1\) with respect to changes in \(dx\) and
\(\partial E_2<\partial E_1<dx>>\) represents the differential of \(E_2\) with respect to changes in the modified
code block \(\partial E_1<dx>\), then

\begin{equation*}
\partial E_2<\partial E_1<dx>> = \partial E_2<\partial^* E_1<dx>>
\text{\(\partial E_2<dx>\)}
\text{\(\partial E_2<\partial^* E_1<dx>>\)}
\end{equation*}

\begin{equation*}
= \partial E_2<\partial^* E_1<dx>>
\partial^* E_2<dx>
dx
\partial^* E_2<dx>
\partial E_2<\partial^* E_1<dx>>
\end{equation*}
The same result might be generalized to state the chain rule.
\[ \partial E_n, E_{n-1}, \ldots, E_1 \langle B \rangle = \partial E_n, E_{n-1}, \ldots, E_2 \langle \partial E_1 \langle B \rangle \rangle \]

### 3.4.8.1 Example

We display the applications of this chain rule with the help of an example taken from [Pai82]

\[ E_1 = S + T \]
\[ \partial^* E_2 \langle S \text{ with } := y \rangle = \begin{cases} y \in T & \text{if } y \in T \\ E_1 \text{ with } := y & \text{end if} \end{cases} \]

\[ E_2 = \{ x \in S \mid K(x) \} \]
\[ \partial^* E_2 \langle S \text{ with } := y \rangle = \begin{cases} k(y) & \text{if } k(y) \text{ then} \\ E_2 \text{ with } := y & \text{end if} \end{cases} \]

\[ E_3 = \mathcal{S} \]
\[ \partial^* E_3 \langle \mathcal{S} \text{ with } := y \rangle = E_3 := 1; \]

\[ E_3 = \# \{ x \in (S + T) \mid k(x) \} \]
\[ \partial E_3, E_2, E_1 \langle \mathcal{S} \text{ with } := y \rangle \rightarrow \partial E_3, E_2 \langle \text{if } y \in T \text{ then} \\ E_1 \text{ with } := y & \text{end if} \\ \mathcal{S} \text{ with } := y \rangle \]

\[ \rightarrow \partial E_3 \langle \text{if } y \in T \text{ then} \\ \text{if } k(y) \text{ then} \\ E_2 \text{ with } := y; & \text{end if} \\ E_3 := 1; \\ \mathcal{S} \text{ with } := y \rangle \]

\[ \rightarrow \text{if } y \notin T \text{ then} \\ \text{if } k(y) \text{ then} \\ E_2 \text{ with } := y; & \text{end if} \\ \mathcal{S} \text{ with } := y \]

\[ ^1 \text{ The set cardinality function} \]
This code shows a considerable improvement over the usual ordered sequence:

\[ E_1 = S + T \]
\[ E_2 = \{ x \in E_1 \mid k(x) \} \]
\[ E_3 = \#E_2 \]

3.4.9 Initialization Transformation: INIT

This converts a code block into a transformed block where every contiguous achieve \( C = f(x_1, \ldots, x_n) \) statement is evaluated and its value stored into \( C \). We explain this with an example from [Pai82].

\[ C_1 = \{ x \in S \mid k_1(x) \} \]
\[ C_2 = \{ x \in C_1 \mid k_2(x) \} \]

The normal pattern of execution in that order would lead to two full iterations over possibly large sets. \( C_1 \) may be evaluated as code block \( B: \partial^+ C_1 < S = S > \)

\[
C_1 = \{ \} ;
(\forall x \in S)
\]
\[
\text{if } k_1(x) \text{ then}
\]
\[
C_1 \text{ with } := x ;
\]
\[
\text{end if} ;
\]
\[
\text{end } \forall ;
\]

Differential of \( C_2 \) with respect to code block \( B \) yields an efficient initialization of \( C_2 \) jammed into the \( C_1 \) initialization loop.

\[
C_2 = \{ \} \\
C_1 = \{ \} \\
(\forall x \in S)
\]
\[
\text{if } k_1 \text{ (x) then}
\]
\[
\text{if } k_2 \text{ (x) then}
\]
\[
C_2 := x ;
\]
\[
\text{end if} ;
\]
\[
C_1 \text{ with } := x ;
\]
\[
\text{end if} ;
\]
\[
\text{end } \forall ;
\]

This is a typical example of what may be termed as vertical jamming.
3.4.10 Cleanup Transformation: Clean <P>

This is the final step of finite differentiation, and may be affected using standard dead code elimination techniques. According to Kennedy [Ken81], the essential statements may be determined and all other statements considered dead and eliminated.

3.5 Graph Based Approach

These are quite popular, and in conjunction with Attribute Grammar (AG) formalism are widely used in prototype environments designed to validate this approach [Hoo87]. The problem domain is transformed into a dependency graph, which is simply a directed graph, with nodes containing the results of intermediate computations and the edges representing the dependencies among the various computations. Any modification in the problem domain are faithfully reflected as changes in the dependency graph, with the algorithms updating the node values. Incrementality is achieved by updating only those nodes which have changed values during a particular iteration [Hud91]. This technique may be termed coarse grained when compared to finite differencing [Yel91], as functions computing the values at individual nodes are in themselves not incremental and have to be worked out from scratch, at each iteration. However, when related to Message Interchange approach, the technique is quite fine grained [Kio92]. Graph based techniques, particularly the ones that work in tandem with Attribute Grammars are dealt with comprehensively in the following chapters.

3.6 Language Based Approach

3.6.1 The RDB Query Language

Horwitz [Hor86] combining the ideas of Relational Database and Attribute Grammars, proposed using incremental algorithms for query evaluation and interactive view updating. The limitations imposed by the poor computational power of relational operators are sought to be overcome by using Attribute Grammars. Using efficient, incremental
attribute evaluation Algorithms [Rep82], to update relations, usual RDB Query Languages can be utilized by the users.

3.6.2 INC

3.6.2.1 Introduction
INC, a language for incremental computations [Ye191], has been proposed as a universal, application independent language, incorporating both coarse and fine grained incrementality. INC includes many RDB operators and can compute transitive closures (not possible with the Graph Based Approach), and perform operations over complex relations.

Internally, the program in such languages is represented by a collection of processes and connections encapsulated into a network [Ye191]. Each function gives rise to its own special network, and different functions are interlinked through their input and output connectors. A network of processes, representing a function, receives messages detailing the changes at the input, and outputs another message detailing changes for consumption by other processes, thereby achieving incremental operations.

At the level of abstraction, this technique might seem quite similar to the graph based approach. The program is converted into a network of processes and interconnections, and therefore, the apparent similarity with graphs is quite obvious. However, it needs to be emphasized, that the methodology is quite distinct with the overall thrust towards incremental computation.

3.6.2.2 The INC Language
The goal of INC designers has been to present a universal technique for incrementality. The language INC, has been tailored to suit incremental computations, while following a FP (functional programming) like syntax at the same time. This makes it application independent, where any problem may be expressed in terms of its Functional Operators.
The INC Compiler further converts these into *dags* (directed acyclic graphs) suitable for efficient incremental execution.

INC has been visualized as a functional language, where the programs basically consist of program forming operations (PFO's), operating with a set of primitive functions on a domain of primitive or constructed data types appropriately called objects. Primitive data types are *integer*, *real*, *boolean* and *character*. Fixed length tuples and *bags* form the constructed Data types and commensurate with the Functional style, each type has a distinguished null element within its domain.

Usual primitive functions like *plus*, *minus*, *times*, *div*, *mod*, *lt*, *le*, *gt*, *gte*, *eq*, etc., operate on primitive data types producing appropriate outputs. A distinguishing feature is the manner in which tuples and bags are manipulated. Two primitive functions allowed to operate on tuples are *tuplify* which consumes *k* objects and produces a tuple, and *project*, which consumes a tuple and produces its *j* component. Bags, on the other hand, allow a variety of primitive functions to operate on them. *Member(b,B)*, for example, would produce a *true* if *b* is in the bag *B* and a *false* if not. Similarly *bagify*, *empty*, *choice*, *merge*, *duplicate* etc. may be allowed to operate on them.

A variety of INC PFO's are defined, and all, due to the basic philosophy are functional in nature. *Apply*[F](B), for example, applies the functions *F* over the objects in bag *B* and produces bag *B'* . Similarly, other PFO's like *Filter*, *Reduce*, *Partition*, *Equijoin*, *Composition*, *Cond* etc. may be used to build an extensive range of user-defined operations.

INC does not allow any explicit recursion of looping but the INC PFO's are quite powerful enough and programs may be written without their explicit use. Meaningful programs can be written with the help of primitive and user-defined functions while operations on bags, more or less compensate for the apparent inadequacies of the language.
3.6.2.3 INC Networks

At the operational level, INC programs are converted into a network of processes, the underlying principle being that each process consumes some data and outputs data which is consumed by another process. Data flows through the network, and in this manner input presented at one and will ensure results at the output connections.

e.g. Partition A may be used as shown in the Figure 3.II

\[\text{Partition } \{\text{mod 2}\} \ A \text{ where } A \text{ is set of natural numbers in a bag.}\]

\[\{1,3,5,2,7,4\} \quad \rightarrow \quad \text{Partition} \quad \rightarrow \quad \{\{1,2\},\{1,3,5,7\}\} \quad \rightarrow \quad \text{mod 2} \]

![Figure 3.II The INC network for a Partition](image)

The selection is quite simple and clear cut for batch programs. In order to ensure efficient incremental execution, processes are designed in a fashion so as to remember their previous states and instead of data flowing through the network, it is the message detailing changes which flow.

3.6.2.4 Incrementality in INC

This technique, even though, quite at contrast with the orthodox programming methodology, has to work under a couple of constraints in order to take care of efficiency considerations. The first being that no process should take more time computing incrementally than it would take to compute from scratch, and the second being that the messages conveying changes cannot be larger than the data itself. At the same time, it is
assumed that the algorithms are versatile enough so that arbitrary sized structures can be discarded in constant time and when efficiency so demands, computation may start from scratch. For example, if some objects are inserted or deleted from a bag, the differential messages may be of the form: \(<\text{Inserts}, \text{Deletes}>\)# or if, say a large number of deletions have taken place over the last iteration, messages may be of the form \(<\text{Inserts}, \text{Remains}>\)#. The idea is to generate the smallest sized messages in conformity with the second bound given above.

3.6.2.5. INC Complexity

*Static* complexity is defined as cost executing a process from the scratch while *incremental* complexity is defined as the cost of executing, when differential information is propagated through the process. *Static* and *incremental* complexity for various functions may be computed and a relative idea of efficiencies got. e.g. \(\text{Apply}[F](B)\) - the *static* complexity is \(O(|B| \times \text{Cost}_F)\) and *incremental* complexity is \(O(|M_B| \times \text{Cost}_F)\) where \(B\) represents the multiset (bag) and \(M_B\) is the incremental change in the bag. These figures apply only when set operations like \(\text{Insert}, \text{Delete}, \text{Member}\) can be performed in expected unit time. For complex data structures, it is difficult to arrive at a measure of complexity as this would solely depend on the data structures and algorithms used by a particular process, and the manner in which messages detailing differential changes are communicated to and from the process.

Nevertheless, worst case or average case INC analysis can still be done, and if the particular application is known, a clear and precise analysis of the complexity can be derived from its constituent functions and PFO's. The basic premise is that incremental complexity should not be worse than the corresponding static complexity.

It is not clear how messages detailing changes in complex data types of INC can be constructed. Nested bags present such a problem. Again, complex types require the use of dynamic networks which are not incorporated into the state INC model. In the face of
differential changes, it becomes very difficult to work out the cost of even simple functions operating on complex data types.

As pointed out earlier, one of the main flaws in the expressiveness of INC comes from the absence of recursive functions. Several possible solutions are visualized. Dynamic networks may solve the problem, where each new call may create a fresh network. Another way would be to include fixed point operators (FP) or higher-order functions.

Storage becomes the major casualty in the thrust towards incrementality. This is particularly so because each process has to maintain its own internal data structures. Depending upon the particular situation, a number of storage optimization techniques may be put into use. If the cost of computing the output from the input is nearly equal to searching it through a hash table, the latter may be deleted resulting in a major saving in the table costs. Algebraic Transformations of INC networks into equivalent networks with less storage costs can be affected on the lines of RDB query optimisation. The third and most viable approach seems to be the use of President Data Structure. Finally, it might be possible to break programs into fragments, and convert only those functions into incrementally executable versions where the running cost greatly overweighs the overhead storage.

3.7 Message Interchange Approach

Systems utilizing this approach try to overcome the lacunae in the traditional AG models, which happen to be more applicative rather than procedural [Dem85]. The rigidity imposed by AG formalisms is substituted by a more relaxed message passing paradigm. [Kio92] has proposed a model where information sources and information sinks are identified and located, so that the flow of information occurs from the sources to the sinks. Although appearing to be quite similar to the Graph Based Approach, the primary importance is on information flow rather than transformation of the model into a graph. [Dem85] introduced the concept of "message classes" and "message passing" which may loosely be related to semantic attributes and attribute flow. This system, even though
working on an underlying Graph, allows widely spaced modes to interchange an arbitrary number of messages. Incremental changes are propagated by passing messages round the system.

The model of message passing as proposed by [Dem85] may be viewed as analogous to evaluators based on Attribute Grammar Formalism [Wai84]. Both the models operate on an underlying derivation tree, or its optimized version, the abstract syntax tree. A derivation tree is said to be consistent if all the semantics equations are satisfied locally at each node. Any change in the attribute values at nodes, or a modification involving subtree replacement may result in an inconsistent derivation tree. Incremental attribute Grammar based systems suffer from two major lacunae, the problem of Aggregates and copy rule chains [Hoo87], for which Hoover has given efficient algorithms which get over these shortcomings. In this thesis, we propose algorithms which take care of copy rule chains where the efficiency of the proposed algorithms is better than that of Hoover's algorithm. The message passing paradigm described here involves passing messages between the nodes of the derivation tree.

3.7.1 Introduction

In this model, as proposed by [Dem85], the notion of attribute is replaced by message classes and attribution by message passing. The class of the message determines its essential properties and a message may belong to - at the most - one of the finite extendible set of message classes. Acceptor functions decide whether to accept or propagate a particular message when it arrives at a given node. A particularly useful paradigm is to allow a given node to process an arbitrary number of messages. At the same time, when the node does not have an acceptor function for a particular class of messages, it propagates the message by default.

When a message has been accepted at a particular node, generation functions may be used to produce a new message. Query or assertion messages are sent to the next node in its proper order, while a reply message is sent directly to its originator. The deficiencies in
AG-based systems are avoided here as messages can propagate by a default propagation rule, making explicit copy rules redundant.

Another advantage is that new message classes may be added without any explicit need to rebuild the interpreter. In AG-based systems, adding new semantic rules may result in a major reorganization of the evaluator.

### 3.7.2 Description of the Model

The basic model of information propagation is through message passing. Since duplicate messages are possible, multisets (bags) are used for collection of messages. The propagation specification, which describes the propagation of messages is a 6 tuple,

\[ PS = \langle G, V, C, R, P, S \rangle \]

Where \( G \) is a context grammar
- \( V \) the domain of possible messages values,
- \( C \) the message class system,
- \( R \) the reply class map,
- \( P \) the propagation order assignment and,
- \( S \) the semantic rule assignment.

There are three types of messages: *queries, replies* and *assertions*. A query is a request for information, a reply is the response to a query and an assertion just conveys information. Message types are further subdivided into a finite number of classes which depends on the message type and propagation order. Primary inputs are intrinsic information or input from an external source and are treated as of primary class.

A class system \( C \) is defined as a collection of four finite disjoint sets of message classes

\[ C = \langle C_s, C_q, C_r, \{C_0\} \rangle \]
There is only one primary inputs class, while there may be a number of classes of other types. R, the reply class map is used to map query classes to the corresponding reply classes.

Message - If \( T \) is a derivation tree, a message on \( T \) is a 5-tuple \(<c,v,r,s,d>\) where,

- \( c \in C \) is the message class,
- \( v \in V \) is the message body,
- If \( c \in C_q \) then \( r \in T \) is the return address.
- If \( c = C_o \) then \( s = \text{none} \).
- \( d \in T \) is the destination node of the message.

**Propagation order**: A propagation order is associated with every query or assertion class as only these class of messages needs propagation. A function, mapping derivation trees to successor function, which are acyclic partial functions, define the propagation order.

A propagation order assignment for \( C \) is a function \( P \) mapping elements of \( C_q \cup C_a \) to acceptable propagation order.

**Semantic rules**: These specify the operations that are performed at each node. There are two kinds of semantic rules - input rules and output rules. There are two kinds of inputs, received message or a primary input. Primary inputs and messages detailing replies to queries are always accepted. For query or assertion messages, an acceptance rule is invoked to decide whether to accept or reject it. There are three types of output rules: For assertions and query messages, a generation rule, when applied to the bodies of accepted messages specifies the generation of new message. Reply rules specify how a reply of a particular class is generated in response to a query. When a query is accepted, the manner in which it is forwarded is specified by the forwarding rules.

Semantic Rule Assignment 'S' for a CFG, \( G \) is a 4-tuple
S=<forwarding rules, generating rules, reply rules, accepting rules>
where each component is a function mapping the productions of G to a set of semantic rules.

3.7.3 Attribution
Incremental updation in response to changes require that a static presentation of messages be defined. Hence, each node calls for labeling by a multiset of message received or sent by it. An attribution of a derivation tree T is a 6-tuple,

\[ A=\langle acc, rej, prop, gen, fwd, rep \rangle \]

Where \( acc(n) \) and \( rej(n) \) contain the messages accepted and rejected at \( n \), \( prop(n) \), \( gen(n) \), \( fwd(n) \) and \( rep(n) \) contain the messages which have been propagated, generated, forwarded and replied to. The attribution to be valid needs to satisfy a number of constraints such that the message processing at a node is determined locally at the node in accordance with the local semantic equations.

Acceptable Priority Assignment: There can be more than one valid attribution for a given tree and in order to ensure a unique attribution, pass numbers are assigned to message classes and a priority assignment is made.

3.7.4 Interpreter
The algorithm to update a valid attribution incrementally in response to editing takes as input an old valid attribution "Old A" together with a set of editing changes. The editing changes include changes in the primary input values and changes in the successor functions resulting from the subtree replacement. Since the propagation specification has an acceptable priority assignment, there is a unique new valid attribution "new A". The algorithm computes \( \delta y(n,C) \), the incremental change in the multiset of messages of any given class at a node. It also computes \( \delta acc(n,C), \delta rej(n,C) \), etc., the incremental changes in each of the attribution components. The components of "new A" can be constructed
from the components of "old A" and incremental values. The algorithm scans all the nodes in the list of affected nodes i.e., the ones in which the editing changes have been accepted. At every node it checks for those semantic rules wherein some change in the accepted input has occurred in a message class on which the rule depends. The semantic action on these rules are reevaluated.

For each class in a pass the algorithm examines the incremental change $\delta r(n,C)$ in received messages and uses this to compute $\delta \text{acc}(n,C)$ and $\delta \text{rej}(n,C)$. It also computes $\delta \text{prop}(n,C)$, $\delta \text{gen}(n,C)$, $\delta \text{wd}(n,C)$ and $\delta \text{rej}(n,C)$, the incremental changes in messages sent from a node in a pass. Once the changes in Messages sent from a node in a given pass have all been computed, these changes are distributed to or and oacc values at their destination nodes.

This methodology, which appears to be quite complicated, can be used for a wide class of applications, particularly when the nodes are distributed over a number of locations.

### 3.8 Alphonse

A basic disadvantage in using the INC system is that, it is completely conceptualized around the FP approach. On the other hand, most of the programming effort has been generally in the direction of imperative languages. To convert algorithms into their incremental versions, transformations are required to convert them into functional versions. This requires a major paradigm shift as the functions generated have to be strictly combinators (i.e. depending on only the arguments). These functions are unable to examine the global state, and therefore, very complex transformations are required and in some cases, it might be quite impossible to do so.

Another language which utilizes the concept of incremental techniques and dynamic dependency analysis, on imperative programs has been suggested by [Hoo92]. Alphonse, is a program transformation system which automatically generates efficient dynamic implementation from simple exhaustive imperative program specifications. Imperative
code fragments might be visualized as performing two broad categories of functions. The first part, the *mutator* is the normal code which performs arbitrary operations. The second, the *maintained portion* is an algorithm, which as the name suggests, maintains some property on a program data structure after every change. This property is reestablished after every modification of the data. The *mutator*, makes some environmentally-dictated changes to the data while the *maintained portion* reacts and firmly establishes the property on it.

Two established incremental technique are used to maintain the Alphonse properties. *Quiescence propagation* [Rep84] and *function caching* [Pug88, Pen92] are combined in the Alphonse strategy to provide a framework for incremental computation. *Quiescence propagation* uses a large network of graphs to store all incremental values in a complex computation. The value at each node is a result of some interim computation. A directed edge from node $u$ to $v$ implies a dependency of $v$ on $u$. After a change to the graph, computations are performed beginning at the change points and traversing the dependent nodes. Propagation becomes quiescent when the new result of intermediate computations matches the old cached values.

*Function caching* [pag88, Pen92] computes and stores the values of individual functions for subsequent use. Calls with argument whose values have been cached need not be computed.

The programmer is probably in the best position to state which portions of the program need to be maintained and what function or procedure values may be cached for reuse. Alphonse, therefore enhances the capabilities of a normal imperative program by two pragmas which are different from the rest of the program.

1. Inserting the `<**Maintained**>` pragma before method or override declarations to indicate that they need not be evaluated if they produce results identical to previous computations.
2. The `<Cached>` pragma before a procedure declaration denotes a procedure whose returned value is to be remembered.

Additional pragmas like `DEMAND`, will evaluate the procedure lazily only when it is called, while `EAGER` would effect updations before subsequent procedure calls.

We display the usage of Alphonse with an example from [Hoo92].

```
Type Tree = object left, right : tree;
Methods <"maintained"> height := Height;
end;

Type Tree Nil = Tree object
    Overrides <"Maintained"> height := Height Nil;
end;

Procedure Height (t:Tree) : Integer =
Begin
    Return max(t.left.height(), t.right.height()) +1
db Height;

Procedure HeightNil(t:Tree) : Integer =
Begin
    Return 0
End HeightNil;
```

**Algorithm 3.1: Maintained Height Tree**

The algorithm 3.1 recursively computes the height of the tree from bottom up. Let $\text{subtree}(t)$ be a subtree rooted at vertex $t$. Non-incremental execution of $t.height$ requires $O(|\text{subtree}(t)|)$ time. When the method is maintained, the time for the first call is the same, but any subsequent call to $t$ or any of its descendants requires only $O(1)$ time, as the heights at various levels are cached. Changes in a child field pointing to a node $x$ in the tree will require $O(height)$ time plus the bookkeeping time for quiescent propagation, to update all the cached values on new and old paths from $x$ to the root. Here $t.height()$ is an incremental procedure instance created by dynamic allocation of object $t$ of type tree. The referenced argument set, which includes any incremental procedure instances or top level variables that are transitively referenced, is
**Incremental Computation**

\[ R(t.\text{height}()) = \{ t.\text{left}, t.\text{left.height}(), t.\text{right}, t.\text{right.height}() \} \]

\[ R(n.\text{height}()) = \Phi \quad \text{when } n \text{ is treeNil object.} \]

### 3.8.1 Dynamic Analysis

Most other algorithms use static dependency analysis for incremental systems [Yeh87, RT88, Hud86, Hug89] and most such systems are based on grammars, in a situation where non-local referencing is needed and procedures need to traverse data structure, examine global data and call other functions. A static determination of the exact set of referenced arguments is, however, not feasible. Moreover, with the inclusion of pointer types, this becomes totally redundant. Dynamic dependency analysis is, therefore the only mechanism to achieve incrementallity.

### 3.8.2 Dependency graphs and caching

Let \( p \) be an incremental procedure instance represented by a dependency graph node \( n \). Edges to \( u \) come from nodes representing the elements of \( R(p) \), the set of referenced arguments of \( p \). \( R(P) \) does not contain the arguments of \( p \) i.e. if \( p = \text{O.m} \ (a_1, \ldots, a_k) \) for maintained method or \( p = \text{f}(a_1, \ldots, a_k) \) for a cached procedure call.

Calls to \( \text{O.m. or f} \) are stored in an argument table, containing all nodes that represent calls to \( p \), one for each argument vectors and is indexed by this vector.

Now all the states accessed by \( p \) are encoded in \( R(p) \) and \( <a_1, \ldots, a_k> \) and any change to \( r, r \in R(P) \), can be effectively translated into an update of the cached return value.

Dependencies are dynamically recorded by pushing on to a stack the node representing the currently executing Alphonse procedure instance, say \( u \). If there is a node at the top of the stack, a call to another alphonse procedure instance or a reference to a non-local storage results in other nodes (representing the call or reference) being pushed on to the stack(\( v \)). Further, an edge is added from \( v \) to \( u \) to record the dependency. The dependency graph,
therefore, contains a snapshot of all dependencies that influenced the result of the most recent computation of the Maintained portion of the program.

### 3.8.3 Quiescence propagation

The storage locations are checked whenever an access is made to them for modification, if they have a corresponding node in the dependency graph. If so, then procedures in such nodes may result in different values. If the new value of the node is different from the previously cached value, then the node is placed in a globally inconsistent set.

Upon calls to alphonse procedures, the inconsistent set is checked. If this set is not empty, then inconsistent nodes are removed and processed till the inconsistent set is empty.

Different strategies may be followed for eager or lazy computation. A node, if it represents a storage location, all its successor nodes are entered in the inconsistent set in lazy evaluation, the consistency flag is set to false and its successors put into the inconsistent set.

In eager evaluation, the procedure is reevaluated if the results are different from previously cached values. All elements of the successor set are added to the inconsistent set.

Alphonse can be used to automatically generate dynamic data structures which are maintained according to the specified properties.

### 3.8 Hybrid and Other Approaches

A host of other techniques have been proposed by researchers to make systems incremental. Quite a few of these techniques rely on heuristics and ad-hoc methods like retraction, function inversion, caching etc. Some of the widely used systems adopt hybrid techniques by combining two or more of the approaches. Cornell Synthesizer Generator [Rep81, Rep84], a system for generation of Interactive Programming
Environments, utilizes the methods of "differential propagation", which may be placed somewhere in between the graph-based approach and finite differencing. The technique described in the CSG [Rep84], is to represent the object (the program under development) by an Abstract Syntax tree, where semantic information is attached to the nodes as attributes. Optimality is achieved by updating only those nodes which are influenced by an incremental change. A variety of incremental attribute evaluation techniques have been proposed in [Alb87, Alb89, Fen91, Hoo87, Hor86, Hud91, Kio92, Tei81, Ten88, Vog91].

A thorough evaluation of these techniques is given in [Fen91]. Another major technique worth mentioning here is Function Caching. Description of efficient hash tables and specialised data structures for the purpose may be found in [Pug88]. [Pen92] has enhanced the concept for efficient incremental attribute evaluation.

3.9. Conclusions

The issue of achieving incrementality has been tackled at various levels. INC and Alphonse have attempted comprehensive solutions to the problem by designing their own language and describing the low level implementation, at the same time. Message interchange approach and graph based approaches describe the implementation strategies for incremental operations. Finite differencing visualizes the problem as a program optimization issue.

It would be pertinent to note here that all these techniques try to arrive at a consensus on trade-offs between issues like storage requirement, time complexity, granularity and, above all, the level of incrementality. A fact that needs to be mentioned at this stage, is that none of the above mentioned methods has emerged as a universal technique as far as applications are concerned. With faster hardware platforms and advances in parallel and distributed computing, acceptable and optimal algorithms for conversion of batch programs into their incremental versions still remains an open problem. In the following chapters, we take a detailed look at Graph based evaluators and particularly the ones where the graph is based on the Attribute Grammar formalism.