TECHNICAL REPORT

No. 02-11

Incremental context-sensitive
evaluation in context

Phil Cook    Jim Welsh
Ian Hayes

September 2002

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au
Note: Most SVRC technical reports are available via anonymous ftp, from svrc.it.uq.edu.au in the directory /pub/techreports. Abstracts and compressed postscript files are available via http://svrc.it.uq.edu.au
Incremental context-sensitive evaluation in context

Phil Cook           Jim Welsh           Ian Hayes

Abstract

Although techniques for implementing or generating incremental semantic evaluators have been explored and refined for more than two decades, several pragmatic concerns still impede the use of such techniques in practical development environments. This report not only addresses some of these concerns, but furthermore demonstrates the need to consider the problem of incremental semantic evaluation in context. The practical concerns addressed here stem from both user interaction and architectural requirements. In particular an innovative preemptive evaluation scheme is presented which helps to reduce delays associated with semantic evaluation over a sequence of edits. Furthermore, a technique for assigning attributes to syntactically erroneous material (the introduction of which is inevitable in a syntax recognition editor) is described, as well as a novel approach to handling "long-distance" semantic effects using fine-grained incremental evaluation of relations.

1 Introduction

Language-based editing systems have the potential to provide software engineers with many benefits, by exposing language-sensitive program information to the user via an appropriate interface, and keeping such information up to date in response to user edits.

This report describes new techniques for generating incremental semantic evaluators for use in such environments, and the implementation of those techniques in the UQ* generic language-based environment. Although incremental semantic evaluation has been a subject of research for many years, it has yet to gain widespread adoption outside the research community. We argue that there are several pragmatic concerns yet to be addressed in this area which impede the use of incremental semantic evaluation in realistic editing systems. The context in which an incremental evaluator is to be used places certain demands upon its performance and the means by which it exposes the semantic information it computes. Hence, the design of successful incremental evaluation schemes requires an understanding of this context.

The techniques presented here are designed for use in a language-based editing system that supports a modeless syntax recognition editing paradigm, with a flexible and generic architecture for tool integration. The scheme presented enables the construction of loosely-coupled evaluators that operate cohesively by building and/or augmenting shared program representations. Relations that can span arbitrary syntax tree nodes are employed to capture semantic information in these program representations. The generated tools are also designed to be tolerant of the syntax errors that are inevitable in an editing system based on syntax recognition. Moreover, since a modeless editing paradigm is assumed, each keystroke is a complete editing operation and a technique for performing efficient incremental evaluation in this setting is discussed. A prototype implementation of these techniques has been developed as part of the UQ* language-based environment [29, 3].

The requirements for incremental evaluation in UQ* are described in Section 2, and an overview of the approach proposed to meet these requirements is given in Section 3. A brief
tour of the salient aspects of UQ’s Environment Description Language (EDL), from which incremental attribute evaluators can be generated, is given in Section 4. Section 5 describes the basic strategy for incremental evaluation. A novel technique for fine-grained incremental evaluation of relations is given in Section 6. This technique solves the performance problems related to “long-distance” semantic effects with aggregate values (e.g., symbol tables), and demonstrates the utility of relations for augmenting shared program representations. Section 7 details techniques for coping with user interaction via modeless syntax recognition. In particular, an approach to attributing syntactically erroneous material is given, as well as an innovative preemptive evaluation scheme. The latter is designed to aid evaluators in keeping pace with user input, while not sacrificing automatic evaluation. Section 8 presents the results of an empirical performance evaluation of the prototype implementation of the techniques presented here. In Section 9 previous approaches to incremental semantic evaluation are described and compared to the approach advocated here. Section 10 summarizes and concludes the report.

2 Requirements for incremental evaluators

Incremental semantic evaluation is a problem that must be considered in context. User interface, architectural, and ideological issues all contribute to the requirements set out here. These requirements were formulated with the UQ language-based environment in mind. UQ acts as a useful vehicle for exploring user needs for incremental systems because it lies at the extremes of the user interface, architectural, and ideological spectra.

2.1 Coping with modeless editing

The modeless syntax recognition editing paradigm assumed here places significant demands on the nature of incremental tools.

1. *Tools must accommodate syntactically erroneous and incomplete programs.* In a syntax recognition editor syntax errors are inevitable, and under a modeless editing paradigm they tend to arise frequently. Elsewhere we have described an incremental parsing strategy that constructs an error-tolerant syntactic program representation [3]. Incremental evaluators must also be tolerant of syntax errors, and must handle the representation of erroneous sub-trees built by the parser. Syntax errors must not preclude the evaluation of attributes in the surrounding tree, as this could hide important semantic errors from the user as well as preventing further analysis or translation by other tools. Furthermore, due to the frequent occurrence of syntax errors under modeless editing, syntax errors must be handled efficiently. Although, in many cases, it may be undesirable or impossible to evaluate erroneous sub-trees themselves, tools should maintain semantic information associated with a sub-tree that moves into error, and should exploit this information if the sub-tree is later reused in a well-formed context.

2. *Tools must be able to respond to each keystroke.* Although the strategy presented here is intended to be applicable to a wide class of tools, one of the primary applications is the construction of static semantic analyzers. Given that syntax analysis is performed on each keystroke, it is fitting that any semantic analyzer associated with a language should also operate automatically in response to each keystroke. This concern stems from
the premise that static semantic checking is as important to the user as syntax checking. Hence, feedback should be available after each editing operation.

3. **Tools must minimize delays over sequences of keystrokes.** For the feedback provided by an interactive tool to be of benefit, it is crucial that delays between edit operations and the display of any feedback from them be minimized. In static semantic analysis, however, many changes to a program can have far-reaching consequences. Even the most efficient incremental evaluation techniques can incur large delays in response to certain changes. Such delays pose a significant problem in a modeless editing system. Clearly it is useless to mandate that all tools should achieve any particular level of responsiveness, as the very nature of semantic analysis would make any such requirement impossible to meet.

A more realistic requirement is that tools should attempt to reduce the delay associated with a sequence of editing operations. Given a sequence of changes to a document, the cost of evaluating each in turn is often greater than the cost of a single evaluation of their composition. Such situations should be exploited by tools to avoid the problem of an ever increasing delay between the user's keystrokes and the feedback provided in response to them.

### 2.2 Communicating with the Environment

The results computed by an incremental tool must be made available to the rest of the environment in some way.

4. **Tools should provide their output by constructing and/or augmenting shared program representations.** The architecture for constructing incremental compilation systems for UQ* is based on a blackboard model, where multiple tools cooperate via a central document server (see Figure 1). Tools communicate through shared program representations. These representations are constructed from abstract syntax trees and relations that can span arbitrary nodes in those trees. For the results of an analysis or translation activity to be of use to other tools in the environment it is necessary that these results are made available from the construction or augmentation of shared program representations.

5. **Some tools must construct and/or augment program representations that are suitable for user inspection and navigation.** This concern is particularly apparent with analysis tools, where many analysis results are useful to both humans and other tools (e.g., declaration-use relationships). The most effective way to enable the construction of such tools is to use representations that capture program information in a form that is suitable for both humans and tools; in particular, relations.
2.3 Tool specification

UQ* is a generic environment, providing facilities for description of syntactic and relational structure of documents (source code documents being the primary interest here), as well as presentation and navigation rules [29]. Generic implementation is a cost-effective method of supporting multiple languages which exploits commonalities between them and helps to ensure the consistency of multiple tools with respect to a particular language. The natural extension of this ideology is the following requirement.

6. **Tools should be generated from declarative specifications.** Generation of tools from declarative specifications is widely recognized as an important cost-saving technique in the development of language-based environments. Employing a declarative description language helps to shield the tool builder from the complexities of incremental tool operation. Declarative languages, such as attribute grammars [17, 18], enable incremental computation by allowing sub-computations to be performed in isolation, without any dependence upon, or effect on, global state. A suitably designed language does not impede the asymptotic running time of evaluations, and thus can still meet the performance requirements set out above.

3 Overview of the approach

The system described here is based on a synthesis of attribute grammar and relational techniques. Syntheses of this form and their benefits have been discussed in detail elsewhere [11, 20, 22]. The primary reason this approach is adopted stems from the premise that attribute grammars provide an elegant and powerful method of specifying semantic aspects of a language while being amenable to incremental evaluation, and that relations provide a superior form of semantic feedback to attributes alone. We strengthen these ideas with concerns which arise from the blackboard architecture we propose: that tools must construct and/or augment shared program representations, and that relations are the primary means by which program representations may be augmented. As discussed above, the relations used to augment a representation in this approach should generally be useful not only to the user, but to other tools in the environment.

The particular combination of attribute grammar and relational concepts advocated here is derived largely from that presented by Maddox [22], as are many of the implementation techniques used. The meta-language presented below is heavily influenced by that of Maddox, and bears a striking resemblance at the abstract syntax level. The concrete syntax of the language is different, however, as it is designed as an extension of UQ*'s existing Environment Description Language (EDL) [29].

Two flavours of relations are provided: external and internal relations. External relations (in combination with the syntax tree being evaluated) form the interface of a tool; they are the relations that are used to communicate analysis results to the environment; e.g., definition-use relationships and error annotations. Internal relations act as aggregate attributes that are accessed “by reference”. They can be used to model information such as symbol tables efficiently and conveniently. While the treatment of relations is similar in notation to that proposed by Maddox, their implementation is superior, permitting fine-grained incremental evaluation.

Implementation of the attribute grammar aspects of the system follows the visit caching approach of Maddox. It is a simple technique that is based on memoization of the computations
performed by visit procedures. It permits multiple sub-tree replacements, and is amenable to efficient evaluation in the presence of the long-distance propagation effects inherent in semantic analysis.

One of the main innovations in the technique presented here is its support for preemptive evaluation: a scheme that enables efficient operation under modeless editing by allowing an evaluation to be restarted upon the arrival of new input. This scheme is made possible by the choice of visit caching, effectively making evaluators generated in this way capable of handling multiple sub-tree replacements where not all the replacements are known prior to commencing an evaluation.

To meet the requirements of syntax recognition editing, semantic evaluators must also be capable of attributing syntactically erroneous program fragments. The scheme presented here is based on a simple approach to attributing erroneous constructs that exploits default attribution rules provided in a tool description. These default rules are applied to certain tree nodes that form part of UQ*’s error-tolerant syntax tree representation [3].

This combination of features is derived from the requirements described above. The requirements for coping with modeless editing have a direct impact on the evaluation scheme; this is reflected in the detailed exposition of features relevant to those requirements. The requirements for communication with an evaluator’s environment are dealt with by the scheme’s fine-grained incremental treatment of relations, which assists the tool builder in developing tools that inter-operate effectively with their environment. Genericity is achieved by using attribute grammar technology combined with relations; the latter providing a powerful and efficient aggregate data type without appealing to ad hoc techniques.

4 A language for describing tools

To satisfy the requirement that tools should be generated from declarative specifications (requirement 6) a meta-language for tool specification is required. This section tours the meta-language used in UQ*, which is an extension of UQ*’s Environment Description Language (EDL) [29].

Each tool description has a header which describes the structures that the tool uses and provides. Each tool may use one syntax tree and several relations. A tool always accesses a syntax tree via a view [29] that is used to specify a more abstract syntax than is usually present in a document’s base (typically concrete) representation. Tools may provide both relations and syntax trees (translation tools may be specified as tools that provide a syntax tree). For example, the header for a semantic analyzer for the hypothetical language PL0 could take the following form, in which the imported PL0_Relations is defined below.

```plaintext
import relations PL0_Relations .
import TK_NONE, TK_INT, TK_BOOL, TK_STRING from relations PL0_Relations .

tool checker
  uses view PL0::default
  provides relations
    PL0_Relations::Error, PL0_Relations::DeclUse, PL0_Relations::TypeInfo .
```

1The full range of syntactic transformations permitted by views have yet to be implemented in UQ*. As such, the examples presented here are based on concrete syntax.
Relations mentioned in this way are external relations. A relation may not be both used and provided by the same tool, otherwise ill-formed (circular) tool descriptions could be constructed, since a relation could be defined to depend (directly or indirectly) upon itself. This restriction precludes some well-formed relation definitions in favour of efficient static checking of tool definitions. In this example, the relations provided by the tool are declared in the relation collection PL0_Relations as follows, where parseTree and string are primitive types in the meta-language.

```c
import language PL0.
relations PL0_Relations.

/*@ Errors */
relation Error(parseTree, string).

/*@ Semantic structure */
relation DeclUse(PL0::VarDecl, PL0::UseIdent).

/*@ Type representation */
enum TypeKind { TK_NONE, TK_INT, TK_BOOL, TK_STRING }
relation TypeInfo(PL0::VarDecl, TypeKind).
```

Attribution of the tree is described in two parts: lexeme and phyllum declarations, and rule definitions. Each variable-spelling terminal symbol and each non-terminal symbol in the abstract syntax is given a lexeme or phyllum declaration respectively. Phyllum declarations define the attributes associated with a tree node, their types, and default values. Lexeme and phyllum declarations are illustrated in the following example, which also shows the definition of the polymorphic typeBindable that is used for representing language elements which may be bound to names.

```c
lexeme ident.
lexeme number.

/*@ ConstDef, VarDecl, and ProcDecl are phyla of the PL0 language */
datatypeBindable = const_binding(ConstDef)
                 | var_binding(VarDecl)
                 | proc_binding(ProcDecl).

phyllum ForStatement {
    context:
        relation Bindings(string, Bindable).
        relation VarTypes(VarDecl, PL0_Relations::TypeKind).
    attributes:
        boolean has_errors.
    defaults:
        this->has_errors = true.
}
```
Each lexeme has a predefined attribute spelling of type string that can be used to obtain the character sequence representing a lexical symbol. The inherited and synthesized attributes of a phylum are declared in context and attributes sections, respectively. In the example above, Bindings and VarTypes are both internal relations that are used to represent symbol table information. The defaults section of a phylum declaration merely indicates the default values of the phylum's synthesized attributes used when the phylum appears as a placeholder (this occurs when there is a neighbouring syntax error, as described in Section 7.1).

Attributes and fields within relation tuples can be either simple values or references. The value types supported are primitive types (such as integer or string), enumeration types (e.g., TypeKind above), structured disjoint-union types (e.g., Bindable above). Objects accessed by reference are either tree nodes or relations. References to tree nodes, whether they are attributes or fields of relation tuples, do not expose the attributes of the nodes to which they refer; i.e., the attributes of a node accessed via a reference cannot be used in attribute equations. This constraint enables the use of efficient static evaluation plans (see Section 5 below).

Rule definitions are used to define attribute equations for each form that a phylum may take. If the phylum ForStatement above has the following syntax:

ForStatement = "for" UseIdent "::=" Exp "to" Exp "do" Statement .

then one rule must be constructed, as shown below.

rule ForStatement = "for" UseIdent "::=" < e1: Exp > "to" < e2: Exp > "do" Statement {
    UseIdent->Bindings = this->Bindings .
    e1->Bindings = this->Bindings .
    e2->Bindings = this->Bindings .
    Statement->Bindings = this->Bindings .
    ...}

PL0_Relations::Error(this, UseIdent->name + " is not an integer") :-
    UseIdent->type_info != TK_INT, UseIdent->type_info != TK_NONE .

this->has_errors =
    UseIdent->has_errors ||
    e1->has_errors ||
    e2->has_errors ||
    Statement->has_errors ||
    UseIdent->type_info != TK_INT .
}

The header of a rule describes the form of the syntactic construct to which it applies. Syntax elements on the right hand side of the rule may be named to avoid ambiguity (e.g., e1 and e2 above), but where names have been omitted and no ambiguity exists they may be referred to by the type of the symbol. Constraints in the body of a rule are either attribute equations, relation clauses, or switch constraints (see below). Attribute equations and relation clauses can be given guards, which have a Prolog-like syntax. A reference to an attribute is always
qualified by the name of the symbol to which it belongs (attributes of the phylum to which the rule applies are qualified by the keyword this). External relations are global to a tool description and are not prefixed by a symbol name; e.g., PLO_Relations::Error above. The clauses that define an external relation are distributed throughout the attributed tree, whereas an internal relation is defined by clauses at a single rule instance.

Relation-valued attributes (i.e., references to internal relations) may be defined either by a copy rule (e.g., the treatment of Bindings above) or by relation clauses. In the former case, a reference to the relation is copied. An example of a clause defining a relation-valued attribute is given below. Here the relation NewBindings is defined to bind a name to a variable declaration.

```plaintext
rule VarDecl = DeclIdent "::" ident {
  this->NewBindings(DeclIdent->name, var_binding(this)) .
  ...
}
```

In this case a single tuple is asserted, but in general, clauses can use pattern matching and guards to construct larger relations. The following example illustrates how this can be applied to merge symbol table information from a sequence of variable declarations. It also shows the notation for switch constraints mentioned earlier. The use of "< rep: {..} >" in the rule’s header associates the name "rep" with an otherwise anonymous repetition non-terminal.

```plaintext
rule Variables = VarDecl "::" < rep: {..} > {
  forall (string s, Bindable b)
    this->NewBindings(s, b) := VarDecl->NewBindings(s, b) .
  forall (string s, Bindable b)
    this->NewBindings(s, b) := rep->NewBindings(s, b) .

  ...

  switch (rep) {
    attributes:
      relation NewBindings(string, Bindable) .
      relation NewVarTypes(VarDecl, PLO_Relations::TypeKind) .
    case empty:
      rep->NewBindings = empty .
      rep->NewVarTypes = empty .
    case VarDecl "::":
      rep->NewBindings = VarDecl->NewBindings .
      rep->NewVarTypes = VarDecl->NewVarTypes .
    case left ^ right:
      forall (string s, Bindable b)
        rep->NewBindings(s, b) := left->NewBindings(s, b) .
      forall (string s, Bindable b)
        rep->NewBindings(s, b) := right->NewBindings(s, b) .
    ...
  }
}
```

8
A switch constraint is used to handle syntactic elements that are used in the EBNF syntax definition style of UQ+ and not found in BNF syntax rules upon which attribute grammars are usually based. They can be used to describe analyses for optional constructs, sub-rules, and repetitious structure. In the first two cases a switch constraint can be viewed merely as syntactic sugar. For repetitions (as shown above), however, they are used to facilitate a vital efficiency improvement: the representation of repetitious structure as balanced binary trees. This improvement is used to guarantee the logarithmic access times needed in incremental applications, as described originally by Pugh and Teitelbaum [23], and further by Wagner and Graham [32] and Maddox [22]. A switch constraint effectively defines an attribution scheme for an anonymous phylum. Each switch constraint may define inherited and synthesized attributes for its anonymous phylum in context and attributes sections. One case section is required for each syntax rule defining the anonymous phylum.

5 Incremental attribute evaluation with visit procedures

This section details an incremental attribute evaluation strategy based on the evaluator presented by Maddox [22]. Although this section is based largely on Maddox's work, a detailed description of the basic evaluation strategy is necessary to understand the more novel features detailed in the sequel. Furthermore, this section serves to highlight the key design decisions that were necessary to enable the development of these more novel features. The aspects of the evaluation strategy discussed in this section are concerned primarily with the more conventional parts of the semantic meta-language presented above. Discussion of evaluation of relation-valued attributes is deferred to Section 6, and details of handling syntax errors and preemptive evaluation are given in Section 7.

The evaluation scheme advocated here is based on a certain class of “tree walking” evaluators developed for ordered attribute grammars [15]. The scheme is simple and efficient due to the use of statically computed visit sequences, or evaluation plans, and it can evaluate a sufficiently large class of attribute grammars to be applicable to practical programming languages. The use of pre-computed visit sequences reduces the amount of data that must be maintained by the evaluator at run-time. Multiple sub-tree replacements are accommodated in the scheme in a simple, elegant, and efficient manner. Furthermore, by avoiding the maintenance of complex scheduling information at run-time, the scheme is easily extended to form a more sophisticated preemptive evaluator, as discussed in Section 7.2.

For each rule in an ordered attribute grammar (OAG) we can construct a visit sequence using the algorithm described by Kastens [15], although the notation used here for describing visit sequences is that used by Reps [26]. A visit sequence defines an evaluation plan for a rule instance (tree node) by defining an order in which to:

- evaluate synthesized attributes of the node (denoted by EVAL(0, a), where a is the name of a synthesized attribute),
- evaluate inherited attributes of child nodes (denoted by EVAL(i, a), where a is the name of an inherited attribute of the ith child),
- perform visits to child nodes (the rth visit to the ith child is denoted by VISIT(i, r)),

and
Figure 2: UML class diagram depicting visit procedures and attributes for rule and phyla classes

- perform visits to the parent node (denoted by SUSPEND(r), where r is the number of the current visit).

The instructions in a visit sequence can be grouped into subsequences terminated by SUSPEND instructions. Each such subsequence describes one visit. There are a number of ways to implement visit sequences; the strategy adopted here is based on mutually recursive procedures. The strategy is described in an object-oriented style to facilitate a simpler explanation and because it corresponds closely with the implementation of these techniques in UQ*. For reasons of exposition the implementation of visit sequences for a non-incremental evaluator is discussed first, followed by a description of how the technique can be extended to provide incrementality.

5.1 Non-incremental evaluation

Each node in the attributed tree is considered as an object in the underlying implementation language, as shown in Figure 2. An abstract class is generated for each phylum of the OAG, with each attribute of the phylum represented as a field in that class. A concrete sub-class of the phylum’s class is generated for each rule of that phylum. This arrangement enables the use of dynamic binding to aid the implementation of visit sequences.

The subsequence of instructions for each visit can be translated into a method in the underlying implementation language. Each of these methods, or visit procedures, is obtained by a straightforward translation from the EVAL and VISIT instructions for the appropriate subsequence. A visit procedure has no return value and takes no parameters, other than the implicit “self” parameter representing the node object to which it is applied. Each EVAL instruction is translated into an assignment statement for the relevant attribute (the left hand side of the assignment will involve a reference to a child node in the case of inherited attributes), and each
VISIT instruction is translated into a method invocation of the appropriate visit procedure for a child node. For example, if a rule has the subsequence

\[
\text{EVAL}(3, w), \text{VISIT}(3, 5), \text{EVAL}(0, x), \text{EVAL}(4, y), \text{VISIT}(4, 1), \text{EVAL}(0, z), \text{SUSPEND}(2)
\]

for visit 2, and is represented by the class Rule1 Instance, then a method of the following form would be generated, where \( f_1, \ldots, f_4 \) are the relevant semantic functions.

\[
\text{method Rule1 Instance.visit}_2; \\
\text{begin} \\
\quad \text{child}[3].w \leftarrow f_1(...); \\
\quad \text{child}[3].\text{visit}_5; \\
\quad x \leftarrow f_2(...); \\
\quad \text{child}[4].y \leftarrow f_3(...); \\
\quad \text{child}[4].\text{visit}_1; \\
\quad z \leftarrow f_4(...) \\
\text{end}
\]

Since a phylum may have more than one rule, when a sub-visit is performed on a child node the visit procedure for the relevant rule must be selected. Dynamic binding may be exploited to this effect. If each rule for a particular phylum has visit methods named \( \text{visit}_1, \ldots, \text{visit}_k \), then corresponding abstract visit methods with the same names appear in the phylum class, as shown in Figure 2.

To evaluate all the attributes in a tree using this scheme, all that is required is an invocation of the single visit method for the root node of the tree. The visit methods will walk the tree recursively calculating attribute values in the correct order.

5.2 Incremental evaluation

The evaluation scheme outlined above is easily extended to accommodate incremental evaluation. The incremental evaluator presented here is based on memoization of visit procedures.

Consider a visit procedure in terms of its “inputs” and “outputs”. The inputs to a visit procedure are inherited attributes computed by previous visits to the parent node, synthesized attributes computed by previous visits to the current node, and the sub-tree rooted at the current node (this accounts for any computation which is performed by sub-visits from the current node). The outputs of a visit procedure are the synthesized attributes it computes. The key idea of the memoizing incremental evaluator is that if the inputs to a visit procedure have the same value as during the previous evaluation, then the synthesized attributes computed by the visit will be identical to those computed previously. In this situation the visit (and all its sub-visits) need not take place.

To enable memoization of visit procedures the inputs and outputs of visit procedures must be stored in tree nodes, and an efficient means of comparing inputs is required. For synthesized and inherited attributes this presents no problem (the non-incremental evaluator described above stores all attribute values anyway). Efficient comparison of sub-trees is more problematic. Rather than perform a structural comparison of sub-trees a simple technique for marking changed sub-trees is employed. Each tree node is endowed with a \text{need}_\text{visit} flag which is used to control the walk performed by the evaluator.

The input to the overall evaluation process is a set of sub-tree replacements. Before updating attribute values the evaluator grafts the replaced sub-trees to the appropriate locations in the
tree. All the nodes within the replacements have their \textit{need\_visit} flags set. Furthermore, to ensure that the tree walk will reach all the replacement sites, the \textit{need\_visit} flag is also set in all the ancestors of each replacement site. Sub-tree replacements received by a tool are in an efficient form of “tree differences”, where maximal sub-trees of the replacement that are unchanged from the previous version of the tree are transmitted as \textit{reuse markers}. Any sub-tree that is reused in this way can have its attribute values reused as well, since the \textit{need\_visit} flag will not be set in reused nodes unless their inherited attributes change.

Visit procedures for incremental evaluation are generated similarly to the non-incremental case. The key differences are that as inherited attributes are calculated they are compared with their previous values, and \textit{need\_visit} flags are set and examined as required. Visit sequence instructions for evaluating inherited attributes, i.e., of the form EVAL\((i, a)\), where \(i > 0\), are translated into code of the form:

\[
\begin{align*}
tmp_{b,a} & \leftarrow f(x_1, \ldots, x_k); \\
\text{if } tmp_{b,a} \neq \text{child}[i].a \text{ then} & \\
& \quad \text{child}[i].\text{need\_visit} \leftarrow \text{true}; \\
& \quad \text{child}[i].a \leftarrow tmp_{b,a} \\
\end{align*}
\]

where \(tmp_{b,a}\) is a variable not used elsewhere in the visit procedure, \(f\) is the semantic function used to evaluate the attribute, and \(x_1, \ldots, x_k\) are the attribute values used as arguments to \(f\). This code template ensures that if any inherited attributes of a node are changed during reevaluation, then all subsequent visits to that node will be performed.

The code template below shows the structure of a memoized visit procedure corresponding to the \(r\)th visit to a node. The boxed line of code is only generated for final visit procedures: a final visit must reset the node’s \textit{need\_visit} flag to ensure that the node is not revisited unnecessarily on subsequent evaluations.

\begin{verbatim}
method SomeRuleInstance.visit_x;
begin
  if need_visit then
    {Evaluate attributes and perform sub-visits}
    need_visit \leftarrow false \{Final visits only \}
  end if
end
\end{verbatim}

To demonstrate the correctness of this incremental evaluation scheme, it is necessary to show that the node marking facilitated by \textit{need\_visit} flags ensures that all affected attributes are reevaluated in response to an edit. Suppose we have a set \(Changed\) that, at any point in the evaluation, contains all the attribute instances that have changed value. Initially \(Changed\) is equal to \textit{Newborn}, the set of all new attribute instances introduced to the tree.

The visit \(i\) to sub-tree \(t\) only needs to be performed if \(\text{Args}_{i,t} \cap Changed \neq \emptyset\), where \(\text{Args}_{i,t}\) is the set of attribute instances that are arguments to the attribute instances computed (directly or indirectly) by visit \(i\). The only arguments to visit \(i\) that could have changed are inherited attributes of \(t\) and attributes computed (directly or indirectly) by previous visits to \(t\). Therefore the above condition can be weakened to

\[
I_i \cap Changed \neq \emptyset \lor \text{Computed}_{i-1,t} \cap Changed \neq \emptyset
\]

where \(I_i\) is the set of inherited attribute instances of \(t\), and \(\text{Computed}_{i-1,t}\) is the set of attribute instances computed by the first \(i-1\) visits to \(t\).
Whenever an attribute in $I_t$ is changed, $t.need_visit$ is set to true. Therefore,

$$I_t \cap Changed \neq \emptyset \Rightarrow t.need_visit$$

The attributes in $Computed_{i-1,t} \cap Changed$ either must have been evaluated on a previous visit to $t$, or must be in Newborn. If an attribute was reevaluated in a previous visit to $t$, then $t.need_visit$ must be true (otherwise the visit would have been skipped). When an edit occurs, each ancestor of each node containing an attribute in Newborn has its $need_visit$ flag set. Therefore,

$$Computed_{i-1,t} \cap Changed \neq \emptyset \Rightarrow t.need_visit$$

and hence,

$$Args_{i,t} \cap Changed \neq \emptyset \Rightarrow t.need_visit$$

Since the evaluation scheme only skips a visit to $t$ if $t.need_visit$ is false, it will correctly recompute all necessary attributes.

The number of attribute instances reevaluated by this scheme may be reasoned about in terms of the sets affected, that is the set of attribute instances whose values change in response to an edit, and edit_ancestors, that is the set of tree nodes that are ancestors of all the replacement sites for an edit. If an attribute instance is reevaluated it must belong to a tree node that has its $need_visit$ flag set at some point during the evaluation. That tree node will either be a member of edit_ancestors or will have at least one attribute instance that has changed. Hence the number of tree nodes with attributes that are reevaluated is $O(|affected| + |edit_ancestors|)$. Therefore, under the assumption that there is some constant bound for the number of attributes per phyllum, the number of attributes reevaluated is $O(|affected| + |edit_ancestors|)$. In the worst case this is equivalent to $O(|affected| + k.h)$, where $k$ is the number of sub-tree replacements and $h$ is the height of the tree. Judicious use of repetitions (rather than recursion) in the EBNF grammar allow the tree structure generated to be of height $O(\log N)$, where $N$ is the number of nodes in the tree.

6 Fine-grained incremental evaluation of relations

Relations that are able to span arbitrary tree nodes (as well as refer to primitive values) play two roles in the scheme presented here. External relations are part of the interface of an evaluation tool; they are the primary means by which a tool can augment a program’s representation with semantic information. Internal relations provide similar power for representing semantic information, but are used as attribute values within a tool. Both types of relation have an implementation that permits efficient propagation of changes to semantic information.

Relations are implemented as objects in the underlying implementation language and relation-valued attributes as references to these objects. Treating relation-valued attributes as references enables efficient treatment of aggregate values; e.g., if a symbol table is represented by a relation, then changes to the declaration of a variable can be propagated to only the sites of its use rather than throughout the entire scope, since the tuples of which the relation is composed can change without changing the value of any references to it. A form of differential propagation is employed to propagate changes to the tuples in a relation to affected locations in the attributed tree. The combination of pass-by-reference and differential propagation results in a fine-grained treatment of aggregate values that uses statically computed (coarse) attribute
dependencies to ensure the well-formedness of the tool description, and refines those coarse attribute dependencies at run-time to gain the necessary efficiency.

Two distinct, yet complementary, implementation strategies are provided for internal relations: maintained and unmaintained. A maintained relation has an associated tuple cache that provides efficient querying, whereas queries of an unmaintained relation are handled by reevaluation of the relation’s constituent tuples.

6.1 The role of relations in incremental evaluation

As noted above, analysis and translation tools constructed using the techniques presented here must communicate with the environment in terms of shared program representations. The basic building blocks of representations in UQ* are abstract syntax trees and relations. The primary role of an analysis tool, however, is to augment some program representation with semantic information. The types of semantic information that a tool should make available can be readily captured by relations that span arbitrary tree nodes, and fall into the following three categories.

Error annotations. Information about the presence of static semantic errors can be captured as tree annotations. For example, a binary relation Error can be used to associate the point where an error was encountered (i.e., some tree node) with an error message.

Semantic annotations. Much of the semantic information computed by an analysis tool is in the form of semantic annotations or attributes associated with tree nodes. Relations can be used to map tree nodes to semantic values, such as type information. A tool can use this technique to make some of the semantic information it computes available to other tools.

Structural information. The most interesting application of relations in semantic analysis is modeling the semantic structure of a program. This category of semantic information is concerned with capturing relationships among program components; e.g., declaration-use relationships.

The environment builder clearly has many decisions to make regarding the modeling of semantic information. These decisions can be summarized as what information should be made available by a tool, and how that information should be captured. The environment builder should be aware of the compositional power of relations; this can often be used to combine information from the latter two categories above. For example, type information need not be associated with each use of a variable, but can be readily available through the composition of declaration-use relations and type information relations. Although internal relations do not play the same role as external relations do, they have the same power for representing semantic information. Their use enables efficient treatment of internal representations and can aid the definition of external relations.

6.2 Maintained vs. unmaintained relations

The attribute evaluation scheme described above assumes that the values of all attributes are stored in the tree. Imposing this requirement on the contents of internal relations is not necessarily wise for reasons of space efficiency. Consider the definition of the NewBindings relation described in Section 4:
case < left ^ right >:
  forall (string s, Bindable b)
    rep->NewBindings(s, b) :- left->NewBindings(s, b) .
  forall (string s, Bindable b)
    rep->NewBindings(s, b) :- right->NewBindings(s, b) .

If each NewBindings relation was implemented as a maintained relation then each node of
the repetition would contain a tuple cache containing duplicates of all the tuples provided by
its two children. In this instance, maintaining all the tuples of each NewBindings relation is
unnecessary: the differential propagation technique described below handles this case efficiently
without the aid of a tuple cache.

Therefore, the decision as to which internal relations are maintained is left to the envi-
ronment builder: the maintained modifier for relation declarations may be used to this end.
A relation declared as maintained has a tuple cache associated with it that enables efficient
querying.

Maintained relations are best used selectively. Relations that are queried at many sites,
such as relations that represent symbol tables for a particular scope, are good candidates for
declaration as maintained, whereas relations that are primarily used to synthesize tuples from
two or more relations are best left unmaintained.

6.3 Differential propagation

Fine-grained incremental evaluation of relation queries is achieved by a form of differential
propagation. Given an attribute A that depends on a relation R, A must be updated in
response to changes in R. Differential propagation exploits the possibility that A may depend
on only a subset of the tuples in R, ensuring that A is only reevaluated in response to changes
to the relevant tuples in R. In the case that A is itself a relation, changes are propagated
efficiently through A to any sites that query it.

Although relations are traditionally treated as sets of tuples, differential propagation requires
them to be treated as bags (or multisets). The Prolog-like notation for relation definitions
permits different clauses of a relation to contribute tuple sets that overlap. In a non-incremental
evaluator, set union would eliminate duplicate tuples. For differential propagation to operate
correctly, however, these duplicate tuples should not be lost. For example, consider a relation
definition of the form:

forall (integer x, integer y) this->R(x, y) :- this->S(x, y) .
forall (integer x, integer y) this->R(x, y) :- this->T(x, y) .

and suppose the tuple (0, 0) occurred in both S and T, and hence also in R. If that tuple
were to be removed from S, but not from T, it should remain in R. If R was represented as a set
it would be impossible to achieve this without reevaluating T.

Differential propagation relies on deltas to represent changes to a relation in terms of tuples
that are removed from and added to it. This information may be conveniently captured using
generalized bags [6], allowing positive, negative, and zero occurrences of tuples. Conceptually a
generalized bag is a partial function from bag elements to the non-zero integers. Let \( B \vdash x \) denote
the number of occurrences of \( x \) in the generalized bag \( B \), and \([ ]\) denote the empty generalized
bag.
Generalized bag addition and subtraction are used throughout the following section for manipulation of deltas and relations. Generalized bag multiplication is used in the calculation of deltas for relation clauses, as discussed in Section 6.4. Generalized bag addition, subtraction, and multiplication are defined, respectively, as:

\[
(B \uplus C) \uplus x = B \uplus x + C \uplus x \\
(B \setminus C) \uplus x = B \uplus x - C \uplus x \\
(B \cap C) \uplus x = B \uplus x * C \uplus x
\]

Note that \((G, \uplus, \cap, \setminus)\), where \(G\) is the set of all generalized bags, is a ring.

Differential propagation relies on storage of dependency information at run-time. Relations maintain a set of query site objects to represent any locations in the attributed tree which depend upon them. A query site consists of

- a tuple template, which describes a query of the relation (tuple templates are similar to tuples except that they may contain “holes” to represent variables in the query),

- a pointer back to the tree node from which the query originates,

- a delta which is used to accumulate changes to the queried relation for later use at the location of the query.

A change to a relation is represented as a delta. A delta is either a generalized bag of tuples, or the special value \(\top\) which represents an arbitrary change to a relation. A \(\top\) delta is propagated whenever a more accurate delta cannot be determined; e.g., due to changes to the underlying definition of a relation. A delta may be filtered using the tuple template associated with a query site. Non-empty deltas that have been filtered in this way are associated with query sites to facilitate fine-grained incremental reevaluation of any affected queries. In the algorithms presented here, the functions \texttt{MergeDeltas} and \texttt{FilterDelta} are used to aid understanding. They are defined as follows, where \(\delta, \delta_1, \) and \(\delta_2\) are deltas (generalized bags), and \(tt\) is a tuple template.

\[
\begin{align*}
\text{MergeDeltas}(\top, \delta) & = \top \\
\text{MergeDeltas}(\delta, \top) & = \top \\
\text{MergeDeltas}(\delta_1, \delta_2) & = \delta_1 \uplus \delta_2 \\
\text{FilterDelta}(tt, \top) & = \top \\
\text{FilterDelta}(tt, \delta) & = [x \mid x \in \delta \land x \text{ matches } tt]
\end{align*}
\]

Query sites are used both to observe and to accumulate changes to relations. A query site observes changes by "subsribing" to the changes relevant to it. Each relation object has \texttt{subscribe} and \texttt{unsubscribe} methods that are used to control the registration of query sites with it (similarly to the Observer pattern [5]). The changes of interest to a query site are further constrained by its tuple template. A query site accumulates (relevant) changes to a relation in its delta. This facilitates communication of changes from a relation to the sites that query it in a manner compatible with the statically computed visit sequences used to drive an evaluator.

Relations are defined by clauses in the meta-language, and each instance of a relation object in the target language is an aggregate of corresponding clause objects. A clause object is composed of query sites. Each query site helps define the relation that contains it, and observes the relation it queries. The relationships between relations, clauses, and query sites are shown in Figure 3. For example, given the clause:
forall (integer x, integer y) this->R(x, y) :- this->S(x, y).

there is a single query site corresponding to the query of this->S, and that query site plays two roles:

- it is related to the clause under the query_sites relationship shown in Figure 3, and thus helps to define this->R, and
- it observes this->S.

Each relation object has a method query which takes a tuple template as a parameter and returns the bag of tuples that match the tuple template. The query method is shown below. Note that each clause object has a method calculateTuples, which is generated from its definition, to enumerate the tuples provided by the clause.

**Method** Relation.query(tuple_template);
begin
return \[ \bigcup_{c \in \text{clauses}} c.\text{calculateTuples} (\text{tuple}\_\text{template}) \]
end

When a relation requires reevaluation its update method is called by the appropriate visit method; i.e., for a relation-valued attribute that is defined by one or more relation clauses, a call to the attribute’s update method is made in the body of the appropriate visit method, rather than an assignment to the attribute. The update method calculates a new delta for a relation by combining deltas computed for each of its constituent clauses. There are two main cases to consider here.

- If any attributes (relational or otherwise) that are used in the definition of a relation have changed then the relation needs to be reevaluated in its entirety. The potentially significant cost of a complete reevaluation is mitigated by the use of a mixture of maintained
and unmaintained relations. If the relation is maintained, then a new tuple cache is calculated, and the old and new caches are compared to construct a delta for propagation to all affected queries. If the relation is not maintained, then differential propagation is disabled, and $T$ is propagated to all dependent queries. Propagation of $T$ has a similar effect on dependent relations as changes to attribute values: if the dependent relation is unmaintained then $T$ deltas are also propagated to the sites where it is queried, otherwise the relation is reevaluated in its entirety. For this reason it is important that the environment builder uses an appropriate combination of maintained and unmaintained relations.

- Otherwise, the delta for each clause is determined by the $calculateDelta$ method of the clause. The $calculateDelta$ method of a clause is generated from its definition, and is similar to the $calculateTuples$ method, except instead of simply performing each query on the right hand side of the clause, the deltas for each query site are used to guide the evaluation. If a relation is maintained, then once the delta for it has been computed in this manner, its tuple cache is updated by applying the delta to it.

The update method is shown below. It requires three steps: calculating a delta ($calculateDelta$), optionally applying the delta to the tuple cache and converting $T$ into a more accurate delta ($applyDelta$), and propagating the change to dependent query sites ($propagateDelta$).

```
method Relation.update;
begin
    propagateDelta(applyDelta(calculateDelta))
end
```

The $calculateDelta$ method computes a delta for the relation and resets the deltas of the relation's constituent query sites. Three passes are made over the clauses of the relation. The first pass updates attribute values used by each clause (clauses maintain copies of any attribute values they use: those used in relation queries are stored in the appropriate query sites, the remaining values are stored as part of the clause itself; these values are updated relative to attributes accessed via the relation's tree_node field, which refers to the node in which the relation is defined). If any of these attributes have changed since the last time the relation was updated, then the delta that is computed must be set to $T$, since it is impossible to determine which tuples should be removed from the relation. As a query site is updated, the relation to which it is subscribed is changed as appropriate. If none of the relevant attributes have changed, however, then the second pass calculates the delta for each clause (as described in Section 6.4 below), and merges these to form the delta for the relation. The third pass resets the deltas of the relation's constituent query sites.

```
method Relation.calculateDelta;
begin
    delta ← [];  
    for clause ∈ clauses do
        for query_site ∈ clause.query_sites do
            Update the fields of query_site.tuple_template from attribute values associated with tree_node and its children; 
            if query_site.tuple_template has changed then
                delta ← T
        end if
    end for
end
```

18
end for;
Update any other attribute values used in clause from values associated with tree_node and its children;
if clause has changed then
delta ← ⊤
end if
end for;
if delta ≠ ⊤ then
for clause ∈ clauses do
delta ← MergeDeltas(delta, clause.calculateDelta)
end for
end if;
for clause ∈ clauses do
for query_site ∈ clause.query_sites do
query_site.delta ← []
end for
end for;
return delta
end
To illustrate the need for ⊤ deltas in calculateDelta, consider the following relation definition:

forall (integer x, integer y) this→R(this→t, x) :- this→S(x, y).

If the value of the attribute t were to change, then all the tuples in R must be removed and new ones computed. However, by the time R is reevaluated, the tuples to be removed cannot be determined (unless R is a maintained relation, as discussed below) because S may also have changed.

If the relation is unmaintained then the delta application step is trivial. A maintained relation, on the other hand, requires its tuple cache to be updated. If the delta to be applied is ⊤, then the tuple cache is reevaluated in its entirety, and a more accurate delta is computed by comparing the old and new values of the cache. Delta application is performed by the relation’s applyDelta method, the implementation of which is specialized for unmaintained and maintained relations.

method UnmaintainedRelation.applyDelta(delta);
begin
return delta
end
method MaintainedRelation.applyDelta(delta);
begin
if delta = ⊤ then
Let templ be a tuple template which matches all tuples;
t ← ⋃_{c ∈ clauses} c.calculateTuples(templ);
delta ← t ⊓ tuple_cache;
tuple_cache ← t
else
tuple_cache ← tuple_cache ⊓ delta
end
end if;
return delta
end

The final step in the update process is to notify all the relevant query sites that depend upon the relation that it has changed. A filtered delta is constructed for each dependent query site and merged with the query site’s delta so a dependent relation may use it in its own update method. If the query site’s delta is non-empty after the merge then the notify method of the corresponding tree node is called to ensure the dependent attribute is scheduled for reevaluation. Two techniques discussed in Section 7 make it possible that a query site’s delta has not been consumed before a further change is made to the relation it queries. The first of these is the treatment of syntactically erroneous material, which can hide certain sub-trees from the evaluation process. The second is preemptive evaluation, which can restart an evaluation before it has been completed. Hence, filtered deltas are merged with existing deltas. Propagation is performed by the propagateDelta method, as shown below.

method Relation.propagateDelta(delta);
begin
for query_site ∈ observers do
query_site.delta ←
MergeDeltas(query_site.delta, FilterDelta(query_site.tuple_template, delta));
if query_site.delta ≠ [] then
query_site.tree_node.notify
end if
end for
end

The notify method schedules the tree node for visiting by the tree walking evaluator described above by ensuring that the need_visit flag is set in the node and all its ancestors.

method Node.notify;
begin
if not need_visit then
need_visit ← true;
if parent ≠ nil then
parent.notify
end if
end if
end

The correctness of the differential propagation scheme is demonstrated in two parts: the correctness of delta calculation, and the correctness of propagation.

The tuple bag for a relation is defined as \( R = \bigcup_{i \in 1..k} C_{R,i} \), where \( C_{R,i} \) is the tuple bag corresponding to the \( i \)th clause of \( R \). When \( R \) changes to a new value \( R' \), \( \delta_R \) must be computed such that either

- \( R' = R \uplus \delta_R \), or
- \( \delta_R = \top \).

The second case is trivial: it will arise whenever any attribute instances appearing in the definition of \( R \) have changed. In the first case, the calculateDelta method above will compute
the delta such that \( \delta_R = \bigcup_{i=1}^{k} \delta_{R,i} \), where \( \delta_{R,i} \) is the delta for the \( i \)th clause. This delta is correct according to the following:

\[
R' = \bigcup_{i=1}^{k} (C_{R,i} \uplus \delta_{R,i}) \\
= (\bigcup_{i=1}^{k} C_{R,i}) \uplus (\bigcup_{i=1}^{k} \delta_{R,i}) \\
= R \uplus (\bigcup_{i=1}^{k} \delta_{R,i})
\]

Furthermore, if the \texttt{calculateDelta} returns \( \top \), and the relation is maintained, then the delta is computed as \( \delta_R = R' \uplus R \), which can be rearranged trivially to \( R' = R \uplus \delta_R \). Hence the delta computed for a relation is correct.

Delta propagation must ensure that all affected queries of a relation are scheduled for reevaluation when the relation changes. Consider a query site \( q \), for which \texttt{q.tree_node} has an attribute instance \( x \) that depends upon the value of the bag of tuples in \( R \) that match \( q \)’s tuple template, \( T_{q,R} = \{ t : R \mid t \text{ matches } q.\text{tuple_template} \} \), where \( R \) is the relation queried. When \( R \) changes, \( x \) must be scheduled for reevaluation if \( T_{q,R'} \neq T_{q,R} \). The \texttt{propagateDelta} method computes a filtered delta \( \delta_{q,R} = \text{FilterDelta}(q.\text{tuple_template}, \delta_R) \). If \( \delta_{q,R} \neq \bot \) then \( x \) is scheduled for reevaluation by setting the need\_visit flag in \texttt{q.tree_node} and all its ancestors. Thus the algorithm correctly schedules \( x \), since \( T_{q,R'} \neq T_{q,R} \Rightarrow \delta_{q,R} \neq \bot \).

Setting need\_visit flags in this way prevents any further visits to \texttt{q.tree_node} being skipped. Furthermore, \( x \) will not have been reevaluated yet, nor will the visit responsible for evaluating it have been skipped, since \( x \) depends on \( R \). Hence the differential propagation scheme reevaluates relations correctly.

### 6.4 Calculating clause deltas

The algorithm for calculating deltas shown above relies on each clause object having a method \texttt{calculateDelta}. These methods are generated from clause definitions in a similar manner to the \texttt{calculateTuples} methods. Rather than enumerating all the tuples defined by a clause, however, a \texttt{calculateDelta} method evaluates a clause incrementally by using the deltas previously computed for each of the clause’s constituent query sites.

Given a clause of the form \( H : \vdash Q_1, \ldots, Q_k \), when any of the tuple bags generated by the queries \( Q_1, \ldots, Q_k \) change, a delta for the clause can be computed by using both the deltas associated with the relevant query sites and their new tuple bags. Conceptually, the tuples generated by the queries \( Q_1, \ldots, Q_k \) are “extended” so that all the generalized bags involved in the calculation involve the same variables. The tuple bag for a clause can now be written as an equation of the form:

\[
T_0 = \bigcap_{i=1}^{k} T_i
\]

When a change to any of the \( T_i \) occurs, a delta \( \delta_0 \) must be computed such that \( T_0' = T_0 \uplus \delta_0 \), based on \( T_i' = T_i \uplus \delta_i \). When computing \( \delta_0 \) the only information available is the current state of each queried relation (i.e., each \( T_i' \)) and the delta for each queried relation (i.e., each \( \delta_i \)). The equation for \( T_0' \) can be rearranged in terms of each \( T_i' \) and \( \delta_i \) as follows. The proof of the third step may be found in the appendix.
\[ T'_0 = \bigcap_{i=1}^{k} T'_i \]

\[ = \bigcap_{i=1}^{k} (T_i \uplus \delta_i) \]

\[ = \bigcap_{i=1}^{k} T_i \uplus \bigcup_{j=1}^{i-1} \bigcap_{j=i+1}^{k} (T'_j \uplus \delta_j) \]

Therefore,

\[ \delta_0 = \bigcup_{i=1}^{k} (\delta_i \uplus \bigcap_{j=1}^{i-1} T'_j \uplus \bigcap_{j=i+1}^{k} (T'_j \uplus \delta_j)) \]

The `calculateDelta` method that is generated for a clause computes its result according to this formula. The delta for each query on the right hand side of clause is considered in turn, and used to "drive" the evaluation process. Tuples from each delta are used to generate variable bindings, limiting the number of tuples that must be examined from re-queried relations (represented by the \( T'_j \) terms in the formula above). This leads to efficient propagation in the average case, even though the worst case still involves enumerating all the tuples for a clause.

To illustrate this method of calculating clause deltas, consider the following relation clause:

\[ \text{forall (integer x, integer y) this} \rightarrow \text{R(x, y) :- S(x, y), T(x, y), U(x, y).} \]

The delta for this clause is computed according to

\[ \delta = (\delta_S \uplus (T'_T \uplus \delta_T) \uplus (T'_U \uplus \delta_U)) \uplus (\delta_T \uplus T'_S \uplus (T'_U \uplus \delta_U)) \uplus (\delta_U \uplus T'_S \uplus T'_T) \]

Suppose the tuple \((0, 0)\) is added to \(T\). In this instance \(\delta_T = \[0, 0]\) and \(\delta_S\) and \(\delta_U\) are both empty. The delta for the clause is computed according to:

\[ \delta = \delta_T \uplus T'_S \uplus T'_U \]

The computation proceeds by iterating over each tuple in \(\delta_T\), generating variable bindings from the tuple under consideration, using those bindings to query \(S\) and \(U\).

### 6.5 Implementing external relations

External relations are also implemented using deltas. Since an external relation can be either used or provided by a tool, but not both, there is no need to propagate changes in the same way as for internal relations. Queries of an external relation are treated similarly to the scheme described above, and changes to a queried external relation (as provided by the environment) trigger notification of the appropriate tree nodes using the `propagateDelta` method shown above.

Handling an external relation that is defined by a tool is complicated by the fact that its definition may be distributed among multiple tree nodes. Each site that defines tuples for an external relation is treated as an internal relation containing that site’s contribution. An external relation has a single query site that is used to query each contribution. As contributions are updated, changes to them are propagated to the external relation, in preparation for transmission to the environment upon completion of the current tree walk.
7 User interaction and incremental evaluation

The incremental evaluation scheme presented in this report differs from previous approaches in that it is designed to be effective in an interactive setting. In particular, the scheme is intended for use in a language-based editing system which employs a modeless syntax recognition input paradigm. This leads to two main demands for semantic evaluators: the need to handle syntactically erroneous or incomplete programs, and the need for efficient processing of input provided a keystroke at a time. The former is achieved by a simple strategy for attributing syntax error-tolerant trees, and the latter by a preemptive evaluation scheme.

7.1 Syntax errors

Syntax errors necessarily occur in a syntax recognition editor [3]. Requiring that the program be in a syntactically correct state before allowing semantic evaluation places an undue constraint on the usefulness of a semantic evaluator and may hinder the effectiveness of analyzers for detecting semantic errors. Therefore it is vital that semantic evaluators generated using the scheme described here be capable of performing evaluation in the presence of syntactically erroneous or incomplete constructs.

The incremental parsing scheme used in UQ* [3] constructs an error-tolerant syntactic representation. For a semantic evaluator to handle syntactically erroneous programs it must perform evaluations on this error-tolerant representation.

The parsing strategy described in [3] isolates erroneous material in a sub-tree rooted at a node called a repair root (Figure 4). A repair root is a non-terminal node which has children consisting of:

- syntactically correct sub-trees,
- placeholder nodes for children for which a correct sub-tree cannot be determined, and
- an error node that contains material surrounding the point where a syntax error was detected and for which a correct location in the tree cannot be determined.

Semantic evaluation over such a structure is performed by ignoring the material stored below error nodes, and using a default attribution scheme for placeholders. Material stored below an error node cannot be evaluated since either it is not in its correct place in the tree, it is syntactically incorrect, or its syntactic correctness is undetermined. By ignoring error nodes a repair root may be attributed using the normal constraints for its rule. A placeholder must be attributed using the default rule provided for its phylum (as described in Section 4). The responsibility for providing sensible attribution rules for placeholders is given entirely to the environment builder.

In some situations UQ*'s incremental parser cannot determine appropriate placeholder nodes to place below a repair root (this occurs when the parser is unable to determine, from a repair root's context, which rule it corresponds to) [3]. In this situation the nodes below the repair root do not match any rule in the grammar, and the evaluation scheme must treat the entire sub-tree rooted at the repair root as a placeholder; i.e., default attribution rules are applied at the repair root rather than placeholder nodes.

Since a sub-tree will never be visited when it is below an error node, there is no need to set need_visit flags for all the ancestors of an error node when either a replacement occurs
Figure 4: Example of a repair root and the sub-trees arranged below it

below it or a query site below it is updated. Hence the notify method may be amended to:

```java
method Node.notify;
begin
  if not need_visit and self is not an error node then
    need_visit ← true;
    if parent ≠ nil then
      parent.notify
    end if
  end if
end
```

An error node will be removed from the tree in response to a user edit which eliminates the corresponding syntax error. When this occurs, the incremental parser attempts to reuse non-trivial sub-trees below the error node. Likewise, a semantic evaluator may reuse the attribute values associated with such sub-trees, with a corresponding saving in evaluation time. Such a saving is particularly important in a modeless editing system, where syntax errors are introduced and eliminated often. The relative ease with which the evaluation scheme is extended to handle syntax errors is a validation, not only of the scheme itself, but also of the design choices in UQx’s incremental parser [3].

7.2 Preemptive evaluation

Preemptive evaluation is a scheme which helps incremental evaluators to keep pace with the rate of user input. In a modeless editor, where each keystroke is a complete editing operation, a naive approach to processing user input will either lead to unacceptable delays between edits and feedback, or will require the user to explicitly invoke an evaluation via some command. Preemptive evaluation overcomes these problems by allowing evaluators to handle sequences of editing operations, rather than single edits. The scheme works by periodically checking for new input during an evaluation, and preempting the completion of that evaluation in favour
of incorporating the new input. Preemptive evaluation may be viewed as a form of multiple
sub-tree replacement handling where not all replacements are known at the beginning of an
evaluation.

The incremental evaluation scheme described in Sections 5 and 6 is designed to be tolerant
of preemption. Very little state information is maintained during an evaluation other than
information that actually forms part of the attributed tree structure. The state information of
the evaluator is comprised of:

- the run-time stack,
- scheduling information in the form of need_visit flags, and
- relation deltas propagated to query sites.

The run-time stack poses no problem for preemption; abandoning part of an evaluation simply
requires the stack to be unwound (the algorithm below uses exception handling to achieve
this). The ways in which scheduling information and propagated deltas are recorded in the
tree are specifically designed to enable preemption. Both these forms of state information
accumulate in the tree until they are “consumed” by the evaluation process. If a tree node
contains any accumulated state information when an evaluation is abandoned, then it will have
been scheduled for evaluation and will remain so until a restarted evaluation reaches it.

The current implementation of this scheme checks for new input, at the beginning of (some)
initial visits. Since this is relatively expensive it is not performed at the beginning of each
initial visit. Instead its execution is controlled by a counter so it is performed periodically. The
algorithm used is shown in the CheckForNewInput procedure below. Sub-tree replacements are
held in the queue pending. CheckForNewInput relies on the function LeastCommonAncestor
which is defined as

\[
\begin{align*}
\text{LeastCommonAncestor}(n, \text{nil}) &= n \\
\text{LeastCommonAncestor}(\text{nil}, n) &= n \\
\text{LeastCommonAncestor}(n_1, n_2) &= n_1 \sqcup n_2
\end{align*}
\]

where \(\sqcup\) is the join operation on the \(\sqcup\)-semi-lattice based on the “descendant of” weak
partial order. The implementation of LeastCommonAncestor has running time \(O(h)\), where \(h\)
is the height of the tree. Assuming the EBNF grammar makes judicious use of repetitions the
running time will be \(O(\log N)\), where \(N\) is the size of the tree. The scheme also relies on each
tree node having an in_first_visit flag which is set for the duration of an initial visit to the node.

CheckForNewInput scans the queue of pending input, incorporating sub-tree replacements,
and computing restart_at which indicates the point in the tree at which evaluation will recom-
ence. If the evaluation is to be restarted an exception is raised to commence stack unwinding.
The operation of CheckForNewInput is illustrated in Figure 5.

**procedure** CheckForNewInput;
**begin**

while there is input to be read do

Read a sub-tree replacement, and append it to pending

end while;

restart_at \(\leftarrow\) nil;

while pending is not empty do

new \(\leftarrow\) the root of head(pending);

old \(\leftarrow\) the node that will be replaced by new;

25
Figure 5: Operation of CheckForNewInput

\[
\text{restart}_\text{at} \leftarrow \text{LeastCommonAncestor} (\text{restart}_\text{at}, \text{old}) ; \\
\text{if } \text{restart}_\text{at} = \text{old} \text{ then} \\
\quad \text{restart}_\text{at} \leftarrow \text{new} \\
\text{end if} ; \\
\text{while not } \text{restart}_\text{at}.\text{in}_\text{first}_\text{visit} \text{ do} \\
\quad \text{restart}_\text{at} \leftarrow \text{parent} (\text{restart}_\text{at}) \\
\text{end while}; \\
\quad \text{Incorporate head} (\text{pending}); \\
\quad \text{new.notify}; \\
\quad \text{Remove head} (\text{pending}) \text{ from} \text{ pending} \\
\text{end while}; \\
\text{if } \text{restart}_\text{at} \neq \text{nil then} \\
\quad \text{raise} \text{ RestartException} (\text{restart}_\text{at}) \\
\text{end if} \\
\text{end}
\]

Restarting is handled by initial visit methods. When a restart exception is caught the visit method will check if the exception has reached the restart point. If so the evaluation is restarted at that point, otherwise the node’s \text{in}_\text{first}_\text{visit} flag is reset and the propagation of the exception is continued. The revised template for \text{visit}_\text{I} methods is shown below. The boxed line of code is only generated if the initial visit for a rule is the only visit for that rule.

\textbf{method} \text{SomeRuleInstance} . \text{visit}_\text{I} ; \\
\text{begin} \\
\quad \text{if } \text{need}_\text{visit} \text{ then} \\
\quad \quad \text{restart} : \\
\quad \quad \begin{align*}
\text{begin} \\
\text{in}_\text{first}_\text{visit} & \leftarrow \text{true} ; \\
\text{CheckForNewInput} ; \\
\text{Evaluate attributes and perform sub-visits;} \\
\text{in}_\text{first}_\text{visit} & \leftarrow \text{false} \\
\text{need}_\text{visit} & \leftarrow \text{false} \quad \{ \text{Final visits only} \}
\end{align*} 
\end{align*}
\text{end}
on exception RestartException (restart_at)
    if restart_at = self then
        goto restart
    else
        in_first_visit ← false:
        raise RestartException (restart_at)
    end if
end

8 Empirical evaluation

An empirical performance evaluation was carried out on the prototype implementation of the techniques discussed in this report. The tests were executed on an evaluation tool that implements static semantic analysis for the hypothetical language PL0, discussed in Section 4. Each test involved a series of modifications to the (initially ill-formed) PL0 program shown in Figure 6 using a controlled input source that produced input at the rate of ten keystrokes per second. The tests were carried out on a Sun Ultra-5 with a 270 MHz CPU and 64 MB of RAM.

Each test consisted of 50 iterations of the following steps.

- A sequence of ten insertions and deletions of a "y" character after the "x" on line 514 is performed. Each insertion causes each of the 500 uses of the variable "x" to be invalidated, and provides a good test of the evaluator's preemptive capabilities.

- The declaration "yi: integer;", where i indicates the current iteration, is inserted, one character at a time, on line 515. This insertion introduces a syntax error on all but the last keystroke, at which point the ten previously invalid uses of the variable become valid.

- The procedure header "procedure Pi;" is inserted, one character at a time, on line 526. This leaves most of the remaining program in syntax error on each keystroke except the ";".

- The procedure body "begin a01 := b01 end;" is inserted, one character at a time, on line 1033. This restores the syntactic correctness of the program and exercises the evaluator's ability to reuse attribute values as they move in and out of error.

On each iteration the size of the program increases and the 500 assignment statements move one level of nesting deeper. The front-end editor and the evaluator were synchronized at the end of each iteration. Each iteration consists of 68 keystrokes.

Four different tests were executed:

Keystroke by keystroke: a test with the evaluator running on every keystroke (with differential propagation enabled),

On demand: a test with the evaluator running on demand just once at the end of each iteration (with differential propagation enabled),
1 var
2    z001: integer;
3    z002: integer;
4      ...
501  z500: integer;
502  procedure Q;
503  var
504      a01: integer;
505      ...
513      a10: integer;
514      x: integer;
515
516      b01: integer;
517      ...
525      b10: integer;
526
527  procedure P0;
528  begin
529      y01 := x + z001;
530      ...
578      y50 := x + z050;
579      y01 := x + z051;
580      ...
628      y50 := x + z100;
629      ...
630      ...
1028      y50 := x + z500
1029  end;
1030  begin
1031      a01 := b01
1032  end;
1033
1034  begin
1035      z001 := z002
1036  end.

Figure 6: PL0 program used for performance evaluation
<table>
<thead>
<tr>
<th>Test</th>
<th>Evaluation time (sec)</th>
<th>Time per keystroke (sec)</th>
<th>Preemptions</th>
<th>Attributes evaluated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial load</td>
<td>7.57</td>
<td>3.52</td>
<td>16.13</td>
<td>48901</td>
</tr>
<tr>
<td>Keystroke by keystroke</td>
<td>172.50</td>
<td>2.54</td>
<td>11.62</td>
<td>1793850</td>
</tr>
<tr>
<td>On demand</td>
<td>3.52</td>
<td>2.54</td>
<td>11.62</td>
<td>42488</td>
</tr>
<tr>
<td>No diff. prop.</td>
<td>0.24</td>
<td></td>
<td>6</td>
<td>48079</td>
</tr>
<tr>
<td>Normal</td>
<td>0.17</td>
<td></td>
<td>7</td>
<td>58884</td>
</tr>
</tbody>
</table>

Table 1: Results of performance evaluation

No differential propagation: a test with preemption enabled but differential propagation disabled (this was achieved by modifying the applyDelta method to always return \( T \), and replacing all copy rules of relation-valued attributes with equivalent tuple assertions), and

Normal: a test with both preemption and differential propagation enabled.

These tests were chosen to evaluate the effectiveness of preemption and differential propagation independently.

Each test was executed three times and the results were averaged. The data for each test consists of the evaluation time (not including time spent by the evaluator waiting for input), the mean evaluation time per keystroke, the number of preemptions made, the number of visits begun, the number of visits completed, and the number of attributes evaluated. Note that the instrumentation of the evaluator to count attribute evaluations has an effect on the time taken by the evaluator. It is impossible to separate the testing of evaluation time and counting of attribute evaluations, however, as evaluation time has an effect of the number of preemptions made. The results of the tests are shown in Table 1. Each entry in the table shows the average value per iteration. Data for the initial attribution of the tree was also collected, which gives some indication of the time required for a non-incremental evaluator.

The results for keystroke by keystroke evaluation are clearly unacceptable and highlight the necessity and utility of preemption. The results for evaluation on demand show that there is a significant overhead in performing automatic evaluation. This overhead is acceptable, however, given that the average delay experienced by the user in the automatic case is significantly less than the 3.52 seconds exhibited by evaluation on demand. The evaluation time when differential propagation is disabled is significantly greater than the normal case. Note that fewer preemptions occurred in this case, since the delays between initial visits were longer and the evaluator could incorporate more changes per preemption. Hence fewer attributes were reevaluated. For these tests, disabling differential propagation resulted in an approximately 40% decrease in the number of attributes evaluated per second. When viewed in these terms the benefits of differential propagation are clear.

The time per keystroke for the normal case provides strong evidence for the viability of the approach advocated here, and compares favourably with the requirement that users experience no more than a 200ms delay in response to a keystroke [27]. Note, however, that the data here does not reflect the maximum delay experienced by the user. Some keystrokes necessarily imply more processing than others (in particular, compare keystrokes that eliminate syntax errors with those that introduce or maintain errors). It is difficult to measure the maximum delay due to the effects of preemption. Examining the performance of the evaluator in terms of
average delays is acceptable, however, given the requirement to minimize delays over sequences of keystrokes discussed in Section 2.

9 Related Work

In this section the techniques discussed in this report are compared and contrasted with previous approaches to constructing incremental semantic evaluators.

Previous work at the University of Queensland has focussed on a block-based strategy for incremental semantic evaluation [16]. This strategy was designed to operate with a minimum of storage overhead, and to minimize the effects of distant propagation of semantic changes. The Kiong and Welsh approach is a framework for the manual, but systematic, implementation of evaluators. The main design choice was a sacrifice in granularity in exchange for a lower cost per unit of reevaluation. In principle their approach was adaptable to automatic asynchronous evaluation, but in practice was only implemented for analysis on request. In contrast, the scheme advocated here supports automatic reevaluation in modeless editors at an acceptable cost, and is generically implemented.

Work on the Centaur environment [2] has been based on Natural Semantics specifications for programming languages. The use of such specifications turns the problem of semantic evaluation into one of automated proof of program properties. Attali et al. [1] demonstrate that, under some circumstances, such proofs can be carried out incrementally. However, the overhead of incremental evaluation in this case appears to be too great: in some cases an incremental evaluation requires more time than a non-incremental evaluation. The use of ordered attribute grammars advocated here permits efficient implementation through static typing and statically computed evaluation plans.

Systematic approaches to incremental semantic evaluation typically use some form of declarative language specification. In particular, attribute grammars [17, 18], or extensions to the attribute grammar formalism, have been the basis of many incremental semantic evaluation strategies. The remainder of this section is devoted to such approaches.

Reps et al. [25] discuss the use of attribute grammars in language-based editors and present an asymptotically optimal incremental attribute evaluation scheme for non-circular attribute grammars. Elsewhere Reps also provides incremental evaluators that avoid storing all attribute instances, and efficient optimal evaluators for absolutely non-circular attribute grammars, and ordered attribute grammars [26]. Alternative strategies for the latter are presented by Yeh [34], Yeh and Kastens [35], and Maddox [22]. The evaluation scheme presented here achieves incrementality in a similar manner to that proposed by Maddox, through the use of visit caching. Several approaches support multiple sub-tree replacements [24, 35, 22], all with the same asymptotic complexity as the scheme advocated here. The preemptive evaluation scheme discussed in Section 7.2 is a novel addition to incremental evaluation and can be viewed as a generalisation of earlier techniques for handling multiple sub-tree replacements.

Although Reps' approach is optimal, it suffers from efficiency problems due to limitations of the attribute grammar formalism; in particular, updating long copy-rule chains and aggregate values. Reps et al. [24] present solutions to these problems based on modest extensions to the attribute grammar formalism to allow explicit references to "remote" attributes, thus avoiding the need for copy rules in many situations, as well as a table attribute type to handle updates to aggregate values efficiently. Hoover presents a method for handling copy rule chains [8] which requires no extension to the attribute grammar formalism. This model is extended by
Hoover and Teitelbaum [9] to support efficient handling of aggregate values. Several alternative solutions to these problems also exist [12, 7, 4, 22].

One of the most interesting extensions to the attribute grammar formalism is the addition of relations. Relations play an important role in a software development environment [21, 33, 13], being used for purposes such as documentation, traceability, and capture of context sensitive structure. Horwitz and Teitelbaum [10, 11] provide a method of combining relations with attribute grammars, and claim that the two are symbiotic: attribute grammars can facilitate more powerful relational computations than conventional relational query languages, and relations provide a convenient solution to long-distance attribute flow problems. Li [20, 19] takes this notion one step further by providing a model of relational equations which replace attribute equations. The authors of both of these models recognize the power of a relational model, both for static semantic checking and for provision of sophisticated query facilities.

Maddox [22] presents an incremental evaluation scheme based on a combination of attribute grammars, functions, relations, and objects. The approach is based on visit caching for ordered attribute grammars, and refinement of static dependency information at run-time. Visit caching was explored earlier by Vogt et al. [30] in the context of incremental evaluation in a pure functional language.

The treatment of relations proposed in this report is similar to the schemes presented by both Li and Maddox. Extending attribute grammars to support relation-valued attributes provides many benefits, including an effective solution to problems with long-distance semantic effects and handling of aggregate values. The scheme proposed here also subsumes the Horwitz and Teitelbaum model of relations through its support for external relations that can be queried and navigated by the user. The differential propagation scheme discussed in Section 6 provides a single efficient fine-grained approach to incremental evaluation of relations that scales well from relation-valued attributes to external relations. In addition, the scheme presented here provides a more natural treatment of relations than Li's by employing a Prolog-like notation for relation definitions rather than relational algebra. The differential propagation scheme presented here enables a fine-grained approach to incremental evaluation of relations that is missing from the schemes presented by both Li and Maddox.

There have been many other extensions to the attribute grammar formalism to make it more useful in an interactive environment. Kaplan et al. [14] discuss the use of attributed graph grammars to provide a natural representation of context sensitive information, and to overcome the efficiency problems associated with attribute grammars. A similar approach is taken by Vorthmann [31], by providing a graph substrate to an attributed tree. Teitelbaum and Chapman [28] discuss the use of higher-order attribute grammars in an editing environment. In practice, such extensions make the formalism more difficult to implement efficiently due to the absence of statically computable evaluation plans. In contrast, the marriage of Ordered Attribute Grammars and relations advocated here provides a formalism that is relatively simple to implement and enables efficient incremental treatment.

No techniques similar to those presented in Section 7 (evaluation around syntax errors and preemptive evaluation) have been proposed elsewhere. Both of these techniques stem from the demands placed on an evaluator by user interaction, a topic which has been largely overlooked in the literature.
10 Conclusions

This report has presented new techniques for incremental semantic evaluation. These techniques arise from putting the problem of semantic evaluation in context; in particular, the context of an interactive programming environment with its associated user needs. The particular combination of visit caching, differential propagation of changes to relations, syntax error handling, and preemptive evaluation arose from the context of a modeless syntax recognition editing system. Visit caching was chosen to handle basic incremental operation due to its relative simplicity, making it a useful basis for the more sophisticated extensions discussed here. Differential propagation of changes to relations serves as a powerful technique for handling aggregate values and long-distance effects efficiently. The need to handle syntax errors efficiently is a result of the syntax recognition editing paradigm, and the technique for doing so was made simple by virtue of the visit caching approach. Preemptive evaluation makes automatic reevaluation in a modeless environment feasible, presenting significant benefits to the user. Furthermore, the generic implementation of these techniques makes the task of building incremental semantic evaluators simpler and less error prone than if they were constructed by hand.

Several possibilities exist for extension of this work. A more flexible meta-language could make tool description easier and more widely applicable; e.g., support for programmer-defined abstract data types to provide access to library components in a way that maintains the declarative semantics of the meta-language. The particular combination of relations and attribute grammars used here may be applicable to building non-incremental tools as well, but further work would be required to determine how useful the formalism is in practice. The algebraic approach to reasoning about the incremental treatment of relations (Section 6) could have wider appeal for constructing other incremental data types, whether for use in an attribute grammar, or as a general technique for program improvement. Finally, the application of the preemptive evaluation technique may be useful in other interactive systems as a general approach for coping with type-ahead.

Incremental semantic evaluation has been an established area of research for many years. Previous approaches to the problem have largely overlooked the impact of user and architectural requirements. Consequently, we believe, the adoption of such techniques by the wider software engineering community has been disappointingly small. It is only with consideration of this context that incremental semantic evaluators can be developed that will meet the needs of real programmers. We believe that the techniques discussed in this report represent a significant step forward in this area, and that they make the promise of realistic, widely used language-based editing systems not only more feasible, but more desirable.

Appendix

This appendix proves the following theorem, which is used in Section 6.4.

\[
\bigcap_{i=1}^{k} (A_i \uplus B_i) = \bigcap_{i=1}^{k} A_i \uplus \bigcup_{i=1}^{k} \left( B_i \cap \bigcap_{j=1}^{i-1} (A_j \uplus B_j) \uplus \bigcap_{j=i+1}^{k} A_j \right)
\]

where \(A_i\) and \(B_i\) are generalized bags.
The proof involves induction on $k$. The induction hypothesis (for $k = n$) is:

$$\bigcap_{i=1}^{n} (A_i \uplus B_i) = \bigcap_{i=1}^{n} A_i \uplus \bigcap_{i=1}^{n} \left( B_i \uplus \bigcap_{j=1}^{i-1} (A_j \uplus B_j) \uplus \bigcap_{j=i+1}^{n} A_j \right)$$

The proof for $k = n + 1$ is as follows:

$$\bigcap_{i=1}^{n+1} (A_i \uplus B_i) = \bigcap_{i=1}^{n} (A_i \uplus B_i) \uplus (A_{n+1} \uplus B_{n+1})$$

$$= \bigcap_{i=1}^{n} (A_i \uplus B_i) \uplus \bigcap_{i=1}^{n} (A_i \uplus B_i) \uplus B_{n+1}$$

$$= \left( \bigcap_{i=1}^{n} A_i \uplus \bigcap_{i=1}^{n} \left( B_i \uplus \bigcap_{j=1}^{i-1} (A_j \uplus B_j) \uplus \bigcap_{j=i+1}^{n} A_j \right) \right) \uplus A_{n+1} \uplus B_{n+1}$$

$$= \bigcap_{i=1}^{n+1} (A_i \uplus B_i) \uplus (A_{n+1} \uplus B_{n+1})$$

(from the induction hypothesis)

$$= \bigcap_{i=1}^{n+1} A_i \uplus \bigcap_{i=1}^{n+1} \left( B_i \uplus \bigcap_{j=1}^{i-1} (A_j \uplus B_j) \uplus \bigcap_{j=i+1}^{n+1} A_j \right) \uplus B_{n+1} \uplus \bigcap_{j=1}^{n+1} A_j$$

(note that the last factor is the identity of $\uplus$)

$$= \bigcap_{i=1}^{n+1} A_i \uplus \bigcap_{i=1}^{n+1} \left( B_i \uplus \bigcap_{j=1}^{i-1} (A_j \uplus B_j) \uplus \bigcap_{j=i+1}^{n+1} A_j \right)$$

Hence the theorem is true for $k = n + 1$.

All that remains to be shown is that the theorem is true for $k = 1$:

$$A_1 \uplus B_1 = A_1 \uplus (B_1 \uplus A_2) = A_1 \uplus B_1$$

\(\Box\)

References


