References

[1] Gregor V. Bochmann. Semantics evaluation from left to right. *Communications of the ACM*, 19(2):55–62, February 1976.

[2] Scott R. Dietzen, Mary Ann Pike, Anne M. Rogers, and William L. Scherlis. *User's Guide to the Ergo Syntax Facility.* Ergo Report In preparation, Carnegie Mellon University, Pittsburgh, 1988.

[3] Rodney Farrow. Experiences with an attribute grammar-based compiler. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 95–107, ACM, February 1982.

[4] Harald Ganzinger, Robert Giegerich, Ulrich Moncke, and Reinhard Wilhelm. A truly generative semantics-directed compiler generator. In *SIGPLAN 82 Symposium on Compiler Construction*, pages 172–184, ACM, 1982. In: SIGPLAN Notices 17(6).

[5] M. Jazayeri and K.G. Walter. Alternating semantic evaluator. In *ACM Annual Conference*, pages 230–234, ACM, 1975.

[6] Martin Jourdan. Strongly non-circular attribute grammars and their recursive evaluation. In *SIGPLAN 84 Symposium on Compiler Construction*, pages 81–93, ACM, 1984. In: SIGPLAN Notices 19(6).

[7] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.

[8] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann. *GAG: A Practical Compiler Generator.* Volume 141 of *Lecture Notes in Computer Science*, Springer-Verlag, 1982.

[9] Ken Kennedy and Scott K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *Third ACM Symposium on Principles of Programming Languages*, pages 32–49, ACM, 1976.

[10] Kai Koskimies. A specification language for one-pass semantic analysis. In *SIGPLAN 84 Symposium on Compiler Construction*, pages 179–189, ACM, 1984. In: SIGPLAN Notices 19(6).

[11] Kai Koskimies, Kari-Jouko Raiha, and Matti Sarjakoski. Compiler construction using attribute grammars. In *SIGPLAN 82 Symposium on Compiler Construction*, pages 153–159, ACM, 1982. In: SIGPLAN Notices 17(6).

[12] Peter Lee, Frank Pfenning, John Reynolds, Gene Rollins, and Dana Scott. *Research on Semantically Based Program-Design Environments: The Ergo Project in 1988.* Technical Report CMU-CS-88-118, Carnegie Mellon University, Pittsburgh, March 1988.

[13] Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis. The Ergo Support System: an integrated environment for prototyping integrated environements. In *Proceedings of the SIGPLAN'88 Symposium on Software Development Environments, Boston, November 28–30*, ACM Press, 1988.

[14] Robert L. Nord. *A Framework for Program Flow Analysis.* Ergo Report 87–038, Carnegie Mellon University, Pittsburgh, November 1987.

[15] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, ACM Press, June 1988. Available as Ergo Report 88–036.

[16] William Pugh. An improved replacement strategy for function caching. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah*, pages 269–276, ACM Press, July 1988.

[17] Kari-Jourko Raiha. Bibliography on attribute grammars. *SIGPLAN Notices*, 15(3):35–44, 1980.

[18] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *Software Engineering Symposium on Practical Software Development Environments*, pages 42–48, ACM, 1984. In: SIGPLAN Notices 19(5).

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

[20] William L. Scherlis. Abstract data types, specialization and program reuse. In *International Workshop on Advanced Programming Environments*, Springer-Verlag LNCS 244, 1986.

[21] Reinhard Wilhelm. *Global Flow Analysis and Optimization in the MUG2 Compiler Generating System*, pages 132–159. Prentice Hall, 1981.

```
program(^value) = ...
   ;final value;

exp(!env,^value) =  ...
  | vref(id(^token))
      with bound(value)
        else "unbound variable"
      where value = lookup(token,env)
```

```
(defattr-attrfun attr program (term) () (value)
   (opcase term (prog (vls value))))

(defcon-deltafun con exp (n term) (env)
   (opcase term ...
      (vref (argcase n
               (0 (vls))
               (otherwise (delta-error 'vref))))

(defsyn-sigmafun syn exp (term) (env)
   (opcase term ...
      (vref (mvb (token) (get-syn-argn syn exp id 0 term env)
               (let ((value (rt-fun lookup token env)))
                  (constraint-check (rt-fun bound value)
                     "unbound variable")
               (vls value))))
```

Figure 3: Attribute ADT Code

the attribute-grammar compiler itself, and a programmable formatting unparser. The flow analyzer compiles high-level semantic descriptions for control flow and abstract interpretation to produce attribute-grammars that perform data flow analysis. The iterative nature of this problem makes demand-driven analysis and caching essential in obtaining adequate performance. Additionally, the flow analyzer itself is implemented using the Ergo Attribute System, since an attribute-grammar provides a natural representation for specifying the syntax-directed translation from the high-level descriptions for control flow to attribute-grammars. The attribute-grammar compiler takes advantage of the optimizations in the underlying attribute ADT to provide good performance when tools are combined in this way. The formatting unparser does not use attribute-grammars, but the attribute ADT directly. It is fast enough to allow real time highlighting of subterms during mouse-waving. Its efficiency is largely due to attribute sharing between different versions of a program.

Among the extensions we are considering at this point is the introduction of *key functions* into the abstract data type. Instead of caching the mapping from the context to the syntext, the implementation would apply a key function to the context and cache the mapping from the resulting *key* to the syntext. This could result in significant improvement in the efficieny of looking up cached values, and also allows more sharing. On the downside: it puts the responsibility for correct caching again into the client's hand, since the abstract data type has no way of checking whether equivalent keys mean equivalent syntext values.

A very useful extension of the attribute specification language is to allow direct quotation of concrete syntax with free variables. We could thus avoid the explicit and tiresome references to term constructors in attribute grammars.

Finally, we plan to exploit the *higher-order view* of abstract syntax which is provided and used by other tools in the ESS. The higher-order view incorporates name binding information in a static and language-independent way into the representation, and currently has to be "rediscovered" when writing an attribute-grammar. Higher-order abstract syntax and how it could be exploited in the context of attribute-grammars is described in [15].

6 Acknowledgements

Earlier versions of the abstract data type of attributes and the main ideas of its implementation were developed during an intensive month of brainstorming within the "Attribute Task Force" within the Ergo group consisting of Scott Dietzen, Conal Elliott, and the second author. The first implementation of the attribute-grammar compiler in Common Lisp was done by Bill Maddox, the total reimplementation of the compiler using attribute-grammars and an overhaul of the specification language syntax were done by the first author.

```
Grammar calc
Terminals id, number

program(^value) = prog(exp(!ienv, ^value))      where ienv = nullenv()
   ; final value;
bindpair(!oenv, ^nenv) = bp(id(^token), exp(!oenv, ^value))
                                                where nenv = bind(token,value,oenv);;
exp(!env, ^value) =
     let(bplist(bindpair(!env/etag, ^etag\nenv)*), exp(!nenv, ^value))
   | vref(id(^token))
        with bound(value) else "unbound variable"
                                                where value = lookup(token, env)
   | cref(number(^litvalue) )                   where value = litvalue
   | add(exp(!env, ^val1), exp(!env, ^val2))    where value = val1 + val2
   | sub(exp(!env, ^val1), exp(!env, ^val2))    where value = val1 - val2
   | sum(exp(!env, ^val)*)                      where value = sumlist(val*);;
```

Figure 2: Calculator Attribute-Grammar

are used to maintain an environment of variables and their values. The environment is enlarged in bindpair and passed down to exp. Whenever a variable is referenced, its value is looked up in the environment. The synthesized attribute value (prefixed with '^') is used to compute the numerical value of the expression. We make use of predefined numerical functions, (e.g. $+$, $-$, $*$, $/$) and user-defined Lisp functions such as nullenv, lookup, bound, sumlist, and bind.

The example also illustrates how the attribute specification language provides two ways of dealing with operators with a variable number of arguments. One is to distribute inherited attributes to all immediate subterms and collect all synthesized attributes from immediate subterms into a list, the other is to pass attributes from one member of a list to an adjacent member in a *bucket brigade*. In the calculator example, the list bindpair+ is specified as a left-to-right bucket brigade — if we instead intended let to bind all variables simultaneously we would have used distribution and collection instead as in the case of the operator sum.

4.2 Attribute-Grammar Compilation

Compilation of an attribute-grammar proceeds in two phases. The input attribute-grammar is parsed to produce an abstract syntax tree, that is then analyzed to produce the attribute ADT code. This latter phase can be further subdivided into three subphases: (1) the abstract syntax is translated into a dependency structure where (2) dependency analysis is done to produce a code graph which is (3) translated into the attribute ADT code.

Dependencies that need to be constructed during subphase (2) are the computation ordering of local attribute definitions and the sequence in which syntext is to be computed for the immediate subterms of a term. There is a close correspondence between the dependencies and the attribute ADT code. Synthesized attribute definitions translate into syntexts with Σ-functions, inherited attribute definitions translate into contexts with Δ-functions, and final attribute definitions translate into attributes with A-functions. The corresponding references translate into calls to get-syn, references to Lisp variables, and calls to get-attr. Additional code is required for variable-length argument lists, optionals, and iterated attributes.

Figure 3 shows the Common Lisp code produced by the attribute-grammar compiler for a small portion of the calculator attribute-grammar in Figure 2. This code calls functions like `get-syn` and `get-syn-argn` which are implementations of functions with the same name in the attribute ADT.

5 Conclusions and Future Research

The view of attributes presented in this paper describes a formalism for specifying the data type of attribute that ensures attribute consistency without constraining the implementation. We have not fully characterized the class of attribute languages that can be supported by the attribute ADT, it is clear that we do not exploit all of its flexibility in the present attribute-grammar compiler. Our experience has been with three major applications: a language-generic program flow analyzer,

same, so it is returned without further computation.

Attr Calc Val program $\langle t, \langle\,\rangle, val\rangle \Leftarrow val$

It should be noted that none of the Δ or Σ functions are directly recursive. Wherever an implementation without attributes would make a recursive call, our implementation will access (and perhaps compute first, if it was not cached already) a syntext value for a subterm. This mechanism allows the implementation to intervene and cache and retrieve syntexts. This mechanism is similar to function caching, but under the control of the client.

4 An Attribute Specification Language

In the previous section we presented an abstract data type of attributes and showed through an example how it can be used to define attributes without ever writing an attribute grammar. However, attribute grammars provide a convenient way of specifying attributes, so the Ergo Attribute System also contains an attribute-grammar compiler. This compiler generates code in the attribute ADT, given a relatively conventional attribute grammar in our *attribute specification language.*

The attribute specification language is basically applicative and similar in structure to the grammar language used by the Syntax Facility, except that it follows the abstract syntax rather than the concrete syntax. This style of grammar provides a declarative view with emphasis on nonterminals and attributes (as in *Lisa* [10]), rather than on productions and semantic rules, (e.g. ALADIN [8] is rule-based, ADELE is semantics-directed) and may thus be described as "abstract-syntax-directed". An attribute-grammar consists of a series of declarations followed by a sequence of rules. A *rule* consists of a (unique) nonterminal on the left-hand side and a sequence of productions on the right-hand side. Productions may contain local definitions, constraints, and declarations. External attributes may be referenced within a grammar by simple qualification of an attribute name. This supports decomposition of large applications and allows sharing among different applications.

The class of languages accepted by the attribute specification language is similar to ADELE in MUG2 [21]. Attribute dependencies are restricted to those which allow attribute evaluation in a fixed number of passes over the abstract syntax tree. A pass is any depth-first tree traversal where the order of the visits to the subtrees of a given node is known at compile time. This classification falls between ordered attribute-grammars [7] and attribute-grammars that are evaluable in alternating passes [5]. Like ADELE, our attribute-grammars are of type *n-sweep* where the decomposition into the single sweeps has to be explicitly provided for. Experience with MUG2 indicated that this class of attribute grammars is sufficiently general while allowing efficient pass-oriented evaluation techniques. Requiring explicit separation into passes helps to keep descriptions modular and comprehensible, and also aids demand analysis.

In order to provide good support for flow analysis the attribute-compiler has an additional feature that allows attributes that are to be computed iteratively if they are declared as such (as in MUG2).

It should be pointed out that the attribute ADT does not restrict us to this particular class of attribute-grammars. Rather, the attribute specification language has evolved based on the demands made by the clients in our program derivation environment. This evolution was greatly facilitated by the separation of attribute ADT and attribute grammar compiler, and also by the bootstrap of the compiler within the Ergo Support System (ESS). This "bootstrap" includes the generation of the attribute-grammar parser by our parser generator and the attribute-grammar compiler by the attribute-grammar system itself.

4.1 The Calculator Example Revisited

Let us return to the calculator example introduced in Section 3.3. We will illustrate how the value attribute could have been specified in the attribute specification language, rather than writing it directly in the attribute ADT. The attribute-grammar compiler will compile this into attribute ADT code which is relatively close to the hand-coded version. In this subsection we follow that standard terminology in attribute-grammars, that is, we use the terms *inherited attribute* and *synthesized attribute.* It is important to remember, that an inherited attribute will be compiled into a member of a *context*, a synthesized attribute will be compiled into a member of a *syntext.* Only attributes that are declared *final* in the attribute-grammar will be compiled into attributes in the attribute ADT.

The attribute-grammar is given in Figure 2 (the reader may want to refer back to the grammar in Figure 1). Inherited attributes are prefixed with '!' and

```
Comment Character       newline '%'
Lexical Terminals        id, number
Precedence Information
    exp  '+' medial left, '-' medial left
program ::= { exp }                                  <prog(exp)>;
bindpair ::= { id '=' exp }                          <bp(id,exp)>;
exp ::=  { 'let' {bindpair ++ ','} 'in' exp }        <let(bplist(bindpair+),exp)>
       | { id }                                      <vref(id)>
       | { number }                                  <cref(number)>
       | { exp^1 '+' exp^2 }                         <add(exp^1,exp^2)>
       | { exp^1 '-' exp^2 }                         <sub(exp^1,exp^2)>
       | { 'SUM' '(' { exp ++ ',' } ')' }            <sum(exp+)>;
```

Figure 1: Calculator Grammar

such a collection of definitions a *context family* — Calc in this example. (**opcase** performs case analysis on the abstract syntax operator of the term.)

ΔCalc program $\langle i, t, \langle\,\rangle\rangle \Leftarrow$ nullenv

ΔCalc exp $\langle i, t, env\rangle \Leftarrow$
  opcase t **of**
    let $\rightarrow$ **case** i **of**
        $0 \rightarrow env$
        $1 \rightarrow$ get-syn-argn Calc $\langle 0, t, env\rangle$
    otherwise $\rightarrow env$

ΔCalc bplist $\langle i, t, env\rangle \Leftarrow$
  case i **of**
    $0 \rightarrow env$
    otherwise $\rightarrow$ get-syn-argn Calc $\langle (i-1), t, env\rangle$

ΔCalc bindpair $\langle i, t, env\rangle \Leftarrow$
  case i **of**
    $0 \rightarrow \langle\,\rangle$
    $1 \rightarrow env$

The context for an expression will be an environment associating variable names with values. When entering the body of a let, this must be updated to the environment *synthesized* from the list of additional bindings. This is achieved by the call to get-syn-argn in line 6 of the example. For other expressions, the environment for the subterms is identical to the environment for the term.

A list of bindings is processed from left to right, so that the environment for binding pair i is the environment synthesized by binding pair $i-1$, or the environment that is passed down if $i = 0$. This is embodied in the definition of ΔCalc bplist.

The following is the definition of the *syntext family* Calc that depends on the context family of the same name. The syntext of an expression should be the result of evaluating it in the current environment, the syntext of a list of bindings is the result of augmenting the environment by additional bindings.

ΣCalc program $\langle t, \langle\,\rangle\rangle \Leftarrow$ get-syn-argn Calc $\langle 0, t, \langle\,\rangle\rangle$

ΣCalc exp $\langle t, env\rangle \Leftarrow$
  opcase t **of**
    let $\rightarrow$ get-syn-argn Calc $\langle 1, t, env\rangle$
    vref $\rightarrow$ **let** id = term-argn $\langle 0, t\rangle$ **in**
        let val = lookup $\langle id, env\rangle$ **in**
          if bound val **then** val
            else error 'Unbound Variable.'
    cref $\rightarrow$ numval (term-argn $\langle 0, t\rangle$)
    add $\rightarrow$(get-syn-argn Calc $\langle 0, t, env\rangle$)
        +(get-syn-argn Calc $\langle 1, t, env\rangle$)
    sub $\rightarrow$(get-syn-argn Calc $\langle 0, t, env\rangle$)
        −(get-syn-argn Calc $\langle 1, t, env\rangle$)
    sum $\rightarrow$ sumlist (get-syn-list Calc $\langle t, env\rangle$)

ΣCalc bplist $\langle t, env\rangle \Leftarrow$
  get-syn-last Calc $\langle$term-argn $\langle 0, t\rangle, env\rangle$

ΣCalc bindpair $\langle t, env\rangle \Leftarrow$
  let id = term-argn $\langle 0, t\rangle$ **and**
      val = get-syn-argn Calc $\langle 1, t, env\rangle$ **in**
    bind $\langle id, val, env\rangle$

The definition of the Σ-function for expressions should be easy to understand. The calculation of the new environment for a bindpair involves synthesizing the value of the expression the variable is bound to and adding a new binding to the environment that is returned. The new environment synthesized from a whole list of bindings is simply the one synthesized by the last element of the list (achieved by a call to get-syn-last).

The definition of the *attribute* family in this example depends only on the syntext. In fact it is the

a context, syntext, or attribute during debugging of an attribute-grammar. Since our efficient version of the implementation provides no simple way of erasing all cached attribute information, we often switch to an implementation where the mapping from context to syntext values is stored in a global hashtable indexed by terms. This is somewhat less efficient, but the hash table can be erased completely at any time without affecting the value of attributes (though their computation may take more time).

Inter-program structure sharing. Any given attribute slot may cache the mapping from different contexts to different syntexts. Therefore a term may appear in many versions of the program with different attribute values. Since syntext values (and therefore attributes) are computed uniformly as a function of the context and the term, the implementation does not have to deal with creating and maintaining versions for each change to the program, and a client does not have to request "new versions."

Demand-driven attribute evaluation. This is clearly built into the implementation, since no attribute is computed, unless access is attempted (via get-syn). We also provide a function to explicitly traverse the term and compute syntext, context, and attributes for each subterm. This is sometimes useful to force the checking of constraints embedded in the attribute definition.

Incremental attribute update. We implicitly use the simple criterion that a syntext (and consequently, attribute) of a term does not need to be recomputed if the context it depends on has not changed. This criterion is correct by definition of a syntext. Thus many attribute values will be shared between closely related versions of a program, achieving the benefits of "change propagation" of Reps et al. [19] within demand-driven evaluation.

Attribute persistence. When a program is modified during a transformation, attributes for unchanged parts of the program will persist. The fact that we cache a mapping guarantees attribute consistency, and if a transformation is "undone," the old attribute values will again be accessible. As noted above, for debugging purposes this is almost too persistent, since attributes will exist as long as the term they are cached in exists, even if the definition of the attributes is changed.

Genericity. An implementation of the attribute ADT is required to provide the primitives for local propagation of context and syntext information (through such functions as con-delta and get-syn-argn). It does *not* provide evaluation strategies: those are the responsibility of the client. This is the primary source of the genericity of the attribute ADT. A client (such as an attribute-grammar compiler) can *program* different strategies for ordering of local dependencies, ordering of global dependencies (*e.g.* list traversal, circular dependencies, non-local propagation), program traversal, and multiple passes over the program.

Efficiency. The primary source of inefficiency in our implementation is the fact that we have to cache a mapping from contexts to syntexts in a slot of the term. In a conventional implementation, the syntext value could be looked up directly, while here an association list must be traversed. However, efficiency gains through other advantages of our scheme outlined above clearly outweigh the additional overhead of caching in our applications. If the need arises, our implementation of caching could be greatly improved using techniques developed for efficient function caching (see, for example, [16]).

3.3 A Simple Example

The Ergo Attribute System can best be explained with an example that will illustrate how to create an application, and how the implementation processes it. We present a simple calculator language and first show how the attribute ADT can be used to implement an evaluator for this language. This example will also be used to demonstrate our attribute specification language (see Figure 2).

The concrete syntax of the language is defined through a grammar (see Figure 1). The Syntax Facility produces a parser, lexer, and unparser for the language as well as a language definition table that can be referenced by other tools. The structure of the abstract syntax for a given calculator expression is listed in the grammar augments (between <>) and shows the operator and its arguments. (bindpair+ denotes the list of bindings). The nonterminals are program, bindpair, exp, and bplist (inferred by the Syntax Facility). Abstract syntax operators are, for example, let, vref, or add.

In this example, a context consists of the environment and a syntext consists of the computed value (except for bindpair which synthesizes a new environment). There is only one attribute, namely the result of evaluating the program.

As is often convenient, we will define the Δ-function separately for each nonterminal. We call

Next we define *syntexts*. The syntext values of a given term are defined through a function from the term and context values to a tuple of values to be computed simultaneously. We call such a function a Σ-function. σ is the type of the information conveyed by the syntext, usually a tuple of values.

ADT `Syn` **with**
mk-syn : `Con` $\rightarrow$ ((`Term` $\times \delta$) $\rightarrow \sigma$) $\rightarrow$ `Syn`
syn-con : `Syn` $\rightarrow$ `Con`
syn-fun : `Syn` $\rightarrow$ ((`Term` $\times \delta$) $\rightarrow \sigma$)
end

In the auxiliary functions and axioms below, $f(t_1, \ldots, t_n)$ is a term with operator f and arguments t_1 through t_n. Σ is a Σ-function, Δ is a Δ-function, and i, t, and c are variables denoting an index, term, and context.

Auxiliary functions of `Syn`
get-syn : `Syn` $\rightarrow$ `Term` $\times \delta \rightarrow \sigma$
get-syn (mk-syn $con\ \Sigma$) $\langle t, c\rangle = \Sigma\ \langle t, c\rangle$

get-syn-argn : `Syn` $\rightarrow$ `Nat` $\times$ `Term` $\times \delta \rightarrow \sigma$
get-syn-argn (mk-syn (mk-con Δ) Σ)
$\langle i, f(t_1, \ldots, t_n), c\rangle$
$=$ get-syn (mk-syn (mk-con Δ) Σ)
$\langle t_i, \Delta\ \langle i, f(t_1, \ldots, t_n), c\rangle\rangle$

get-syn-list : `Syn` $\rightarrow$ `Term` $\times \delta \rightarrow$ (σ `list`)
(get-syn-list $syn\ \langle t, c\rangle)_i$
$=$ get-syn-argn $syn\ \langle i, t, c\rangle$

get-syn-last : `Syn` $\rightarrow$ `Term` $\times \delta \rightarrow \sigma$
get-syn-last $syn\ \langle f(t_1, \ldots, t_n), c\rangle$
$=$ get-syn-argn $syn\ \langle n, t, c\rangle$
end

The auxiliary function get-syn and the destructor function syn-fun are extensionally equal, but they will be implemented differently. The computation of synthesized information is often recursive and this recursion will be exposed in the definition of the Σ-function by an access to the syntext of the immediate subterms of a term. This access is done through the function get-syn-argn, which in turn makes use of the Δ-function to compute the correct context for the subterm.

Since the term t itself is an argument to the Σ-function, it may perform any recursive computation on t to obtain the syntext. However, it is most efficient if the information needed about the subterms of t is embodied in the syntext of the subterm so it can be cached there and retrieved by get-syn-argn.

Our implementation of get-syn will by default try to look up a cached value for the given term and context. If no such value exists, it will be computed by the appropriate Σ-function and the result will be cached. Caching and lookup can be explicitly enabled or disabled at compile-time, so that in reality we support several alternative implementations of this abstract data type simultaneously. Syntext values are cached in a slot of the term as an association list indexed by the syntext name and the context. The equality check between contexts defaults to Common Lisp `equal`.

Finally, we consider *attributes*, which are defined through a function from terms, context, and syntext information to the attribute value. This function is called an A-function (alpha function). In the definition below, δ and σ are the type of the context and syntext information on which the attribute depends, respectively. α is the type of the information conveyed by the attribute. A is an A-function.

ADT `Attr` **with**
mk-attr : `Syn` $\times$ ((`Term` $\times \delta \times \sigma$) $\rightarrow \alpha$) $\rightarrow$ `Attr`
attr-syn : `Attr` $\rightarrow$ `Syn`
attr-fun : `Attr` $\rightarrow$ ((`Term` $\times \delta \times \sigma$) $\rightarrow \alpha$)
end

Auxiliary functions of `Attr`
get-attr : `Attr` $\rightarrow$ ((`Term` $\times \delta \times \sigma$) $\rightarrow \alpha$)
get-attr (mk-attr $syn\ A$) $\langle t, c, s\rangle = A\ \langle t, c, s\rangle$
end

The relationship between attr-fun and get-attr is the same as between syn-fun and get-syn. As in the case of contexts, we do not cache the result of this computation.

3.2 Requirements Revisited

In this section we describe how our implementation of the attribute ADT meets the requirements we identified in the introduction: attribute consistency, inter-program structure sharing, demand-driven attribute evaluation, incremental attribute update, and attribute persistence.

Attribute consistency. Since we cache a *mapping* rather than a value, and that mapping uniquely determines the syntext and therefore the attribute value, correctness of attributes with respect to its definition as a function is always guaranteed. There is one notable exception, namely a change in the *definition* of

in which a term occurs. Typical examples of contexts are binding environments, paths from the root to the current subterm occurrence, etc. Contexts are analogous to sets of inherited attributes in an attribute-grammar.

A *syntext* is a tuple of values that distills relevant information from the subterms of a term, given the current *context*. Typical examples of syntexts are computed values, translated terms, etc. Syntexts are analogous to sets of synthesized attributes in an attribute-grammar.

Finally, an *attribute* is a single value that depends on the current context and syntext. Attributes seem to be the simplest way of isolating values from context and syntexts, which are the central notions of the abstract data type.

3.1 The Specification of the Abstract Data Type

We now give the definition of the abstract data type in the pieces outlined above and intersperse comments about our implementation, and how it satisfies the requirements of a program derivation system. Of course, the abstract data type allows other implementations, and we will occasionally indicate alternatives.

The abstract data type definition is somewhat unusual in that the principal objects given as arguments to constructors are functions. We therefore think of it as a higher-order data type, and this has several consequences. Normally, the signature of an algebraic data type is that of a many-sorted algebra — here functions are arguments of constructors, and the types of these functions are not fixed. Thus a completely formal specification of the data type would have to use dependent function types or explicit polymorphism. Such a formulation would exceed the scope of this paper and also make the specification less readable, so we have chosen to use type variables in the specification and explain their dependencies informally. We omit the obvious axioms for the destructor functions associated with an abstract type and focus on the auxiliary functions. We consider a function *auxiliary* if it could be defined from the basic constructors and destructors, but, for efficiency reasons, should be implemented differently.

A brief word on notation: we use "$\rightarrow$" as the function type constructor and "$\times$" as product type constructor, where $\times$ binds more tightly. Functions are constructed using "λ," products are constructed with "$\langle\,\rangle$." We use typewriter fonts for data types, lower-case greek letters for type variables, sans-serif font for functions, italics for variables, and upper-case greek letters for function variables. "=" stands for equality in a data type, "$\Leftarrow$" stands for function definition.

Following the dependencies between the abstract types, we begin with the definition of *contexts*. Contexts are specified by a function that, given a term and its context, computes the context information for an immediate subterm. This function is called a Δ-function and takes three arguments: the index for an immediate subterm, a term, and a context. δ is the type of the information conveyed by the context, usually a tuple of values. The type of context information associated with a term may depend on that term, and therefore δ and δ' may differ.

ADT Con **with**
  mk-con : $((\mathtt{Nat} \times \mathtt{Term} \times \delta) \rightarrow \delta') \rightarrow \mathtt{Con}$
  con-deltafun : $\mathtt{Con} \rightarrow ((\mathtt{Nat} \times \mathtt{Term} \times \delta) \rightarrow \delta')$
  end

The following auxiliary function could be defined in terms of con-deltafun but may be implemented quite differently.

Auxiliary functions of Con
  con-delta : $\mathtt{Con} \rightarrow ((\mathtt{Nat} \times \mathtt{Term} \times \delta) \rightarrow \delta')$
  con-delta (mk-con Δ) $\langle n, t, c\rangle = \Delta\langle n, t, c\rangle$
  end

Δ-functions often need synthesized information, but this information will typically be cached; hence Δ-functions should involve little computation. Our implementation of the attribute ADT relies on the assumption that Δ-functions can be executed quickly to obtain context for the subterms of a given term if the necessary synthesized information has been cached. If this assumption should fail, it would be possible to cache the result of Δ-functions without changing the abstract data type specification, but we not yet found a need for this.

In the following two examples of contexts, Unit is the unit type with "$\langle\,\rangle$" as its sole element, which we use to denote the empty context. A path is a sequence of indices determining a path through the abstract syntax tree representing a term, and pcons extends a path by an index. A number of useful contexts such as these are predefined in our implementation.

empty-con : Con
empty-con = mk-con $(\lambda\langle n, t, c\rangle\ .\ \langle\,\rangle)$

path-con : Con
path-con = mk-con $(\lambda\langle n, t, c\rangle\ .$ (pcons $\langle n, c\rangle))$

attribution.

This conclusion has been confirmed by our experience with three major applications that have been completed using the Ergo Attribute System: a language-generic program flow analyzer [14], the attribute-grammar compiler itself, and a programmable formatting unparser with an interface to the X window system supporting mouse pointing and real-time highlighting.

2.2 An Abstract Data Type

One of the guiding principles in the design of the ESS is the use of formally specified *abstract interfaces* as the "glue" binding together the system components. As far as we know, this principle has never been applied to attributes. An abstract data type of attribute should fulfill the requirements laid out above, and also satisfy the following criteria.

Genericity. It should not unduly constrain the class of attribute grammars that may be compiled into the attribute ADT.

Efficiency. It must be possible to implement the specification of the abstract data type faithfully and efficiently.

With these goals in mind we formulated the following main principle guiding the design of the abstract data type.

> **Functional character of attributes.** An attribute value should be completely determined by the term and the context it appears in, where the definition of context is made together with the definition of an attribute.

Roughly, the abstract data type allows the definition of an attribute as a function from a term and selected information about its context to a value. In our implementation, a mapping from this context information to the attribute value is cached in the term as an association list between contexts and values.

2.3 An Attribute Specification Language and Compiler

Our attribute specification language and the underlying compiler are both very flexible and can easily be (and have often been) changed. In part, this is due to the fact that it is completely bootstrapped within the ESS, that is, it is written using our parser generator (for the specification language) and the attribute-grammar facility itself (for the compilation into code using the abstract data type of attributes). Another reason is that the underlying data type of attributes makes few constraints on the attribute-grammar.

The Ergo Attribute System is similar to the MUG2 [21] system in many respects, including the language and class of grammars accepted. The language is nonterminal-oriented like *Lisa*, a language developed by the HLP group [10] rather than production-oriented like most other systems. There is a similarity to the Cornell Synthesizer [18] where (unlike MUG2, for example) equivalence between transformed terms cannot be presumed.

The Ergo Attribute System compiles an attribute-grammar written in the attribute specification language into a collection of attribute evaluator functions that rely on the attribute ADT. These evaluator functions can be viewed (1) as operating on abstract syntax to produce an attributed abstract syntax tree, or (2) as an algorithm for computing attributes on demand.

3 An Abstract Data Type of Attributes and Its Implementation

In Section 2 we listed the requirements that must be fulfilled by the abstract data type of attributes. We will reiterate here one of the principal requirements.

> **Attribute Consistency.** The data type should be formulated in such a way that consistent attribution is ensured by the data type and is not the responsibility of the client, that is, every attribute accessed is guaranteed to be valid with respect to its definition.

A natural consequence of this goal is that caching is not part of the specification, but merely of our implementation. In other words, the *specification* does not distinguish between attribute access and attribute computation, though, of course, the *implementation* will. The principle idea underlying our abstract data type specification is that *an attribute value is completely determined by a term and the context it appears in.* Thus attributes in the abstract data type are *specified* through a function which maps a term and information about its context to a value.

A *context* is a tuple of values that distills relevant information from the total of the environment

To this end we have designed and implemented a collection of tools called the Ergo Support System (ESS, see [13]), which includes a parser/unparser generator, an attribute-grammar compiler, a program flow analyzer, a higher-order unifier, a facility for definition of user interfaces, and some system management utilities. These tools are integrated by means of abstract interfaces, two principal ones being an abstract data type of *terms* (for representing abstract syntax of specifications, programs, proofs, etc.) and an abstract data type of *attributes*.

The motivation for including the Ergo Attribute System in the ESS comes from the need for semantic analysis within the context of a derivation system [20]. Flow analysis for instance, has a critical role in computing global information to complement more "local" pattern matching and rewriting in program transformation systems.

The ESS itself has been used to a great extent to develop the ESS. In the case of the Ergo Attribute System, both the Ergo Syntax Facility [2], which generates parsers and unparsers from grammar specifications, and the attribute-grammar compiler itself (after an initial bootstrap) were used.

2.1 Requirements

The use of attributes to represent semantic information is a well-studied formalism. There are numerous strategies for evaluating attributes, see for example, Bochmann [1], Farrow [3], Jourdan [6], Kastens [7], and Kennedy and Warren [9]. Raiha [17] provides an extensive bibliography on attribute-grammars. The notion of attribute is well-known from compilers and compiler writing systems, (e.g. GAG [8], MUG2 [4], HLP [11]) and systems like the Cornell Program Synthesizer [18]. Attributes are also very useful in a program derivation environment. Many natural transformations depend upon non-local conditions that must be checked, and attributes provide for a convenient way of defining and localizing this information. Moreover the syntax-directed nature of many applications within the derivation system can be reflected within an attribute-grammar.

In this section we outline the requirements that should be fulfilled by an attribute-grammar system, in particular in the context of a program derivation environment. Our implementation satisfies these requirements at the expense of some loss of efficiency over conventional implementations *for some conventional applications*. However, the conceptual simplicity, the degree of independence between tools achieved through the abstract data type, and the efficiency gained when there are many related versions of a program more than make up for this loss of efficiency.

Attribute consistency. Attributes should always be correct with the respect to the current version of the program and the definition of the attribute (typically in the form of an attribute-grammar). This is an obvious requirement that should be *de rigeur* for any attribute-grammar system, but it can be violated in many of them. Attribute consistency in such systems is typically *ensured by the client* through an explicit attribution phase. In the Ergo Attribute System, attribute consistency is *guaranteed by the abstract data type* and cannot be violated by the client.

Inter-program structure sharing. During a sequence of transformation steps, many different, but closely related versions of a program may exist. Since programs are generally large, but most transformations local, it is important that the different versions of a program share structure. This presents a consistency problem with respect to attributes, because even though a subprogram may be unaffected by a transformation, its attributes may have changed.

Demand-driven attribute evaluation. Clearly, many attributes may be very expensive to compute, but are needed for only a small subset of the transformations. Moreover, the system cannot predict in general when an attribute may be needed. Therefore, demand-driven attribute evaluation is extremely desirable in this setting.

Incremental attribute update. The simple solution of complete attribute recomputation after each transformation step is unnecessarily expensive, since many attributes in a program will not change during a transformation. Thus incremental attribute update is a highly desirable feature.

Attribute persistence. The user or transformation metaprogram may backtrack or explore the space of programs in an unpredictable fashion, thus requiring attribute values from earlier versions of a program. These should not always have to be recomputed. This also explains why destructive changes of the program during transformations are generally not desirable.

Our solution combines demand-driven evaluation and incremental attribute update in a new way. The overhead in storing and accessing attributes incurred by our scheme is more than offset by the gains from structure and attribute sharing, and from the demand-driven, incremental, and persistent nature of

The Ergo Attribute System

Robert L. Nord Frank Pfenning

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

The Ergo Attribute System was designed to satisfy the requirements for attributes in a language-generic program derivation environment. It consists of three components: (1) an abstract data type of attributes that guarantees attribute consistency, (2) a Common Lisp implementation which combines demand-driven and incremental attribute evaluation in a novel way while allowing for attribute persistence over many generations of a program, and (3) an attribute-grammar compiler producing code based on this abstract data type from a high-level specification. Our experience with three major applications (one being the attribute-grammar compiler itself) confirms that the overhead in storing and accessing attributes incurred by our implementation scheme is more than offset by the gains from the demand-driven, incremental, and persistent nature of attribution.

This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

The authors can be reached via electronic mail on the ArpaNet as `rln@cs.cmu.edu` and `fp@cs.cmu.edu`.

To appear at the Third ACM SIGSOFT Symposium on Software Development Environments, Boston, MA, November 28-30, 1988.

1 Introduction

The major features that distinguish the Ergo Attribute System from other attribute-grammar systems are (1) a formal specification of an abstract data type (ADT) of attribute separated from the attribute-grammar compiler and (2) a new way of ensuring attribute consistency while allowing demand-driven attribute evaluation and a large amount of sharing of attribute values between different generations of a program. The attribute-grammar compiler itself is written as an attribute-grammar.

The novel features were developed in direct response to the requirements imposed by a language-generic program derivation system. These requirements differ from those in a compiler or compiler generator system in that potentially many more distinct, but closely related versions of a program or specification exist. Moreover, the relationship between different versions and the time when an attribute value may be needed are much less predictable. As far as we know, other systems do not deal with this matter directly, since they are only concerned with the latest version of a program and thus attributes for previous versions do not persist.

The body of this paper consists of three sections: Section 2 presents an overview of the Ergo Attribute System and its motivation, Section 3 describes the abstract data type of attributes and its implementation, Section 4 describes the attribute specification language and the process of compiling attribute-grammars into the attribute ADT.

2 The Ergo Attribute System

The Ergo Attribute System is part of the Ergo Project's [12] ongoing effort to build an environment that supports experimentation in program derivation.