Journal of Universal Computer Science, vol. 7, no. 1 (2001), 54-70
submitted: 1/9/00, accepted: 13/10/00, appeared: 28/1/01 © Springer Pub. Co.

Declarative Term Graph Attribution for Program Generation

Wolfram Kahl
(University of the Federal Armed Forces Munich, Germany
kahl@informatik.unibw-muenchen.de)

Frank Derichsweiler
(University of the Federal Armed Forces Munich, Germany
deri@informatik.unibw-muenchen.de)

Abstract: We show how the declarative spirit of attribute grammars can be employed to define an attribution mechanism for *term graphs*, where the non-uniqueness of inherited attributes demands an appropriately generalised treatment.

Since term graphs are a useful data structure for symbolic computation systems such as theorem provers or program transformation systems, this mechanism provides a powerful means to generate concrete programs (and other relevant text or data structures) from their abstract term graph representations.

We have implemented this declarative term graph attribution mechanism in the interactive term graph program transformation system HOPS and show a few simple examples of its use.

Categories: D.1.2 — Automatic Programming, D.2.6 — Programming Environments, F.4.2 — Grammars and Other Rewriting Systems, D.2.2 — Tools and Techniques, D.1.1 — Applicative (functional) Programming

Key Words: Program generation, term graph attribution, declarative attribute grammars, graph traversals

1 Introduction and Related Work

Attribute grammars have been developed by Knuth for specifying and implementing the (static) semantic aspects of programming languages [Knu68, Knu90]. Since then, attribute grammars have grown into a recognised field of study with numerous applications; for one of many surveys see [Paa95].

Attribute grammar ideas also have found their way into graph transformation research. One kind of approaches, like those of [Göt82, Sch87, ZM96], attributes graph-grammar parse trees instead of string-grammar parse trees. Most current approaches consider attributed graphs and their derivation or transformation. One of the most well-known frameworks in this context seems to be PROGRES [Sch90, Sch97], where, however, the declarative nature of attribute grammars is given up in favour of an operational approach. In the same way, also the algebraic approach of [LKW93, WG96] implemented in AGG [ERT99] is not oriented towards a declarative view of attributions, but towards describing transformations of attributed graphs.

The approach documented in [Ber96] attempts to move closer to the original attribute grammar setting by concentrating not only on attributions, but also on the traversals necessary to calculate the attributions; it thus necessarily stresses the operational aspects of the attribute grammar view, so this approach, too, abandons pure declarativity.

Only in the field of incremental graph attribution there are approaches that maintain pure declarativity for the definition of graph attributions via attribute-grammar-like formalisms. In many respects quite similar to our approach is that of [ACR+88], which however still sends unique attribute values along its “cables”. Since this is realised via restricting nodes to have a predefined number of incoming edges, this is not applicable in a general term graph setting.

In this paper we present a *purely declarative* approach to *term graph attributions* in a formalism which is essentially a straight-forward transfer of the attribute-grammar paradigm to the slightly more general setting of term graphs, and appropriately deals with *sharing*. Furthermore, we present an implementation of this approach in the term graph programming system HOPS, and applications of an implementation of this approach to program generation in Smalltalk, Ada, and Haskell.

2 From Syntax Tree Attributions to Term Graph Attributions

We now shortly sketch the principles behind our declarative approach to term graph attribution. We keep this completely general and independent of our specific implementation, which will be presented in [Section 4] together with choices for the involved languages and further details, as well as example rules.

Attribute grammars are an extension of context-free grammars which consists of *semantic rules* added to the syntactic rules of the context-free grammar. In our view, a semantic rule consists of

– a *tree pattern* P determining applicability of the rule,

– an *attribute name* N for the attribute to be defined, and

– an *expression* E defining the values of the N-attributes of those nodes where the pattern P matches; this expression
  - is written in an *attribute definition language*, and
  - contains *attribute references* to other attributes, written in an *attribute reference language* which allows access to attributes of nodes accessible via navigation primitives or as images of nodes in the pattern P.

In purely declarative attribute grammars, attribute definition languages are referentially transparent, i.e., cannot express side-effects.

Such an attribute grammar is then used to define *attributions* of *syntax trees*, and since syntax trees can be considered as a special kind of graphs, we consider an attribution as a (partial) function mapping a node of the graph and an attribute name to an attribute value from an attribute value set depending on the attribute name.

Usually only certain attribute values at the root of the tree are relevant, but we may as well consider the *attribution of the whole tree* (or graph) as the result of applying the semantic rules of an attribute grammar to a given syntax tree (or graph).

This view of attributions carries over to graphs without any problems, and the *syntactic* definition of semantic rules also does not need noteworthy adaptation. What changes, however, is the *semantics* of an attribute grammar, since the interpretation of the attribute reference language will have to change, according to the following discussion.

If we consider the attribute flow over the syntax tree in a conventional attribute grammar, then attribute values usually flow along single edges, and they may flow along an edge in either direction. So a single node may have *synthesised* attribute values coming in from below (i.e., via its outgoing edges), and it is relevant via which edge each attribute value arrives, and *inherited* attribute values come from above, and here, too, it may be relevant which edge among the node's parent's outgoing edges this is.

When considering trees as term graphs, then the ordering among outgoing edges is replaced by edge labels attached to these edges, and for every node, no two outgoing edges have the same label. Since every tree node has only at most one incoming edge, it is also true that for every tree node, no two incoming edges have the same label. Generalising, we now consider the direction in which an edge is attached to a node together with its label as one "*input channel*" of the node in question (somewhat similar to the "cables" of [ACR$^+$88]). Then conventional attribute grammars (and the attributed graph specifications of [ACR$^+$88]) are always confronted with single attribute values coming from any input channel. A corresponding formalism for general term graphs, however, has to cope with arbitrary numbers of attribute values in the input channels coming in from above, i.e., from the direction of the parent nodes.

Furthermore, in contrast to the operational definition of [Ber96], where different values are arrived at in a sequential manner (see also [Section 5]) and can therefore be considered to be organised in lists, in our purely declarative formalism there is no such obvious structure organising the different attribute values — the only fact that must not be hidden is the possibility of multiple occurrences of the same attribute value. Therefore, multisets are exactly the structure that naturally organises attribute values in general labelled graphs.

As a result, elements of the attribute reference language that refer to inherited attributes have to change their interpretation: instead of referring to a single attribute value, they now refer to a multiset of attribute values. Furthermore, the attribute definition language will have to provide primitives to implement *declarative* functions mapping these multisets to single values that can be used for the defined attributes.

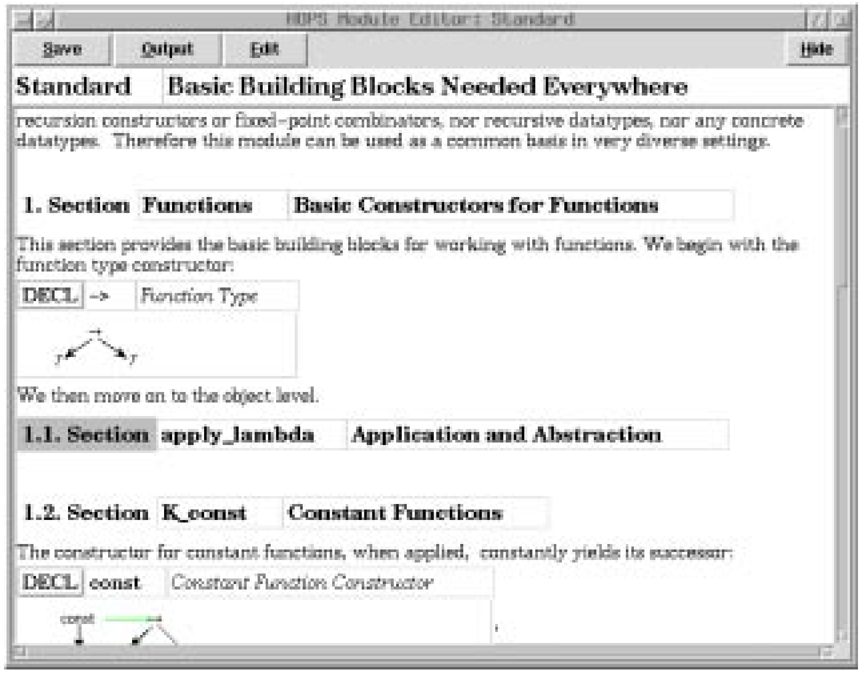
Summarising, we may observe that the move from syntax tree attributions to term graph attributions can easily be realised without violating the purely declarative attribute grammar principles. Only appropriate machinery for representing and manipulating multi-sets is required and needs to be employed for inherited attributes.

Given these principles of declarative term graph attributions, we may now further develop our approach by turning to attribute value sets that are CPOs. As in the conventional attribute grammar setting [Far86], this lets even cyclic attribute dependencies and cyclic term graphs be dealt with easily by defining the attribution via a fixed-point construction. In implementations, this may be realised via lazy evaluation, which is the path that we choose by relying on Haskell [HPJW$^+$92] as our attribute definition language, see [Section 4].

3 HOPS **Overview**

The **H**igher **O**bject **P**rogramming **S**ystem HOPS is a graphically interactive term graph programming system designed for transformational program development, see also [Kah99, Der99, Kah98a, Kah94, ZSB86].

In the spirit of Literate Programming [Knu84], HOPS modules are documents containing program fragments. In HOPS, these are mostly declarations, transformation rules, and attribution definitions — declarations and rules are created and manipulated as *term graphs*. An example view of a module editor window is shown in [Fig. 1].



Standard | Basic Building Blocks Needed Everywhere

recursion constructors or fixed-point combinators, nor recursive datatypes, nor any concrete datatypes. Therefore this module can be used as a common basis in very diverse settings.

1. Section | Functions | Basic Constructors for Functions

This section provides the basic building blocks for working with functions. We begin with the function type constructor:

DECL | -> | *Function Type*

We then move on to the object level.

1.1. Section | apply_lambda | Application and Abstraction

1.2. Section | K_const | Constant Functions

The constructor for constant functions, when applied, constantly yields its successor:

DECL | const | *Constant Function Constructor*

Figure 1: A HOPS module editor window

HOPS manipulates arbitrary second-order term graphs, where all the structure usually encoded via name and scope is made explicit. Term graphs in HOPS therefore feature nameless variables, explicit variable binding (to denote which node binds which variable), explicit variable identity (to denote which nodes stand for the same variable) and metavariables with arbitrary arity; for a detailed introduction to this term graph concept see [Kah98c].

Term graphs are in fact a standard representation of terms, used mostly for efficiency reasons in symbolic computation systems as well as in many implementations of functional programming languages. Most of the literature, however, when drawing term graphs for λ-expressions still establishes variable binding via names and scope — here HOPS differs in that it uses a term graph concept where variable bindings are explicit and denoted by additional edges. As an example we show, on the left side of [Fig. 2], a term graph corresponding to the following expression in a λ-calculus enriched with arithmetic operators and constants (the thick curved arrow denotes variable binding):

$$(\lambda\, x\, .\, x \cdot x + x \cdot 2 + 3)(3 + 5)$$

HOPS types are integrated into the HOPS program term graph structure, and term graph nodes are connected to their types via special *typing edges*. (For the purposes of attribution, variable binding, variable identity, and typing can be considered as edges with designated labels.) In the right part of [Fig. 2], the same graph is shown again as in the left part, but with the display of its typing nodes enabled. Usually we shall show term graphs with display of typing nodes disabled for better readability; in the interactive system, however, the omnipresence and accessibility of the typing is extremely valuable.

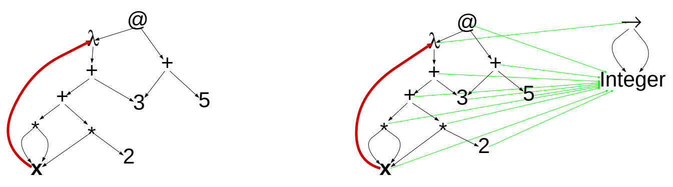


Figure 2: Example term graph — without and with typing

The basis for the typing system are *typing elements*, i.e., simple term graphs that introduce a new node label together with its typing schema making explicit how the typing of a node with this new label is related to the typing of its successors and bound variables. Every HOPS declaration contains one such typing element. Since sets of typing elements define HOPS languages, typing elements also serve as *attribute rule patterns*, so we shall explain this concept in some detail.

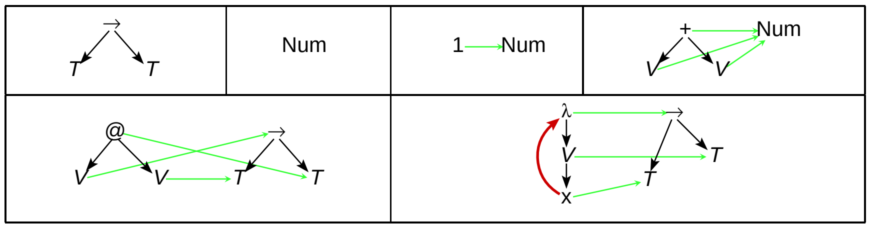


Figure 3: Example typing elements

In [Fig. 3] we show six example typing elements for simply-typed λ-calculus and for arithmetics — the typing function is denoted by thin, light arrows with tiny heads. Nodes labelled with "*T*" are type variables; nodes labelled with "*V*" are instantiatable (typed) program metavariables, and nodes labelled with "**x**" are typed bound variables. Different variable nodes always belong to *different variables* (unless joined by variable identity lines — not to be seen in this paper).

The typing elements of [Fig. 3] may be interpreted as follows:

"→": Nodes with this label are untyped (i.e., type nodes) and have two untyped successors. (Nodes with this label represent *function types*.)

"Num": Nodes with this label are untyped (i.e., type nodes) and have no successors.

"1": Nodes with this label have a typing arrow towards a node with label "Num", i.e., are typed with "Num".

"+": Nodes with this label are typed with a "Num" node and have two successors which are both typed with that same "Num" node.

"@": Every node with this label (*function application*) has two successors, where the left successor is typed with a function type f, the right successor is typed with the argument type of f, and the node itself is typed with the result type of f.

"λ": Nodes with this label (*function abstraction*) are binders and can bind at most one variable, which may occur below their first outgoing edge (which is also the only outgoing edge). Every λ-node is typed with a function type f; its successor is typed with the result type of f, and the bound variable (if any) is typed with the argument type of f.

The typing elements for $\rightarrow$ and Num, although they do not contain any typing arrows, are necessary for introducing their respective label and for fixing this label's arity.

A term graph G is *well-typed* if for every node there is a homomorphism from the typing element for that node's label into G at that node; the details of this kind of type system have been introduced in [Kah98b].

Via the typing elements it contains, every set of HOPS modules (closed in an appropriate sense) defines a *term graph language*. Although the languages of all term graphs in this paper are based on typed λ-calculus and include the above typing elements, it is perfectly possible to define completely different languages as long as their type system is expressible via such typing elements. We also have experimented e.g. with π-calculus.

Having no hard-coded language is considered to be one of the advantages of HOPS (only the variables are fixed within the implementation): Different languages for different domains and/or levels of abstraction give flexibility. The HOPS user declares and uses the bricks which are appropriate for his situation.

While declarations serve to introduce the language as such, rules may be used to turn the language into a calculus, i.e., to constrain the semantics of language constructs. Since the term graph rewriting mechanism of HOPS is not central for this paper, we describe this aspect of HOPS only very shortly: Rules are term graphs with an additional *rule arrow* connecting the left- and right-hand sides. Rules may contain typed meta-variables with successors, i.e., second-order rules are allowed; the details of this rule mechanism have been defined in [Kah96]. Application of matching rules is always possible during editing; this may be manual application of single rules or invocation of *transformation strategies* [Der99].

An example transformation sequence may be started from the term graph of [Fig. 2] and is shown in [Fig. 4]; in the first step, the rule for distributivity of multiplication over addition is applied inside the body of the λ-abstraction and yields a term graph corresponding to $(\lambda\, x\, .\, x \cdot (x + 2) + 3)(3 + 5)$; in the second step the λ-calculus rule of β-reduction is applied, yielding $(3+5)\cdot((3+5)+2)+3$. Note that sharing is preserved as far as possible — this corresponds to the usual definition of lazy evaluation via graph reduction.

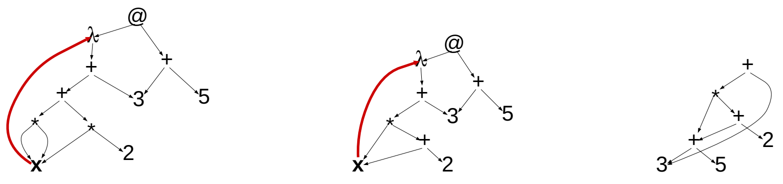


Figure 4: Example term graph transformation sequence

Transformation strategies may also be used to selectively apply rules, which provides flexibility similar to instantiation of parametric modules. E.g. within a specification on an abstract level, a brick called `sort` might be used for a sorting function. No decision about the special sorting algorithm is made; only type and arity are fixed. During a refinement step the decision for a special sorting algorithm is reflected by transforming the `sort` e.g. into e.g. a `heapsort` via a selective transformation step. Continuing the development, the HOPS transformation engine may then be used to expand an implementation for `heapsort` by applying an appropriate rule.

4 Declarative Term Graph Attribution in HOPS

HOPS realises *semantic rules* as module entries of their own, called *attribution definitions*. As mentioned above, the typing elements of HOPS declarations serve as attribute rule patterns, so every attribution definition has to refer to some declaration. The separation of node-label declarations and the corresponding attributions gives flexibility: Using different attribution sets allows to produce different output from the same source (e.g. producing source code in different programming languages from one term graph).

Each attribution definition consists of a target at which it is directed and the definition text proper. These definition texts take on the shape of series of Haskell definitions interspersed with attribute reference expressions written in a syntax similar to that of FunnelWeb, a powerful literate programming system [Wil92].

This means that Haskell here serves as the *attribute definition language* used to produce the real result values, where the central attributes will usually (at least in the examples of this paper) carry strings or functions delivering strings as their values (these strings may then be interpreted in the target language). However, the HOPS attribution mechanism only very weakly depends on Haskell as its attribute definition language; adapting this mechanism to a different language would be extremely easy.

The FunnelWeb-like *attribute reference language* is a typed functional language with only the *String* type allowed for direct embedding into Haskell code, but also featuring a *Node* type and certain function types.

A typical fragment is the following, where the attribute *type* is defined in terms of the *type* attributes of the successors, and the *label* of the node is used inside a string constant.

```
@<type@>@(@1@) = Constr "@<label@>@(@1@)" (map tp succtypes)
    where succtypes = @<successors@>@(@1@,@<type@>@)
```

In the document output, however, and for easier reading, HOPS renders the attribute reference language without "`@`"-characters and in different fonts — note that attribute reference language parentheses and commata are large, bold, and not in typewriter font:

```
type(1) = Constr "label(1)" (map tp succtypes)
    where succtypes = successors(1,type)
```

To give a first flavour of how this attribution mechanism is used, we show here the definitions of two attributes *expr* and *pexpr* in a simplified Haskell conversion that does not respect sharing in any way and takes care of parenthesisation in only a rather crude way. Note that in this application not only the attribute definition language is Haskell, but the generated strings are going to be Haskell code, too. The value of the attribute *expr* at a node n is a string containing a Haskell expression corresponding to the subgraph induced by the node n, and this Haskell expression is not parenthesised on the outside if easily avoidable. The string value of the attribute *pexpr* has parentheses added at least if this makes a difference.

The definition for the conversion of function application (with node label "@") shows how to use the natural numbering of the typing element nodes for referring to the attributes of different nodes:

HaskellAttrib for $\boxed{\textbf{Standard.@}}$

```
expr(1) = expr(2) ++ ' ' : pexpr(3)
pexpr(1) = '(' : expr(1) ++ ")"
```

The "*1*" is an attribute reference language expression of type *Node* and refers to the image source node of the actual pattern, which here is the node labelled with "@". This definition therefore means that the *expr* attribute of an application node is the concatenation (with a space character in-between) of the *expr* attribute of the first successor (number *2*) and the *pexpr* attribute of the second successor (number *3*); the *pexpr* attribute is calculated from the *expr* attribute by just adding parentheses.

In the definition for λ-abstraction, we have to take care whether there is a bound variable or not; we choose to use different conversions for this purpose. This is implemented via the built-in macro *bvar* which takes five arguments: The first argument refers to a node; if this node has a bound variable, then the call evaluates to the fourth argument in an environment where the second argument, considered as a macro name, is bound to the result of applying the third argument (which has to be a function with argument type *Node*) to the bound variable node in the term graph. Otherwise it evaluates to the fifth argument. E.g., if *1* refers to a binder having a bound variable node with the built-in *number* attribute being `1005` then the macro call "*bvar*(*1*,`bv`,*number*,`"x`*bv*`"`,`[]`)" evaluates to "`x1005`". If *1* however refers to a binder that does not bind any variable (as e.g. in $\lambda x.3$), then that macro call evaluates to "`[]`".

Here this is used to implement the distinction between a λ-abstraction in Haskell and an application of `const`, which is more readable than a λ-abstraction where there is no occurrence of the bound variable in the body. Since λ-abstractions already need parentheses in our context, we make the distinction between the two cases on the top-level:

HaskellAttrib for $\boxed{\lambda}$

```
bvar(1,bv,expr,
expr(1) = "(\\ " ++ bv ++ " -> " ++ expr(2) ++ ")"
pexpr(1) = expr(1)
,
expr(1) = const pexpr(2)
pexpr(1) = '(' : expr(1) ++ ")"
)
```

Since HOPS is a language-independent term graph programming framework, it is easy to define term graph languages for different purposes and oriented at different programming language paradigms. We now present two applications where programs in other languages are generated from more or less specialised HOPS languages.

5 Specialised Graph Traversals in Smalltalk

Many problems on graphs can be solved via algorithms that are instances of a depth-first traversal coupled with inherit-synthesise attribute calculations [Ber96]. Examples for instances of this inherit-synthesise scheme are the computation of strongly connected components of a graph, or unification.

The general principles of this algorithm scheme are:

– start with a given *initial value* associated with the algorithm call and compute the *initial inherited attribute*;

– calculate the *inherited attribute for the first successor*, and traverse the first successor, yielding a synthesised attribute for that node;

– combine that synthesised attribute with the current node's inherited attribute to yield the *inherited attribute for the next successor*;

– when all successors are exhausted, use their synthesised attributes together with the current node's inherited attribute to calculate the *synthesised attribute*; and

– whenever encountering a node that has been visited before, use the synthesised attribute calculated at that time (if already available) together with the old and new inherited attributes to calculate the *new synthesised attribute* for that node.

(The last step is the most direct culprit for the loss of pure declarativity here.)

Building on primitive graph access function, we produced a HOPS definition for this algorithm scheme for inherit-synthesise graph traversals. The whole scheme is represented by a HOPS brick "IS" with 13 parameters — this is defined from several sub-schemes taking three to five arguments each, see [Fig. 5]; the fully expanded version comprises about 250 nodes. (In [Fig. 5], all successors of the tip of the rule's left-hand

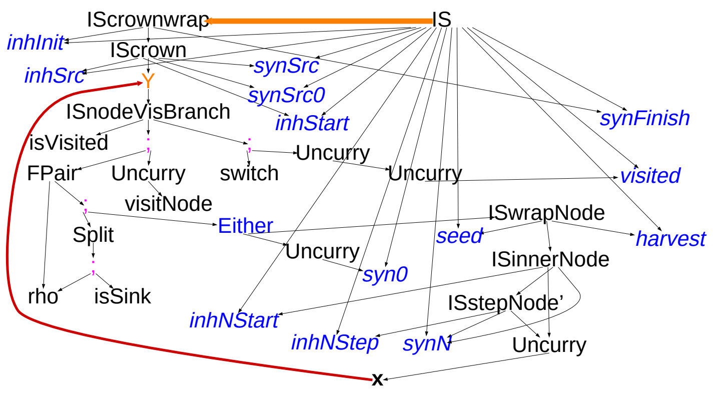


Figure 5: Inherit-synthesise graph traversal "IS"

side, i.e., the node labelled "IS", are metavariables — their *node label* is still "*V*", but they have been equipped with variable names that only serve documentation purposes, and which are displayed in a (blue) oblique sans-serif font.)

Only instantiations for these parameter functions are required in order to produce an algorithm which solves a given problem. Having instantiated the parameter functions, we use partial evaluation to derive a simplified instance of this algorithm scheme. Partial evaluation as a variant of program transformation is realised using the rule application mechanism and transformation strategies of HOPS.

Not every problem needs the whole generalised power of the algorithm scheme for the algorithm solving it; frequently the parameters are instantiated with functions that just select one of their arguments and forward it unchanged. Since we have chosen an approach where "default" instantiations are provided for the parameters, when realising a new algorithm it is possible to specify only the "non-default" parameters via a special selective transformation strategy for the instantiation process.

The result of the simplification strategy application produces an optimised version of the specific algorithm in question. There, e.g. unnecessary handling of values is avoided, and simplification rules have been applied. For simple applications, such an optimised version frequently has significantly fewer nodes than the (expanded) uninstantiated scheme. As an extreme example for this we show the instance that checks for absence of cycles. The use of non-strict boolean operators ensures that in the case of a cyclic graph this result is propagated as fast as possible, and that no unnecessary traversal is performed. For this application, the fully expanded applicative version of the optimised algorithm is given by the 63-node DAG shown in [Fig. 6].

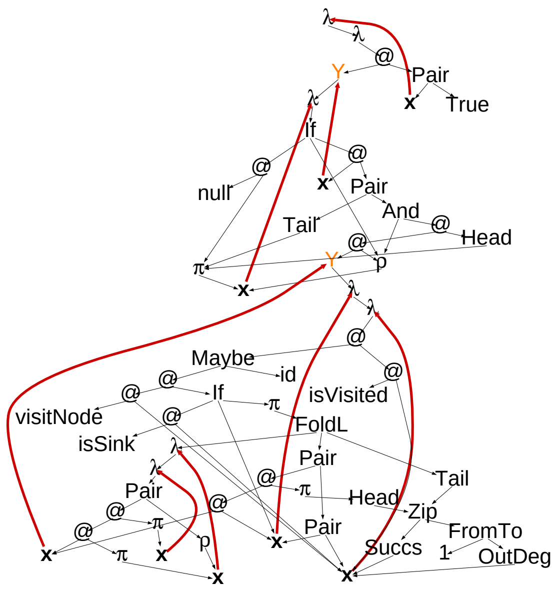


Figure 6: Optimised acyclicity check

On such an optimised version, the attribution mechanism is now used in order to produce textual source code for the algorithm. (Furthermore it is also possible to simulate the application of the produced algorithm by providing some actual parameters and again using the HOPS transformation facilities.) Just as different parameters instantiate the algorithm scheme to solve a different problem, we might also use different attribution definitions to produce implementations very easily in different textual programming languages.

To continue our example of the acyclicity check, we use the code generation mechanism on the optimised version of [Fig. 6] to produce methods in the programming language Smalltalk, thus implementing an efficient cycle-check based on simple graph navigation primitives. The methods are shown below (manually uglified for reasons of space); and the reader may observe how the term graph structure has been translated rather directly into Smalltalk. Apart from the top-level method `cycle_check`, a separate Smalltalk method is generated for every recursive function (i.e. sub-DAG induced by a Y-brick); parameters of subsequent λ-abstractions are accumulated:

```
cycle_check: n_359 with: dummy_n_363
  ^self lf_n_294: (Array with: n_359 with: true)
lf_n_294: n_290
  n_290 fst isEmpty ifTrue: [^n_290 snd]
    ifFalse: [^self lf_n_294:
                (Array with: (n_290 fst cpRemIdx: 1)
                       with: (n_290 snd and: [self lf_n_304: n_290 snd
                                               with: n_290 fst fst]))]
lf_n_304: n_303 with: n_299
  ^(self visValOrNil: n_299)
     isNil:
     [self visVal: n_299 with:
      (n_299 isSink ifTrue: [n_303]
        ifFalse: [(((n_299 succs zip: (1 to: n_299 outDeg)) cpRemIdx: 1)
                  foldLeft: [:zero_n_335 :value_n_335 |
                             self lf_n_314: zero_n_335 with: value_n_335]
                  zero: (Array with: (self lf_n_304: n_303 with:
                                (n_299 succs zip: (1 to: n_299 outDeg)) fst fst)
                              with: (Array with: n_303 with: n_299))) fst])]
     orApply: [:value_n_341 | self lf_n_340: value_n_341]
lf_n_314: n_305 with: n_308
   ^Array with: (self lf_n_304: n_305 fst with: n_308 fst) with: n_305 snd

lf_n_340: n_339 ^n_339
```

The alert reader may have noticed that `lf_n_340:` is an identity function — it corresponds to the single “id” node in [Fig. 6]. Alternatively, we might have chosen not to generate strings containing Smalltalk code, but abstract syntax trees for Smalltalk as an appropriate Haskell datatype inside the attribute definition language. In that case, the application of this identity might have been eliminated as a “peephole optimisation” during the subsequent concrete code generation. As yet another possibility it would be possible to define a HOPS language closer to Smalltalk and transform the acyclicity check into that language before employing the attribution mechanism. This kind of far-reaching transformations is, however, outside the scope of the present paper.

6 Ada Code Transformation

In another example we employ the HOPS transformation mechanism to transform Ada code for a primeness predicate on natural numbers into a more efficient shape.

The starting point is a given Ada algorithm with unnecessary parameters within local functions. First we translate the given program into HOPS rules; then, by using fold/unfold techniques and the transformation facilities, a new and optimised version of the function is generated. Finally the attribution mechanism is used to produce Ada again.

On the whole, we started from a HOPS representation of the following initial, “purely functional” Ada code, where no function has side-effects:

```
function Isprim (n : Nat) return Boolean is
  function Isdiv (k, n : Nat) return Boolean is
    function Divides (k, n : Nat) return Boolean is
    begin if n < k
          then return (n = 0);
          else return Divides (k, n - k);
          end if;
    end Divides;
  begin if k <= 1
        then return False;
        else return (Divides (k, n) or Isdiv (k - 1, n ));
        end if;
  end Isdiv;
begin return not (Isdiv (n / 2, n )) and (2 <= n);
end Isprim;
```

This was then transformed into a version that accesses local variables from within nested functions:

```
function Isprim (n2 : Nat) return Boolean is
  function f4 (n9 : Nat) return Boolean is
    function f6 (n3 : Nat) return Boolean is
    begin if n9 > n3
          then return n3 = 0;
          else return f6(n3 - n9);
          end if;
    end f6;
  begin if 2 > n9
        then return False;
        else return (f6(n2)) or (f4(n9 - 1));
        end if;
  end f4;
begin return (not (f4(n2 div 2))) and (n2 > 1);
end isPrim;
```

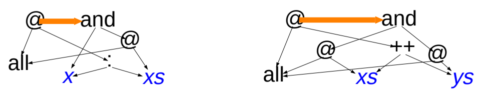
7 Let Sharing Make a Difference

The translation to Haskell as sketched in [Section 4] could equally well be handled via expanding the term graph to a syntax tree first, and then applying conventional attribution techniques — largely this also applies to the Smalltalk and Ada examples of the last two sections.

Term graphs are, however, not in all contexts equivalent to terms, and therefore we now present an application that crucially depends on the possibility to recognise and classify different sharing situations. The problem we shall tackle is the generation of Haskell definitions from term graph rules with the constraint that *sharing* be reflected in the formulation of the Haskell rule. Therefore, we again have the case that Haskell is used both as the attribute definition language and as the language of the strings generated as attribute values.

The following cases have to be distinguished:

1. If a shared node has no successors, then it will be represented by a single identifier and the sharing may be ignored. All other cases therefore assume that the shared node has successors. In [Fig. 7], an example is shown where not only the local



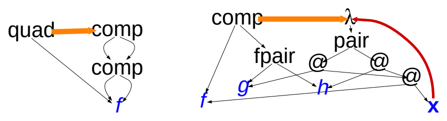
```
all (b:bs) = b && all bs
all (xs ++ ys) =
  (all xs) && (all ys)
```

Figure 7: Laws for `all :: [Bool] -> Bool` with generated Haskell

variables `x`, `xs`, and `ys` need no special device to express their sharing, but also the shared occurrences of `all` — also in the second rule where there are several occurrences of `all` on the right-hand side.

2. If the shared node is only reachable from the rule's right hand side, then:
   - If there is no λ-bound variable occurring free below the shared node, then the expression represented by that node should be the right-hand side of a definition for an identifier for that node in a `where`-clause.
   - Otherwise the corresponding definition has to be put inside a `let`-binding inside the innermost λ-abstraction binding one such free variable.

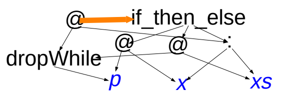
   Examples are in [Fig. 8]; the second example is again not a Haskell definition, but only a law.



```
quad f = v5 . v5
   where v5 = f . f
fpair g h . f = \ x ->
  let v7 = f x in (g v7, h v7)
```

Figure 8: Right-hand-side sharing with generated Haskell bindings

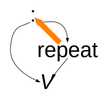
3. If a node with successors is reachable from both rule sides, and if the expression represented by the shared node is a pattern, then this should be converted into an as-pattern (of the shape "*var*`@`*pattern*" in Haskell) on the left-hand side and into a reference to the variable bound to the whole pattern on the right-hand side, as for example in the definition of the Haskell prelude function `dropWhile` in [Fig. 9].



```
dropWhile p v10@(x : xs) =
  if p x then dropWhile p xs
         else v10
```

Figure 9: Inter-rule-side sharing with generated as-pattern

4. If the left-hand side occurs as a whole within the right-hand side, then the right-hand side is replaced by a `where`-bound variable, as in the definition of `repeat` in [Fig. 10], for which, again, exactly the Haskell prelude definition is generated (up to α-conversion). (This kind of definition usually gives rise to cyclic data structures in Haskell implementations.)



```
repeat v3 = v4
  where
    v4 = v3 : v4
```

Figure 10: Generating cyclic `where`-bindings

5. Otherwise, the sharing is ignored. This comprises cases like the shared `dropWhile` node in [Fig. 9], which does not represent a pattern.

The (somewhat contrived) term graph rule of [Fig. 11] illustrates the the co-occurrence of some of these effects in a single rule. This rule is translated automatically into the

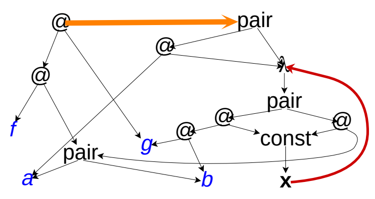


Figure 11: Transformation rule with three different kinds of sharing

following Haskell definition (which nicely demonstrates how the various methods provided by Haskell (and, similarly, by other text formalisms) to encode sharing using names and local definitions obfuscate the overall structure):

```
f (v20 @ (a, b)) g = (a v30, v30)
    where v30 = \ x -> let v26 = const x in ((g b v26), v26 v20)
```

We only sketch our implementation of this solution, which starts from an attribution with values for three attributes:

1. Every node has an attribute containing a unique Haskell identifier for the case that that node needs to be represented by a variable (in the above example, three of these are used: `v20`, `v26`, and `v30`).

2. One attribute contains a Boolean value indicating for each node whether it is the source node of the rule's left-hand side.

3. Another Boolean attribute indicates the source node of the right-hand side.

Starting from these, all in all over twenty attributes are used to implement this only seemingly simple conversion:

- Ten attributes for managing all aspects of determining rule sides and sharing status, among these the following:
  - *inLhs* and *inRhs* are the characteristic functions of the two rule-sides.
  - *spine* is the characteristic function of the spine of the rule's left-hand side.
  - *head* of type `Maybe String` is `Nothing` outside the spine, and on the spine it propagates the name of the defined function.
  - *is_shared* holds true for nodes with more than one predecessor.
  - *is_asvar* holds true for those left-hand-side nodes that need to be represented by an as-pattern.
- Eight attributes for expression generation and expressions on both rule sides, among these the following:
  - *mkexpr* is a function that prescribes how the node is to be turned into Haskell syntax depending on the strings that represent its successors; this syntax prescription is used for building expressions on the right-hand side and patterns on the left-hand side.
  - *expr* is a String containing the Haskell expression corresponding to the node if it occurs on the right-hand side.
  - *pexpr* corresponds to *expr*, but is guaranteed to represent an atomic expression — in doubt, it adds parentheses.
  - *lexpr* contains a representation of the node as sub-expression of the left-hand side; this may include as-patterns.
  - *wheres* contains all information for establishing the bindings of those variables that represent shared nodes below the node carrying the attribute. This synthesised attribute is scanned at λ-nodes for those bindings which contain the bound variable; these bindings are then integrated into a local `let`-binding, while the remainder is further transmitted via this attribute and, at the top of the right-hand side, turned into a `where`-clause.
  - *lhs* and *rhs* are only defined for the respective ends of the rule arrow — for the left-hand side we need the name of the defined function in addition to the *lexpr* standing for the whole left-hand side, and for the right-hand side the *expr* is concatenated with an appropriate representation of *wheres*.
- Three auxiliary attributes for the administration of local bindings, i.e., for calculating *wheres*.

For most of the omitted attributes, their dependencies on other attributes are very simple; quite a few are only used for being able to share the results of intermediate calculations between different attributes. The complete definition of this Haskell conversion is included in the HOPS user manual [Kah98a].

Our implementation converts the resulting attribute value dependencies for an attributed graph into a set of Haskell definitions, for which the top-level attributes are then evaluated by some Haskell implementation. Since Haskell considers top-level bindings and `let`-bindings as mutually recursive, this implements the lazy attribution semantics discussed in [Section 2] without any overhead for the user.

Here the big advantage of our purely declarative approach shines: The user does not even have to think about, for example, the number of traversals necessary to calculate the defined attribution — we estimate that in the operational approach of [Ber96] at least two traversals would be necessary: one for establishing *inLhs* and *inRhs*, sharing status, and initialising some binder-related information, and a second one for assembling the expressions. Consequently, while in the declarative approach the user may concentrate on separation of concerns, in the operational approach the user would have to concentrate on separation of traversals, and on separation of the visits to individual nodes. This obviously will often lead to far less well-structured and maintainable attribution definitions.

Therefore, in our opinion the declarative approach helps to make the analysis of the problem and the structure of the solution much more explicit.

8 Concluding Remarks

We presented a straightforward extension of the attribute grammar approach to cover term graph attributions in close analogy to the original syntax tree attributions. Unlike the mostly operational approaches to be found in the literature, our approach is purely declarative and includes a natural treatment of sharing and the resulting multiplicity of inherited attributes.

Since term graphs are a popular data structure in all kinds of symbolic computation systems, including interpreters, compilers, theorem provers, and proof assistants, the declarative way of defining term graph attributions as presented in this paper is an attractive means of defining output from term graphs, especially where it is relevant to somehow reflect the sharing present in the term graphs into the generated output.

Acknowledgements

We would like to thank the anonymous referees for their useful suggestions.

References

[ACR+88] Bowen Alpern, Alan Carle, Barry Rosen, Peter Sweeney, Kenneth Zadeck. Graph attribution as a specification paradigm. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 121–129, 1988.

[Ber96] Rudolf Berghammer. Wiederverwendbare Algorithmenschemata in ML am Beispiel von Graphdurchlauf-Problemen. *Informatik, Forschung und Entwicklung*, 11(4):179–190, November 1996.

[Der99] Frank Derichsweiler. Strategy Driven Program Transformation within the **H**igher **O**bject **P**rogramming **S**ystem HOPS. In Arnd Poetzsch-Heffter, Jörg Meyer, eds., *Programmiersprachen und Grundlagen der Programmierung*, Informatik Berichte 263 — 1/2000, 165–172. FU Hagen, 1999.

[ERT99] Claudia Ermel, Michael Rudolf, Gabriele Taentzer. The AGG approach: Language and environment. In Grzegorz Rozenberg, ed., *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. World Scientific, Singapore, 1999.

[Far86] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. *ACM SIGPLAN Notices*, 21(7):85–98, 1986.

[Göt82] H. Göttler. Attribute graph grammars for graphics. In Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, eds., *GG '82*, LNCS 153. Springer, 1982.

[HPJW$^+$92] Paul Hudak, Simon L. Peyton Jones, Philip Wadler et al. Report on the programming language Haskell, a non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992. See also `http://haskell.org/`.

[Kah94] Wolfram Kahl. Can functional programming be liberated from the applicative style? In B. Pehrson, I. Simon, eds., *Proc. IFIP 13th World Computer Congress, Vol. I*, 330–335. North-Holland, 1994.

[Kah96] Wolfram Kahl. *Algebraische Termgraphersetzung mit gebundenen Variablen*. Herbert Utz Verlag Wissenschaft, München, 1996. ISBN 3-931327-60-4.

[Kah98a] Wolfram Kahl. *The* **H***igher* **O***bject* **P***rogramming* **S***ystem — User Manual for* HOPS. Fak. für Informatik, Univ. der Bundeswehr München, 1998. electronically available via: `http://ist.unibw-muenchen.de/kahl/HOPS/`.

[Kah98b] Wolfram Kahl. Internally typed second-order term graphs. In J. Hromkovič, O. Sýkora, eds., *Graph Theoretic Concepts in Computer Science, WG '98*, LNCS 1517, 149–163. Springer, 1998.

[Kah98c] Wolfram Kahl. Relational treatment of term graphs with bound variables. *Logic Journal of the IGPL*, 6(2):259–303, March 1998.

[Kah99] Wolfram Kahl. The term graph programming system HOPS. In R. Berghammer, Y. Lakhnech, eds., *Tool Support for System Specification, Development and Verification*, 136–149, Wien, 1999. Springer-Verlag. ISBN: 3-211-83282-3.

[Knu68] Donald E. Knuth. Semantics of context-free languages. In *Mathematical Systems Theory*, vol. 2, 127–145. Springer-Verlag, New York, June 1968.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[Knu90] Donald E. Knuth. The genesis of attribute grammars. In Pierre Deransart, Martin Jourdan, eds., *Attribute Grammars and their Applications (WAGA)*, LNCS 461, 1–12. Springer, 1990.

[LKW93] Michael Löwe, Martin Korff, Annika Wagner. An algebraic framework for the transformation of attributed graphs. In M.R. Sleep, M.J. Plasmeijer, M.C.J.D. van Eekelen, eds., *Term Graph Rewriting: Theory and Practice*, 185–199. Wiley, 1993.

[Paa95] Jukka Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[Sch87] A. Schütte. *Spezifikation und Generierung von Übersetzern für Graphsprachen durch attributierte Graphgrammatiken*. EXpress Edition, 1987. Dissertation, EWH Koblenz.

[Sch90] Andy Schürr. Introduction to PROGRES, an attribute graph grammar based specification language. In Manfred Nagl, ed., *Graph-Theoretic Concepts in Computer Science, WG '90*, LNCS 411, 151–165. Springer, 1990.

[Sch97] Andy Schürr. Programmed graph replacement systems. In Grzegorz Rozenberg, ed., *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, 479–546. World Scientific, Singapore, 1997.

[WG96] Annika Wagner, Martin Gogolla. Defining operational behaviour of object specifications by attributed graph transformations. *Fundamenta Informaticae*, 26:407–431, 1996.

[Wil92] Ross N. Williams. *FunnelWeb User's Manual*, May 1992. Part of the FunnelWeb distribution, available at `http://www.ross.net/funnelweb/`.

[ZM96] Gaby Zinßmeister, Carolyn L. McCreary. Drawing Graphs with Attribute Graph Grammars. In J. Cuny et al., eds., *Graph-Grammars and Their Application to Computer Science, GraGra '94*, LNCS 1073, 443–453. Springer, November 1996.

[ZSB86] Hans Zierer, Gunther Schmidt, Rudolf Berghammer. An interactive graphical manipulation system for higher objects based on relational algebra. In Gottfried Tinhofer, Gunther Schmidt, eds., *WG '86*, LNCS 246, 68–81. Springer, 1986.