Origin Tracking in Attribute Grammars

Kevin Williams and Eric Van Wyk

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN, USA
kwill,evw@cs.umn.edu

Abstract. Origin tracking is a technique for relating the output of a transformation back to its input. In term rewriting systems, where this notion was developed, it relates subtrees in the resulting normal form term to the original term. The technique is useful in several settings, including program debugging and error reporting.

We show how origin tracking can be integrated into higher-order attribute grammars, which construct new syntax trees during attribute evaluation. Furthermore, we extend origins with additional information to track sub trees that correspond to the redex and contractum of rewrite rules when implemented using attribute grammars. The computation of origins and their extensions is formally defined using big-step operational semantics. Finally we describe a program transformation framework as an example use of origin tracking in attribute grammars.

1 Introduction and Motivation

Transformations on syntax trees have many applications, ranging from optimizations which aim to reduce execution time to translating human-readable code down into low-level languages. Such transformations can output trees with non-obvious relations to the transformation's input. Without making explicit relations between the trees, it can be difficult to perceive how the two trees are related. The transformation's output may have been copied from a subtree of the transformation's input or constructed by a transformation based on a specific subtree of the input, but these connections are lost in the transformations.

Origin tracking [5] constructs links from each node in the output tree of a transformation to a node in the transformation's input. In many cases a series of transformations is made to achieve some goal, such as optimization, and origins are traced across multiple steps. Simply put, origins connect a node to the node which introduced it to the tree. Consider a transformation which replaces every negation node *negate* with subtraction from zero. An example of this is shown in Fig. 1. Intuitively, $const(0)$ and the *sub* node were introduced to the tree because the original *negate* node acted as a transformational catalyst. Other nodes in the output tree were not modified by the transformation, and thus have origins pointing back to the nodes they were copied from (the origin of $const(3)$ in the output tree has an origin pointing to the $const(3)$ node in the input tree).

Van Deursen [4] added origin tracking to primitive recursive schemes (PRS), in which evaluation by term rewriting is done in two phases, but we focus on the

B. Combemale et al. (Eds.): SLE 2014, LNCS 8706, pp. 282–301, 2014.
© Springer International Publishing Switzerland 2014

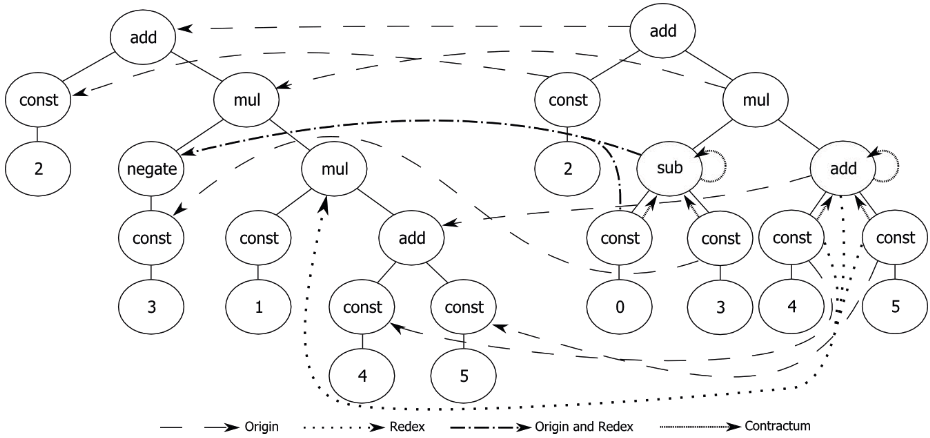

Fig. 1. Input and output for rewrite rules replacing negation with subtraction and removing the multiplicative identity. Links are shown for origins, redex, and contractum.

second here for the purpose of discussing origin tracking and connecting it to attribute grammars. In PRS evaluation, an unordered set of left-linear rewrite rules are applied nondeterministically and exhaustively to a given input tree. For example, the following rewrite rules replace negation with subtraction from zero and reduce multiplication by one on the left:

$$negate(X) \rightarrow sub(const(0), X), \quad mul(const(1), X) \rightarrow X$$

Per the notion of origins in PRS [4], origins for individual nodes are constructed based on where the node is located related to the contractum and if it was explicitly constructed from a (non-variable) term on the right hand side of the rewrite rule. If the node is either disjoint from or above the contractum, then it is given an origin based on the *context* case which points to the node from which it was duplicated in the input tree. For example, this holds for the *add* and *mul* nodes in Fig. 1. If the node is explicitly constructed by the rewrite rule (such as the *sub* and *const* (of 0) nodes in the figure), then it is given an origin based on the *auxiliary symbols* case which points to the root of the redex. Finally, nodes which are copied based on variable bindings in the rewrite rule are given an origin based on the *common variables* case which points to the node from which it was duplicated in the input tree; see the $const(4)$ node in the figure. Note the similarity between origins constructed by the *context* and *common variables* cases: these are origins on nodes copied from the input tree.

In this paper, we migrate this notion of origins into attribute grammars (AGs) [10]. During tree construction, we annotate trees with a path to their origin. We use annotations to hold origins in AGs. Annotations are similar to attributes except they are set on undecorated trees when the tree is built and before its attributes are evaluated or it is used as sub-tree in some other tree construction operation. Annotations are accessed in the same way as attributes

(e.g. *t.anno* where *anno* is an annotation on tree *t*). The operational semantics of this evaluation are presented in Section 2 without origins and in Section 3 with origins.

With origins and without the rewrite rules themselves, it can still be difficult to determine what caused the changes resulting in the *const*(4) node in Fig. 1. By adding a reference for the redex (dotted and dash-dotted lines), it is clear that it was affected by a reducing transformation catalyzed by the lower *mul* node in the input tree, which is not clear with only origin edges. Similarly, adding the contractum arrow for *const*(0) shows that it was not the only node modified by the transformation which constructed it; because its contractum points to its parent, it can assume that its parent is also new. Beyond redex and contractum, we found two additional properties which are useful in exploring transformations: a boolean flag which shows whether the node was newly constructed by the transformation (nodes constructed by the *auxiliary symbols* case in a PRS) and a set of labels which describe the applied transformation. While these four properties have simple implementations within PRS, they are not straightforward to define these within AGs. This is partly due to the abundance of attributes which construct the unmodified nodes and are unnecessary in PRS. The combination of these four new properties with origins are called extended origins and are discussed in Section 4.

In Section 2, we define a simple attribute grammar calculus and the big-step operational semantics of attribute evaluation. The effort to define this operational semantics pays off in Section 3 where it is extended to precisely show how origins can be added to attribute grammars and computed during attribute evaluation, the first contribution of this paper. The second contribution is the definition and specification of *extended* origins in Section 4. This extension adds to each node whether the node was newly constructed by a transformation, the node's redex, the node's contractum, and a set of descriptive labels. Section 5 contains the third contribution, an application to a program transformation specification language based on Halide [12], a transformation tool for optimizing matrix computations. We close with related work in Section 6 and conclude in Section 7.

2 Attribute Grammars

In this section we provide a specification of attribute grammars that is used throughout the paper. After a description of the structure of an attribute grammar we provide a big-step operational semantics for evaluation of expressions in attribute equations without origins. This semantics is then extended in Sections 3 and 4 to compute origins and their extensions during attribute evaluation. Typing rules for expressions are also provided to aid in understanding the distinction between undecorated and decorated trees.

2.1 Definition of the Formalism

In this formulation of attribute grammars we assume a set of primitive types, PT, used in all attribute grammars, where PT includes types $Bool$, Int, Str.

An attribute grammar AG has the form $\langle G, A, O, D\rangle$ where $G = \langle N, P, sig, S\rangle$ is the underlying context free grammar. N is the set of nonterminals. $X = N \cup PT$, and denotes the symbols that appear on the right hand side of productions. P is finite set of production names, each with a signature $sig(p \in P) = x_0 :: N_0 \ ::= x_1 :: X_1 \ ... \ x_{n_p} :: X_{n_p}$ where $n_p \geq 0$. In this formalism, as in our attribute grammar system Silver [13], production signatures provide names for the symbols in a production; these names are then used in attribute equations to refer to nodes in the syntax tree. A function $type_P$ extracts just the type from a production signature such that $type(x_0 :: N_0 \ ::= x_1 :: X_1 \ ... \ x_{n_p} :: X_{n_p}) = N_0 ::= X_1 ... \ X_{n_p}$. $S \in N$ is the type of the root node of a tree representing, for example, a complete program or compilation unit.

The set of attributes $A = \langle A_{syn}, A_{inh}, A_{loc}, type_A\rangle$, contains the finite disjoint sets of names of, respectively, the synthesized, inherited, and local attributes and a mapping of attribute names to types in X. Note that $type_A(a \in A_{syn} \cup A_{inh}) \in X$ since we limit synthesized and inherited attributes to hold only undecorated trees and primitive values. This can easily be generalized to support reference [6] or remote [2] attributes (decorated trees) but we keep things simple in this formalism. $type_A(a \in A_{loc}) \in N$ so that local attributes only hold syntax trees. In the original work on HOAGs [14], this was the case and local attributes were called non-terminal attributes. Note that in Silver and other AG systems, we generalize this to allow local attributes to hold any type, but restrict them here to trees to simplify the discussion.

The "occurs-on" relation $O = \langle O_{attr}, O_{loc}\rangle$ indicates which attributes occur on which nonterminals and which local attributes occur on which productions: $O_{attr} \subseteq (A_{syn} \cup A_{inh}) \times N$ and $O_{loc} \subseteq A_{loc} \times P$. Note that a local attribute has the same type on each production. Though not formalized here, there are no inherited attributes on S.

Attribute equations and functions are specified in $D = \langle EQ, \sigma_f\rangle$. EQ is the set of set of equations indexed by P and have the form $lhs = e$. Expressions e are defined below, and the left hand side lhs, for a production $p \in P$ with $sig(p) = x_0 :: N_0 \ ::= x_1 :: X_1 \ ... \ x_{n_p} :: X_{n_p}$, has the form

$$
\begin{aligned}
lhs ::= \ & x_0.a \text{ where } (a, N) \in O_{syn} \\
| \ & x_i.a \text{ where } i > 0, (a, X_i) \in O_{attr}, a \in A_{inh} \\
| \ & \ell_i.a \text{ where } (a, type_A(\ell_i)) \in O_{attr}, a \in inh \\
| \ & \ell_i \quad \text{ where } (\ell_i, p) \in O_{loc}
\end{aligned}
$$

F is finite set of function names, $F = dom(\sigma_f)$, where σ_f maps function names to lambda-expressions of the form $\lambda y_1 : T_1, ..., y_n : T_n.e$, where y ranges over variables bound in expressions and T ranges over types, defined below.

Fig. 2 shows an attribute grammar, written in Silver, that computes the transformations described in Section 1. Note that here, the process is deterministically driven by a root production $root$ which defines its $doExpd$ local attribute

```
nonterminal Root, Expr;

synthesized attribute expd::Expr
  occurs on Expr;
synthesized attribute simp::Expr
  occurs on Expr;

abstract production negate
e::Expr ::= ne::Expr
{ e.expd = sub(const(0), ne.expd);
  e.simp = negate(ne.simp);  }

abstract production mul
e::Expr ::= l::Expr r::Expr
{ e.expd = mul(l.expd, r.expd);
  e.simp
   = case l of
     | const(1) -> r.simp
     | _ -> mul(l.simp, r.simp)
     end;  }
```

```
abstract production root
r::Root ::= e::Expr
{ local doExpd :: Expr = e.expd;
  local doSimp :: Expr =
    doExpd.simp;  }

abstract production add
e::Expr ::= l::Expr r::Expr
{ e.expd = add(l.expd, r.expd);
  e.simp = add(l.simp, r.simp);  }

abstract production sub
e::Expr ::= l::Expr r::Expr
{ e.expd = sub(l.expd, r.expd);
  e.simp = sub(l.simp, r.simp);  }

abstract production const
e::Expr ::= i::Integer
{ e.expd = const(i);
  e.simp = const(i);  }
```

Fig. 2. Silver syntax specification which replaces negation with subtraction from zero and removes the multiplicative identity

as the expanded tree which replaces negation with subtraction. Similarly, the *doSimp* local attribute removes multiplicative identities from *doExpd*. Two of the attribute equations have obvious connections to the original rewrite rules: *negate*'s *expd* equation resembles the negation expansion rule, and *mul*'s *simp* equation resembles the rule conducting the removal of the multiplicative identity. The remaining attributes serve to reconstruct the tree outside where the rewrite rule would have been applied; in the PRS, this reconstruction is conducted automatically behind the scenes.

Many attributes have dependencies on other attributes on the same production or on its children. Thus attributes without any dependencies are evaluated first, followed by attributes whose dependencies have been evaluated. Thus, for well-defined attribute grammars, evaluation never runs into the case where needed attributes are not defined. Note that references to parent (left hand side), child, and local attribute trees are seen as decorated in attribute equations; this is reflected in the typing rules found in the following section.

2.2 Static and Dynamic Semantics of Expression Evaluation

Here we first discuss the form of expressions (e), values (v), and types (T), as shown in Fig. 3, and present typing and big-step operational semantics evaluation rules for expressions without origins. These rules are relatively straightforward; the only potentially unexpected aspect is that we treat decorated and undecorated syntax trees as having different types and, thus, value representations.

$$
\begin{array}{ll}
e ::= \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & v ::= \texttt{true} \\
\quad \mid\ \texttt{case}\ e\ \texttt{of} & \quad \mid\ \texttt{false} \\
\qquad q_1(y^1_1, ..., y^1_{n_{q_1}}) \Rightarrow e_1 & \quad \mid\ n \\
\qquad ... & \quad \mid\ str \\
\qquad q_n(y^n_1, ..., y^n_{n_{q_n}}) \Rightarrow e_n & \quad \mid\ p(v, ..., v) \\
\quad \mid\ f(e, ..., e) & \quad \mid\ [n, ..., n] \\
\quad \mid\ var & \quad \mid\ \lambda y_1 : T_1, ..., y_n : T_n.e \\
\quad \mid\ var.attr & T ::= PT \\
\quad \mid\ p(e, ..., e) & \quad \mid\ N \\
\quad \mid\ \texttt{new}\ var & \quad \mid\ N ::= X...X \\
\quad \mid\ v & \quad \mid\ T...T \rightarrow T \\
var ::= x_0 \ \mid\ x_i, i > 0 \ \mid\ \ell_i \ \mid\ y & \quad \mid\ Ref\ \ N
\end{array}
$$

Fig. 3. The form of expressions e, variables var, values v, and types T

The primary reason for the formality here is to be provide a precise means for specifying the computation of origin, redex, and contractum information in the later, extended version of these evaluation rules.

Expressions include if-then-else expressions, case-expressions, and function application that behave as one would expect in a functional language; these are listed first in the productions over e. Case expressions also introduce variable bindings which again are denoted by y_i. Expressions also include variable references, var, of which there are four varieties: references to the tree constructed by a production and named by the variable on the root/left-hand side (x_0) and child trees ($x_i, i > 0$). Local attributes (ℓ_i) and bound variables y round out the types of variables, all denoted by var.

Attributes may be referenced on decorated syntax trees, denoted $var.attr$. The restriction of $var.attr$, and not allowing $e.attr$, ensures that attributes are only accessed of the production root, children, locals or variables bound by functions or case-expressions. The restriction is removed in Silver and most AG systems but it keeps things simple here. Tree construction, $p(e_1, ..., e_{n_p})$, constructs new undecorated syntax trees. Synthesized attributes cannot be accessed on such trees; the process of decorating the root node of an undecorated tree with its inherited attributes converts it to decorated tree.

Expressions also include the values, v, to which which expressions evaluate, also shown in Fig. 3. These include boolean, numeric (n), and string (str) literals. Tree literals $p(v, ..., v)$ are undecorated trees; they are simply terms in the language of the grammar G. Paths, $[n_1, ..., n_k]$, are sequences of integers describing a path to a subtree. The empty path [ ] refers to the root node of the original syntax tree, [1] refers to the first child of that root node, and the path $[1, 2]$ refers to the second child of that first child. For example, the *negate* node in Fig. 1 is referenced by the path $[2, 1]$. Finally, lambda expressions are also values.

Types include primitive types PT, undecorated trees with nonterminal of type N at the root, production types $N ::= X...X$, function types $T...T \rightarrow T$ and path types, $Ref\ \ N$, for paths to trees of type N.

Fig. 4 contains the big-step operational semantics of the evaluation of expressions, these rules have the form $\sigma, t \vdash e \to v$ indicating that for an environment σ mapping bound variables y to values, and expression e that is part of an equation for the production that constructed the tree t, evaluates to value v.

The figure also has typing rules to assisting in understanding evaluation. These have the form $AG, p, \Gamma \vdash e\colon T$ indicating that an expression e in an equation associated with production p in AG has type T where Γ maps bound variables to their types.

Before beginning, we note one additional form of type rule for production, function, and attribute names of the form $AG \vdash e\colon T$ since these are done independently of any production or equation. Specifically, $AG \vdash p\colon T$, $AG \vdash f\colon T$, and $AG \vdash a\colon T$ indicate that, respectively, a production p, function f or attribute a has the indicated type. These are straightforward and not formalized here, they simply refer to the appropriate components of AG.

Variable references: Inside of equations for a production we consider the variables representing the root, child nodes, and local attributes to be decorated trees, and thus their type is *Ref N* and their values are paths to the appropriate nodes. For example, in Fig. 4 the rule T-Root indicates that the root node variable x_0 is a reference to the nonterminal on the left hand side of the production p. Rule E-Root indicates that x_o evaluates to the path t on the left hand side of the turnstile — this is the path to the tree on which this expression is being evaluated. Child node variables x_i are typed similarly and evaluate to the path to the tree t extended to denote their sub-tree. Local attributes are declared to have the type of a nonterminal, just like child trees in productions and similarly the production has equations defining the inherited attributes on each local attribute. Thus their type in expressions are decorated trees, represented by paths. The negation of the index i for the local ℓ_i is used in specifying the path to this local decorate tree. Bound variables are bound to types and values and are found in Γ and σ, respectively.

Attribute access: The type rule T-SynInh checks that attributes are accessed on decorated trees only, and that the attribute decorates the tree and thus determines its type. The rule E-SynInh indicates that because the parent node, each child node, and local variables are typed as decorated trees, synthesized and inherited attributes can be accessed from them. (Note that local attributes are accessed by name directly, without the "dot" notation shown above.)

Tree construction: Productions are used like functions to build undecorated trees of some nonterminal type N, and are essentially just terms in the language of G. Child expressions are evaluated to values that match the production's type.

When an equation copies an undecorated tree value into a higher order synthesized or inherited attribute (of the same type), it is simply that same undecorated tree that is stored in the attribute. On the other hand, when an equation copies such a value into a local attribute, then that undecorated tree becomes a decorated tree in the sense that it can now be given inherited attributes and then have synthesized attributes computed on it.

$$\frac{AG \vdash p\colon N ::= X_1 ... X_n}{AG, p, _ \vdash x_0 \colon Ref\ N} \text{ (T-Root)}$$

$$\sigma, t \vdash x_0 \rightarrow t \quad \text{(E-Root)}$$

$$\frac{AG, p, \Gamma \vdash var\colon Ref\ N \quad (attr, N) \in O_{attr} \quad AG \vdash attr\colon T}{AG = \langle G, A, \langle O_{attr}, _\rangle, p, \Gamma \vdash var.attr\colon T} \text{ (T-SynInh)}$$

$$\frac{\sigma, t \vdash var \rightarrow h}{\sigma, t \vdash var.attr \rightarrow h.attr} \text{ (E-SynInh)}$$

$$\frac{i > 0 \quad AG \vdash p\colon N ::= X_1 ... X_n}{AG, p, _ \vdash x_i \colon Ref\ X_i} \text{ (T-Child)}$$

$$\frac{i > 0}{\sigma, t \vdash x_i \rightarrow t \cdot [i]} \text{ (E-Child)}$$

$$\frac{AG \vdash p\colon N ::= X_1, ..., X_n \quad \forall i_n^1 (AG, p, \Gamma \vdash e_i \colon X_i)}{AG, p, \Gamma \vdash p(e_1, ..., e_n)\colon N} \text{ (T-Tree)}$$

$$\frac{\forall i_n^1 (\sigma, t \vdash e_i \Rightarrow v_i)}{\sigma, t \vdash q(e_1, ..., e_n) \Rightarrow q(v_1, ..., v_n)} \text{ (E-Tree)}$$

$$\frac{AG \vdash \ell\colon N}{AG, p, _ \vdash \ell_i \colon Ref\ N} \text{ (T-Local)}$$

$$\sigma, t \vdash \ell_i \rightarrow t \cdot [-i] \quad \text{(E-Local)}$$

$$\frac{AG, p, \Gamma \vdash var\colon Ref\ X}{AG, p, \Gamma \vdash \texttt{new}\ var\colon X} \text{ (T-New)}$$

$$\frac{\sigma, t \vdash var \rightarrow h}{\sigma, t \vdash \texttt{new}\ var \rightarrow *h} \text{ (E-New)}$$

$$\frac{(y, T) \in \Gamma}{AG, p, \Gamma \vdash y\colon T} \text{ (T-BVar)}$$

$$\frac{(y, v) \in \sigma}{\sigma, t \vdash y \rightarrow v} \text{ (E-BVar)}$$

$$\frac{AG, p, \Gamma \vdash e\colon Ref\ N \quad \forall i_n^1 (AG \vdash q_i \colon N ::= X_1^i ... X_{n_{q_i}}^i) \quad \forall i_n^1 (AG, p, \Gamma[r_1^i \mapsto Ref\ X_1, ..., r_{n_{q_i}}^i \mapsto Ref\ X_{n_{q_i}}] \vdash e_i \colon T)}{AG, p, \Gamma \vdash \texttt{case}\ e\ \texttt{of}\ q_1(y_1^1, ..., y_{n_{q_i}}^1) \Rightarrow e_1\ ...\ q_n(y_1^n, ..., y_{n_{q_n}}^n) \Rightarrow e_n \colon T} \text{ (T-Case)}$$

$$\frac{\sigma, t \vdash e \rightarrow h \quad q_i = prod(*h) \quad \sigma[y_1^i \mapsto h \cdot [1], ..., y_{n_{q_1}}^i \mapsto h \cdot [n_{q_i}]], t \vdash e_i \rightarrow v}{\sigma, t \vdash \texttt{case}\ e\ \texttt{of}\ q_1(y_1^1, ..., y_{n_{q_1}}^1) \Rightarrow e_1\ ...\ q_n(y_1^n, ..., y_{n_{q_n}}^n) \Rightarrow e_n \ \rightarrow v} \text{ (E-Case)}$$

$$\frac{AG, p, \Gamma \vdash e_1 \colon Bool \quad AG, p, \Gamma \vdash e_2 \colon T \quad AG, p, \Gamma \vdash e_3 \colon T}{AG, p, \Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \colon T} \text{ (T-If)}$$

$$\frac{\sigma, t \vdash e_1 \rightarrow \texttt{true} \quad \sigma, t \vdash e_2 \rightarrow v}{\sigma, t \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightarrow v} \text{ (E-IfTrue)}$$

$$\frac{\sigma, t \vdash e_1 \rightarrow \texttt{false} \quad \sigma, t \vdash e_3 \rightarrow v}{\sigma, t \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightarrow v} \text{ (E-IfFalse)}$$

$$\frac{AG \vdash f\colon T ::= T_1, ..., T_n \quad \forall i_n^1 (AG, p, \Gamma \vdash e_i \colon T_i)}{AG, p, \Gamma \vdash f(e_1, ..., e_n)\colon T} \text{ (T-FuncApp)}$$

$$\frac{\sigma_f(f) = \lambda y_1 \colon T_1, ..., y_n \colon T_n . e \quad \forall i_n^1 (\sigma, t \vdash e_i \rightarrow v_i) \quad \sigma[y_1 \mapsto v_1, ..., y_n \mapsto v_n], t \vdash e \rightarrow v}{\sigma, t \vdash f(e_1, ..., e_n) \rightarrow v} \text{ (E-FuncApp)}$$

Fig. 4. Typing and evaluation rules for expressions without origins

New: As described by rule E-NEW, the *new* operator extracts the value (an undecorated tree or primitive value) that a path refers to. (confirmed by the type rule T-NEW). It uses a dereference operator $*$ to do this. A path refers to a decorated tree or a primitive value, the dereference operator extracts a new undecorated tree from that path. In the case of a primitive value it just returns it. Note that in this formulation only variable accesses evaluate to references.

Case: The type rule T-CASE requires that the expression to be matched, e, be a reference to a tree with type N, each production to be matched, p_i, must have n_{p_i} children, and each expression e_i has the same type. Note that the types added to Γ for evaluation of the case clause expression are converted to *Ref* types. This is the same process used in the type rules of parent, child, and local variables since all of these will be seen as decorated (*Ref*) trees in the evaluation of the expression. The rule E-CASE matches the result of evaluating e with one of the given productions p_i, binds each y_j^i to the jth child of the value of e, and evaluates the ith expression e_i.

Other constructs: The typing and evaluation rules for if-then-else expressions and function application are the same as in simple functional languages.

3 Origin Tracking in Attribute Grammars

In Section 2, we defined how attributes are evaluated within AGs without origins. In this section, we define how attributes are evaluated with origins. The semantics in that section were defined so that only a few key modifications need to be made to compute origins during expression evaluation, as described below.

As discussed above, the origin is defined as an annotation which contains a reference to the node's origin. In the case of initial trees, the origin is defined as $\perp$. We redefine the language of values v to replace the tree value $p(v, ..., v)$ with the tree value with an origin with a vertical bar to divide it from the node's children: $p(v, ..., v|o)$. None of the typing rules require modification, but two evaluation rules must be updated. These two rules (E-TREE and E-NEW) are replaced by the two rules shown in Fig. 5. The rule E-O-TREE is only different from E-TREE in that it gives the constructed tree an origin pointing to the tree on which the attribute is defined.

Where E-NEW used $*h$, the rule E-O-NEW uses $duplicate(h)$, the function *duplicate* is defined in Fig. 6. If *duplicate* is passed a path to a primitive value, then it returns that value. If *duplicate* is passed a path to a (decorated) tree it constructs an undecorated copy of the tree with origins on the new tree pointing to the corresponding nodes on the original tree. Note that *duplicate* mimics the *common variables* case of PRS origins discussed in Section 1 in that a subtree is copied into the result of the transformation's output such that its origins point back to the transformation's input.

If we replace the expression for the *simp* equation on production *mul* in Fig. 2 with $simplify(l, r)$ where *simplify* is a function whose body is the `case` expression

$$\frac{\forall i_n^1 (\sigma, t \vdash e_i \rightarrow v_i)}{\sigma, t \vdash q(e_1, ..., e_n) \rightarrow q(v_1, ..., v_n | t)} \quad \text{(E-O-Tree)}$$

$$\frac{\sigma, t \vdash var \rightarrow h}{\sigma, t \vdash \texttt{new}\ var \rightarrow duplicate(h)} \quad \text{(E-O-New)}$$

Fig. 5. New rules required to add origins to AGs. The "E-O-" prefix in the name of each of the above rules means that the rule replaced the similarly named rules from Fig. 4 with the "E-" prefix. Note that adding origins does not affect the typing relations.

$$\begin{array}{l} duplicate(h) = \\ \quad \textit{if } type(*h) \in PT \textit{ then } *h \\ \quad \textit{else case } *h \textit{ of} \\ \qquad q(t_1, ..., t_k|_) \rightarrow q(\ duplicate(h \cdot [1]), ..., duplicate(h \cdot [k]) \mid h\ ) \end{array}$$

Fig. 6. Definition of duplicate with origins using pseudo code

currently in the figure, then the origin computed for any tree now constructed or duplicated in that function is the same as if the function *simplify* was not called and the original specification was used. This is because the evaluation rule E-FuncApp in Fig. 4 uses the same tree t in the context of evaluating $f(e_1, ..., e_n)$ as in the context of evaluating the body of f. Thus, origins are dependent on the attribute being evaluated, not the functions used in that evaluation.

To simplify interaction with the generated origins, we define the function *getOrigin* such that $getOrigin(p(t_1, ..., t_k|o)) = o$. A tree's origin *path* is generated by repeatedly calling *getOrigin* on its output until it returns $\perp$ (signifying the initial tree has been reached). Note that origin paths and paths are different: *origin paths* are ordered sequences of trees, and paths $[n, ..., n]$ as seen in v are ordered sequences of integers used to locate decorated trees. This function will be added to the interface defined in the next section.

4 Extending Origin Tracking with Transformation Information

Origins are useful for constructing paths from the result of a set of transformations to the initial tree. However, the information provided by origins does not always provide all of the information that we may want from a transformation. Specifically, the answers to the following four questions are missing:

- Was the tree newly constructed by the transformation in question?
- What is the root of the transformation's input (its redex)?
- What is the root of the transformation's output (its contractum)?
- Why did the transformation happen?

We define a set of functions to provide an interface for answering these questions. The first question is answered by a function *getIsContractum* that returns true

on subtrees which were not just copied from the previous tree (i.e. *true* for nodes with *auxiliary symbols* origins). The second question is answered by *getRedex* which returns a path to the redex of the transformation, and the third is answered by a function *getContractum* which returns a path to the contractum of the transformation. The fourth question is answered by a function *getLabels* which returns a set of *label*s for a given subtree where each *label* contains a characterization of the transformation which constructed the subtree.

These four functions, along with origins, make our interface for extended origins. Two of these functions (*getIsContractum* and *getLabels*) directly return annotations pulled off of their argument while the others compute their results from new annotations.

4.1 The Extended Origins Interface

In this section, we define the interface functions and state some invariants on their behavior.

The function *getIsContractum* returns whether a node was newly introduced by the last applied transformation, and requires a new annotation of type *bool* called *isContractum* such that $getIsContractum(t) = t.isContractum$. This annotation is set so that the nodes with *context* or *common variables* origins in the PRS setting define *isContractum* to be *false* and those nodes with *auxiliary symbols* origins define *isContractum* to be *true*.

To set *isContractum* we must be able distinguish between attribute equations that implement a rewrite rule and set *isContractum* to *true* (such as the definition of *expd* on *negate* and *simp* on *mul* in Fig. 2) and those that direct the transformation and set *isContractum* to *false* (such as the both attributes on *sub*). The expression $p(e_1, ..., e_n)$ is evaluated with $isContractum = true$ unless three conditions hold, indicating that *isContractum* should be set *false*. These are:

- p matches the production of the tree the attribute is evaluated on,
- each e_i is either x_i or $x_i.attr$ for some attribute *attr*, and
- the constructed tree will be the root (not some subtree) of the tree eventually computed as the value of the attribute whose equation is being evaluated.

The first two conditions are simple to validate, and the third is determined by a new boolean flag *er* which is added to the left of the turnstile in the evaluation relation defined below. In our running example, in the *expd* attribute on *negate*, the *sub* node is evaluated where $er = true$ and the $const(0)$ node is evaluated where $er = false$.

getLabels requires a new finite set L with type $labels \times P \times A$ which statically defines labels for each attribute on each production. Calling $getLabels(t)$ returns the set of labels associated with the production p and attribute *attr* which constructed t, denoted L^p_{attr}. These labels may be different for every application, but possible labels for AGs include “translation”, “rephrasing”, “local”, “inherited”, and “synthesized”. Other customizable labels refer to the task completed

```
getRedex(t) =                                  getContractum(t) =
    if t = ⊥ then ⊥                                if t = ⊥ then ⊥
    else if t.redex ≠ ⊥ then t.redex               else if t.redex ≠ ⊥ then t
        else getRedex(parent(t))                       else getContractum(parent(t))
               (a)                                            (b)
```

Fig. 7. Definitions of getContractum and getRedex using pseudo code

by a given attribute, such as “replace negation with subtraction”. Though these labels are strings, we do not exclude the possibility for labels of other types.

getRedex and *getContractum* require a single new annotation called *redex* which contains either a path to the redex of the tree the annotation resides on or $\perp$, indicating that there is no redex. Both *getContractum* and *getRedex* are defined using a helper function *parent* which returns the parent node of its given subtree or $\perp$ if it does not have a parent. *getRedex* is defined in Fig. 7(a), and *getContractum* is defined in Fig. 7(b).

Consider the following nodes in the output tree in the example from Section 1: *sub*, the inner *add*, and *mul*. The *sub* node, t_s, was constructed by the *expd* attribute on *negate* and $getIsContractum(t_s) = true$, $getRedex(t_s)$ returns a path to the *negate* node, and $getContractum(t_s)$ returns a path to t_s. The *add* node, t_a, was copied by the inner *mul* in the input tree using a new *copy* which defines t_a's redex as a path to the inner *mul* in the input tree. Also, $getIsContractum(t_a) = false$ and $getContractum(t_a)$ returns a path to t_a. The *mul* node, t_m, was unchanged by the transformation and is not new, so $getIsContractum(t_m) = false$, and $getRedex(t_m) = getContractum(t_m) = \perp$.

Below are invariants relating the above functions and origins on a tree t with children $t_1, ..., t_n$. Each invariant is followed by a brief description.

$$getOrigin(t) = \perp \implies \neg getIsContractum(t) \wedge getRedex(t) = \perp \wedge getContractum(t) = \perp \wedge getLabels(t) = \{\}$$

If the origin is undefined (which only occurs on initial trees) then the above are default values for each of the properties.

$$getIsContractum(t) \implies getOrigin(t) \neq \perp \wedge getRedex(t) \neq \perp \wedge getContractum(t) \neq \perp$$

If the tree was constructed by a transformation, then its origin, redex, and contractum are defined.

$$getOrigin(t) \neq \perp \implies \forall t_i (getOrigin(t_i) \neq \perp)$$

If the origin is defined, then the origin of every child of t is defined.

$$(getRedex(t) \neq \perp \wedge getContractum(t) \neq \perp) \implies \forall t_i (getRedex(t_i) \neq \perp \wedge getContractum(t_i) \neq \perp)$$

If a tree defines both its redex and contractum, then each of its children define their redexes and contractums.

$$getRedex(t) \neq \bot \iff getContractum(t) \neq \bot$$

The redex is defined if and only if the contractum is defined. This is should be clear from each of their definitions.

4.2 Evaluating Extended Origins in Attribute Grammars

As seen above, extending origins requires three new annotations: *isContractum*, *redex*, and *labels*. Thus the tree value form $p(v_1, ..., v_n|v)$ in v is replaced by $p(v_1, ..., v_n|v, v, v, v)$ where the first annotation is the node's origin, the second holds *isContractum*, the third holds *redex*, and the last holds *labels*.

Also, two items are added to the left of the turnstile in the evaluation rules: *er* (used for setting *isContractum*) and the name of the attribute being evaluated, a, to find the correct set of labels. Thus evaluation rules have the form

$$\sigma, t, a, er \vdash e \rightarrow v.$$

Many of the evaluation rules used for origins are only changed to use this extended form and thus are not shown. Some only require the addition of the two variables in the consequent, as shown here:

$$\frac{i > 0}{\sigma, t, a, er \vdash x_i \rightarrow t \cdot [i]} \quad \text{(E-EO-CHILD)}$$

This applies to E-EO-ROOT, E-EO-LOCAL, and E-EO-BVAR. Others, including E-EO-IFTRUE, E-EO-IFFALSE, E-EO-CASE, and E-EO-FUNCAPP, simply use the new form in the antecedent, passing along the new values a and er in the evaluation of their component expressions. Recall that function application with origins constructs origins based on the tree on which the attribute is being evaluated. Similarly, the annotations introduced in extended origins are constructed independently of the function being evaluated as they are also passed along as values to the left of the turnstile.

The rule for attribute access requires a notable modification. Consider the reducing transformation conducted by *mul* in the example in Fig. 1. If the left child of *mul* is *const*(1), then the node's *simp* attribute returns a copy of the *simp* attribute on the node's right child. If tree copying remains unchanged and copies every annotation on the tree, then the resulting attribute might not define the correct redex. In our example, it would not define any redex. This is inconsistent with the description of *getRedex* which should define a redex because a transformation has changed the tree. We explicitly define the *copy* functionality for attribute access for extended origins. The *copy* is shown in Fig. 8(a), and the new rules are shown here:

$$\frac{\sigma, t, a, true \vdash var \rightarrow h}{\sigma, t, a, true \vdash var.attr \rightarrow copy(h.attr, t)} \quad \text{(E-EO-SYNINHR)}$$

$$
\begin{array}{l}
copy(t', r') = \\
\quad if\ type(t') \in\ PT\ then\ t' \\
\quad else\ case\ t'\ of \\
\qquad q(t'_1, ..., t'_k | o, n, r, l) \rightarrow \\
\qquad\quad q(copy(t'_1, \bot), ..., copy(t'_k, \bot)), \\
\qquad\qquad | \ o, n, if\ r' \neq \bot\ then\ r'\ else\ r, l))
\end{array}
$$

(a)

$$
\begin{array}{l}
duplicate(h, r', l') = \\
\quad if\ (type(*h) \in PT\ then\ *h \\
\quad else\ case\ *h\ of \\
\qquad q(t'_1, ..., t'_k | o, n, r, l) \rightarrow \\
\qquad\quad q(duplicate(h \cdot [1], \bot, l'), ..., \\
\qquad\qquad duplicate(h \cdot [k], \bot, l') \\
\qquad\qquad | \ h, false, r', l')
\end{array}
$$

(b)

Fig. 8. Definitions of *copy* and *duplicate* for extended origins using pseudo code. *copy* only modifies the redex if r' is not $\bot$, and *duplicate* specifies every annotation.

$$\frac{\sigma, t, a, false \vdash var \rightarrow h}{\sigma, t, a, false \vdash var.attr \rightarrow copy(h.attr, \bot)}\ \text{(E-EO-SynInhNR)}$$

In E-EO-SynInhR, the expression will return a value which is the root of the value computed for attribute a, so the value of the attribute *attr* on h is modified to have a redex pointing to t. In E-EO-SynInhNR, the expression will not be the root of the value on attribute a, so it is copied with an undefined local redex.

The rules for *new*, and thus the *duplicate* function, must be modified to construct correct values for new annontations *isContractum*, *redex*, and *labels* for duplicated trees. Our original example does not include any such *common variables* cases, for example if in the *simp* equation on *mul* we replaced $r.simp$ with just r. In this case the new tree should have *isContractum* set to *false* and *redex* set to a path to the *mul* node. We define a new *duplicate* which modifies the one in Fig. 6 and inserts the new annotations. The new definition of *duplicate* is shown in Fig. 8(b), and the new rules that replace E-New are shown here:

$$\frac{\sigma, t, a, true \vdash var \rightarrow h}{\sigma, t, a, true \vdash \texttt{new}\ var \rightarrow duplicate(h, t, L_a^{prod(t)})}\ \text{(E-EO-NewR)}$$

$$\frac{\sigma, t, a, false \vdash var \rightarrow h}{\sigma, t, a, false \vdash \texttt{new}\ var \rightarrow duplicate(h, \bot, L_a^{prod(t)})}\ \text{(E-EO-NewNR)}$$

In E-EO-NewR, *new* is evaluated such that the given path is duplicated and given t as a new redex if $er = true$ and $\bot$ if $er = false$.

This last set of rules demonstrates the greatest difference between the evaluation of origins and extended origins. Since we need to determine if a tree is part of the contractum to set *isContractum* and set its *redex* annotation the single rule E-O-Tree is replaced by three rules shown in Fig. 9. Rule E-EO-NotCntr defines the case in which the constructed tree has a *context* or *common variables* type of origin and is not a constructed as part of the contractum (abbreviated Cntr in rule names). In this case the constructed tree does not have a redex and sets *isContractum* to *false*. The *mul* node in the

$$\frac{q = prod(*t) \quad \forall i_n^1(e_i = \texttt{new}\ x_i \vee e_i = x_i.attr) \quad \forall i_n^1(\sigma, t, a, false \vdash e_i \Rightarrow v_i)}{\sigma, t, a, true \vdash q(e_1, ..., e_n) \Rightarrow q(v_1, ..., v_n | t, false, \bot, L_{attr}^{prod(t)})} \text{(E-EO-NotCntr)}$$

$$\frac{\neg(q = prod(*t) \ \wedge \ \forall i_n^1(e_i = \texttt{new}\ x_i \vee e_i = x_i.attr)) \quad \forall i_n^1(\sigma; t, a, false \vdash e_i \to v_i)}{\sigma; t, a, true \vdash q(e_1, ..., e_n) \to q(v_1, ..., v_n | t, true, t, L_a^{prod(t)})} \text{(E-EO-CntrRoot)}$$

$$\frac{\forall i_n^1(\sigma; t, a, false \vdash e_i \to v_i)}{\sigma; t, a, false \vdash q(e_1, ..., e_n) \to q(v_1, ..., v_n | t, true, \bot, L_a^{prod(t)}} \text{(E-EO-CntrChild)}$$

Fig. 9. Tree construction rules for extended origins

original example's output is an example of this. E-EO-CntrRoot defines the case where the tree being constructed may be the root of the computed attribute value and is part of the contractum, resulting in a node which defines its redex to be t and $isContractum = true$. This resembles the *auxiliary symbols* origin case, and the *sub* node in the original example's output is an example of this. The final rule, E-EO-CntrChild, the constructed tree sets *isContractum* to *true* and has no redex since it is not the root of the value of the computed attribute. This resembles the *auxiliary symbols* origin case, and $const(0)$ in the original example's output is an example of this. Recall, setting *redex* to $\bot$ does not mean that the *getRedex* function will not be able to find the root of the redex on a parent node.

5 Applying Extended Origins

This section explores an application of extended origins to a language extension built using Silver. This extension is for parallel matrix programming [15] based on ideas from Halide [12], a tool intended for writing high-performance image processing code which separates the "algorithm" (the operations to be evaluated) from the "schedule" (the transformations which specify the order in which the operations are evaluated). The schedules in Halide are designed to not affect the semantics of the algorithm and only modify where and when operations take place (e.g. by tiling, parallelizing, or vectorizing loops).

As an example of this, the code in Fig. 10(a) constructs a 2-dimensional gradient matrix *grad* based on indexes x and y. The result of applying the two schedules is shown in Fig. 10(b). The two schedules have parallelized the y dimension (`parallelize y`) and designated the y loop as the outermost loop (`reorder y, x`). These are the only schedules discussed in this paper, but we do not claim that these two schedules are sufficient for high performance computing; instead, they were selected based on their transformations and how they interact with extended origins.

In this small example, many relations are obvious. The OpenMP pragma must have been generated in some way by the *parallelize* schedule and the y iteration occurs outside of the x iteration due to the *reorder* schedule. Consider if this

```
grad(x,y) = x + y {
  parallelize y;
  reorder y, x;
}
```

(a)

```
#pragma omp parallel for ...
for y from 0 to yMax {
  for x from 0 to xMax {
    grad[x][y] = x + y;
  }
}
```

(b)

Fig. 10. Example's input and output

example included more schedules which closely interacted with each other and were more invasive, thereby obfuscating relationships between the output code and the initial schedules and algorithm. Such a set of schedules would output code without any simple connection back to the original code.

By adding extended origins to this implementation, we can connect each node of the output tree to the schedules which affected it. Intuitively, each OpenMP pragma should be connected to a parallelize schedule, and each reorder schedules should be connected to the nodes they rearrange.

Here, we briefly describe how an AG transforms the input code shown in Fig. 10(a) into the code in Fig. 10(b). First, the algorithm is expanded into nested for loops, each of which is encapsulated within its own *forMarker* node. The expression nested in the deepest loop is a transformed version of the assignment statement in the original code: `grad[x][y] = x + y` under a *bodyMarker*. The marking nodes are used to mark where the tree should be cut when applying the reorder schedule. This simplifies the reorder schedule because other schedules which add new nodes must decide whether the added nodes should stay inside a given loop (inside a loop and above the nested marker) or outside a given loop (below a marker and above its loop). The first schedule is transformed into its *ScheduleAsRoot* variation which has the remaining schedules and the current state of the algorithm as its children. After applying its transformation, it replaces itself with the next schedule's *ScheduleAsRoot* node. After all schedules have been applied, the markers are removed and the final tree is returned.

The *parallelize* schedule inserts an OpenMP pragma immediately before the loop iterating over the given variable. To do this, a new higher-order synthesized attribute *parallel* is defined on all nonterminals which replicates constructs not affected by the transformations using equations similar to those on the *add* production in the running example. On the loop which iterates over the variable to be parallelized, the *parallel* attribute holds the sequence of the new pragma followed by a copy of the original loop. Initially, the for-loop compared its iterating variable against an inherited attribute *parWith* which held the variable to be parallelized and, if the two variables matched, constructed the new pragma. However, this gave the pragma an origin pointing to the loop, and therefore cannot connect the pragma to the *parallelize* schedule.

We define *parWith* to have type *ParWith*, a nonterminal which defines the new pragma as one of its attributes and with only one production which contains the variable to be parallelized as a child. The *ParWith* node is constructed by the *parallelizeAsRoot* node, which was in turn constructed by the *parallelize*

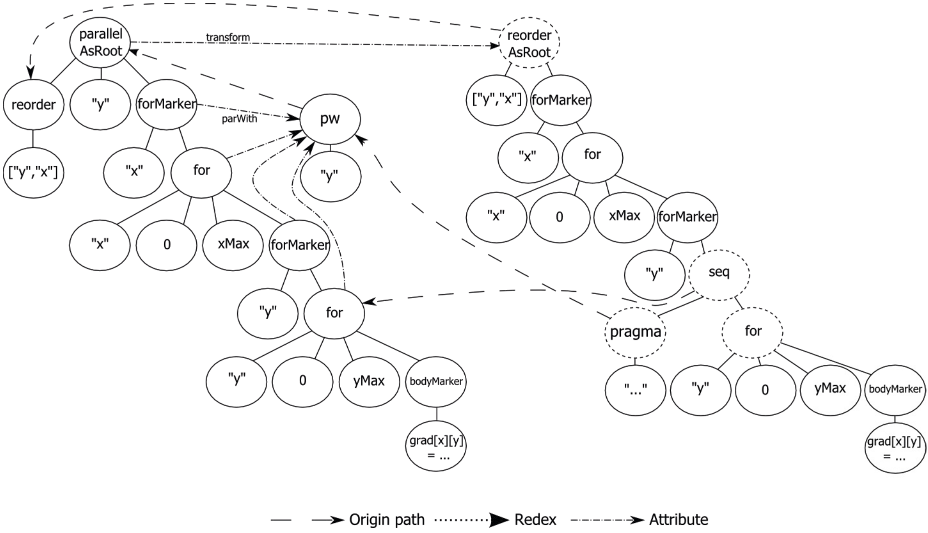

Fig. 11. Diagram showing the result of applying the parallelization schedule using *parWith* where the type of *parWith* is *ParWith*. Origins are shown with dashed arrows, attributes are shown with dot-dash arrows, and new nodes are shown with dashed ovals. Note that the pragma's origin path includes both the *pc* and *parallelizeAsRoot* nodes, and therefore also includes the *parallelize* node.

schedule; thus the *parWith* tree's origin path includes the *parallelize* schedule. In the loop's definition of *parallel*, the loop copies the pragma attribute from its *parWith* attribute. Thus the origin path of the pragma leads through the *parWith* tree to the *parallelizeAsRoot* node and the *parallelize* schedule. This relation is depicted in Fig. 11. Had we instead defined *parWith* to be a string instead of a tree, this origin path would not exist and we would lose the relationship between the parallel loop and the parallel schedule.

The reorder schedule acts as one would expect: it splits the loops into fragments rooted at *forMarker*s or *bodyMarker* nodes, rearranges the fragments, and re-nests the fragments in the new order. Note that this transformation outputs nodes which are duplicates of the input nodes, so none of the output tree's nodes have the *reorder* schedule in their origin path. Instead, the connection to the schedule is facilitated by the *redex* property. The reordering of the schedules is conducted within local attributes on *reorderScheduleAsRoot*, so the nodes in the ordered list of fragments have redexes pointing to it. Thus, each of the output nodes are connected to the schedule via origins to the ordered list, a redex to *reorderScheduleAsRoot*, and an origin to the *reorder* schedule. Though this connection seems hard to find, the local attribute holding the reordered fragments can be given a label which suggests following the redex property to find the schedule which conducts the reordering.

6 Related Work

The example in Section 1 is based on van Deursen's description of origin tracking in primitive recursive schemes (PRS) [4]. In our addition of redex and contractum information to origins in attribute grammars we designed the evaluation rules for tree construction to distinguish the equations which correspond directly to rewrite rule transformations (whose origins correspond to the *auxiliary symbols* case in a PRS) from those that simply reconstruct the tree (whose origins correspond to the *context* and *common variables* cases). The reason we focus on the rewrite rules from the second phase of a PRS is that the first phase includes rewrite rules that more closely resemble attribute grammar equations. The expansion of *negate* would be specified by the following rules:

$$\begin{aligned} expd(negate(X)) &\rightarrow sub(const(0), expd(X)) \\ expd(mul(X,Y)) &\rightarrow mul(expd(X), expd(Y)) \\ expd(sub(X,Y)) &\rightarrow sub(expd(X), expd(Y)) \\ expd(const(N)) &\rightarrow const(N) \end{aligned}$$

Here, *expd* corresponds to a synthesized attribute in attribute grammars; the rules above can be easily transcribed into attribute grammar equations.

In fact, this is done in previous work [11] in which bidirectional transformations are specified as rewrite rules and then implemented in attribute grammars. In that work, the translation of rewrite rules to attribute equations defines a similar notion of origins, called "links-back", but these are not implemented on general attribute equations. "Links-back" are only generated from rewrite rules, significantly simplifying the process.

PRSs and AGs can be encoded in the other formalism [3], but adding origins to attribute grammars by encoding a PRS with origins as an AG is not as intuitive as a direct approach. Additionally, the translation approach does not support the extension of redex and contractum information to origins.

Various language processing systems have implemented origins tracking. These include Spoofax [8], based on strategic term rewriting; CENTAUR [1], implemented in Lisp and Prolog with some notion of attributes similar to annotations as described here; and in the meta-programming language Rascal [9].

The annotations for origins and redexes are implemented in Silver as reference [6]/remote [2] attributes; these allow graph structures to be defined on top of syntax trees using attributes that point to other nodes in the syntax tree. They are useful in many settings such as linking variable uses to their declarations.

7 Discussion and Conclusion

In Silver, many of the above restrictions imposed by the simple attribute grammar calculus in Section 2 are removed since the restrictions can easily be generalized. In addition to the generalizations mentioned earlier, the **new** construct is not used in Silver because Silver uses the context of a reference to a tree

such as x_i to determine if it should be seen as a decorated or undecorated tree. For example, it is decorated on the right hand side of an attribute equation for attribute evaluation and case expressions, but undecorated otherwise.

One concern regarding this definition of evaluation is that two transformations which result in the same output without origins can result in trees with different annotations. When inserting the OpenMP pragma in the application given in Section 5, the designer has a choice to either define the *parallel* attribute on the loop as $seq(pragma(...), for(...))$ or $seq(pragma(...), x_0)$. The former constructs a new tree for the loop which defines $isContractum = true$, while the latter duplicates the original tree such that $isContractum = false$. This is inconsistent, and one could argue that $isContractum = false$ is the best result for this transformation. However, such a decision would disagree with the currently held correlation between PRS origin cases in Section 1 and the *isContractum* annotation. Currently, nodes with origins constructed by either *context* or *common variables* cases define $isContractum = false$, and nodes with origins constructed by the *auxiliary symbols* case define $isContractum = true$. This is a classic case of two unique transformations which construct the same tree (excluding annotations). We expect to find no issues with allowing some nodes with *auxiliary symbols* origins to define $isContractum = false$, but more research is required before any further claim can be made.

One area of future work is to determine how best to use the information tracked by extended origins. How can we effectively present the data collected in the Halide-inspired language extension to the programmer? This is beyond the scope of this paper, but we can be assured that we have the raw data required.

Extended origins may also be useful in debugging attribute grammars. Algorithmic debugging [7] is a search technique applied to attributed syntax trees, following the structure of the tree and (local) higher order attributes. Extended origins provide additional "edges" that may be traversed during debugging in searching for the errant attribute equation, but more research into this is needed to determine how useful that would be in practice.

We have not yet analyzed how tracking origins affects the amount of memory Silver uses. More trees are kept in memory and not garbage collected due to the origin and other references. In many applications using origins such as debugging and transformation visualization we may run Silver in a "debug" mode to track origins and pay the memory cost, but then turn it off for other applications.

To conclude, in this paper we defined origin tracking in attribute grammars according to core themes shown in their construction in PRS. After showing that origins provide little context, four additional properties and their accessors were defined and added to define extended origins. These properties were shown to provide meaningful connections between nodes and schedules through complex transformations. Future work includes applying other complex transformations and analyzing how they interact with extended origins.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. This work is partially supported by NSF Awards No. 0905581 and 1047961.

References

1. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: Centaur: The system. SIGPLAN Not 24(2), 14–24 (1988)
2. Boyland, J.T.: Remote attribute grammars. J. ACM 52(4), 627–687 (2005)
3. Courcelle, B., Franchi-Zannettacci, P.: Attribute grammars and recursive program schemes I and II. Theoretical Computer Science 17(2), 163–191, 235–257 (1982)
4. van Deursen, A.: Origin tracking in primitive recursive schemes. In: Conf. Proc. Computing Science in the Netherlands. pp. 132–143, available as technical report CS-R9401. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands (1993)
5. van Deursen, A., Klint, P., Tip, F.: Origin tracking. Journal of Symbolic Computation 15, 523–545 (1992)
6. Hedin, G.: Reference attribute grammars. Informatica 24(3), 301–317 (2000)
7. Ikezoe, Y., Sasaki, A., Ohshima, Y., Wakita, K., Sassa, M.: Systematic debugging of attribute grammars. In: Proc. 4th Int. Workshop on Automated Debugging, pp. 235–240 (2000)
8. Kats, L.C.L., Visser, E.: The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In: Proc. of ACM Conf. on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA). ACM (2010)
9. Klint, P., van der Storm, T., Vinju, J.: Rascal: a domain specific language for source code analysis and manipulation. In: Proc. of Source Code Analysis and Manipulation, SCAM 2009 (2009)
10. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968), corrections in 5 (1971)
11. Martins, P., Saraiva, J., Fernandes, J.P., Van Wyk, E.: Generating attribute grammar-based bidirectional transformations from rewrite rules. In: Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM), pp. 63–70. ACM (2014)
12. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proc. of ACM Conf. on Programming Language Design and Implementation (PLDI), pp. 519–530. ACM (2013)
13. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Science of Computer Programming 75(1–2), 39–54 (2010)
14. Vogt, H., Swierstra, S.D., Kuiper, M.F.: Higher-order attribute grammars. In: Proc. of ACM Conf. on Programming Language Design and Implementation (PLDI), pp. 131–145. ACM (1989)
15. Williams, K., Le, M., Kaminski, T., Van Wyk, E.: A compiler extension for parallel matrix programming. In: Proc. of the International Conf. on Parallel Programming (ICPP) (September 2014)