Origin Tracking in Attribute Grammars

Kevin Williams and Eric Van Wyk

University of Minnesota

Stellenbosch, WG2.11, January 20-22, 2015

First, some advertising

- Multiple new faculty positions at Minnesota
- Part of MnDrive - a state-funded effort in robotics, sensors, and advanced manufacturing.
- Several areas including computer science, very broadly focused
- Program generation with even a hint of applicability to these areas would be a great fit.
- Talk to me for details.
- `http://cse.umn.edu/research/mndrive/`
- (Minneapolis is a great city, as you all know from our previous meeting there....)

A refresher and introduction.

Extensible languages frameworks

Allow programmers select the features to be added to a "host" programming language.

- new syntax (notations)
- new semantic analyses
  (error-checking, optimization opportunities)
- new optimizations

Why would anyone want to do that?

Programming language features

General purpose features in, *e.g.*, C or Java

- assignment statements, loops, if-then-else statements
- functions (perhaps higher-order) and procedures
- I/O facilities
- modules
- data: integer, strings, arrays, structs

Domain-specific features

- matrix operations (MATLAB)
- regular expression matching (Perl, Python)
- statistics functions (R)
- computational geometry operations (LN)
- parallel computing (SISAL, NESL, SAC, PQL etc.)

Many similarities, needless differences.

Working with multiple (domain-specific) languages is a headache.

For example

```
#include <stdio.h>

Matrix gradient ;

int main( int argc, char **argv) {
  match argv[1] with
  | /$(@s=[0-9])+.*/
      -> gradient(x,y) = s * (x + y) {
                                  parallelize y;
                                  reorder y, x;
                                 }
  | _ -> ...
}
```

- A program in extended ANSI C
- Compilation: `scaled_grad.xc` $\Longrightarrow$ `scaled_grad.c`

Reliably composable language extensions

- Extension developers work independently.
- Extension users use multiple extensions.
  - not experts in language design
  - combinations of extensions must "just work"

Challenges for composable language extensions

1. syntax — context free grammars, assoc. regexs
   - context-aware scanning [GPCE'07]
   - modular determinism analysis [PLDI'09]
   - Copper
2. semantics — attribute grammars
   - attribute grammars with forwarding, collections and higher-order attributes
   - set union of specification components
     - sets of productions, non-terminals, attributes
     - sets of attribute defining equations, on a production
     - sets of equations contributing values to a single attribute
   - modular well-definedness analysis [SLE'12a]
   - modular termination analysis [SLE'12b, SCP'14]
   - Silver

Origin Tracking in Attribute Grammars

J. Symbolic Computation (1993) **15,** 523–545

Origin Tracking

A. VAN DEURSEN, P. KLINT, AND F. TIP
arie@cwi.nl, paulk@cwi.nl, tip@cwi.nl

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

We are interested in generating interactive programming environments from formal language specifications and use term rewriting to execute these specifications. Functions defined in a specification operate on the abstract syntax tree of programs and the initial term for the rewriting process will consist of an application of some function (e.g., a type checker, evaluator or translator) to the syntax tree of a program. During the term rewriting process, pieces of the program such as identifiers, expressions, or statements, recur in intermediate terms. We want to formalize these recurrences and use them, for example, for associating positional information with messages in error reports, visualizing program execution, and constructing language-specific debuggers. *Origins* are relations between subterms of intermediate terms and subterms of the initial term. *Origin tracking* is a method for incrementally computing origins during rewriting. We give a formal definition of origins, and present a method for implementing origin tracking.

1 Introduction

Origins in Halide-inspired language extension

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

becomes

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

- `#pragma omp` connects to `parallelize` via origins
- reordered for-loops connect to `reorder` via origins and redex edges.

A simple example

Sorts: E, Int

Operators:

$add : E\ E \to E$ $\quad sub : E\ E \to E$

$mul : E\ E \to E$ $\quad div : E\ E \to E$

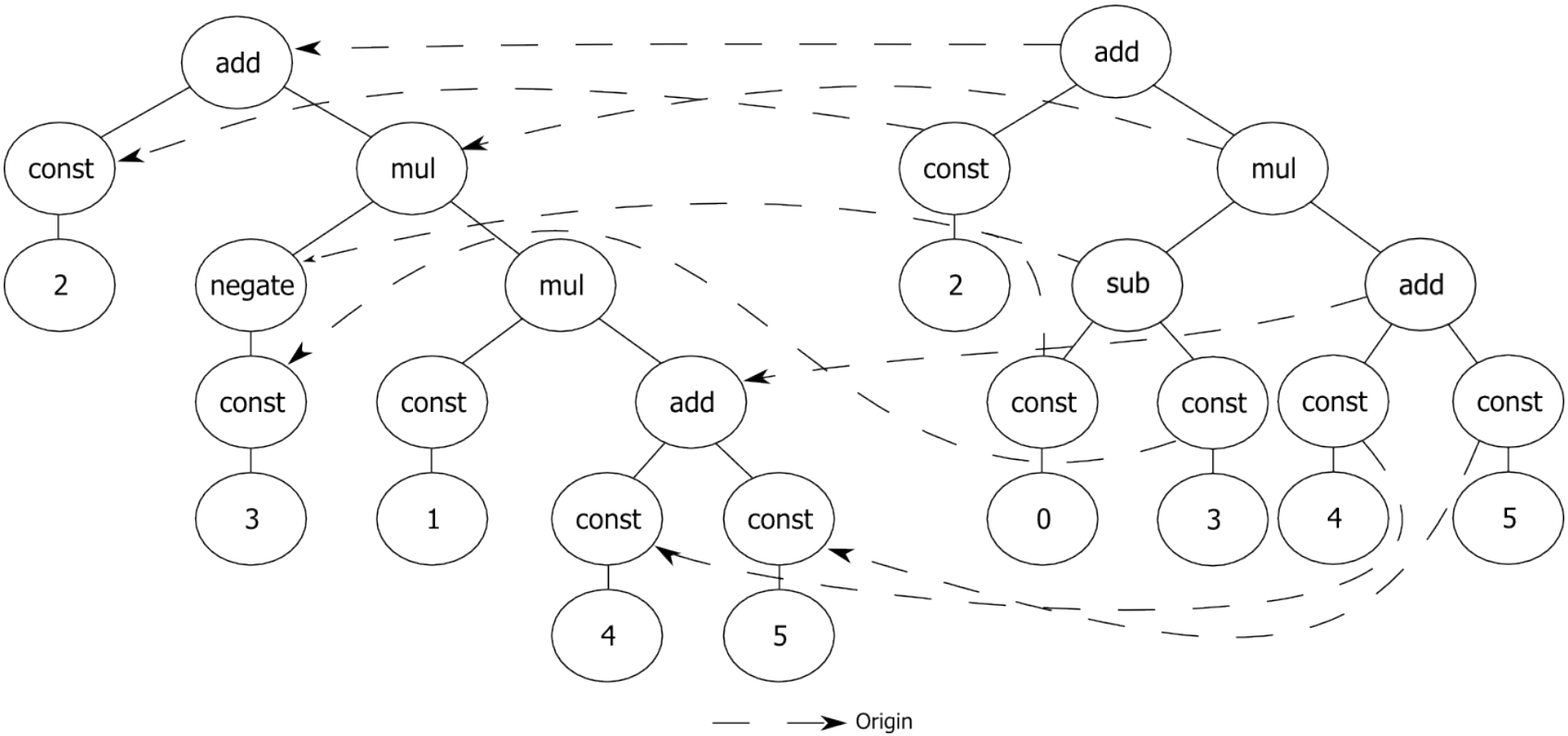
$negate : E \to E$

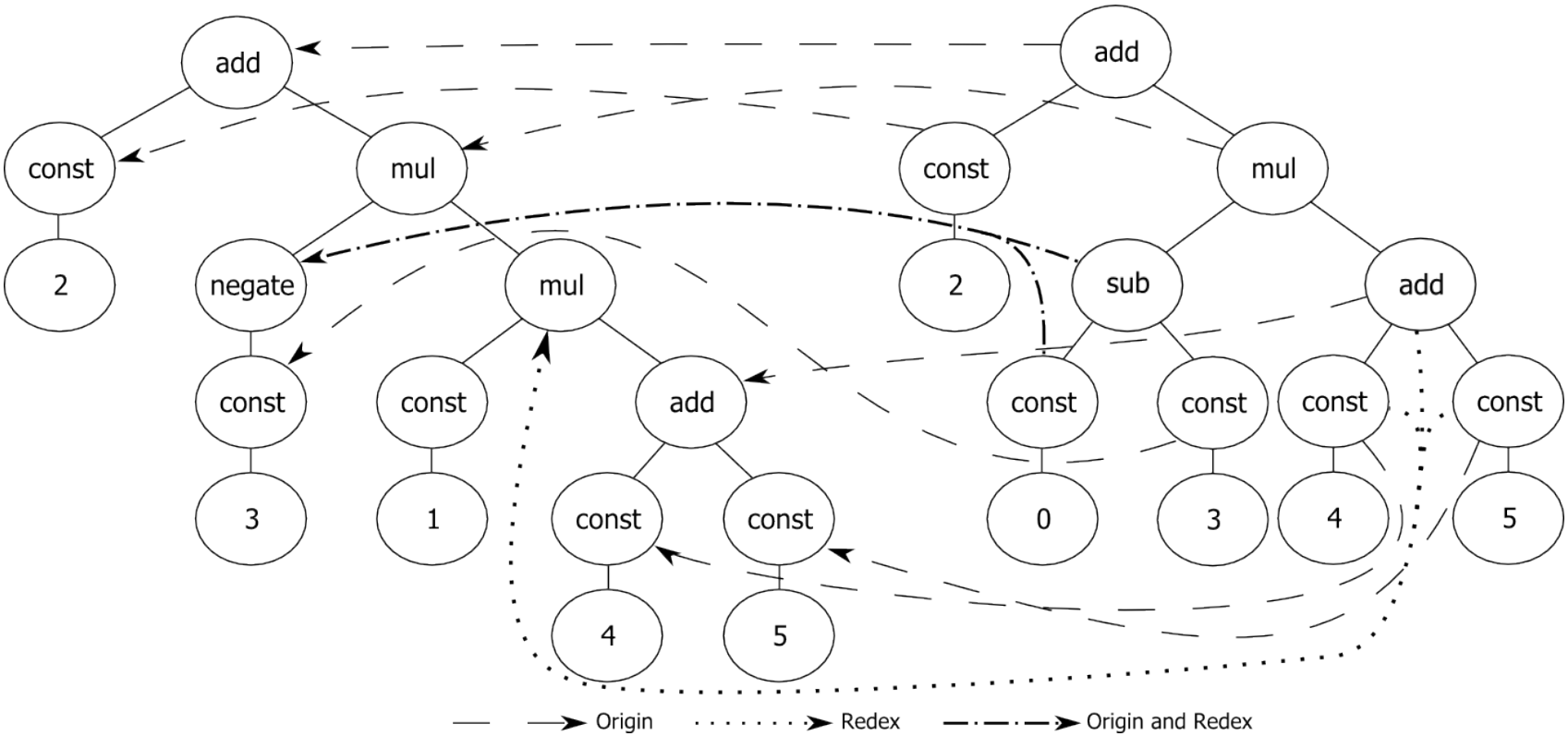
$const : Int \to E$

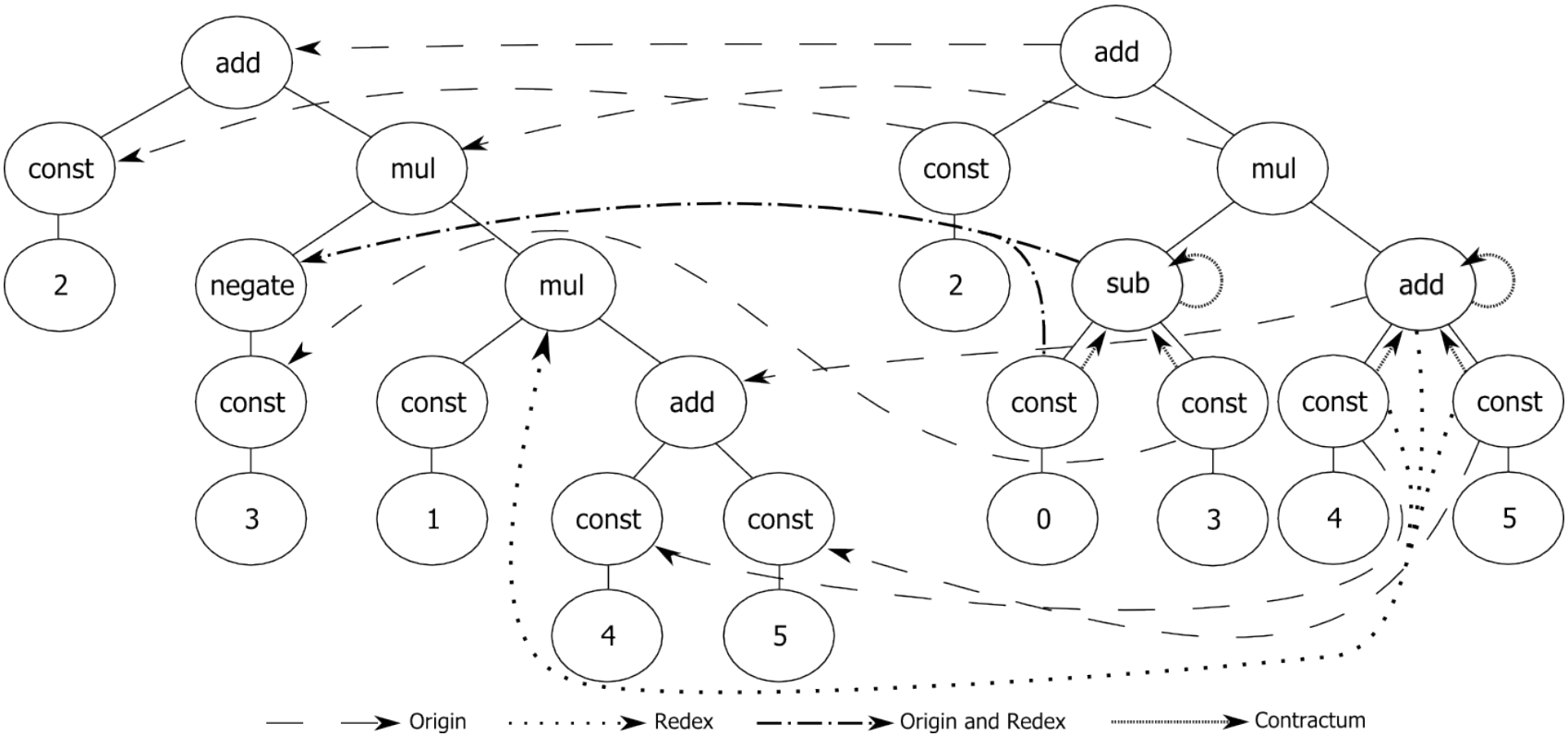
Rewrite rules:

- $negate(X) \to sub(const(0), X)$
- $mul(const(1), X) \to X$

Apply to $2 + (-3 * (1 * (4 + 5)))$


add
const
mul
2
negate
mul
const
const
add
3
1
const
const
4
5
add
const
mul
2
sub
add
const
const
const
const
0
3
4
5
Origin


add
const
2
mul
negate
const
3
mul
const
1
add
const
4
const
5
add
const
2
mul
sub
const
0
const
3
add
const
4
const
5
Origin
Redex
Origin and Redex


add
const
2
mul
negate
const
3
mul
const
1
add
const
4
const
5
add
const
2
mul
sub
const
0
const
3
add
const
4
const
5
Origin
Redex
Origin and Redex
Contractum

Adding (extended) origins to AGs

Challenges

- For transformations, AGs are more verbose.
  - Explicit equations for implicit copies rules.
- Simple origins are straightforward,
  - New trees have origins pointing to the root node of the production
- Extended origins, not so simple.
  - Need to identify the equations that are “interesting”

```
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;  }
```

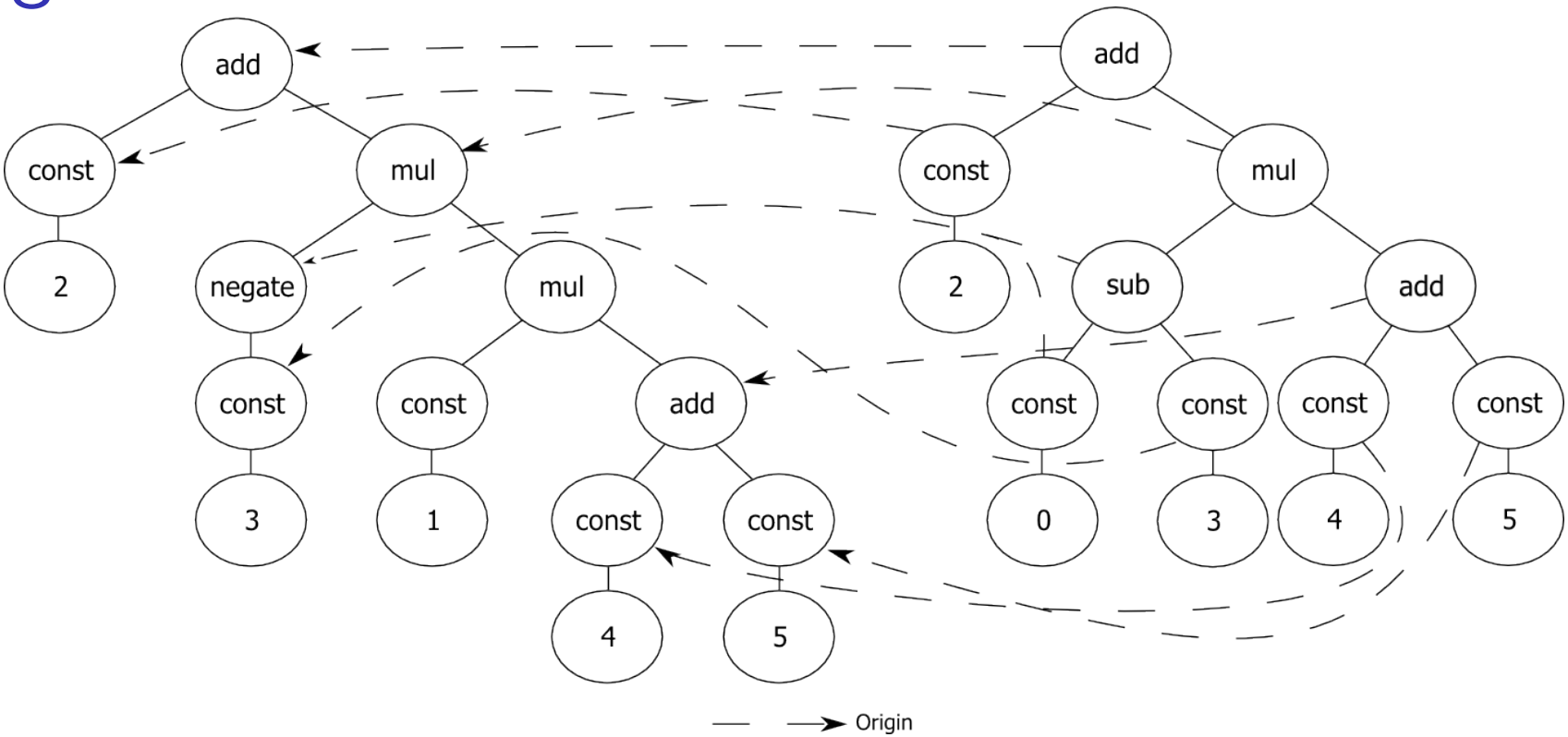
```
abstract production root
r::Root ::= e::Expr
{ local doExpd :: Expr = e.expd;
  local doSimp :: Expr =
    doExpd.simp;  }

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);  }
```

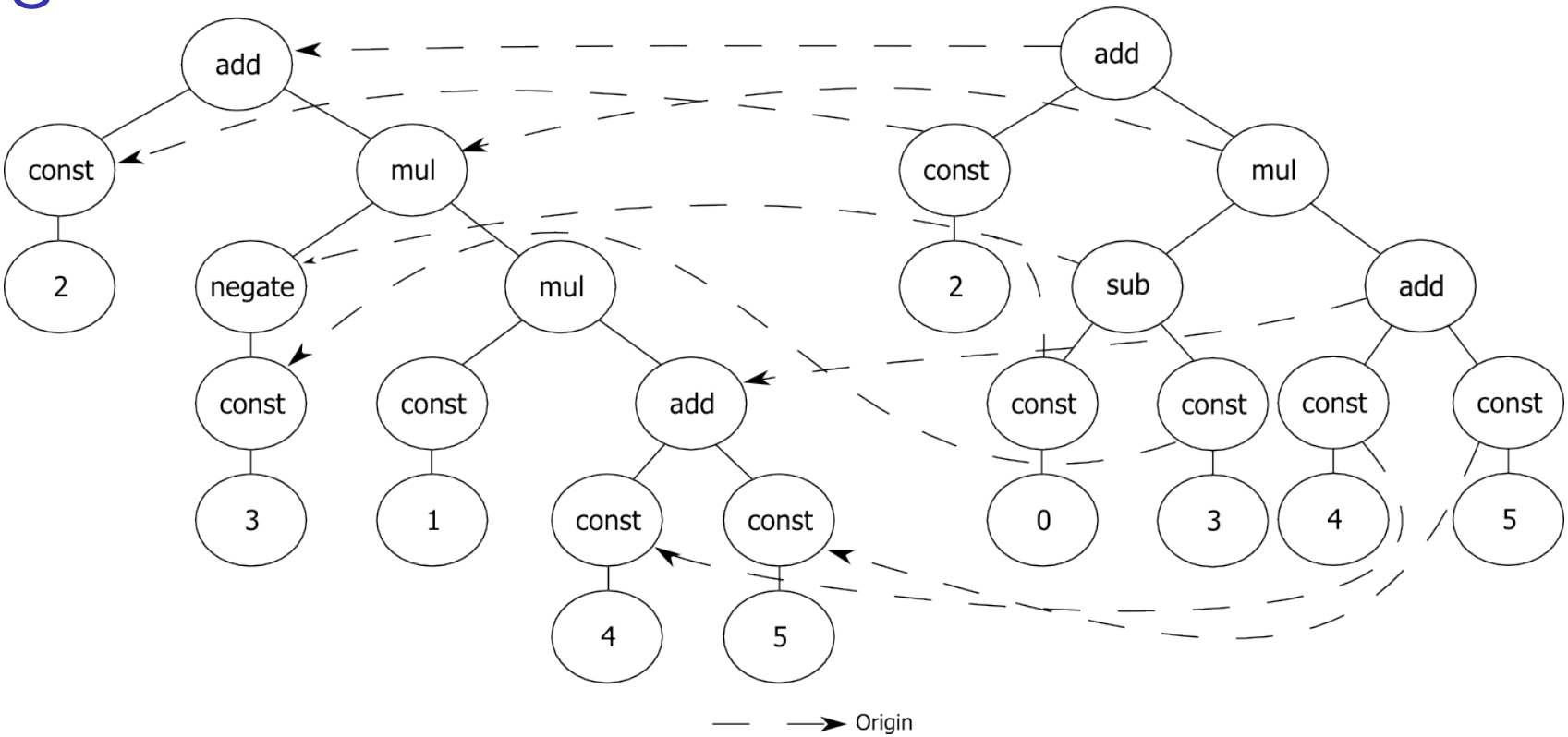
Origins




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

Local higher-order attributes for transformed trees.

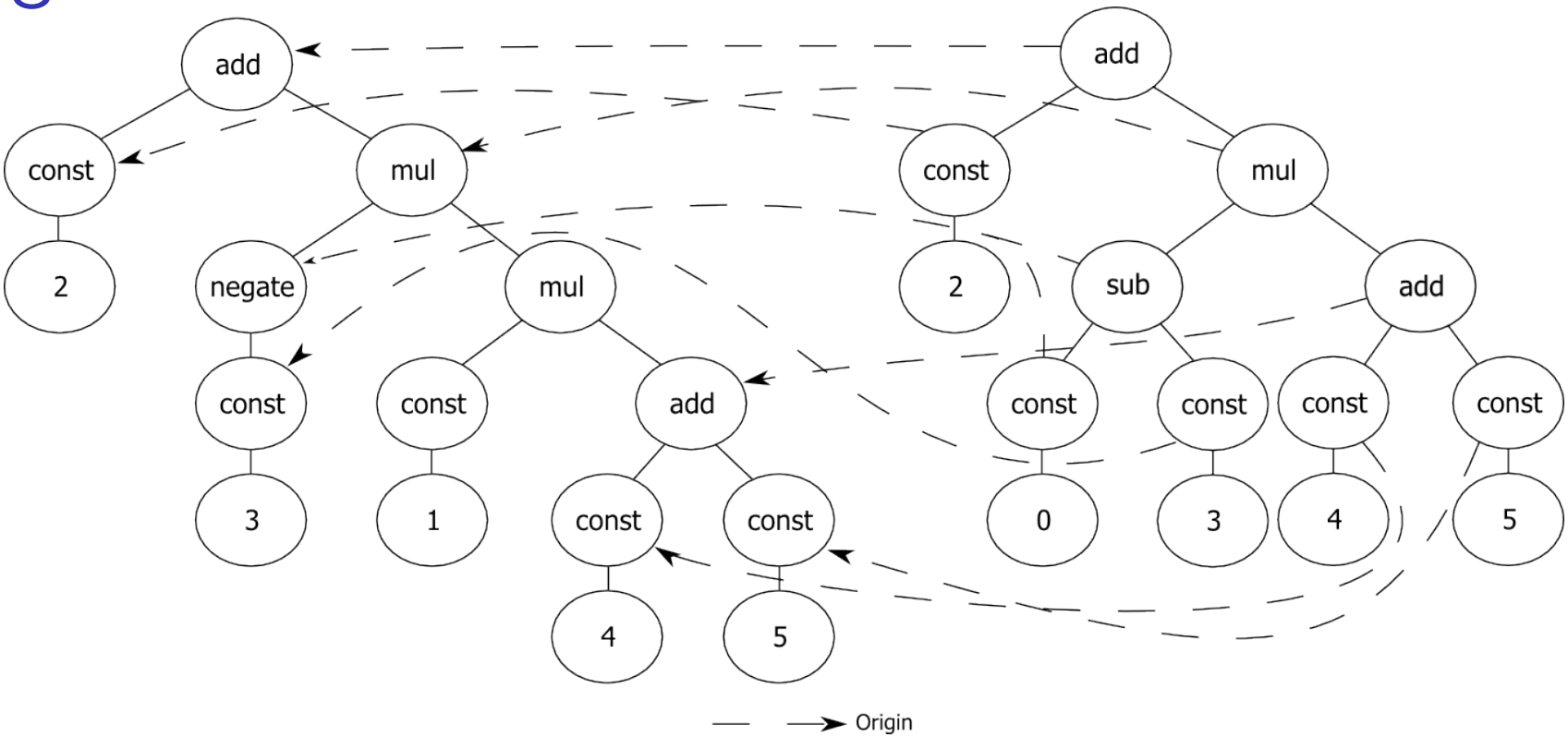
Origins




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

boiler plate equations

Origins




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

$mul(const(1), X) \rightarrow X$

So, how is this accomplished?

- Big-step operational semantics of expression evaluation
- Different types for undecorated trees (terms) and decorated trees

$$
\begin{array}{lcl}
e & ::= & \texttt{if } e \texttt{ then } e \texttt{ else } e \\
& | & \texttt{case } e \texttt{ of} \\
& & \quad q_1(y_1^1, ..., y_{n_{q_1}}^1) \Rightarrow e_1 \\
& & \quad ... \\
& & \quad q_n(y_1^n, ..., y_{n_{q_n}}^n) \Rightarrow e_n \\
& | & f(e, ..., e) \\
& | & var \\
& | & var.attr \\
& | & p(e, ..., e) \\
& | & \texttt{new } var \\
& | & v \\
var & ::= & x_0 \mid x_i, i > 0 \\
& | & \ell_i \mid y
\end{array}
$$

$$
\begin{array}{lcl}
v & ::= & \texttt{true} \\
& | & \texttt{false} \\
& | & n \\
& | & str \\
& | & p(v, ..., v)|o) \\
& | & [n, ..., n] \\
& | & \lambda\ y_1 : T_1, ..., \\
& & \quad y_n : T_n.e \\
T & ::= & PT \\
& | & N \\
& | & N ::= X...X \\
& | & T...T \to T \\
& | & Ref\ N
\end{array}
$$

Without origins:

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

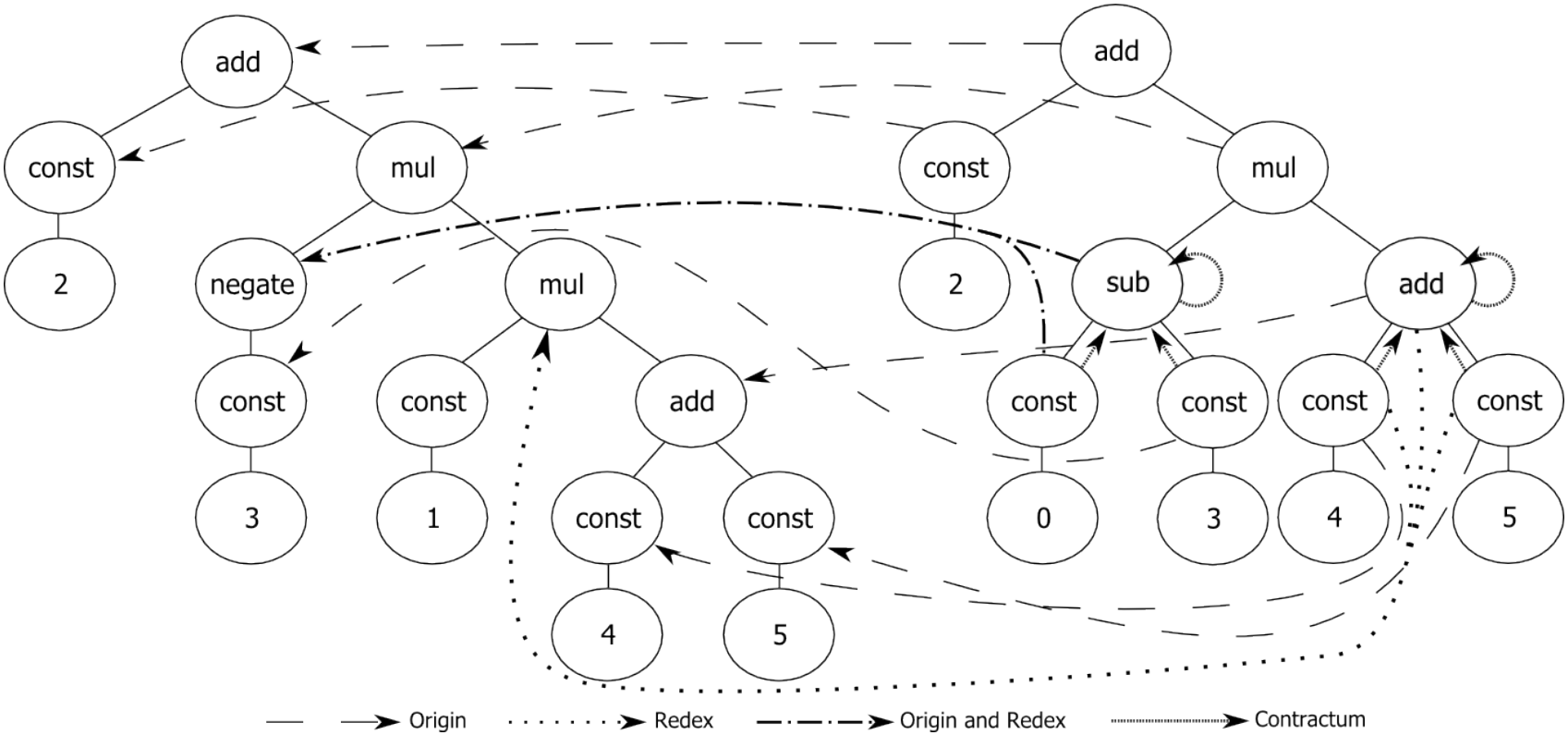
$$\frac{\sigma \vdash var \rightarrow h}{\sigma \vdash \texttt{new } var \rightarrow *h} \quad \text{(E-New)}$$

With origins:

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

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

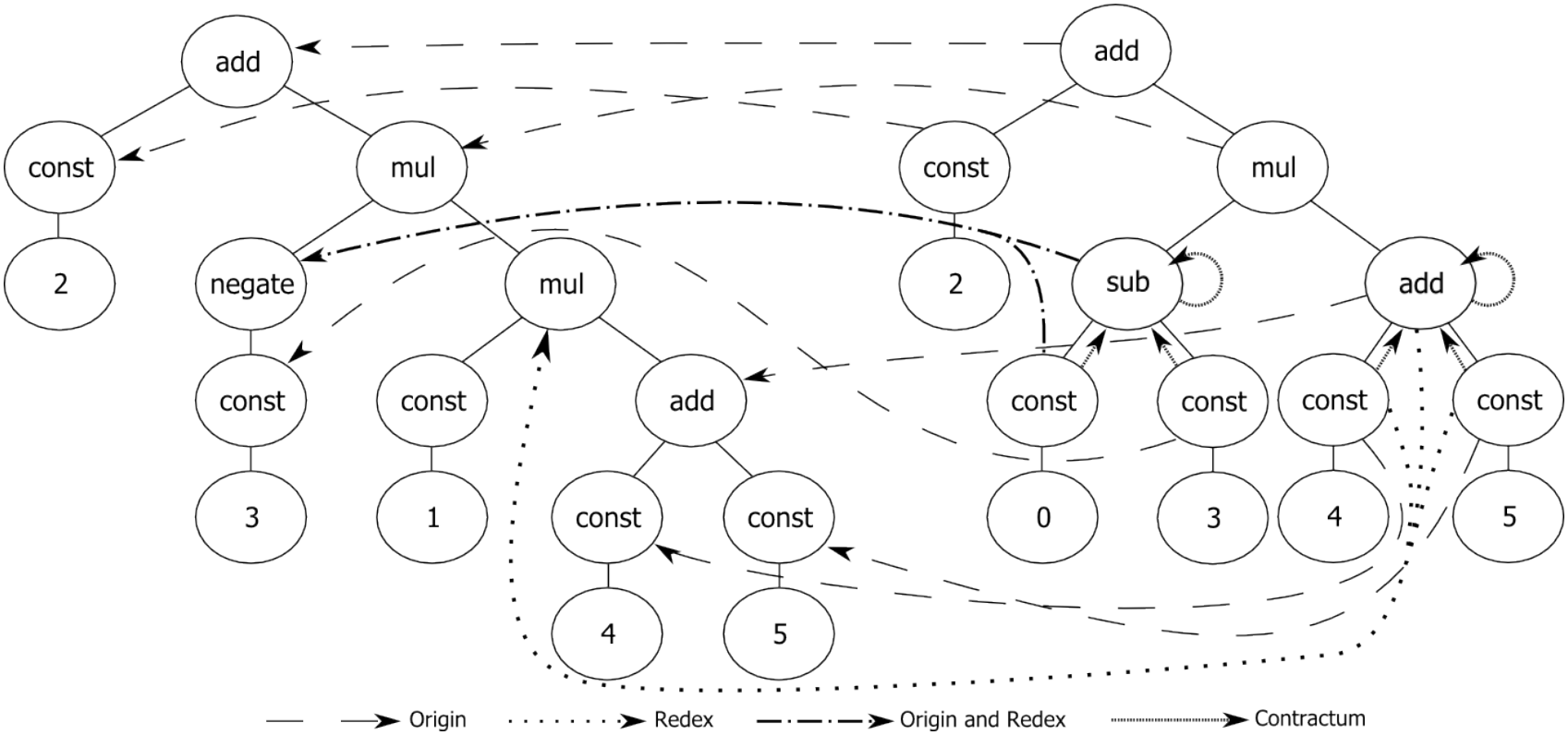
Redex and Contractum




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

$negate(ne) \rightarrow sub(const(0), ne)$

Redex and Contractum




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

$mul(const(1), X) \rightarrow X$

Extended Origins

Questions answered by extended origins:

- Is a tree newly constructed, is it a contractum?
- What is the redex, if one exists?
- What is the contractum root, if one exists?
- Why was the transformation made?

$$
\begin{array}{lcl}
e & ::= & \texttt{if } e \texttt{ then } e \texttt{ else } e \\
& | & \texttt{case } e \texttt{ of} \\
& & \quad q_1(y_1^1, ..., y_{n_{q_1}}^1) \Rightarrow e_1 \\
& & \quad ... \\
& & \quad q_n(y_1^n, ..., y_{n_{q_n}}^n) \Rightarrow e_n \\
& | & f(e, ..., e) \\
& | & var \\
& | & var.attr \\
& | & p(e, ..., e) \\
& | & \texttt{new } var \\
& | & v \\
var & ::= & x_0 \mid x_i, i > 0 \\
& | & \ell_i \mid y
\end{array}
$$

$$
\begin{array}{lcl}
v & ::= & \texttt{true} \\
& | & \texttt{false} \\
& | & n \\
& | & str \\
& | & p(v, ..., v|o), n, r, c, l) \\
& | & [n, ..., n] \\
& | & \lambda\ y_1 : T_1, ..., \\
& & \quad y_n : T_n.e \\
T & ::= & PT \\
& | & N \\
& | & N ::= X...X \\
& | & T...T \to T \\
& | & Ref\ N
\end{array}
$$

Origins interface: $p(v, ..., v|o, n, r, c, l)$

- o - origin
- n - isNew, Boolean
- r - redex or $\perp$
- c - contractum or $\perp$
- l - set of descriptive labels, specific to each attribute equation

Invariants

$$\mathit{getOrigin}(t) = \bot \implies \neg\mathit{getIsNew}(t) \land \mathit{getRedex}(t) = \bot \land \mathit{getContractum}(t) = \bot \land \mathit{getLabels}(t) = \{\}$$

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

$$\mathit{getIsNew}(t) \implies \mathit{getOrigin}(t) \neq \bot \land \mathit{getRedex}(t) \neq \bot \land \mathit{getContractum}(t) \neq \bot$$

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

Invariants

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

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

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

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

Origins Implementation: $p(v, \ldots, v | o, n, r, l)$

- r, the redex, is only defined on nodes where n (isNew flag) is true.
- Thus, we compute the contractum from these values.
- $\sigma, t, a, er \vdash e \rightarrow v$
  - a - attribute being computed
  - er - tree is root of final value

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

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

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

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

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

In the context of $\sigma; t, a, er \vdash ... \rightarrow ...$, the value er flows through if-then-else expressions and function calls.

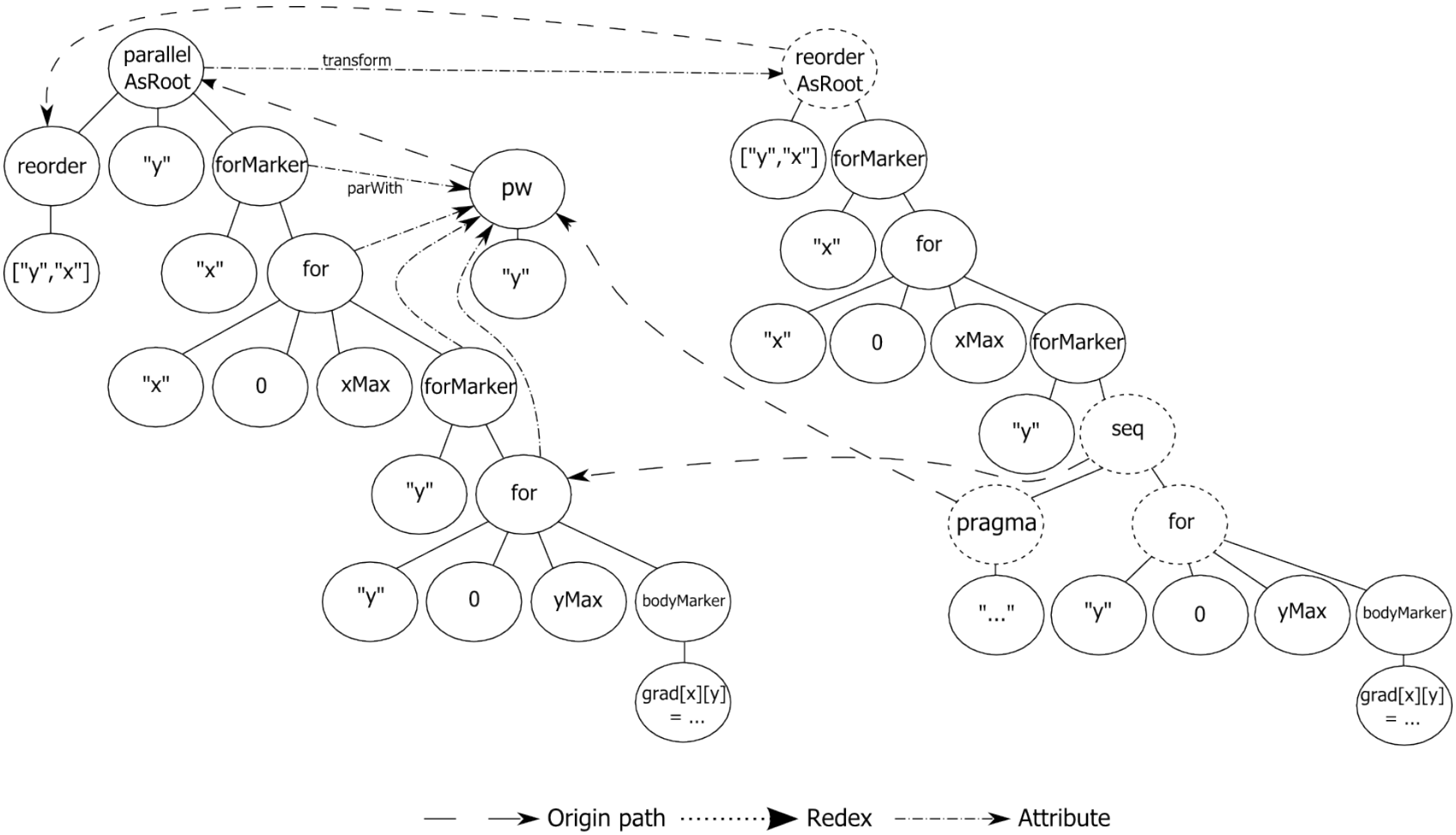
Halide-inspired language extension

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

becomes

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

- `#pragma omp` connects to `parallelize` via origins through auxiliary "LoopIndex-ToParallelize" node
- reordered for-loops connect to `reorder` via origins and redex edges.


parallel
AsRoot
transform
reorder
AsRoot
reorder
"y"
forMarker
parWith
pw
["y","x"]
"x"
for
"y"
"x"
0
xMax
forMarker
"y"
for
"y"
0
yMax
bodyMarker
grad[x][y]
= ...
["y","x"]
forMarker
"x"
for
"x"
0
xMax
forMarker
"y"
seq
pragma
for
"..."
"y"
0
yMax
bodyMarker
grad[x][y]
= ...
Origin path
Redex
Attribute

Future Work

- Full integration into Silver.
- Tools to make use of all of this data.
  - This may be the genuinely hard part.

Questions?

Thanks for your attention.

`http://melt.cs.umn.edu`
`evw@cs.umn.edu`


Eric Van Wyk and August Schwerdfeger.
Context-aware scanning for parsing extensible languages.
In *Intl. Conf. on Generative Programming and Component Engineering, (GPCE)*, pages 63–72. ACM, 2007.


August Schwerdfeger and Eric Van Wyk.
Verifiable composition of deterministic grammars.
In *Proc. of Conf. on Programming Language Design and Implementation (PLDI)*, pages 199–210. ACM, June 2009.


Ted Kaminski and Eric Van Wyk.
Modular well-definedness analysis for attribute grammars.
In *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, volume 7745 of *LNCS*, pages 352–371. Springer-Verlag, September 2012.


Lijesh Krishnan and Eric Van Wyk.
Termination analysis for higher-order attribute grammars.
In *Proceedings of the 5th International Conference on Software Language Engineering (SLE 2012)*, volume 7745 of *LNCS*, pages 44–63. Springer-Verlag, September 2012.


Lijesh Krishnan and Eric Van Wyk.
Monolithic and modular termination analysis for higher-order attribute grammars.
Science of Computer Programming, 96(4):511–526, December, 2014.