Composable Language Extensions for Computational Geometry: a Case Study

Eric Van Wyk
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota 55455

Eric Johnson
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota 55455

***Abstract*— This paper demonstrates how two different sets of powerful domain specific language features can be specified and deployed as composable language extensions. These extensions incorporate analyses and transformations that simplify the process of writing efficient and robust computational geometry programs and can be automatically added to a host language and used simultaneously. This is not possible in domain-specific language and library-based implementations of these features.**

One extension relies on characteristics of geometric algorithms to implement efficient exact-precision integers; the other employs a technique that symbolically perturbs geometric coordinates to safely and automatically handle degeneracies in the input data. These language extensions are implemented in an extensible language framework based on higher-order attribute grammars and forwarding. Attribute evaluation on the new language extension constructs is used to implement the static analysis and code transformations that enable the generation of efficient code.

I. Introduction

Domain specific languages (DSLs) [1] can simplify programming because they have features tailored to the specific problem domain of the task at hand. The features typically include (i) language constructs such as new data types and high-level operations on values of those types, (ii) domain-specific optimizations that ensure that efficient code can be generated, and (iii) domain specific analyses that guide these optimizations and check for their correct use. These constructs allow one to specify the problem solution at the (higher) level of abstraction of the problem instead of encoding the solution at the lower level of abstraction of a general purpose programming language. This allows one to more quickly specify a solution and also helps one avoid errors. Because the solution is specified in terms specific to the domain, domain specific optimizations and program analyses can be employed.

However, DSLs suffer from some practical problems that inhibit their widespread use [1]. First, while rich in domain specific features these languages typically lack many of the features found in modern full-featured languages such as Java (classes, packages, foreign-function interfaces) or ML (higher-order functions and modules). The useful data types, analyses and transformations of a DSL are often trapped in a small general purpose imperative or functional programming language that adds little to the problem solving process. Second, it is not uncommon for a particular problem to have aspects from different domains. These domains are typically supported by different domain specific languages and it is often difficult to make use of multiple DSLs. Even when they can be used together the level of granularity at which constructs from the different languages can be composed is too coarse – each program source file contains constructs from only one language. The issue also arises for single-domain problems. In the case study presented in this paper we see that in computational geometry one often wants to make use of static analyses and transformations from separate DSLs and/or libraries in the same program and even in the same expression. The basic problem is that both general purpose and domain-specific languages often lack the right collection of features for the problem at hand. Language developers decide what features are available in the language - once fixed, languages are not easily adapted to particular problems.

Libraries provide another way to add domain-specific functionality to a language; here classes, methods, functions, and procedures represent abstractions in the domain. Their primary advantage is that one can use different libraries addressing different domains in the same program. But with libraries one cannot add new syntax, semantic analysis or optimizations. The exception is templates in C++; despite limitations they can sometimes be used to generate efficient code.

Both libraries and DSLs provide developers with abstractions 5 for their programs that match the domain of their problems. Elements of the solutions design are often directly represented by the domain-specific abstractions of the DSL or library. Programs with such abstractions often easier to adapt when changes must be made since the design elements were not lost in the original program because they had to be translated to lower-level programming language constructs.

The primary problem with domain specific languages is the abstractions provided by different DSLs cannot be used in the same program. Although abstractions provided by libraries can be, libraries cannot provide the new syntactic constructs and many of the analyses and optimizations that are possible in DSLs. Extensible languages and compilers offer a potential solution to these problems in that they offer a means to let the programmer decide what features to include in a language so that the feature set closely matches the problem at hand. Programmers can chose the appropriate general purpose and domain-specific features. In the approach we are exploring, an extensible "host" language can be extended, under the guidance of the programmer, with the unique combination lan-

guage extensions that support different domains that define the domain specific language features she desires. This extended language raises the level of abstraction to that of the problem at hand. When the host language is a full-featured language like Java or ML, the programmer also has access to the modern general-purpose features she expects. This language acts as a host for the desired domain specific features. The extended language thus has both the general purpose and domain specific features desired by the programmer.

The case study presented here investigates two language extensions applicable to problems in the domain of computational geometry. The first extension implements a technique called symbolic perturbation that removes degeneracies from the input data making it considerably easier to write robust geometric programs. The best implementation of these techniques are implemented in the CGAL computational geometry library [2]. The second extension implements exact precision integers - due to a characteristic of geometric algorithms certain optimizations can be made that are not applicable in general purpose applications. The optimizations here come from a DSL called LN [3]. Both make writing efficient and robust programs considerably easier. But one cannot write a program that combines the perturbation techniques of CGAL with the efficient exact-precision integers of LN. Thus, programmers for geometric applications have a very limited set of combinations of perturbation techniques and numeric representations to choose from. Most commonly a library like CGAL provides a limited set of pre-coded choices.

Our primary goal in implementing computational geometry abstractions as language extensions is to validate that an extensible compiler framework can support the types of analysis and transformational techniques found in DSLs in such a way that different extensions, written by different language features designers, can be combined by the programmer to create a custom language with the desired set of language features.

For extensible compiler frameworks to have wide appeal, they must provide a fully developed host language and a rich set of language features implemented as extensions. The work presented here reports our results of developing two sophisticated language extensions and showing that they can be used together in the same program (in this case, in the same geometric expression). That said, the host language used here is a very simple language as our focus is on validating that language extensions can used together in the same program.

In Section II we provide a brief introduction to the problems encountered in writing geometric programs, some solutions to these problems. Section III describes our extensible compiler framework based on attribute grammars and defines a simple imperative host language. Section IV provides the specifications of the CG language extensions and explains how they can be used together. This provides an example of how different language extensions that provide the new syntax, analyses, and optimizations found in DSLs can be used together; thus overcoming the aforementioned problems with DSL-based and library-based implementations of domain-specific abstractions. Section V concludes and discusses related and future work.

II. Computational geometry

In this paper we demonstrate the utility of extensible languages and domain specific language extensions in the domain of computational geometry (CG). We show how they can make it significantly easier to implement robust geometric programs. This will also show the fine-level of granularity at which different language extensions can interact. Below we present two language extensions; each addresses a fundamental problem in developing robust implementations of geometric algorithms. These are based on existing domain specific languages [3] and libraries [2] [4] that cannot be used together, even though the problems they address both occur in many algorithms. Thus, a programmer can only use one because there is no common framework on which these separately developed solutions can be based. Extensible host languages provide such a framework, making the use of both extensions possible.

There are two fundamental difficulties in developing robust implementations of geometric algorithms that these extensions help to solve. First, the text-book algorithms typically assume that exact-precision real numbers and operations are available. Simply using machine-supported floating point numbers instead can introduce round-off errors that adversely affect the behavior of the algorithm, but using general purpose exact precision numbers can be too slow. Similarly, the correct operation of textbook algorithms may require the application of a technique called symbolic perturbation [4] that masks degeneracies in the data. For example, the randomized incremental trapezoidal decomposition algorithm (see [5, Chapter 6]) depends on the assumption that each trapezoid has at most four adjacent trapezoids, a condition that can be guaranteed by symbolic perturbation. Without symbolic perturbation, each trapezoid may have an arbitrary number of adjacent trapezoids. Masking degeneracies simplifies the algorithm and its data structures as well as its proof of its correctness. We present language extensions to address each of these two problems.

Many CG algorithms are combinatorial algorithms that do not create any new geometric entities but instead perform some query over the input geometric entities. A convex-hull algorithm for example tells which points belong to the convex hull of a given set of input points. It is common practice in CG to base such algorithms on a *geometric primitives*. These are expressions that return a qualitative, not quantitative, result about some relationship between geometric objects. For example, the *in-circle* primitive tests if a given point lies inside, outside, or exactly on, a given circle. The *left-right-side* primitive tests whether a given point is to the left, the right, or directly on a vertical line. Primitives are typically implemented by computing the sign of an expression as either 1, -1, or 0 indicating if the expression is, respectively, greater than, less than, or equal to zero. These values correspond to the three possible results of the primitives described above. It is only in these primitives where geometric entities are compared or examined. Thus the problems of limited-precision and data degeneracies can be addressed in the primitives and the rest of the algorithm remains the same.

The two sets of domain specific features that we implement as language extensions have previous implementations that are incompatible. LN [3] implements efficient exact-precision integers as a DSL that statically analyzes geometric primitives in order to generate efficient C++ implementations. The generated code is then called by the main algorithm. CGAL [2] is a C++ template library that provides hand-coded implementations of standard geometric primitives for several symbolic perturbation schemes. These are implemented as C++ templates that can be instantiated with the programmer's numeric type of choice, including exact-precision types. However, the limitations of C++ template meta-programming prevent the application of the important analyses and optimizations of LN. Even though both LN and CGAL are based on C++ their respective efficient implementations cannot be used together.

Extensible languages offer a solution to this problem. Each symbolic perturbation scheme and exact-precision numerical implementation can be implemented as a modular, composable language extension that encapsulates the transformations and analyses that are used to generate the implementation from a programmer provided primitive expression. Below we describe in more detail an efficient exact-precision integer implementation and a symbolic perturbation scheme. In Section IV, we will illustrate how the transformations sketched below are implemented as language extensions.

A. Degeneracies in input data

As mentioned above, geometric algorithms can be significantly simplified if the input data are free of degeneracies. A common degeneracy occurs when geometric entities have the same x-coordinate. This may cause the left-right-side primitive to indicate that a point lies exactly on a vertical line. When this primitive is implemented by computing the sign of an expression, the $sign$ operator returns 0. Simply stated, when we take the $sign$ of an expression we do not want it to return 0 as this indicates the presence of a degeneracy. Consistently treating equality (0) as the same as greater than (1) or less than (-1) does not solve the problem and can lead to non-termination or incorrect results in some algorithms [6]. The domain experts in computational geometry have devised a technique called *symbolic perturbation* [7][4][6] that is used to perturb the coordinates by symbolically adding an infinitesimally small *perturbation value* to coordinates so that these degenerate cases do not occur or occur extremely rarely, and then can be detected.

In the *randomized linear perturbation* scheme [4] implemented as a language extension here, when a programmer writes, for example, $r = sign(z - x * y)$, we want this to be *transparently changed*, through the language extension defined transformations, into the code in Figure 1. In this scheme, the perturbation value for a coordinate x is $E_x *_s e$ where E_x is a random value specific to x and e is a symbolic infinitesimally small constant value. We subscript symbolic operations with s to indicate that these operations are symbolic and are not computed. When the sign of an expression is computed, the sign of the original coordinates is computed first; if it is 0, the

$\{\ r = sign(z - x * y);$
  if $(r == 0)$ **then** $\{$
    $r = sign(E_z - (x * E_y + y * E_x));$
    if $(r == 0)$ **then** $\{$
      $r = sign(0 - E_x * E_y);$
      if $(r == 0)$ **then**
        halt $(\text{"}perturbation\ error\text{"});\ \}\ \}\ \}$

Fig. 1. The implementation of perturbation of $r = sign(z - x * y)$.

sign of the perturbation values are computed. To achieve this, $r = sign(z - x * y)$ initially transforms to

$$r = sign_s(\ (z +_s E_z *_s e) -_s (x +_s E_x *_s e) *_s (y +_s E_y *_s e)\ ) \quad (1)$$

Next this expression is transformed so that the $sign$ computations in Figure 1 can be made. We convert the coordinate-valued expression (the expression under the $sign_s$ operator) to a *polynomial over* e in which the coefficients are computable (non-symbolic) expressions by essentially applying the distributive laws $(a + b) * c \Rightarrow (a * c) + (b * c)$ and $a*(b+c) \Rightarrow (a*b)+(a*c)$ as rewrite rules and then collecting like-power e coefficients. Expression (1) thus becomes

$$\begin{aligned} r = sign_s(\quad & (z - x * y) *_s e^0 \\ +_s\ & (E_z - (x * E_y + y * E_x)) *_s e^1 \\ +_s\ & (0 - E_x * E_y) *_s e^2 \quad) \end{aligned} \quad (2)$$

Finally, because e is infinitesimally small, we can convert $sign_s$ to an expression that implements the symbolic $sign_s$ operation by computing the non-symbolic sign of the coefficients of the polynomial in increasing order of the associated powers of e. This final expression is shown in Figure 1. The original expression is rewritten without symbolic operations so that the evaluation is possible and the perturbation values are not computed unless they are needed. This optimization is crucial for good performance of symbolic perturbation. Note that this perturbation scheme does not guarantee that all degeneracies are removed and thus halts with an error message in these rare cases. Other perturbation schemes, like Simulation of Simplicity [7], can make this guarantee and can also be implemented as language extensions.

B. Efficient exact-precision integers

It is a common practice in CG to use exact-precision integers to represent the original floating point representations of the input data. Representations of the exact-precision integers specified here include the original double-word floating point representation and the array-based exact-precision representation. This is possible since we assume a fixed lower bound on the precision of the floating point values that allow an accurate conversion from doubles to exact integers. Exact precision types are used in the geometric primitives where the bit length of precise integers needed to store intermediate results will often exceed that supported by the hardware. Because exact-precision types are used only in expressions and not stored in variables whose value may be changed inside

```
{ double fp;  exact ep;    r = 0;
  fp = z_d − x_d ∗ y_d;
  if ( fp < ⟨maxErr⟩ ∧ fp > −⟨maxErr⟩ ) then {
    // compute exact precision version
    ep = ... z_e ... y_e ... x_e ... ;
    if (ep < 0) then r = −1;
    if (ep > 0) then r = 1; }
  else {
    if (fp < 0) then r = −1;
    if (fp > 0) then r = 1;   }   }
```

Fig. 2. The implementation of exact-precision integer $r = sign(z - x * y)$.

loops or branching statements, a number of static analyses are possible. The number of bits that will be required to store the exact-precision value can be statically determined as can an upper bound on the maximum error that would arise if the primitive expression was implemented using hardware supported floating point numbers.

In our language extension implementation of the LN [3] analyses and transformations the example $r = sign(z - x * y)$ is transformed into the code in Figure 2. In LN and our extension, the expression is computed using the original floating point representations. This shown in line 2 of the figure where the floating point values are represented variables subscripted by d. LN and the extension compute, at compile time, the maximum error value ($\langle$*maxErr*$\rangle$) of the floating point expression. If the magnitude of the resulting value is larger that the error value, then the sign of this value can be returned as the sign of the expression. If it is not, the expression is computed using exact precision integers. Since the maximum bit length required for exact-precision integer evaluation is known at compile time much of the bookkeeping overhead of calling subroutines and the use of loops to handle arbitrary bit lengths is not needed. All the loops can be unrolled and the subroutines in-lined to generate a fast implementation of the exact precision expression. In our implementation, in Section IV, exact precision integers are introduced as a new type and transformations over expressions of that type.

C. Using both exact integers and symbolic perturbation:

An advantage of our approach to extensible languages is that these extensions and their associated transformations can be used together so that one can write geometric primitives whose intermediate results are exact-precision integers that will by symbolically perturbed in the case of any degeneracies, thus addressing both fundamental problems mentioned above. The final implementation of such a primitive will look like the expression in Figure 1 in which each *sign* expression is replaced by an expression similar to the one in Figure 2. This demonstrates the fine-grained level of interaction of language extensions that allows programmers to take advantage of domain specific features specified by different feature designers.

III. Extensible compiler framework

Attribute grammars [8] provide the foundation of our extensible compiler framework. Language constructs are specified by productions and their explicit semantics and translations can be defined by attribute definitions. Two extensions to AGs are used in defining extensible languages: *higher-order attributes* [9] that allow abstract syntax trees (ASTs) to be attribute values and *forwarding* [10], a technique for providing default values for attributes that is similar to macro expansion.

In our framework for building extensible compilers, the host language is defined by an AG. Language extensions are specified as AG fragments that contain productions defining new language constructs and attribute definitions for these productions and possibly those in the host language. The activity of creating an extended language specification entails combining all the productions and attribute definitions in the host language and language extension specifications. This combined specification defines the extended language. This task is performed by the framework tools and maintains the the distinction between the feature designer, who implements a language construct, and the programmer, who builds an extended language by selecting a host language and a set of appropriate language extensions.

A. The Simple host language

Figures 3, 4, and 5 show some of the AG for the abstract syntax of a very simple imperative host language called Simple. This AG is written in (an abbreviated version of the syntax of) Silver, an AG specification language. The concrete syntax of Simple and the extensions described below is not shown; the current implementation relies on traditional scanner and parser generators like lex and yacc.

Figure 3 first defines 5 nonterminals for the root of the AST (`Root`), statements (`Stm`), declarations (`Dcl`), expressions (`Exp`), and type expressions (`Typ`). Terminal symbols and their defining regular expressions for identifiers and integer literals are also shown. Next, a synthesized attribute `pp` is defined to be of type string (`String`) and to decorate (`@`) all the nonterminals. Values for these attributes are defined by attribute definition equations that are associated with each production in the abstract syntax grammar; attribute values are computed lazily, that is, on demand. An inherited environment attribute `env` is a list of pairs of strings and `Typ` trees that is used by identifier reference productions to look up the type of identifiers. The attribute `defs` is used to collect declarations from `Dcl`s. Their definitions are as expected and not shown. A collection of `host` attributes (`hostRoot`, `hostStm`, `hostDcl`, `hostExp`, and `hostTyp`) are defined (though not all shown) to be of the indicated type and are used in translating programs written in an extended language to programs written only in the host language. These attributes are defined on the the type-appropriate productions using the same pattern that is given for productions `assign` and `block`. Some of the remaining statement and declaration productions are shown but their attribute definitions are elided as they can be inferred from the other examples. Simple

expressions and types are discussed in the following sections on forwarding (Section III-B) and production-valued attributes (Section III-C).

```
grammar simple ;
nonterm Root, Stm, Dcl, Exp, Typ ;
term Id /[a-zA-Z] ([0-9] | [a-zA-Z] )* / ;
term IntLit / [0-9]+ / ;

syn attr pp :: String @ Root, Stm,
                        Dcl, Exp, Typ;
syn attr defs :: [(String,Typ)] @ Dcl ;
inh attr env  :: [(String,Typ)] @ Stm,Exp;

syn attr hostStm :: Stm @ Stm ;
syn attr hostExp :: Exp @ Exp ;
syn attr hostTyp :: Typ @ Typ ;
...
prod root    r::Root ::= s::Stm
prod block   b::Stm ::= d::Dcl s::Stm
{ b.pp = "{" ++ d.pp ++ s.pp ++ "}" ;
  b.hostSmt = block(d.hostDcl, s.hostStm); }
prod assign  a::Stm ::= l::Exp r::Exp
{ a.pp = l.pp ++ " = " ++ r.pp ++ ";" ;
  a.hostStm = assign(l.hostExp,r.hostExp); }

prod ift    s::Stm ::= c::Exp t::Stmt
prod ifte   s::Stm ::= c::Exp t::Stmt e:Stmt
prod while  w::Stm ::= c::Exp b::Stmt
prod halt   h::Stm ::= e::Exp

prod dcl d::Dcl ::= v::Id  t::Type
{ d.defs = [ (v.lex, t) ] ; }

prod dclSeq dd::Dcl ::= d1::Dcl d2::Dcl {...}
prod stmSeq ss::Stm ::= s1::Stm s2::Stm {...}
```

Fig. 3. Partial AG specification of simple imperative host language.

B. Forwarding in attribute grammars

Forwarding [10] is an extension to AGs that allows languages to be specified in a highly modular manner. Feature designers will not know all of the attributes that will occur in the final specification of the extended language created by the programmer since programmer chosen extensions may introduce new attributes on host language productions. But, since language extensions need to work closely together new constructs introduced as productions in one extension must provide definitions for attributes introduced in a different extension. Thus, a construct specified as an AG production must be able to *implicitly* specify its value for these attributes. (It *explicitly* specifies the semantics that are of particular concern to it using traditional attribute definitions.) Forwarding allows the feature designer to implicitly define the semantics for a new language construct by specifying a means to construct a *semantically equivalent* construct. If the new construct is queried for an attribute that it does not explicitly define it "forwards" that query to the semantically equivalent construct it specifies. In AG terms, a production defines a distinguished attributed AST, indicated by the `forwards to` syntax, that provides default values for synthesized attributes that are not explicitly defined.

Some examples in Figure 4 will help to clarify. This figure specifies Simple expressions and defines, in the order given, conjunction, negation, disjunction, subtraction, multiplication, identifier reference and integer literal expressions. The production `or` makes use of de Morgan's laws to build the semantically equivalent expression composed of uses of `not` and `and` productions, that is $a \vee b$ forwards to $\neg(\neg a \wedge \neg b)$. When an AST node created by `or` is queried for its `pp` attribute, which it *explicitly* defines, the defined value is returned. If it is queried for the value its `type` attribute, that query is forwarded to the `forwards to` construct and its value is returned.

Because the forwards-to tree will require inherited attributes, these are automatically copied from the forwarding-node (in this example the one created by `or`) unless they are explicitly defined. Note that in some cases the forwards-to node may also forward the query; we will eventually find a value for the attribute since language extensions forward (directly or indirectly) to constructs in the host language. Consider another attribute that is not shown in these specifications; `ctrans`, an attribute which specifies a constructs straightforward translation to C++. Defining this attribute on `and` and `not` but not on `or` results in the C++ translation of `or` being just the C++ translation of the equivalent and/not construct it forwards to. Forwarding is similar to macro expansion in that both reuse the semantics of existing language constructs, but unlike macros, forwarding productions also define semantics, as attributes, that can generate proper error messages, something traditional macro systems cannot do.

```
syn attr type :: Type @ Exp ;

prod and   e::Exp ::= l::Exp r::Exp {...}
prod not   e::Exp ::= n::Exp {...}
prod or    e::Exp ::= l::Exp r::Exp
{ e.pp = l.pp ++ "||" ++ r.pp ;
  forwards to not(and(not(l),not(r))) ; }

prod sub e::Exp ::= l::Exp r::Exp
{ e.pp = ...;  e.type = l.type ;
  forwards to l.type.subProd (l,r); }

prod mul e::Exp ::= l::Exp r::Exp
{ e.pp = ...;  e.type = l.type ;
  forwards to l.type.mulProd (l,r); }

prod idRef e::Expr ::= id::Id
{ e.pp = ...;  e.type = lookup(id.lex, e.env);
  forwards to e.type.idRefProd (id) ; }

prod intLit e::Expr ::= i::IntLit {...}
```

Fig. 4. AG specification of Simple expressions.

C. Production-valued attributes

The productions `sub`, `mul`, and `idRef` in Figure 4 use forwarding and production-valued attributes. Production-valued

attributes [10] contain productions build new trees. In the case of `sub`, the expression `l.type.subProd` evaluates to a production that defines subtraction for the type of the expression `l`. For example, if the type of `l` is integer, then its attribute `type` will be the tree `intT()` defined by the production `intT` in Figure 5. Since `intT` defines its `subProd` attribute to be `subInt`, (the subtraction production specific to the integer type), then `subInt` is used to build the tree to which this instance of `sub` will forward to. In another instance where the type of `l` is a different type, then that type will define the type-specific production to use. This is how *operator overloading* is implemented in Simple. We will use this in the language extensions when we introduce new types for exact-precision integers and symbolic perturbation to overload subtraction and other operations. We also overload the `idRef` production so that identifier references forward to type-specific productions as well. This will be used in the perturbation extension to implement the transformation of x to $x +_s E_x *_s e$ for the example $r = sign(z - x * y)$ from Section II-A.

Note that in this simple host language there is no type coercion. Also, there are no type names, so the type information on `Typ` trees does not depend on it environment. Thus, there is no harm in copying them to new locations in the AST as the value of `type` attributes.

```
syn attr subProd :: Prod (Exp ::= Exp Exp) ;
syn attr mulProd :: Prod (Exp ::= Exp Exp) ;
syn attr idRefProd :: Prod (Exp ::= Id) ;
syn attr zeroTree :: Exp ;

subProd, mulProd, idRefProd, zeroTree @ Type;

prod intT    t::Type ::=
{ t.subProd = subInt ;  t.mulProd = mulInt ;
  t.idRefProd = idRefInt ;
  t.zeroTree = intLit(term(IntLit,"0")) ;  }
prod floatT  t::Type ::= { ... }
prod arrayT  t::Type ::= ct::Type
{ t.compType = ct ; }
prod subInt  e::Exp ::= l::Exp r::Exp
{ e.pp = ... ;    e.type = intT();  }
prod mulInt  e::Exp ::= l::Exp r::Exp
{ e.pp = ... ;    e.type = intT();  }
```

Fig. 5. AG specification of Simple types and type-specific productions.

We have implemented this simple host language using Silver, an attribute grammar specification language we have built for specifying extensible languages. The Silver compiler translates Silver specifications to Haskell which is used as the implementation language. The Haskell implementation performs the attribute evaluation. The grammar specifications above comprise the `simple` grammar module. It will be imported by language extensions that extend Simple in the following sections. Silver is freely available on the internet at `www.melt.cs.umn.edu`.

IV. Computational geometry language extensions

To add exact-precision integers as a language extension we specify a new type `exactT` and new operations (productions) on values of this type. To add symbolic perturbation we specify a new type constructor (production) `rlpT` that perturbs a specified (numeric) type and operations over these types. In doing so we use the host-language operator overloading feature and higher-order attributes to build trees representing the transformed expressions. The host language, its abstract syntax, and its operator overloading facility provide the common framework on which both language extensions can be based. This allows the features specified in both language extensions to be used together. The programmer can then define variables, such as $x, y,$ and z in the example, to have the type `rlpT(exactT())`. These are also easy for the programmer to use since they simply need to change the type of their variables to trigger the necessary transformations.

To understand the AG specification of the perturbed and symbolic types and the exact integers and their associated transformations we will trace the compilation process of our example expression $sign(z - x * y)$. This will parallel the explanation and expressions given in Section II. Because the evaluation of attributes is demand-driven, the transformation from $r = sign(z - x * y)$; to its implementation in the host language can be more easily understood by starting at the root node of the statement's abstract syntax tree. We will see how the perturbed-type construct forwards to a symbolic-type construct that forwards to an expression over exact integer types. These exact integer type expressions then forward to their implementations which are written entirely using constructs from the host language. This process is described in the remainder of this section.

```
grammar cg ;   import  simple ;

prod sign s::Stm ::= lhs::Exp e::Exp
{ s.pp = lhs.pp ++ "= sign(" ++ e.pp ++ ");" ;
  forwards to e.type.signProd(lhs,e);   }

syn attr signProd::Prod(Exp::=Exp Exp) @ Type ;
aspect prod intT t::Type::=
{ t.signProd=signInt; }
prod signInt  s::Stm ::= lhs::Exp  e::Exp
{ forwards to ...seq of if-then stmts ... }
```

Fig. 6. The generic `sign` production and `signProd` attribute.

An extension for computational geometry, on which the extensions for randomized linear perturbation and exact-precision integers will be built, is shown in Figure 6. This extension introduces the `sign` statement that assigns the sign of its second expression to its first, for example $r = sign(z - x * y)$;. Based on the type of the second expression (which in this case is based on the types of the variables x, y, and z), the `sign` production forwards to a type specific sign production. If the expression has the type `exactT()` then `sign` will forward to `sign_e` - the exact precision integer specific sign production shown in Figure 9. If the type is `rlpT(intT())`

or `rlpT(exactT())` then `sign` forwards to the perturbed sign production `sign_p` shown in Figure 7. The `aspect` production adds the definition of the `signProd` attribute to the others defined on the production `intT`.

A. Symbolic perturbation as a language extension

To indicate that the randomized linear perturbation is to be applied to the expression used in `sign` we require that the variables in the expression have perturbed types. Thus, in our example $r = sign(z - x * y)$ the variables x, y, and z are declared to have type `rlpT(X)` where `X` could be the integer or exact-precision integer type. Thus the `type` attribute on the nodes in the AST for the $sign(z - x * y)$ will be `rlpT(X)`, for some `X`. This allows the host `sign` production to forward to the perturbed-type specific sign production `sign_p`.

The productions in Figure 7 use the synthesized attribute `symTree` to compute the representation of the expression using symbolic operators and expanding the perturbed variables to the symbolic representations. The `sign_p` production forwards to the symbolic sign production `sign_s` with the symbolic expression. For the example $r = sign(z-x*y)$ this corresponds to (1) in Section II-A. The only difference is that in the `symTree` expression variables, e.g. x, are not represented as symbolic expressions, e.g. $x +_s E_x *_s e$, but instead represented using the convenience production `idRefTrans_s` as shown in the production `idRef_p`. The tree `e.symTree` in `sign_p` is constructed using the symbolic productions `add_s`, `sub_s`, `mul_s`, and `idRefTrans_s` shown in Figure 8.

```
grammar RLperturb ;  import  simple ;

prod rlpT  t::Type ::= pt::Type
{ t.subProd = sub_p;  t.signProd = sign_p;
  t.mulProd = mul_p;  t.compType = pt;    }

syn attr symTree :: Expr  @ Expr ;
prod sign_p s::Stm ::= lhs::Exp  e::Exp
{ s.pp = lhs.pp ++ "= sign(" ++ e.pp ++ ");" ;
  forwards to sign_s (lhs, e.symTree) ;   }

prod sub_p  e::Exp ::= l::Exp r::Exp
{ e.pp = ...;    e.type = l.type ;
  e.symTree = sub_s(l.symTree,r.symTree) ;  }

prod mul_p e::Exp ::= l::Exp r::Exp
{ e.pp = ...;    e.type = l.type ;
  e.symTree = mul_s(l.symTree,r.symTree) ;  }

prod idRef_p  e::Exp ::= i::Id
{ e.pp = i.lex;  e.type =lookup(i.lex, e.env) ;
  e.symTree = idRefTrans_s(i,e.type.compType);}
```

Fig. 7. The randomized linear perturbation type and operator productions.

From the symbolic expression, we could apply the distributive laws $(a + b) * c \Rightarrow (a * c) + (b * c)$ and $a * (b + c) \Rightarrow (a*b)+(a*c)$ to collect the like-power e coefficients. Instead of applying these rules directly, each node in the symbolic tree is decorated by two attributes that keep track of the coefficients of the polynomial over e. The first is `coefs` and is a list of `Exp` trees. The second, `ncoefs` is highest power of e with a non-zero coefficient. Both attributes are defined in Figure 8. Starting at 0, the i^{th} element of `coefs` is the coefficient of e^i. Thus, for our example $r = sign(z - x * y)$, on `sign_s` we want `e.coefs` to be the list

$$[z - x * y, E_z - (x * E_y + y * E_x), (0 - E_x * E_y)] \quad (3)$$

and `e.ncoeffs` to be 2. This list then represents the transformed symbolic expression shown in equation (2) in Section II-A. From this list, it is straightforward for the production `sign_s` to create the construct that it forwards to - for the example this is the construct shown in Figure 1. In its forwards-to clause, the block and initial assignment are created. Note that the zero values used in the conditions of the if-thens must have the same type as the type being perturbed by the `rlpT` production. If that type is `intT()`, then the integer zero constant must be provided to the `mkRest` function. If that type is `exactT`, then the exact-precision zero must be provided. This tree is stored in the `zeroTree` attribute on the component type attribute (`compType`) of the expression `e`'s `type` attribute. The straight forward implementation of `mkRest` (not shown) creates the nested if-then statement in Figure 1. Note that we also overload the equality operator `==` in the same manner as is done for `sign`, `sub`, and others.

```
syn attr coefs :: [Exp] @ Exp ;
syn attr ncoefs :: Integer @ Exp ;

prod symT  t::Type ::= pt::Type
{ t.compType = pt;  }

prod sign_s s::Stm ::= lhs::Exp  e::Exp
{ s.pp = ...;
  forwards to block( emptyDcl(),stmCons(
       assign(lhs, head(e.coefs)), rest));
  local rest :: Stm ;
  rest = mkRest( tail(e.coefs), e.ncoefs,
           lhs, e.type.compType.zeroTree ); }

prod sub_s e::Exp ::= l::Exp r::Exp
{ e.pp = ...;   e.type = l.type ;
  e.ncoefs = max (l.ncoefs, r.ncoefs) ;
  e.coefs = subF(l.type.compType.subProd,
                 l.coefs,r.coefs );    }

prod mul_s e::Exp ::= l::Exp r::Exp
{ e.pp = ...;   e.type = l.type ;
  e.ncoefs = l.ncoefs + r.ncoefs ;
  e.coefs = mulF ( l.type.compType.addProd,
   l.type.compType.mulProd, l.coefs, r.coefs);}

prod idRefTrans_s
e::Exp ::= i::Id  ct::TypeExpr
{ e.pp = ...;   e.type = symT (ct);
  e.ncoefs = 1;
  e.coefs = [ (ct.idRefProd)(i),
   (ct.idRefProd)(term(Id,"E_"++i.lex)) ];}
```

Fig. 8. The symbolic type and operator productions.

The definition of `coefs` and `ncoefs` by the productions `sub_s`, `mul_s`, and `idRefTrans_s` is straight for-

ward and such things are common in higher-order AGs. In `idRefTrans_s` where we map, for example, x to $x +_{ct} E_x *_{ct} e$ (where ct is the component type of the perturbation type), we set `ncoefs` to 1 and define the list `coefs` using the identifier reference production `idRefProd` and the terminal creation function `term`. By passing the component type to `idRefTrans_s` as the parameter `ct` it is used to define the `type` attribute and used further in the productions `sub_s` and `mul_s` to get the type-specific subtraction, addition, and multiplication productions from the component type. Thus, the type specific productions are used to specify the construct that `sign_s` forwards to and are used in the expressions in `coefs` shown in (3) above. This is allows different types, such as integers or exact-precision integers to be the component type of perturbation type. The productions that specify these types need only define the appropriate attributes (`subProd`, `mulProd`, etc). We do not provide definitions of the helper functions `subF` and `mulF` that compute `coefs`; the implementation is simple and more verbose than interesting.

1) Composing Simple and RLperturb: Given the above specification of the grammar `RLperturb` we can compose it with the generic sign productions in grammar `cg` and grammar `simple` to create the new extended language `simpleRLP`. This is done by the simple Silver specification given below:

```
grammar simpleRLP ;
import simple including syntax ;
import cg including syntax ;
import RLperturb including syntax ;
```

This grammar specification imports the definitions (productions, attribute, etc.) defined in the named grammars along with their concrete syntax specifications (which were not shown). Silver combines these grammar to create the grammar specifying the new extended language.

For programs written in this extended language the AG specification must provide a translation to a target language, be it Java byte-codes or C++. To do this, attributes that specify the translation of the host language are written for the productions in `simple` that define one of the `host` attributes (`hostStm`, `hostExp`, etc). For C++, the string valued attribute `ctrans` may be defined. Thus, programs written in the extended language must in essence be first translated to the host language and from there the translation to byte-codes or C++ is performed using traditional AG techniques. Forwarding is the means for realizing this translation. Where `r` is the root of the AST of the program, the value of `r.hostRoot.ctans` is the translation of the original tree `r` first to its host language tree representation `r.hostRoot` and then to C++. On a tree node created by `sign` with a perturbed-type expression, the `hostStm` attribute is computed for `sign` by forwarding to `sign_p` which computes the symbolic sign tree rooted by production `sign_s` which then forwards to the `block` production which defines its `hostStm` attribute as shown in Figure 3. The block may contain extension constructs and their host language translation is defined similarly. according to what host language construct they forward to.

B. Exact precision integers

To implement exact precision integers in our framework we follow the pattern established above for symbolic perturbation types but due to space limitations fewer actual specification can be shown. A grammar `exact` defines a new type production `exactT` that has no parameters and defines the type-specific production attributes `subProd`, `mulProd`, `signProd` to be those shown in Figure 9. This allows for operator overloading of $-$, $*$, and $sign$ as was done in `RLperturb`. The `sign_e` production is responsible for building the the expression in Figure 2 and forwarding to that tree. This tree is specified in the definition of `sign_e` in Figure 9 where it uses the attributes `fpTree`, `maxErr`, and `eTree` of the child expression `e`.

```
prod sign_e s::Stm ::= lhs::Exp  e::Exp
{ forwards to block ( ... e.fpTree ...
     ... e.maxErr ... e.eTree ...  ) ; }

prod sub_e e::Exp ::= l::Exp r::Exp
{ e.maxBits = 1 + max(l.maxBits,r.maxBits);
  e.maxErr = if e.maxBits <= 53 then 0 else
   l.maxErr + r.maxErr + exp(2,e.maxBits-53);
  e.fpTree = sub_d(l.fpTree,r.fpTree);   }

prod mul_e e::Exp ::= l::Exp r::Exp
{ e.maxBits = l.maxBits + r.maxBits;
  e.maxErr = if e.maxBits <= 53 then 0
       else l.maxErr * r.maxBits +
            r.maxErr * l.maxBits +
            exp(2,e.maxBits-53) ;
  e.fpTree = mul_d(l.fpTree,r.fpTree);   }
```

Fig. 9. Exact precision integer productions.

We can statically analyze expressions under `sign_e` to compute the maximum error, `maxErr`, that is possible if the expression was computed using floating-point types. We can also compute the maximum number of bits, `maxBits`, that are needed to precisely store any intermediate values in such an expression. In our framework, this static analysis is implemented by the feature designer by specifying attribute definitions on the overloading productions. Some of these are shown in Figure 9. The error value, $\langle$*maxErr*$\rangle$ in the condition in Figure 2, determines if the exact precision expression needs to be evaluated. These are the same definitions used in LN [3] implemented in the attribute grammar framework.[1]

In LN, exact precision numbers are represented as an unevaluated sum of hardware supported numbers. Thus, an exact precision integer value a is represented as $a_0 + a_1 + \ldots + a_m$ where $a_i = a'_i \times 2^{ri}$ and a'_i is an integral and r is the predefined radix, the number of bits that used to store each a'_i.[2] The size of this sum, m, is statically computed

[1] 53 is used because there are 53 bits of precision in double precision floating point numbers, the chosen hardware supported numeric type.

[2] Terms in this sum may, after a number operations, have more than r bits and thus must be *normalized* when the number of bits approaches the number of available bits, 53 in LN's implementation. The required normalization operations are inserted at statically-computed points in the expression.

as the attribute `cNum` (not shown). We implement this sum as an array of statically known size. For two such numbers, $a = a_0 + \ldots + a_m$ and $b = b_0 + \ldots + b_n$, where $n \geq m$, the sum $c = a + b$ is defined as $c_0 = a_0 + b_0$, $c_1 = a_1 + b_1$, $\ldots c_m = a_m + b_m$, $c_{m+1} = b_{m+1}$, $\ldots c_n = b_n$. The multiplication $c = a * b$ is similarly defined as $c_k = \sum_{0 \leq j \leq m} a_j * b_{k-j}$. In the implementation, each x_i is an array access with a constant index expression. In our framework, we implement this sum and product by defining a code fragment, using host language constructs, that implements the sequence of assignments indicated by the above expressions. It is just a series of assignment statements that are generated by unrolling the loops suggested by the above definitions of addition and multiplication. These code fragments are stored in the higher-order attribute `eTree` and is defined on all the exact-type productions. It is used in Figure 9 in the expression that `sign_e` forwards to as the code that efficiently implements the exact precision integers using host language constructs.

We also need to compute the floating point implementation of the expression; we define a higher-order attribute `fpTree` that is used to build the tree of this expression. On `mul_e`, for example, this is defined using the double sized floating point multiplication production `mul_d`. We've left out many of the details since they are simply an implementation of the analyses and transformation given in [3] and they do not add to the understanding of how they can be implemented in language extensions. The key point is that they can be specified by the feature designer by writing attribute definitions.

To extend Simple with exact precision integers, one needs only write a Silver specification like the one given above for `simpleRLP`. To create a specification of an extended language that uses both extensions, one needs only add the extra statement "`import exact include syntax;`" to the `simpleRLP` Silver specification. This composition of language features is carried out by the Silver attribute grammar tools. When Simple is extended with both `exact` and `RLperturb` one can use symbolic perturbation and exact precision numbers together by creating values of the type `rlpT(exactT())`. This is possible through the use of operator overloading, forwarding, and the fact that the transformations performed by `RLperturb` always query the component type for the type specific arithmetic operations (productions) to use in its transformations. (This last point is seen in Figure 8 in the use of the `subProd`, `subProd`, and `mulProd` attributes of the component type attribute `compType`.)

This implementation of exact precision numbers as an extension to Simple does not restrict the programmer from creating exact-precision type variables and assigning updated values to such variables inside of loops. Doing so would invalidate the analysis done in Figure 9 to compute `maxErr` and `maxBits`. In a more complete implementation of the host language and the extension this restriction would need to be enforced. Given the analysis that can be supported this does not present an exceedingly difficult challenge.

1) Performance:: One of our goals has been to demonstrate that our framework for extensible compilers makes it easy to use several geometric language extensions in the same program. It is also important that the code generated by our language extensions is efficient. Here we briefly report some performance measurements that compare our extensible language framework to existing techniques. However, there are no existing tools that allow one to easily write programs that use the *combination* of computational geometry techniques that we have demonstrated above.

Instead we have attempted to demonstrate that the C++ code generated by our extensible compiler framework is faster than equivalent code constructed using the non-native numerical types provided with CGAL. This prevents us from making any comparisons of the performance of perturbed expressions, since CGAL only provides perturbed primitives for a fixed precision type. We settled on measuring the speed of execution for expressions over CGAL's non-native numerical types, including the high-performance CORE::Expr type.

We tested our approach by running tests of 1,000,000 executions of the in-circle primitive, implemented as a sign-of-determinant expression, using our exact integer extension and various CGAL types. We have made every effort to ensure that the CGAL types are not disadvantaged in our test application, including (i) configuring in-lining and pass-by-reference to produce the fastest results for the CGAL types, (ii) doing all allocation and initialization of input values outside of our timing loop, (iii) running in batches of 10,000 cases to keep memory overhead low, and (iv) performing a code review with colleagues to look for inefficiencies in the test application. Our generated code consistently ran faster than the CGAL numeric types. The measured speed-up over MP_Float was 7.81, over Gmpz (GNU MP Integers) was 10.00, over Gmpq (GNU MP Rationals) was 23.47, and over CORE::Expr was 3.51. We attribute this speedup to (i) the avoidance of any allocation/construction calls in our code and (ii) the ability of the optimizer to work effectively on our (simple, repetitive, and monolithic) generated code.

We conclude that in addition to providing unique compatibility with perturbation and other potential extensions, our implementation of exact arithmetic provides a substantial performance improvements compared to object-based techniques. We believe the factors that benefit our exact arithmetic implementation are advantageous to symbolic perturbation. The performance improvements are *only* indicative of what is possible with the extensions in this domain. While the results may vary when the extensions are composed with a fully developed extensible C or Java we expect similar results.

V. Conclusion

We have shown how two language extensions can be made to work closely together in the same host language. These extensions both perform sophisticated semantic analyses and program transformations. This illustrates the power of attribute grammar-based extensible compiler frameworks as effective generative programming tools.

Feature designers and programmers play distinct roles in our approach. Programmers are not expected to be knowl-

edgeable about language implementation techniques such as attribute grammars. They only need to identify the language extensions that they want to use. The framework tools create the specification for the extended language from the attribute grammar specifications of the language extensions and the host language. Feature designers, however, are expected to have enough of an understanding of language processing to be able to implement their language extensions as attribute grammar fragments. Attribute grammars provide a high-level declarative language for specifying new language constructs, static analyses, and code transformations (via forwarding).

A. Related Work

Traditional syntactic, hygienic [11], and programmable [12] macros systems and embedded domain specific languages [13] allow new constructs to be added to a language but lack an effective way to optimize language constructs or statically report domain specific error messages. Meta-object protocol systems [14], [15] provide limited opportunities to add new language constructs but can check for errors and perform abstract syntax based optimizations, as can some modern macro systems [16], [17]. C++ template meta-programming [18] has a limited means of specifying optimizations but cannot easily specify the semantic analysis in the extensions shown here.

The problem of modular language definition and extensibility has received much attention from the AG community [19], [20], to mention just a few. Traditional AGs do not make the distinction between the feature designer, who writes the AG fragment, and the programmer, who combines extensions in building an extended language. Thus, the modularity and reuse of language features specified as AG fragments is achieved only by writing attribute definitions that "glue" new fragments into the host language AG. Of particular interest are the rewritable reference attribute grammars [21] in the JastAddII system in which an extensible Java 1.4 compiler has been specified. New (extension) constructs are translated to host language constructs by destructive rewrites on the syntax tree. These rewrites are triggered on an AST node when it is first queried for an attribute value. The value returned is the attribute value on the rewritten tree. Thus productions that implement a new construct cannot define attributes both explicitly via attribute definitions and implicitly via translation. Although forwarding is similar to rewriting, it is non-destructive; the original tree and the forwards-to tree exist simultaneously. This allows both the explicit and implicit (via forwarding) specification of semantics, a capability that we have found to be crucial in the highly modular language specifications required for extensible languages and composable language extensions.

B. Future Work

We are currently exploring several avenues of research to further develop the ideas and tools described in this paper. After validating the effectiveness of the CG extensions in the Simple host language we are currently implementing a full version of C as an extensible host language using the Silver AG system. A version of a significant subset of Java has also being developed in which we have explored an extension that embeds SQL into Java [22] and performs static syntax and type-checking of SQL queries. A more foundational aspect of our current work includes understanding the precise conditions under which language extensions can be safely composed and codifying these conditions as a set of analyses over the AG specifications of language extensions. (Silver currently performs an number of sanity checks and the well-definedness analysis [10] to ensure that in an extended language specification all needed attributes have exactly one definition.)

REFERENCES

[1] A. v. Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography." *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, June 2000.

[2] A. Fgabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schoenherr, "The CGAL kernel: A basis for geometric computation," in *Proc. Workshop on Applied Computational Geometry*, May 1996.

[3] S. Fortune and C. J. van Wyk, "Static analysis yields efficient exact integer arithmetic for computational geometry," *ACM Trans. on Graphics*, vol. 15, no. 3, pp. 223–248, 1996.

[4] J. Z. Emeris and J. F. Canny, "A general approach to removing degeneracies," *SIAM J. of Comp.*, vol. 24, no. 3, pp. 650–664, 1995.

[5] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2nd ed. Springer-Verlag, 2000.

[6] R. Seidel, "The nature and meaning of perturbations in geometric computing," *Discrete Computational Geometry*, vol. 19, pp. 1–17, 1998.

[7] H. Edelsbrunner and E. P. Muecke, "Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms," *ACM Transactions on Graphics*, vol. 9, no. 1, pp. 66–104, 1990.

[8] D. E. Knuth, "Semantics of context-free languages," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127–145, 1968, corrections in **5**(1971).

[9] H. Vogt, S. D. Swierstra, and M. F. Kuiper, "Higher-order attribute grammars," in *ACM PLDI Conf.*, 1990, pp. 131–145.

[10] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski, "Forwarding in attribute grammars for modular language design," in *Proc. 11th Intl. Conf. on Compiler Construction*, ser. LNCS, vol. 2304, 2002, pp. 128–142.

[11] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, "Hygienic macro expansion," in *Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM Press, 1986, pp. 151–161.

[12] D. Weise and R. Crew, "Programmable syntax macros," *ACM SIGPLAN Notices*, vol. 28, no. 6, 1993.

[13] P. Hudak, "Building domain-specific embedded languages," *ACM Computing Surveys*, vol. 28, no. 4es, 1996.

[14] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the MetaObject Protocol*. Cambridge, MA. USA: MIT Press, 1991.

[15] S. Chiba, "A metaobject protocol for C++," in *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*. ACM Press, 1995, pp. 285–299.

[16] D. Batory, D. Lofaso, and Y. Smaragdakis, "JTS: tools for implementing domain-specific languages," in *Proceedings Fifth International Conference on Software Reuse*. IEEE, 2–5 1998, pp. 143–53. [Online]. Available: citeseer.nj.nec.com/171171.html

[17] J. Baker and W. Hsieh, "Maya: Multiple-dispatch syntax extension in java," in *Proc. of ACM PLDI Conf.* ACM, 2002, pp. 270–281.

[18] T. Veldhuizen, "Using C++ template metaprograms," *C++ Report*, vol. 7, no. 4, pp. 36–43, May 1995.

[19] H. Ganzinger, "Increasing modularity and language-independency in automatically generated compilers," *Science of Computer Programing*, vol. 3, no. 3, pp. 223–278, 1983.

[20] U. Kastens and W. M. Waite, "Modularity and reusability in attribute grammars," *Acta Informatica*, vol. 31, pp. 601–627, 1994.

[21] T. Ekman and G. Hedin, "Rewritable reference attributed grammars." in *Proc. of ECOOP '04 Conf.*, 2004, pp. 144–169.

[22] E. Van Wyk, D. Bodin, and P. Huntington, "Adding syntax and static analysis to libraries via extensible compilers and language extensions," in *Proc. of LCSD 2006, Library-Centric Software Design*, 2006.