From Models to Efficient Code: It's All in the Middle

Eric Van Wyk Mats P.E. Heimdahl Yousef Saad
Department of Computer Science and Engineering
University of Minnesota
200 Union Street SE, Minneapolis MN, 55455, USA
{evw,heimdahl,saad}@cs.umn.edu

1 Introduction

In our opinion, high-performance computing is plagued by two main problems. First, the main users of various high-performance applications are experts in their respective fields (physicists, chemists, biologists, and many other fields other than computer science) and they are solving problems in domains largely unrelated to the computing field. Yet, they are typically forced to work with low-level general purpose programming languages designed for ease of translation rather than ease of modeling. Consequently, productivity is low and many promising new techniques never see the light of day because it is simply too costly and difficult to integrate them into existing code-bases. Second, even though high-performance applications are developed at a low level, the additional performance tuning required to make the code run efficiently on new and evolving platforms is a time consuming process—the theoretical performance of a new platform is rarely (if ever) achieved and manually tuning the code to get close is very difficult. Thus, we see two main problems; (1) the lack of domain specific languages and domain specific language features suitable for the scientific and high-performance modeling tasks at hand and (2) the lack of a highly flexible and efficient translation infrastructure that will allow us to rapidly modify a compiler to take advantage of characteristics of the problem domain as well as the target hardware.

To address both of these problems, we hypothesize that an *extensible language* implemented using attribute grammars with forwarding [13] can serve as a *host-language* for a plethora of domain specific *language-extensions*. There are two primary ways in which we can use such language extensions to create an *extended-language*. First, such extension can be combined to construct new *domain specific modeling-languages* suitable for different audiences and domains with high-performance computing needs. In this case, the programs written in the extended-language use only the high-level constructs introduced by the language extensions. These languages are similar to domain specific languages. A second way to use language extensions is to use them more sparingly in that programs written in the extended-language use many constructs from the underlying host-language as well as construct defined by the language extensions. In this case, the programs are typically recognizable as programs written in the host-language with the exception that there are additional domain specific constructs being used. The attribute grammar based compilers for such languages can be seen as "pre-processors" that compile the language extension features down to their representation in the host language. A traditional compiler can then be used to translate the resulting code to machine code. Extensible languages based on attribute grammars and forwarding can also serve as a framework for the translation of these extension constructs into their semantically-equivalent representation in the host-language and the host-language translation to various hardware architectures.

Domain specific languages: In the embedded systems domain, a collection of languages suitable for different domains have emerged. For instance, MATLAB [10] is often used to develop control laws, and Simulink [10] or SCADE [4] may be used to capture these control laws as block diagrams. These languages, however, are not suited for modeling the complex mode-logic that is required for, for example, air transport flight guidance and flight management systems. Notations such as Statecharts [6], SCR [7], RSML [9], and Safe State Machines [4] are much better suited for this purpose. By selecting the most appropriate modeling language for the task at hand, dramatic quality and productivity improvements can be achieved. Similarly, we believe that those writing scientific applications and solving problems in the high-performance computing area would benefit from the use of domain specific languages or familiar languages extended with domain specific features that aid in writing scientific applications. Scientists, engineers, and other users of high-performance computing can benefit from domain-specific languages and domain-specific language features in two different ways.

First, the domain-specific language or extended language will likely be easy to use, an efficient modeling tool, transparent, and concise because the domain specific notations of the task at hand can be used. Second, when developing large models or application codes, a domain-specific language can be vital in speeding up the development, debugging, and validation efforts before automatically translating it into an efficient package, possibly written in a more traditional language. In fact, domain specific languages have been developed for high-performance computing as well. For example, Sisal is a functional language used on parallel architectures for scientific applications. Many Sisal programs would perform better that the equivalent Fortran programs that were compiled using automatic parallelizing and vectorizing compilers [1]. Although these languages provide the benefits we have cited above, they are sometimes not adopted by practitioners. There are several reasons for this but the two of most significant interest are:

1. They have a legacy code base in FORTRAN or another language and they do not want to translate it.

2. They are not convinced of the advantages of using a domain specific language and thus see the cost of trying it out on their problems to be too high.

Extensible languages provide a means for overcoming these obstacles. If the host-language is the same as the language used in the legacy code, then practitioners can easily experiment with a few domain specific extensions to their existing applications. The extensible language compiler will generate the representation of the extension constructs in the host language and and their familiar compiler can then be used to generate machine code. This significantly reduces the cost to the practitioners of experimenting with domain specific language extensions in their existing applications.

The domain specific language features that are packaged as language extensions will primarily be drawn from two domains. The first domain is the problem domain; be it simulation of fluid flow over an airplane wing or the modeling of a chemical reaction. Scientists working on such problems should be able to use a notation from their own domain—a notation we envision created from language extensions. Such language extensions raise the language's level of abstraction to that of the problem. The second domain is the domain of high-performance computing itself. The programmers of high-performance applications should be able to declaratively specify, for example, the communication patterns and layout of data in memory instead of having to implement this intention as a collection of low-level send/receive messages.

Efficient computations: Unless the executables generated from these high-level programs containing domain specific language features are efficient they are of little use. Because domain specific language features can be used to trigger domain specific optimizations, efficient programs can be generated. In fact, these *optimizations are often not possible without the domain specific information*; this information is easily found on the domain specific constructs. However, this information is very difficult or impossible to gather from the equivalent program written in a general purpose language. In "Impact of economics on compiler optimization" [11], Robison shows that it is not economically feasible to include many important (domain specific) optimizations in traditional compilers. Instead, these optimizations should be language extensions that programmers can add to their compiler's optimization repertoire.

The example of MATLAB as a rapid prototyping tool is worth mentioning here. MATLAB (a product of MathWorks Inc.) started in the 1970s as an interactive interface to matrix libraries which were then just developed (EISPACK and LINPACK). As the package grew in popularity, it progressively became useful across many engineering and scientific disciplines and soon started being viewed as a sort of general purpose "drafting" tool. It gives the user a very fast way of checking whether or not an idea will work before attempting to program it in C or FORTRAN. The trend of the last decade that is interesting to observe is that MATLAB *gained popularity the moment MathWorks started including what they refer to as "toolboxes" geared toward specific applications.* For example, control theorists have access to a control toolbox, while mechanical engineers will find a finite element toolbox. The number of available toolboxes has been steadily increasing in recent years as MathWorks seems to be aware of their importance. MATLAB is not written to provide efficient implementations, but it is a popular interactive package used for rapid proto-typing by scientists and engineers. If efficient codes can be automatically generated the use of domain specific language features would be of use to many more practitioners.

In section 2 of this paper we propose a method for developing extensible programming languages and their supporting tools. In section 3 we show an example from previous work in the domain of computational geometry that has much in common with HPC. Section 4 summarizes and concludes.

2 Extensible Languages

Extensible languages provide a means of overcoming both of the problems mentioned in the introduction. An *extensible language* is one that can easily be extended with a unique combination of domain specific language features. We will refer to this language as a *host language*. The language features that can be added to a host language take the form of

1. new language constructs that raise the level of abstraction to that of a particular problem domain,

2. new domain specific optimizing transformations, or

3. new semantic analysis that ensure that new language constructs are used correctly and determine when the optimizations can be safely applied.

In our framework, a host language defined by an attribute grammar [8] that has been extended with a mechanism called *forwarding* [13]. A language extension is specified as an attribute grammar *fragment* that seamlessly extends the host language with the new language features defined in the extension. The process of building an extended language is carried out by the language framework tools by simply taking the "union" of the host language attribute grammar specification and the attribute grammar specifications of the programmer chosen language extensions to create the extended language specification. This specification is then used by the tools to automatically generate a compiler for the language.

Forwarding: We have extended attribute grammars with a mechanism called *forwarding* that allows attribute grammar productions to specify a *semantically equivalent* expression or statement. If a production is queried for an attribute (say *jbc*, its Java byte-code translation) that it does not define then the query is passed onto the semantically equivalent expression, called the *forwards-to* tree, and its value for the attribute can be safely used as the result of the query. An example will clarify. Consider a *foreach* construct that iterates over elements of a collection (an array, set, list, etc.). The statement "*foreach x :: t in c do s*" would perform statement s for each x in the collection c that contains elements of type t. The production defining the semantics of *foreach* may define an *errors* attribute that generates appropriate error messages and an attribute *unparse* that specifies its concrete syntax and my be used in the messages defined by *errors*. This production would also specify that it forwards to the semantically equivalent statement

```
t x ;
for (Iterator i = c.iterator(); i.hasNext(); ) {
        x = i.next(); s }
```

It may be that this production does not define the attribute *jbc*. In this case, the query for *jbc* on the *foreach* construct is forwarded to the *declaration/for* construct which returns its *jbc* attribute value. In this way, a language extension production can *explicitly* define the semantics, as attributes, that are of interest to it, and *implicitly* define its semantics, via forwarding, of the remaining attributes.

3 An example from computational geometry

In this section we demonstrate some of the capabilities of our framework with a *domain-specific language extension* example from computation geometry. Our goal with this demonstration is to show how in our language framework efficient programs can be generated from the high-level specifications.

This extension does not introduce any significant new syntax, but illustrates the use of *forwarding as rewriting to implement operator overloading and transformational optimizations.* These *optimizations are not possible without domain specific knowledge*; they rely on identifying domain specific constructs and could not be performed after the constructs have been translated to their base language implementation.

3.1 Symbolic perturbation

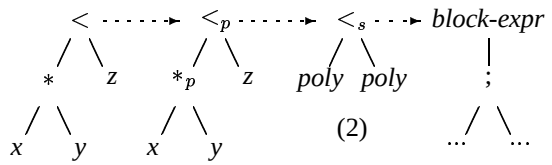
In computational geometry, many algorithms, such as computing the convex hull of a set of 2D points, perform qualitative tests on geometric elements such as points and lines. These tests are called *primitives* and include checking whether a point lies inside or outside a circle or is to the left or right of a directed line in 2D space. The algorithms that use these tests can be significantly simplified if the degenerate cases such as when the point lies exactly on the line do not happen. Simply stated, when we compare coordinate values we do not want them to be equal. Treating equality as always greater than (or always less than) does not solve the problem and can lead to non-termination or incorrect results in some algorithms [2]. Thus, using the proper numerical kernel is very important for computational geometry problems. The domain experts in computational geometry have devised a notion called *symbolic perturbation* [14, 12] that symbolically adds an infinitesimally small *perturbation value* to coordinates so that these degenerate cases never occur or occur extremely rarely, and can then be detected.

In the *randomized linear perturbation* scheme used here [3], when a programmer writes, for example, $x * y < z$, 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 two coordinates are compared the original coordinates are first compared; if they are the same the perturbation values are compared to break the tie. Thus, an expression $x * y < z$ initially transforms to

$$(x +_s E_x *_s e) *_s (y +_s E_y *_s e) <_s (z +_s E_z *_s e) \quad (1)$$

$\{$ *coord* $t_1 = x * y$; *coord* $t_2 = z$;
if $(t_1 < t_2)$ *return true*;
if $(t_2 < t_1)$ *return false*;
$t_1 = x * E_y + y * E_x$; $t_2 = E_z$;
if $(t_1 < t_2)$ *return true*;
if $(t_2 < t_1)$ *return false*;
$t_1 = E_x * E_y$; $t_2 = 0$;
if $(t_1 < t_2)$ *return true*;
if $(t_2 < t_1)$ *return false*;
throw perturbation_exception; $\}$

Figure 1. Fast, computable expression.



(Figure 1)

Figure 2. The generic, perturbed-type, symbolic-type and block expression trees implementing symbolic perturbation. Dashed arrows point to forwards-to trees.

We now transform this expression so that the comparison can be made. We convert coordinate-valued expressions (the expressions under $<_s$) to *polynomials over* e in which the coefficients are computable (non-symbolic) expressions by first 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

$$((x * y) +_s (x * E_y + y * E_x) *_s e +_s (E_x * E_y) *_s e^2) <_s (z +_s E_z *_s e) \quad (2)$$

Finally, because e is infinitesimally small, we can convert $<_s$ to an expression that implements the symbolic comparison by comparing the coefficients of the two polynomials in increasing order of their associated powers of e. The final expression is shown in Figure 1 as a block-expression, essentially an inlined-function body. The original expression is rewritten without symbolic operations, so the evaluation is possible, and that the perturbation values are not computed unless they are needed. This optimization is absolutely critical.

3.2 Symbolic perturbation as a language extension

To implement these constructs and transformations as a language extension we will use a host-language operator overloading feature, higher-order attributes to build trees, and strategically applied rewrite rules. We introduce perturbed types and symbolic types, and type-specific productions for variable references and the operators $+, *$, and $<$. These define attributes used in the transformations. Because the evaluation of attributes is demand-driven, this transformation process can most easily be understood by beginning at the root node of the comparison expression tree $x + y < z$.

Operator overloading: As originally detailed in [13], a host language may use forwarding to enable operator overloading by using a generic production for each overloaded operator that forwards to a type-specific production for that operator. Here, the parser uses a generic $<$ production to construct the initial tree. When this production is queried for an attribute, say *jbc*, that it does not define, it must build its forwards-to tree. To do this, the production queries its children for their *type* attribute that is a node in the attributed abstract syntax tree that defines their types and resolves them to a single type, perhaps their least common super-type. In this case, that type is the perturbed-type node and it is queried for a *production-valued* attribute called *lessthanSpecializer* that is defined on all types that overload $<$. If the type does not define this attribute we raise a program-error since the feature designer that introduced this type has decided, by not defining the attribute, to not overloading this operator. In this case, the perturbed type does define the attribute as the perturbed-type-specific production $<_p$. The generic $<$ production forwards to a tree it builds using this type-specific production and its two child expression trees. Since the generic $<$ production defines no attributes, all attribute queries are forwarded to the type-appropriate generated tree. The resulting trees are the two left-most trees in Figure 2. Since we overload $<, +, *$ for the perturbed type we essentially get a tree built with perturbed-type specific productions.

The $<_p$ node in Figure 2 is now queried for *jbc*, which it does not define, and thus must build a symbolic-expression tree that it can forward to. The $<_p$ production forwards to the symbolic comparison whose subexpressions are polynomials over *e*, that is, expression (2) above. To build this tree, it queries its subexpressions for their symbolic representation, implemented by perturbed-type expressions as the higher-order attribute *symTree*. Perturbed type operators define this attribute as expected as the symbolic version of themselves and perturbed-type variable references, for say x, define it as their symbolic expression $x +_s E_x *_s e$. Thus, $<_p$ defines $symTree$ as the symbolic representation of the subexpressions, that is (1) in the example above. We do not forward to $symTree$ but instead query it for its *distributed* version, $distTree$, which is a sequence of sums of products of real and symbolic values. We then query that for its poly-

nomial version, $polyTree$, in which like-powered terms of e are gathered together, and forward to that symbolic expressions, build by production $<_s$. This tree is (2) above. To complete the transformation, $<_s$ queries its subexpressions for a list of their coefficient expressions and builds a block expression like the one in Figure 1 that it forwards to.

In Figure 2 we see, from left to right, the original tree generated by the parser, the tree specialized for perturbed-types, the polynomial symbolically-typed tree (expression 2) in our example) and finally the block-expression written entirely with host-language constructs that defines the efficient computable implementation of $x * y < z$ under symbolic perturbation. Building these intermediate trees is relatively straight-forward and is not difficult for the language feature designer to specify. The feature designer is an expert in the domain on interest; in this case it is computational geometry.

This example shows that extensible language framework based on attribute grammars with forwarding has two important traits:

1. High-level domain specific constructs make it easier for the programmer to implement a solution to their problem because they can specify this solution in the familiar notation of the domain.

2. Domain specific optimizations and transformations can transform the high-level specifications into efficient code written in the host-language.

4 Summary and Conclusion

Above, we have shown only one aspect of our extensible language framework. Nevertheless, we believe it demonstrates how high-level domain specific notations along with their domain specific optimizations and code generation techniques can be packaged as language extensions that can be seamlessly and modularly added to a host-language.

These same techniques can be applied in to other aspects of the domain of high-performance computing. For example, applications may be written using domain specific notations that declaratively specify the communication patterns and data layout strategies that are to be used in a concurrent application instead of forcing the programmer to write, at a low level of abstraction, the explicit message send and receive commands that one needs to use when using a message passing library like MPI [5].

We conjecture that these extensible language techniques will be useful in high-performance applications in that they would allow non-computer scientist practitioners to specify their problem solutions in a full-featured modern language extended with just the right combination of domain specific language features. We also believe that the flexibility of our approach will provide us with the capability to generate efficient code for various high-performance platforms. The key is to be able to leverage not only knowledge of the target platform, but also knowledge of the problem domain—knowledge that is only available because the programmer has used a domain specific construct.

References

[1] D. Cann and J. Feo. Sisal versus fortran: a comparison using the livermore loops. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 626–636. IEEE Computer Society, 1990.

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

[3] J. Z. Emeris and J. F. Canny. A general approach to removing degeneracies. *SIAM Jounal of Computing*, 24(3):650–664, 1995.

[4] T. Esterel. Corporate web page. www.esterel-technologies.com, 2004.

[5] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI - The complete reference*. MIT Press, 1998. 2nd edition.

[6] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[7] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, 1995.

[8] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in 5(2):95–96, 1971.

[9] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements Specifi cation for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[10] MathWorks. The mathworks inc. corporate web page. http://www.mathworks.com, 2004.

[11] A. D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 1–10. ACM Press, 2001.

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

[13] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th International Conf. on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002.

[14] C.-K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *Journal for Computer and System Sciences*, 40:2–18, 1990.