POET: A Scripting Language For Applying Parameterized Source-to-source Program Transformations *

Qing Yi (qingyi@cs.utsa.edu)
University of Texas at San Antonio

Abstract

We present POET, a scripting language designed for applying advanced program transformations to code in arbitrary programming languages as well as building ad-hoc translators between these languages. We have used POET to support a large number of compiler optimizations, including loop interchange, parallelization, blocking, fusion/fission, strength reduction, scalar replacement, SSE vectorization, among others, and to fully support the code generation of several domain-specific languages, including automatic tester/timer generation, and automatically translating a finite-state-machine-based behavior modeling language to C++/Java code. This paper presents key design and implementation decisions of the POET language and show how to use various language features to significantly improve the productivity of supporting programmable compiler optimization for high performance computing and supporting ad-hoc code generation for various domain-specific languages.

1 Introduction

The development of most software applications today requires a non-trivial number of program transformations and translations between different languages. For example, domain-specific algorithmic designs need to be translated to general-purpose implementation languages such as C/C++/Java, systematic program transformations need to be applied to migrate existing software or improve performance of the developed C/C++/Java code, and compilers are required to translate source programs to machine/byte code for execution. The effectiveness of the program transformations and the efficiency of the generated code are critical concerns that routinely determine the success or failure of a software product.

A large collection of development tools, e.g., Pathfinder [1], Metamill [2], and UModel [3], exist to automatically translate high-level software design to lower-level implementations, and a number of domain-specific systems, e.g., ATLAS [51] and FFTW [23], have been built to automatically generate efficient implementations of key computational kernels on

*This research is funded by NSF through grant CCF0747357 and CCF-0833203, and DOE through grants DE-SC0001770
a wide variety of computing platforms. These existing infrastructures for supporting specialized code generation and domain-specific performance optimization, however, are mostly developed using general-purpose programming languages such as C/C++/Java or string-manipulating scripting languages such as Perl/Python. While existing open-source compilers (e.g., gcc, ROSE [34]) can be used to provide infrastructure support for general-purpose program analysis and transformation, they are dedicated to only a few popular programming languages. Due to the lack of built-in support for parsing/unparsing of the input code, integration of ad-hoc domain-specific notations, and systematic application of structured program transformations, the development cost for building specialized code generators or optimization frameworks are prohibitive and have limited their uses to a small set of design notations and computational kernels that have wide-spread use. Reducing such overhead could significantly improve the variety of domain-specific code generators and optimization frameworks available for software development.

POET is an interpreted language designed for applying advanced program transformations to code in arbitrary languages as well as quickly building ad-hoc source-to-source translators between these languages. It has been used to support the transformation needs of both popular programming languages such as C/C++, Java, FORTRAN, and several domain-specific languages that we have designed on the fly for various purposes. Figure 1 shows the structure of a typical POET transformation engine, which is essentially a POET language interpreter coupled with a set of transformation libraries and language syntax descriptions. The transformation libraries include predefined POET routines which can be invoked to apply a large number of compiler optimizations such as loop interchange, parallelization, fusion, blocking, unrolling, array copying, scalar replacement, among others. The language syntax specifications, on the other hand, are used by the POET interpreter to dynamically parse input code in a variety of different programming languages. The developer needs to write a POET script to specify which input files to parse using which syntax descriptions, what transformations to apply to the input code after parsing, and which syntax to use to unpars the transformation result. The POET script can be extensively parameterized and reconfigured via command-line options when invoking the transformation engine.

A POET transformation engine as illustrated by Figure 1 can play many different roles as it is used for various purposes. In particular, the design of the language has focused on supporting the following software development needs.

• Programmable compiler optimization for high performance computing. POET was initially designed for extensive parameterization of compiler transformations, including sophisticated loop and data layout transformations, so that the configuration of
these optimizations can be empirically tuned [55]. It provides programmable control of compiler optimizations for developers, especially high performance computing specialists, to selectively apply these optimizations as well as conveniently define their own customized algorithm-specific optimizations [57].

- Ad-hoc source-to-source translation and domain-specific code generation. POET is language neutral and uses external syntax descriptions to dynamically parse/unparse code in arbitrary programming languages. We have used POET to automatically generate context-aware timers for computational intensive routines [35], to automatically produce object-oriented C++/Java implementations from a finite-state-machine-based behavior modeling language [54], and to automatically translate parameter declarations in POET to the input languages of independent search engines so that the configurations of the POET scripts can be automatically tuned [28].

This paper focuses on key design and implementation decisions of the POET language to support the above use cases. Section 2 first presents an overview of the language. Sections 3, 4 and 5 then present details of the language design to effectively support dynamic parsing of arbitrary languages, convenient pattern matching and traversal of the input code, and flexible composition and tracing of program transformations. Section 6 presents use case studies of the language. Finally, Sections 7 and 8 present related work and conclusions.

2 Overview of the Language

POET is designed to make compiler optimizations readily available to developers for programmable control and to significantly reduce the development cost of supporting ad-hoc language translation and code generation of domain-specific languages. Table 1 summarizes the core concepts supported by POET to achieve its design goals. Figure 2 shows an example POET script to demonstrate the use of these language concepts.

2.1 The Type System

As shown in Table 1, POET supports two types of atomic values: integers and strings\textsuperscript{1}. The boolean value \textit{false} is represented using integer 0, and \textit{true} can be represented using any of the other integers. Two notations, \textit{TRUE} and \textit{FALSE}, are provided to denote integers 1 and 0 respectively. Additionally, the following compound types are supported within POET.

- \textit{Lists}. A POET list is a singly linked list of arbitrary elements and can be constructed by simply enumerating all the elements. For example, (a \textit{“<”} b) produces a list with three elements, a, \textit{“<”}, and b. Lists can be dynamically extended using the :: operator, shown at line 13 of Table 1, and are extensively used in POET to conveniently store sequentially-accessed elements.

- \textit{Tuples}. A POET tuple is a finite number of statically-composed elements separated by commas. For example, (\textit{“i”}, 0, \textit{“m”}, 1) constructs a tuple with four elements,

\textsuperscript{1}POET does not support floating point values under the assumption that program transformation does not need floating point evaluation. The language may be extended in the future if the need arises.
“i”, 0, “m”, and 1. Tuples cannot be dynamically extended, so they are used to represent only static sequences of values, e.g., the parameters of a function call.

- **Associative Maps.** POET maps are used to associate pairs of arbitrary values and are constructed by invoking the MAP operator shown at line 5 of Table 1, e.g., MAP(3=> “abc”) builds a map that associates 3 with “abc”. The content of associative maps can be dynamically modified using assignments, shown at line 22.

- **Code Templates.** Each POET code template is a distinct user-defined data type and is constructed using the # operator shown at line 6 of Table 1, where each component of the data type (parameter of the code template) is given a specific value. They are used to build pointer-based data structures such as arbitrarily shaped trees and DAGs (directed acyclic graphs) and are used to construct internal representations of the input code that POET scripts operate on. POET offers integrated support for

---

Table 1: Overview of the POET language

<table>
<thead>
<tr>
<th>Types</th>
<th>Compound type construction</th>
<th>POET expressions: operating on different types of values</th>
<th>POET statements: variable assignment and control flow</th>
<th>Global type/variable declarations and executable commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. atomic types</td>
<td>1. e_1 e_2 ... e_n</td>
<td>1. !, &amp;&amp;, ==, !=, &lt;, &lt;=, &gt;, &gt;=, ==, !=</td>
<td>1. a = b</td>
<td>30. &lt;define a b/&gt; Declare a macro variable a and assign b as its value</td>
</tr>
<tr>
<td>2. compound types</td>
<td>2. e_1, e_2, ..., e_n</td>
<td>2. Integer arithmetics and comparison</td>
<td>2. a[i] = b Modify a local or static variable a to have value b; return b as result of evaluation</td>
<td>31. &lt;trace a1, ..., am/&gt; Declare a list of related trace handles a1, ..., am</td>
</tr>
<tr>
<td>integers (e.g., 1, 20, -3), strings (e.g., “abc”, “132”)</td>
<td>3. A list of n elements e_1, e_2, ..., e_n</td>
<td>3. Boolean operators</td>
<td>3. a[i] = b Modify associate map a so that entry i is associated with value b; return b as result of evaluation</td>
<td>32. &lt;parameter p type=t default=v parse=r message=d/&gt; Declare a command-line parameter p which has type t, default value v, can be parsed using parsing specifier r, and its meaning is defined in string d.</td>
</tr>
<tr>
<td>lists, tuples, associative maps, code templates, xform handles</td>
<td>4. A tuple of n elements e_1, e_2, ..., e_n</td>
<td>4. Equality comparison between arbitrary types of values</td>
<td>4. (a1, ..., am) = (b1, ..., bm) Modify variables a1, ..., am to have values b1, ..., bm respectively; return the b tuple</td>
<td>33. &lt;input cond=c from=f syntax=s to=t/&gt; If expression c evaluates to true, parse the input code from file f using syntax descriptions defined in file s, then save the parsing result to variable t</td>
</tr>
<tr>
<td></td>
<td>5. MAP{f1=&gt;t1, ..., fn=&gt;tn}</td>
<td>5. Concatenate two values a and b into a single string</td>
<td>5. a1; a2; ... ; am Evaluate expressions a1 a2 ... am in order; return the result of evaluating am</td>
<td>34. &lt;eval s1, ..., sm/&gt; Evaluate the group of expressions/statements s1, ..., sm</td>
</tr>
<tr>
<td></td>
<td>6. c # (p1, ..., pn)</td>
<td>6. Access the rest of the list after the first element; return c if a is not a list</td>
<td>6. RETURN a Evaluate expression a and then return it as the result of the current xform invocation</td>
<td>35. &lt;output from=t to=f syntax=s cond=c/&gt; If expression c evaluates to true, unparse the expression t to file f using syntax descriptions defined in file s.</td>
</tr>
</tbody>
</table>
Figure 2: An example illustrating the overall structure of each POET file

associating parsing/unparsing specifications with each code template so that input programs can be automatically parsed and converted to their internal representations (for more details, see section 3.1).

• *Xform Handles.* Each POET xform handle is a global function pointer and is constructed by following the name of a POET *xform routine* with an optional list of values to pre-configure invocations of the routine, shown at line 7 of Table 1. Each *xform handle* can be invoked by following it with a tuple of input parameters and will return the result of evaluating the corresponding routine using the given parameters. More details of POET xform routines are provided in Section 2.2.

### 2.2 Components of A POET Script

Figure 2 shows the typical structure of a POET script, which includes a sequence of *include* directives (line 1 of Figure 2), type declarations (lines 2-13), variable declarations (lines 14-18), and executable commands (lines 19-21). Except for the *include* directives, which must start a POET script and specify the names of other POET files that should be evaluated before continue reading the current one, all the other POET declarations and commands can appear in arbitrary order and are evaluated in their order of appearance. As illustrated at lines 2 and 6 of Figure 2, a POET *comment* can appear anywhere and is either enclosed inside a pair of `<* and *>` or from `<<*` until the end of the current line.

POET supports two categories of user-defined types: *code templates*, which define data types that can be used to construct internal representations of the input code, and *xform routines*, which define functions that implement various program analysis and transformation algorithms. Their declarations are illustrated at lines 2-13 of Figure 2. In particular, lines 3-
6 define a code template type named \textit{Loop}, which has four parameters (data fields) named \textit{i}, \textit{start}, \textit{stop}, and \textit{step} respectively. The qualification of the template parameters and the template body at line 4 of Figure 2 are used to describe the corresponding syntax of the code template when it appears in a programming language (for more details, see Section 3.1). Lines 7-14 define a \textit{xform routine} named \textit{ReverseList}, which has a single input parameter named \textit{list} and an optional parameter named \textit{prepend}, which has empty string as value if left unspecified in a routine invocation. When the \textit{xform} routine is invoked, e.g., at line 20 of Figure 2, a value must be given to the required parameter \textit{list}, but the additional argument supplied at line 20 to replace the default value of \textit{prepend} is not required.

Besides user-defined types, each POET script can also declare three categories of global variables, \textit{macros}, \textit{command-line parameters}, and \textit{trace handles}, whose lifetime and scopes span through the entire execution of all POET scripts interpreted by a POET transformation engine. For example, line 14 of Figure 2 declares a \textit{macro} named \texttt{OPT_STMT} and assigns the code template \textit{Loop} as its value. Each \textit{macro} can be used to reconfigure behavior of the POET interpreter and is discussed in more detail in Section 3.4. Lines 15-17 declare three \textit{command-line parameters} named \texttt{inputFile}, \texttt{inputLang} and \texttt{outputFile}, whose values can be redefined via command-line options and are used to extensively parameterize POET scripts. Line 18 declares a \textit{trace handle} named \texttt{inputCode}, which is used to keep track of transformations to the input code. Each trace handle can act as an integral component of a compound data structure to trace transformations to the data. For example, the trace handle \texttt{inputCode} could be modified by the invocation of the \texttt{AnalyzeOrTransformCode} routine at line 20 of Figure 2 although the routine implementation cannot directly access it by name. Section 5.2 discusses the use of trace handles in more detail.

POET supports three types of executable commands, \texttt{input}, \texttt{eval}, and \texttt{output}, illustrated at lines 19-21 of Figure 2. In particular, the \texttt{input command} at line 19 is used to parse a file named by variable \texttt{inputFile} using syntax descriptions contained in a file named by \texttt{inputLang}. The parsing result is then converted into an internal representation and stored to variable \texttt{inputCode}. The \texttt{eval command} at line 20 specifies a sequence of expressions and statements to evaluate. The \texttt{output Command} at line 21 is used to write the transformed internal representation (i.e., \texttt{inputCode}) to an external file named by \texttt{outputFile}.

All POET expressions and statements must be embedded within an \texttt{eval command} or the body of a code template or \textit{xform} routine. Most POET expressions are \textit{pure} in the sense that unless trace handles are involved, they compute new values instead of modifying existing ones. POET statements, as shown in Table 1, are used to support variable assignment and program control flow. Except for loops, which always have an empty value, all the other POET statements have values just like expressions. However, when multiple statements are composed in a sequence, only the value of the last statement is returned.

2.3 Variables And Assignments

POET variables can be separated into the following three categories, each managed using a separate group of symbol tables. All variables can hold arbitrary types of values, and their types are dynamically checked during evaluation to ensure type safety.
• **Local variables**, whose lifetime and scopes span through the bodies of single code templates or *xform* routines. For example, at lines 2-13 of Figure 2, `i`, `start`, `stop`, `step` are local variables of the code template `Loop`, and `list.result`, and `p_list` are local variables of the *xform* routine `ReverseList`. Local variables are introduced by declaring them as parameters or simply using them in the body of a code template or *xform* routine.

• **Static variables**, whose lifetime and scopes are restricted within individual POET files to avoid naming conflicts from other files. Each POET file can have its own collection of static variables, which can be used freely within the file without prior declaration to propagate information across different components of the same file.

• **Global variables**, whose lifetime and scopes span throughout the entire execution of a POET transformation engine and across multiple files. There are three categories of global variables: *macros*, *command-line parameters*, and *trace handles*, each of which serves a special purpose and must be explicitly declared in the global scope before used, as discussed in Section 2.2.

All names that have not been previously declared are treated as local or static variables, depending on the scopes of their appearances. In particular, all names within a code template or *xform* routine body are considered local variables unless an explicit prefix, such as `GLOBAL`, `CODE`, or `XFORM`, is used to qualify the name. An example of such qualified names is shown at line 14 of Figure 2. All local and static variables can be modified within their scopes using the assignment operator, shown at lines 21 and 23 of Table 1. However, assignment cannot be used to modify global variables or any portion of a compound data structure such as a tuple, list, or code template, which cannot be modified unless through the insertion of *trace handles*, discussed in Sections 5.2.

### 2.4 Key Language Features

A key design objective of POET is to dynamically support the parsing/unparsing of arbitrary programming languages and to provide full programming support for defining arbitrary transformations to the input code. Each transformation can be extensively parameterized, and built-in tracing support is provided to ease the composition of parameterized program transformations. The language targets both source-to-source compiler optimizations (e.g., automatic parallelization and loop transformations) and domain-specific program analysis and code generation, via the following key language features.

**Ability to dynamic parse/unparse arbitrary programming languages** POET is language neutral and uses syntax specifications defined in external files to dynamically process different input/output languages. Transforming a different programming language simply requires using an alternative language syntax specification file, e.g., by supplying a different file name to the `inputLang` command-line parameter in Figure 2. POET has been used to support subsets of a wide variety of different programming languages such as C, C++, Fortran, Java, COBOL, and have been used to additionally support a number of ad-hoc domain-specific languages. After being parsed, input codes from different programming languages can be mixed together, and their internal representations can be modified in a uniform fashion via language independent program transformation routines.
Convenience of expressing arbitrary program transformations  In POET, program transformations are defined as xform routines which take a collection of input data and return the transformed code as result. These routines can use arbitrary control-flow such as conditionals, loops, and recursive function calls, can build compound data structures such as lists, tuples, hash tables, and code templates, and can invoke many built-in operations (e.g., pattern matching, AST replacement and replication) to operate on the input code. The full programming support for defining arbitrary customizable transformations distinguishes POET from most other existing special-purpose transformation languages, which rely on template- or pattern-based rewrite rules to support definition of new transformations.

Parameterization of transformation scripts  As illustrated by lines 15-17 of Figure 2, each POET script can define a large collection of command-line parameters to dynamically modify its behavior. A single POET script therefore can be used to dynamically produce a wide variety of different output, effectively allowing different software implementations be manufactured on demand based on different feature requirements.

Composition and Tracing of Transformations  Most compilers and domain-specific code generation tools need to apply a long sequence of different transformations to an input code, with each transformation controlled by various parameters. It is extremely difficult to manually trace the combined transformation result as each transformation could be applied in dramatically different ways. POET provides built-in support to automatically resolve this issue via a collection of trace handles, which can be used to automatically keep track of various code fragments as the input code goes through different transformations. The tracing capability makes the composition of different transformations extremely flexible, where transformation orders can be easily adjusted or even dynamically tuned [55].

Flexibility and ease of use  A POET program can include an arbitrary number of different files, each of which consists of a sequence of global declarations and commands. The declarations serve to specify attributes of special-purpose global names, e.g., variables, code templates, and xform routines, whose scopes span across different POET files. The static and local variables do not need to be declared before used, and their scopes are limited within the files or the code template/xform routines that contain them. All variables can dynamically hold arbitrary types of values. An extensive collection of built-in operations are provided to easily construct, analyze, and modify internal representations of different programming languages to satisfy the development needs of building customized program transformations and domain-specific code generators.

3  Dynamically Parsing Arbitrary Languages

A key language feature of POET is the ability to dynamically parse an arbitrary programming language using a single input command, where both the concrete syntax and the internal representation of the input language are collectively specified using a collection of code templates defined in an external file. These code template specifications are interpreted and matched against the input code in a top-down recursive descent fashion at runtime, and an abstract syntax tree (AST) representation of the input code is automatically constructed as result of the input command. This approach is more flexible than the conventional parser generation
Components of a code template declaration and their meanings

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template body</td>
<td>Concrete syntax of the code template for parsing and unparsing</td>
</tr>
<tr>
<td>pars=(v1;p1,...,vm;p_m,)</td>
<td>Template parameters which specify data fields of the corresponding internal representation</td>
</tr>
<tr>
<td>parse=p1</td>
<td>Use p1 as the alternative concrete syntax to substitute the template body for parsing</td>
</tr>
<tr>
<td>output=p1</td>
<td>Use p1 as the alternative concrete syntax for unparsing of the code template</td>
</tr>
<tr>
<td>lookahead=n</td>
<td>Examine the n leading input tokens when using the code template for parsing (by default, n=1)</td>
</tr>
<tr>
<td>v=INHERIT</td>
<td>Use expression c as the alternative return result after successful parsing using the code template</td>
</tr>
<tr>
<td>match=c</td>
<td>Allow c to be used in place of the code template when matching against the input code</td>
</tr>
</tbody>
</table>

Type specifiers for tokens and compound data objects

<table>
<thead>
<tr>
<th>Type specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>The integer type, which includes all integer values</td>
</tr>
<tr>
<td>STRING</td>
<td>The string type, which includes all string values</td>
</tr>
<tr>
<td>ID</td>
<td>The identifier type, which includes all string values that can be used as identifiers in POET</td>
</tr>
<tr>
<td>CODE</td>
<td>The code template type, which includes all code template objects</td>
</tr>
<tr>
<td>XFORM</td>
<td>The xform handle type, which include all xform routine handles</td>
</tr>
<tr>
<td>TUPLE</td>
<td>The tuple type, which includes all POET tuples</td>
</tr>
<tr>
<td>MAP</td>
<td>The associative map type, which includes all POET associative maps</td>
</tr>
<tr>
<td>-</td>
<td>The ANY type, which includes all values supported by POET</td>
</tr>
<tr>
<td>lb ... ub</td>
<td>The range type, which includes all integers &gt;= lb and &lt;= ub</td>
</tr>
</tbody>
</table>

Parsing specifiers: matching concrete syntax specifications with leading input tokens

<table>
<thead>
<tr>
<th>Parsing specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a constant value</td>
<td>Match the first token with the given value, e.g. 3, 5, 137, &quot;abc&quot;, &quot;de3f7&quot;</td>
</tr>
<tr>
<td>a token type specifier</td>
<td>Match the first token with the type specifier</td>
</tr>
<tr>
<td>&quot;p</td>
<td>Match the first token with anything but the parsing specifier p</td>
</tr>
<tr>
<td>a code template name</td>
<td>Parse leading tokens using the corresponding code template</td>
</tr>
<tr>
<td>EXP</td>
<td>Parse leading tokens as an expression, using the built-in expression parser within POET</td>
</tr>
<tr>
<td>v = p</td>
<td>Parse the leading tokens using parsing specifier p, then save the parsing result to variable v</td>
</tr>
<tr>
<td>p ...</td>
<td>Parse the leading tokens as a list of 0 or more components, each parsed using parsing specifier p</td>
</tr>
<tr>
<td>LIST(p,s)</td>
<td>Parse the leading tokens as a list of 1 or more components, each parsed using parsing specifier p</td>
</tr>
<tr>
<td>TUPLE(p)</td>
<td>Parse the leading input tokens using parsing specifier p, then return a tuple as result</td>
</tr>
<tr>
<td>p1 p2 ... pm</td>
<td>Parse the leading input tokens using m consecutive parsing specifiers p1, ..., pm</td>
</tr>
<tr>
<td>p1</td>
<td>p2</td>
</tr>
</tbody>
</table>

Table 2: POET support for specifying syntax of arbitrary languages

The drawback of the dynamic parsing approach is its runtime overhead, where the interpretation of code template specifications can significantly slow down a POET transformation engine. However, when used to support programmable compiler optimization and to quickly build ad-hoc source-to-source translators, the overhead of interpreting transformations to an input code typically outweighs that of parsing the same code. Note that irrelevant fragments of the input code do not need to be parsed, so the parsing overhead applies only to portions of the input code that need to be analyzed or transformed. POET is designed to be an interpreted language and therefore values flexibility and convenience over the runtime cost.

In the following, Section 3.1 presents how to use POET code templates to collectively specify the syntax and internal representation of an arbitrary language. Section 3.2 presents annotations that can be inserted within an input code to provide additional information for parsing. Section 3.3 presents our dynamic parsing algorithm. Section 3.4 presents POET macros that can be used to dynamically modify behavior of the dynamic parser.
3.1 Specifying Syntax Using Code Templates

POET uses a group of code templates to specify both the concrete syntax and the internal representation (i.e., abstract syntax) of an arbitrary programming language. These code templates are used in the parsing phase to recognize the structure of an input code, in the program analysis/transformation phase to represent the internal data structures, and in the unparsing phase to output results to external files. Table 2 shows the meaning of various code template components as they are used in parsing, unparsing, and IR construction.

Figure 3 illustrates the correlation between syntax specifications using Backus-Naur form (BNF) vs. using POET code templates. In particular, each BNF production $A : \beta$ is translated to a single POET code template definition in the format of `<code A ...> \beta </code>`, where the template name corresponds to the left-hand non-terminal $A$, and the template body corresponds to the right-hand side $\beta$. A template parameter in the format of $a:t$ is created for each non-constant symbol $t$ within $\beta$, where $a$ specifies the name of the data field to store the value of $t$. The parameter name $a$ (surrounded by a pair of @’s) is then used to substitute the original BNF symbol $t$ in the template body. For example, the Ctrl symbol at line 1 of Figure 3(a) is translated to template parameter `ctrl:CODE.Ctrl` in (b), where
CODE.Ctrl declares Ctrl as a new POET code template name that will be defined later. The reserved token, ‘@’, is used for context switching between POET and source strings of the input language within code template bodies.

In summary, each POET code template uses parsing specifiers in the body, parameters, and the parse and output attributes of the code template to recursively define the concrete syntax specified by a BNF production. Table 2 shows the different formats of parsing specifiers supported by POET. After parsing, an internal representation of the input code is automatically constructed and saved into a variable specified within the input command.

The structure of the AST (abstract syntax tree) to represent an input code after parsing is specified using code template parameters. In particular, each code template is a unique user-defined compound data type, where the template parameters are data fields within the data structure. For example, the code template at line 1 of Figure 3(b) is conceptually equivalent to the type definition `struct Nest {Ctrl* ctrl; SingleStmt* body;};` in the C programming language. By default, a code template object is automatically constructed by the POET dynamic parser after using each code template to successfully parse a fragment of the input code. The resulting code template object is then used as the parameter value of a parent code template, and eventually an AST is built in a bottom-up fashion as the result of parsing the entire input code. For example, Figure 4 shows the resulting AST built after using the code templates in Figure 3 to parse an input code fragment at line 6 of Figure 5.

The default AST construction for each code template can be reconfigured using the rebuild attribute shown in Table 2. For example, line 5 of Figure 3(b) specifies that after using the For code template to parse an input code fragment, the parser should invoke the RebuildLoop routine with the respective parameters to generate the parsing result. The POET dynamic parser also uses a special keyword, INHERIT, to provide limited support for inherited attributes during AST construction. For example, line 6 of Figure 3(b) specifies that the previous code template object constructed (i.e., the true-branch of an if-conditional) should be saved in the ifNest local variable of an Else code template object. After the AST is properly constructed, sophisticated program analysis and transformation can then be applied to the internal representation without being restricted by the parsing process.
void dgemm_test(const int M, const int N, const int K,
    const double alpha, const double *A, const int lda,
    const double *B, const int ldb, const double beta,
    double *C, const int ldc) {
    int i, j, l; //=>gemmDecl=Stmt
    for (j = 0; j < N; j += 1) //=> BEGIN(gemmBody=Nest) BEGIN(nest3=Nest)
        { //=> BEGIN(nest2=Nest)
            for (i = 0; i < M; i += 1) //=> BEGIN(nest1=Nest)
                {
                    C[j*ldc+i] = beta * C[j*ldc+i];
                    for (l = 0; l < K; l +=1)
                        C[j*ldc+i] += alpha * A[l*lda+i]*B[j*ldb+l];
                }
        }
}

Figure 5: An example POET input code with embedded annotations

3.2 Annotating the Input Code

The POET dynamic parser accepts annotations embedded within an input code and uses the additional information to guide the parsing of various code fragments. As illustrated by Figure 5, each POET annotation either starts with "//@" and lasts until the line break, or starts with "/*@" and ends with "@*/". These annotations can be naturally treated as comments in C/C++/Java code and can be embedded inside the comments of other languages such as Fortran/Cobol. POET currently supports the following two types of parsing annotations, each annotation specifying which code template should be used to parse a particular code fragment and which global variable to save the parsing result.

- Single-line annotations, each of which applies to a single line of program source and has the format => T, where T is a parsing specifier defined in Table 2. For example, line 5 of Figure 5 indicates that the source code should be parsed using the Stmt code template, and the parsing result should be stored in the global variable gemmDecl.

- Multi-line annotations, each of which applies to more than one line of program source and has the format BEGIN(T), where T is a parsing specifier. For example, line 6 of Figure 5 indicates that the code template Nest should be used to parse the code fragment starting from the for loop and lasting until Nest has been fully matched (i.e., until line 16). Further, the parsing result should be saved to the nest3 and gemmBody global variables. Similarly, the other multi-line annotations in Figure 5 define values of the global variables gemm (lines 1-17), nest2 (lines 8-15), and nest1 (lines 11-14).

3.3 The Parsing Algorithm

Figure 6 shows the algorithm implemented by POET to dynamically parse an arbitrary input language. Compared to conventional predictive recursive descent parsers, the main difference here is that the syntax of the input language is interpreted, i.e. dynamically matched against
**Figure 6: The dynamic recursive descent parsing algorithm in POET**

a stream of input tokens at runtime. Therefore, the POET dynamic parser can be used to process arbitrary programming languages based on varying syntax descriptions instead of being dedicated to any statically defined input language.

The *parse* algorithm in Figure 6 takes three parameters: *tokens*, the input token stream generated from an internal tokenizer; *goal*, the top-level parsing specifier to match the input tokens; and *inherit*, the inherited attribute for the current parsing specifier (i.e., the result of matching the previous parsing specifier). The algorithm proceeds by dynamically matching the leading tokens of the input stream against the targeting parsing specifier. If the parsing is successful, it returns the internal representation of the parsed code and modifies the input stream to contain the rest of unmatched tokens; otherwise, an exception is raised to report the location within the input stream where an error has occurred.

The algorithm first extracts the leading token (*tok1*) from the input stream and examines *tok1* to process all input annotations. Each input annotation associates a parsing specifier with a particular fragment of the input code and is treated as a special token by the POET
tokenizer. Each annotation is categorized as either single-line (illustrated at line 5 of Figure 5), where a complete fragment of the input code is associated with a parsing specifier, or multi-line (illustrated at lines 6, 8, and 11 of Figure 5), where the start of a fragment is indicated to match against a parsing specifier without indicating the end of the fragment. The algorithm processes each single-line annotation simply by recursively invoking itself with the appropriate annotated information. To process a multi-line annotation, it first prepends the annotated code fragment (i.e., the beginning portion of the relevant input code) to the rest of the input stream and then proceeds to match the new stream against the annotated parsing specifier. For both single-line and multi-line annotations, the parsing result is then used as the new leading input token, and the original input stream is modified accordingly.

After step (1) of the algorithm, the value of $tok_1$ could be a single input token (e.g., a string or an integer) or a code template object which is the result of processing an input annotation. Step (2) of the algorithm then continues by matching the top-level parsing specifier $goal$ with $tok_1$ followed by the rest of the input stream. In particular, the algorithm separately considers the following alternative forms that $goal$ could take.

(2.1) $goal$ is a constant value (i.e., a single integer or a string literal). If $goal$ is an empty string, the parsing succeeds without consuming any input tokens, and the empty string is returned as result; otherwise, the parsing succeeds (in which case, $tok_1$ is removed from tokens) if and only if the value of $goal$ matches that of $tok_1$.

(2.2) $goal$ is the name of a code template. If $tok_1$ is already an object of the given code template (the result of processing input annotations), the object is returned as result after removing $tok_1$ from the input stream; otherwise, the input stream is matched against the syntax definition (i.e., the parse attribute or template body) of the given code template, and if the matching is successful, an object of the given code template is created based on values of the template parameters saved during the parsing process (using a temporary symbol table created before parsing).

(2.3) $goal$ is a template parameter. Here the parsing specifier of the template parameter $(var\_constr(goal))$ is used as target to recursively invoke the parse algorithm, and if the parsing succeeds, the result is saved as the value of the template parameter to be later used to build an object of the corresponding code template.

(2.4) $goal$ is a token type specifier such as INT, STRING, ID. Here $tok_1$ is compared with the given type specifier and returned as the parsing result if the matching succeeds.

(2.5) $goal$ is a built-in operator such as assignment, EXP, LIST, and TUPLE, shown in Table 2. To process a variable assignment in the format of $v = p$, the input is parsed using the given parsing specifier $p$, and the parsing result is saved as the value of given variable $v$. To process the EXP specifier, the built-in expression parser is invoked to automatically recognize user-defined operations. The LIST and TUPLE operators are similarly processed by invoking their built-in parsing support.

(2.6) $goal$ is a list of parsing specifiers. Here the algorithm proceeds to match the input tokens with the given sequence of parsing specifiers one after another, and the parsing result for each specifier is concatenated at the end of the resulting list. Note that each parsing result is used as the inherited attribute for parsing the following specifier, which is consistent with the meaning of the inherit parameter for the parse algorithm.
Macros for reconfiguring behavior of the input command

<table>
<thead>
<tr>
<th>Macros</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOKEN</td>
<td>Reconfigure the POET internal tokenizer to treat a list of parsing specifiers as single tokens</td>
</tr>
<tr>
<td>KEYWORDS</td>
<td>Reconfigure POET dynamic parser to treat a list of string literals as keywords of the input language</td>
</tr>
<tr>
<td>PREP</td>
<td>Reconfigure POET dynamic parser to invoke a xform handle to filter the input tokens before parsing</td>
</tr>
<tr>
<td>BACKTRACK</td>
<td>Reconfigure POET dynamic parser to enable/disable backtracking during parsing</td>
</tr>
<tr>
<td>PARSE</td>
<td>Reconfigure POET dynamic parser to use a given parsing specifier as goal to parse all input code</td>
</tr>
</tbody>
</table>

Macros for reconfiguring behavior of the output command

<table>
<thead>
<tr>
<th>Macros</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNPARSE</td>
<td>Reconfigure the POET unparser to invoke a given xform handle to post-process (reformat) output tokens</td>
</tr>
</tbody>
</table>

Macros for reconfiguring the internal expression parser (i.e., the support of the EXP parsing specifier)

<table>
<thead>
<tr>
<th>Macros</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP_BASE</td>
<td>Use a given parsing specifier for base cases of expressions</td>
</tr>
<tr>
<td>EXP_BOP</td>
<td>Accept a given list of binary operators (in increasing order of precedence) within expressions</td>
</tr>
<tr>
<td>EXP_UOP</td>
<td>Use a given list of unary operators within expressions</td>
</tr>
<tr>
<td>PARSE_CALL</td>
<td>Use a given code template as the internal representation of a function call when parsing expressions</td>
</tr>
<tr>
<td>PARSE_ARRAY</td>
<td>Use a given code template as the internal representation of an array access operation</td>
</tr>
<tr>
<td>PARSE_BOP</td>
<td>Use a given code template as the internal representation of all binary operators</td>
</tr>
<tr>
<td>PARSE_UOP</td>
<td>Use a given code template as the internal representation of all unary operators</td>
</tr>
<tr>
<td>BUILD_BOP</td>
<td>Invoke a given xform handle to rebuild internal representations of binary operators</td>
</tr>
<tr>
<td>BUILD_UOP</td>
<td>Invoke a given xform handle to rebuild internal representations of unary operators</td>
</tr>
</tbody>
</table>

Table 3: Macros that can be used to reconfigure the default behavior of POET

(2.7) *goal* is the alternative (|) operator. Here the algorithm examines each of the alternative parsing specifiers in turn and uses the leading input token *tok*1 to predictively determine which specifier to use. If any of the alternative specifiers can potentially match *tok*1 as the first input token, the specifier is used to recursively invoke the *parse* algorithm, and the parsing result is returned as the result of the whole parsing process; otherwise (none of the matching succeeds), an error is reported.

3.4 Modifying The Default Parsing Behavior

POET provides three categories of built-in macros, shown in Table 3, to modify behavior of the POET interpreter when evaluating the input and output commands. The goal is to enable developers to easily adapt POET to conveniently support the parsing/unparsing needs of a wide variety of different programming languages.

Figure 7 shows examples of redefining these POET macros to support the C and Fortran programming languages. In particular, the TOKEN and KEYWORDS macros at lines 1-2 of Figure 7 are used to reconfigure the POET tokenizer. The PARSE macro at line 3 specifies the top-level parsing specifier that should be used to parse an input programming language. The BACKTRACK macro at line 4 controls the tradeoffs between developer productivity and parsing performance. The PREP and UNPARSE macros at lines 5-6 are designed to accommodate peculiar programming languages such as Fortran and Cobol which treat tokens differently based on their column locations within an input file. The large number of expression macros at lines 7-12 are used to easily adapt the POET built-in support for the EXP parsing specifier, which can automatically recognize reconfigurable binary/unary operations, function calls, and array accesses within expressions. If the expression of a language cannot be fully specified using these operators, the additional syntax can be specified as part of the EXP_BASE macro, which can include arbitrary code templates as components and can recursively invoke the EXP parsing specifier if necessary\(^2\). Developers can also provide

\(^2\)Note that left-recursion is not allowed, as required by all top-down predictive parsers
Figure 7: Example: using macros to reconfigure the default behavior of POET

their own expression parsers defined via code templates and can freely combine these code
templates with the built-in EXP specifier.

4 Analyzing the Input Code

The POET language currently places more emphasis on supporting program transformations, discussed in Section 5, than supporting sophisticated program analysis such as iterative data-flow analysis, dependence analysis, and pointer aliasing analysis commonly implemented in full-blown optimizing compilers [37]. Being an interpreted transformation language, POET is not intended as a language of choice for implementing complex program analysis algorithms, e.g., context-sensitive pointer and shape analysis, which are considered overly expensive even when implemented using fully compiled languages such as C/C++. However, it is within our future work to extend the POET language with built-in support for various program analysis capabilities implemented using C++ within the POET interpreter.

The existing program analysis support within POET focuses on strong programming support for conveniently navigating and collecting information from the internal representation of an input code, through flexible pattern matching operations and traversal of the AST (abstract syntax tree), shown in Table 4 and discussed in Sections 4.1 and 4.2 respectively.
Pattern specifier | Values matching the pattern
---|---
an expression $p$ | The result of evaluating $p$
a code template name | Objects of the given code template type
a token type specifier | Values matching the given type specifier (see Table 2)
VAR | Trace handles which can be embedded within POET expressions
an xform handle $f$ | All values s.t. when used as parameters to invoke $f$, the invocation returns TRUE
an uninitialized variable $v$ | All POET values; variable $v$ is assigned with the value after matching
$c \# p$ | All values that can match pattern specifier $p$; variable $v$ is assigned with the value after matching
$(p_1, p_2, ..., p_n)$ | All tuples of $n$ elements which match the pattern specifiers $p_1, p_2, ..., p_n$ respectively
$(p_1 p_2 ... p_n)$ | All lists of $n$ elements which match the pattern specifiers $p_1, p_2, ..., p_n$ respectively
$p_1 :: p_2$ | All lists with the first element matching pattern $p_1$ and rest of the list matching pattern $p_2$
$p_1 \ op \ p_2$ | All expressions built using the given binary $op$ (e.g., $+$, $*$) to combine patterns $p_1$ and $p_2$
$p_1 \ | \ p_2 \ | ... \ | p_n$ | All values that can match one of the pattern specifiers $p_1, p_2, ..., p_n$

Pattern matching operators:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
</table>
| $a : b$ | Return whether value $a$ matches the pattern specifier $b$
| switch (a) { case b1: ... case bn: ... default: ... } | Match value $a$ against pattern specifiers $b_1, ..., b_n$ in turn, evaluate the matching branch; if all matches fail, evaluate the default branch.

AST traversal operators:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
</table>
| foreach(a : b : c) { d } | Traverse and match all values embedded within $a$ against pattern specifier $b$; for each value $x$ that can successfully match $b$, evaluate expressions $d$ and then $c$; if $c$ evaluates to true, skip the inside of $x$ and continue; otherwise, continue traversing inside $x$ to find more matches.
| foreach_r(a : b : c) { d } | Same as the foreach operator, except that values within $a$ are traversed in reverse order

Table 4: POET support for pattern matching and AST traversal

<table>
<thead>
<tr>
<th>Pattern specifier</th>
<th>Values matching the pattern</th>
</tr>
</thead>
</table>
| an expression $p$ | The result of evaluating $p$
| a code template name | Objects of the given code template type
| a token type specifier | Values matching the given type specifier (see Table 2)
| VAR | Trace handles which can be embedded within POET expressions
| an xform handle $f$ | All values s.t. when used as parameters to invoke $f$, the invocation returns TRUE
| an uninitialized variable $v$ | All POET values; variable $v$ is assigned with the value after matching
| $c \# p$ | All values that can match pattern specifier $p$; variable $v$ is assigned with the value after matching
| $(p_1, p_2, ..., p_n)$ | All tuples of $n$ elements which match the pattern specifiers $p_1, p_2, ..., p_n$ respectively
| $(p_1 p_2 ... p_n)$ | All lists of $n$ elements which match the pattern specifiers $p_1, p_2, ..., p_n$ respectively
| $p_1 :: p_2$ | All lists with the first element matching pattern $p_1$ and rest of the list matching pattern $p_2$
| $p_1 \ op \ p_2$ | All expressions built using the given binary $op$ (e.g., $+$, $*$) to combine patterns $p_1$ and $p_2$
| $p_1 \ | \ p_2 \ | ... \ | p_n$ | All values that can match one of the pattern specifiers $p_1, p_2, ..., p_n$

4.1 Dynamic Pattern Matching

The most common operation on the internal representation (i.e., AST) of an input code is examining each node within the AST and performing different operations accordingly. POET provides powerful pattern matching support to conveniently decompose the structure of each AST node, illustrated by the $xform$ routine in Figure 8 which uses pattern matching to recursively check the type consistency of an simple expression. Table 4 shows the varying forms of pattern specifiers supported by POET.

POET provides two pattern matching operators, the $switch$ operator and the $:$ operator, to dynamically test the type and structure of an arbitrary unknown value. For example, the $TypeCheckExp$ routine in Figure 8 uses the $switch$ operator to match the input parameter $exp$ against three pattern specifiers within the $case$ labels at lines 4, 12, and 13 respectively. Each specifier is matched in their order of appearance, and if successful, the statements
following the corresponding case label are evaluated, and the evaluation result is returned as result of the whole switch statement. Note that once a case label is successfully matched, the rest of the labels are simply ignored. So each switch operator is essentially a sequence of if-else branches. At lines 7 and 8 of Figure 8, the : operator is used to match type1 and type2 against different pattern specifiers. Each operation returns TRUE (integer 1) if the matching is successful, and returns FALSE (integer 0) otherwise.

Note that when uninitialized variables appear in a pattern specifier, these variables are treated as place holders which can be matched to arbitrary values. If the overall matching is successful, all the uninitialized variables are assigned with their matching values as part of the evaluation. For example, the pattern specifier at line 4 of Figure 8 includes two uninitialized variables, exp1 and exp2, so if the pattern matching succeeds, exp1 and exp2 will be assigned with the second and third parameters of the Bop code template object respectively. Therefore the pattern matching operations can be used not only for dynamic type checking, but also for initializing and assigning values to variables.

### 4.2 Traversing the AST

When examining the internal representation of an input code, developers frequently need to traverse an entire AST searching for specific code patterns. POET provides two operators, foreach and foreach_r, for this purpose. As shown in Table 4, both operators collectively apply pattern matching to the entire AST representation of an input computation.

As example, the xform routine in Figure 9 uses the foreach operator to identify all basic blocks from an input code that contains only expression statements and loops. Note that in order to process each code fragment that matches a given pattern, the pattern specifier needs to contain assignments or uninitialized local variables to save the matching fragments. For example, lines 6 of Figure 9 traverses the input code to find all loop nests, each of which
1: <code>GraphEdge pars=(from,to) "@{(from)}@"->{$(to)@}" </code>

2: <xform BuildCFG pars=(input, blocks) prev="" edges="" >
3: cur=blocks[input];
4: if (cur != "") <+ the current input code belongs to a new basic block>
5: {
6:   if (prev != "") { edges = GraphEdge#(prev,cur) :: edges; } <+ add CFG edge
7:     prev=cur; <+ save the previous basic block
8: }
9: switch (input) {
10:   case (first second): <+ input is a list of statements *
11:     (e1, b1) = BuildCFG(first,blocks);
12:     if (second != "") { BuildCFG[prev=b1;edges=e1](second,blocks)}
13:   else { (e1,b1) }
14:   case Nest#(loop, body): <+ input is a loop nest *
15:     (e1,b1) = BuildCFG(body, blocks);
16:     (GraphEdge#(b1,cur)::e1, b1) <+ add the loop back edge
17:   case ExpStmt: (edges, prev)
18:   default: <+ input could be a function declaration or an empty string *
19:     foreach (input : (cur=_): FALSE) {
20:       if (HEAD(cur) : ExpStmt|Nest) { RETURN (BuildCFG[prev=""](cur,blocks)); } <+ look inside
21:     }
22:     (edges,prev) <+ no need to look inside
23: }
24: </xform>
25:<parameter inputFile default="" message="input file name"/>
26:<parameter outputFile default="" message="output file name"/>
27:<parameter inputLang default="" message="file name for input language syntax" />
28:<input from=(inputFile) syntax=(inputLang) to=inputCode/>
29:<eval blocks=BuildBasicBlocks(inputCode); (cfg,_) = BuildCFG(inputCode, blocks); />
30:<output syntax=(inputLang) to=outputFile from=(StmtList#cfg) />

Figure 10: Example: control-flow analysis for a simple loop-based language

is first assigned to the local variable cur (with the corresponding loop and body saved to local variables loop and body respectively) and then used to evaluate the foreach loop body. Similarly, line 9 of Figure 9 traverses the input code to process each AST node in pre-order. Both loops at lines 6 and 9 set the third foreach parameter to FALSE, which indicates that after processing each matching code fragment, the pattern matching process should continue by traversing inside the matched fragment. The foreach_r loop essentially has the same semantics as that of foreach, except that it traverses the input code in reverse pre-order (i.e., the opposite order used by the foreach loop).

4.3 Example: Simple Program Analysis

POET can be used to easily implement straightforward program analysis algorithms such as type checking, where a type is automatically determined for each expression within an input code; and control flow analysis, where a graph is constructed to model the control flow between instructions of an input code. A simplified implementation of type checking is shown in Figure 8. Figure 10 presents an implementation of the control-flow analysis, which uses the BuildBasicBlocks routine in Figure 9.

The executable commands of the POET script in Figure 10 start at line 28, which reads the input code from an external file using a given language syntax description file. Line 29 then
analyzes the input code by first invoking the \texttt{BuildBasicBlocks} routine, defined in Figure 9, to identify all basic blocks (i.e., single-entry-single-exit sequences of statements) and then invoking the \texttt{BuildCFG} routine, defined in Figure 10, to connect the identified basic blocks with control flow edges. Finally, the control flow graph is output to an external file at line 30.

In Figure 9, the POET script for identifying basic blocks starts by declaring a code template type \texttt{BasicBlock} (at lines 1-3) to store the analysis result. Lines 4-21 then define the \texttt{BuildBasicBlocks} routine which takes a single input code as parameter and returns an associative table which maps each statement that should start a new basic block with the corresponding block of statements. In Figure 10, line 1 defines a code template type \texttt{GraphEdge} to support the construction and output of the control-flow graph. Lines 2-23 then define the \texttt{BuildCFG} routine which takes the input code together the collection of identified basic blocks and returns a tuple of two components: the list of control flow edges that connect the basic blocks, and the last basic block encountered from traversing the input code.

As illustrated by the \texttt{BuildBasicBlock} routine in Figure 9 and by the \texttt{BuildCFG} routine in Figure 10, POET provides two ways to traverse an input AST: using the \texttt{foreach} or \texttt{foreach_r} operators, and using recursive invocations of AST visiting functions. While the \texttt{foreach} and \texttt{foreach_r} operators provide convenient ways of skipping AST nodes that are irrelevant to the desired solution, the recursive invocation of visiting functions is more powerful and supports implementation of arbitrary complex divide-and-conquer algorithms.

As shown in both Figures 9 and 10, the code templates in POET can be used to easily build complex data structures, e.g., the \texttt{BasicBlock} and \texttt{GraphEdge} types. However, to avoid infinite recursion when traversing ASTs built via code template objects, POET disallow using code templates to build cyclic data structures. Specifically, associative map is the only compound data type whose components can be modified after initial construction in POET. Therefore, associative maps are required to build and navigate a cyclic data structure. For example, to quickly find the successors of an arbitrary basic block, an associative table can be constructed to quickly map each basic block to a list of its successors.

Since POET supports the implementation of complex data structures and arbitrary divide-and-conquer algorithms, it can be used to implement sophisticated program analysis algorithms such as iterative data-flow analysis, where each basic block needs to be associated with a set of information (e.g., a set of expressions or variables). These sets of information can be implemented using either the built-in compound type \texttt{list} or using the associative map in POET. In particular, the associative map can be used to easily support set intersection, union, and subtraction, which are required for solving most data-flow analysis problems. The main difference between using POET to implement data-flow analysis algorithms vs. using a more conventional compiler writing language such as C/C++ is the efficiency of implementation. We are looking to provide built-in support for these analysis problems by internally implementing them within the POET interpreter using C/C++ in the future.

5 Supporting Program Transformations

A key emphasis of the POET language is to support easy construction and composition of parameterized source-to-source program transformations so that the performance of differently optimized code can be automatically tuned on varying architectures. In the following,
Transformation Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLACE(c1,c2,e)</td>
<td>Replace all occurrences of c1 with c2 in AST e</td>
</tr>
<tr>
<td>REPLACE((a1,r1)...(an,rn), e)</td>
<td>Traverse AST e in pre-order to locate and replace each ai (i=1,...,n) with ri</td>
</tr>
<tr>
<td>REBUILD(e)</td>
<td>Rebuild AST e by invoking the rebuild attribute of each code template object inside e</td>
</tr>
<tr>
<td>DUPLICATE(c1,c2,e)</td>
<td>Replicates AST e with copies, each copy replacing c1 by a different component in c2</td>
</tr>
<tr>
<td>PERMUTE((i1,i2,...,im),e)</td>
<td>Reorder list e so the jth (j=1,...,m) element is located at i in the result</td>
</tr>
</tbody>
</table>

Tracing operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT (x,e)</td>
<td>Insert all the trace handles rooted at x to be embedded within AST e</td>
</tr>
<tr>
<td>ERASE (x,e)</td>
<td>Remove all the occurrences of trace handle x from the input AST e</td>
</tr>
<tr>
<td>TRACE((x1,...,xm),e)</td>
<td>Treat variables x1,...,xm as trace handles during the evaluation of expression e</td>
</tr>
<tr>
<td>SAVE(v1,v2,...,vm)</td>
<td>Save the current values of trace handles v1,v2,...,vm to be restored later</td>
</tr>
<tr>
<td>RESTORE(v1,v2,...,vm)</td>
<td>Restore the previous values saved for the trace handles v1,...,vm</td>
</tr>
</tbody>
</table>

Evaluation operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELAY {e}</td>
<td>Delay the evaluation of expression e until later</td>
</tr>
<tr>
<td>APPLY (e)</td>
<td>Force the evaluation of a delayed expression e</td>
</tr>
</tbody>
</table>

Table 5: Built-in POET operators for support program transformation

<table>
<thead>
<tr>
<th>Index</th>
<th>Expressions</th>
<th>Evaluation result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>REPLACE(&quot;x&quot;,&quot;y&quot;, SPLIT(&quot;&quot;,&quot;x*x-2&quot;))</td>
<td>&quot;y&quot; &quot;y&quot; &quot;y&quot; &quot;2&quot;</td>
</tr>
<tr>
<td>2</td>
<td>REPLACE( ((&quot;a&quot;,1) (&quot;b&quot;,2) (&quot;c&quot;,3)), SPLIT(&quot;&quot;,&quot;a+b-c&quot;))</td>
<td>1 &quot;+&quot; 2 &quot;+&quot; 3</td>
</tr>
<tr>
<td>3</td>
<td>REPLACE( ((&quot;a&quot;,1) (&quot;b&quot;,2) (&quot;c&quot;,3)), Bop#(&quot;+&quot;,&quot;a&quot;,&quot;Bop#(&quot;-&quot;,&quot;b&quot;,&quot;c&quot;)&quot;))</td>
<td>Bop#(&quot;+&quot;,1,Bop#(&quot;-&quot;,2,3))</td>
</tr>
<tr>
<td>4</td>
<td>REBUILD(Bop#(&quot;+&quot;, 0, Bop#(&quot;-&quot;,a,3)))</td>
<td>Bop#(&quot;-&quot;,a,2)</td>
</tr>
<tr>
<td>5</td>
<td>DUPLICATE(&quot;var&quot;, (1 2 3), Stmt#(&quot;var&quot;))</td>
<td>Stmt#1 Stmt#2 Stmt#3</td>
</tr>
<tr>
<td>6</td>
<td>PERMUTE((3 2 1), (&quot;a&quot; &quot;b&quot; &quot;c&quot;))</td>
<td>(&quot;c&quot; &quot;b&quot; &quot;a&quot;)</td>
</tr>
</tbody>
</table>

Table 6: Examples of invoking transformation operators

Sections 5.1, 5.2, and 5.3 discuss how to effectively use various POET built-in support, shown in Table 5, to build sophisticated program transformations. Section 5.4 illustrates how to build the top-level transformation scripts for optimizing known input programs.

5.1 Modifying the AST

POET provides four built-in operators, REPLACE, REBUILD, DUPLICATE, and PERMUTE, shown in Table 5 and illustrated in Table 6, to support the replacement, simplification, replication, and permutation of various code fragments within an input AST. Figure 11 uses several xform routines from the POET opt library to illustrate how to invoke these transformation operators to properly modify trace handles embedded inside an input AST. Section 5.2 explains details of these trace handle updates.

The REPLACE operator is invoked to replace various fragments of an input AST with new ones. It can be invoked using two different syntaxes, illustrated by entries 1-3 of Table 6. For example, the REPLACE invocation at line 13 of Figure 11 is used to replace all occurrences of the code fragment handlevalue with a new fragment newvalue in the AST rooted at trace, while the invocation at line 33 performs a sequence of one-time replacement operations (accumulated at lines 29-32) as it traverses the input AST in pre-order. The REPLACE operator can also be invoked to insert a new code fragment into an AST or to remove an existing code fragment. For example, the routine AppendDecl at lines 19-26 of Figure 11 illustrates how to append new variable declarations at the end of existing ones. To remove a code fragment x from the input code, one simply needs to replace x with the empty string.
Figure 11: Examples: modifying the AST

The **REBUILD** operator is invoked to simplify an input AST after it has been recently modified, e.g., with some fragments replaced with empty strings. In particular, when invoking **REBUILD** on an input AST, all the code template objects within the AST are traversed in post-order, and if a code template object has a pre-defined **rebuild** attribute (see Table 2), the **rebuild** expression is invoked, and the rebuilding result is used to substitute the original code template object. For example, entry 4 of Table 6 shows that when applied to a symbolic expression composed of constant numbers, the **REBUILD** operator can be used to evaluate the expression and return the evaluation result. Of course, the effectiveness of the expression evaluation depends on details of the **rebuild** attribute defined within the **Bop** code template, a type defined in our POET opt library.

The **DUPLICATE** and **PERMUTE** operators are provided to support special needs of replicating and reordering fragments of an input code. They are used to implement the loop unrolling and interchange transformations shown in Table 7.
5.2 Tracing of Transformed Code

POET uses a special concept called trace handles to automatically keep track of various fragments of an input code as they go through different transformations. These trace handles can be embedded inside an input AST and therefore be modified within xform routines even if the routines cannot directly access them through their names. In particular, xform routines can invoke the built-in transformation operators shown in Table 5 to effectively keep all embedded trace handles up-to-date. POET dedicates six built-in operators, INSERT, ERASE, COPY, TRACE, SAVE, and RESTORE, shown in Table 5, to properly set up and maintain all trace handles. The tracing capability enables different transformations to the same code to be naturally composed and their ordering flexible and easily adjustable, discussed in detail in Section 6.1.

All trace handles must be declared at the global scope before used, and they must be explicitly inserted inside an input AST using the INSERT operator to keep track of the transformed code. In order to successfully insert trace handles inside an AST, all handles must already contain correct fragments of the input code as their values. This condition is typically satisfied by using trace handles inside parsing annotations, e.g., the variables gemm, gemmDecl, gemmBody, nest3, nest2, and nest1 in Figure 5, to save the parsing result of special fragments within the input code. After all the trace handles have been assigned proper code fragments, they can be collectively embedded inside the input code using a single INSERT operator, illustrated at line 14 of Figure 12 and discussed in Section 5.4.

Since embedded trace handles can be modified by the transformation operators shown in Table 5, special care must be taken to ensure that their new values do not contain the original trace handles as components; otherwise, after modifying the trace handles, cycles will be created inside the resulting AST and will incur a runtime error whenever the AST needs to be traversed. The EraseTraceHandle routine at lines 2-7 of Figure 11 illustrates how to properly remove all nested trace handles from an input AST by invoking the ERASE operator. In particular, ERASE(handlevalue, handlevalue) at line 3 peels off the outermost trace handle contained in handlevalue by returning its value, and ERASE(handlevalue, exp) at line 3 removes the trace handle contained in handlevalue from exp and returns the new exp. The EraseTraceHandle routine is used by the ModifyTraceHandle routine at lines 8-18 to properly update a trace handle with a new value. Both are utility routines within the POET library. In contrast to the ERASE operator, which erases a single trace handle from an input code, the COPY operator can be invoked to erase all trace handles from an input AST and is used to generate independent copies of the original code.

By default, all POET xform routine parameters are passed-by-value so that routine invocations cannot have side effects except for modifying trace handles embedded inside values of the routine parameters. To change the default parameter passing strategy, the TRACE operator can be invoked to temporarily convert a list of static or local variables into trace handles during the evaluation of a single expression, e.g., the invocation of the A_ScalarRepl routine at line 9 of Figure 12, so that routines invoked within the expression can modify the converted static/local variables. After the TRACE evaluation, the static/local variables are automatically reverted back to their original states.

The SAVE and RESTORE operators are used together for saving and restoring information relevant to trace handles, and both return the empty string as result. For example, before
1: include opt.pi
2: <trace gemm,gemmDecl,gemmBody,nest3,nest2,nest1/> 
3: <input to="gemm" syntax="Cfront.code" from="gemm.c" /> 
4: < parameter mu type=1..MB default=4 
    message="Unroll and Jam factors for nest3" /> 
5: < parameter nu type=1..NB default=1 
    message="Unroll and Jam factors for nest2" /> 
6: ......
7: <eval nest3_UnrollJam = DELAY {
    UnrollJam[factor=(nu mu);trace=gemmBody](nest1,nest3);};
8: A_ScalarRepl=DELAY {
    TRACE(Arepl,
        ScalarRepl[init_loc=nest1[Nest.body]; elem_type=ftype;
            trace_repl=Arepl; trace_decl=gemmDecl; trace=nest2]
                ("a_buf",alphaA, LDA*I+L, dim, nest1[Nest.body]));}
9: ......
10: />
11: />
12: />
13: <eval
14: INSERT(gemm.gemm);
15: APPLY Specialize;
16: APPLY A_ScalarRepl;
17: APPLY nest3_UnrollJam;
18: APPLY B_ScalarRepl;
19: APPLY C_ScalarRepl;
20: APPLY array_ToPtrRef;
21: APPLY Abuf_SplitStmt;
22: APPLY body2_Vectorize;
23: APPLY array_FiniteDiff;
24: APPLY body2_Prefetch;
25: APPLY nest1_Unroll;
26: ) />
27: <output to="dgemm_kernel.c"
    from=gemm/>

(a) Transformation definitions
(b) Output definition

Figure 12: A POET script that optimizes the input code shown in Figure 5

applying transformations to an input code, the values of all embedded trace handles can
be saved using the SAVE operator. Then, after a sequence of transformations are finished
and the results output to external files, all trace handles can be restored with their original
values so that a new sequence of transformations can start afresh.

5.3 Delaying Evaluation of Expressions

POET provides two operators, DELAY and APPLY, to support the delay of expression
evaluations so that a block of expressions/statements can be saved to a variable to be used
later. After being saved into an arbitrary local/static variable, the delayed expression can
be later evaluated simply by invoking the APPLY operator with the local/static variable as
parameter. For example, the DELAY operator is used at lines 6-12 of Figure 12 to save all the
potential input code transformations into a list of static variables. Lines 15-25 then execute
these transformations by invoking the APPLY operator on the corresponding variables. Note
that the ordering of applying different transformations at lines 15-25 can be easily adjusted
by swapping the APPLY invocations, thereby allowing flexible composition and reordering
of transformations to the input code. The delayed expressions are in a way similar to xform
routines except they are defined and invoked using a different syntax, don’t have parameters,
and can directly operate on variables of the parent scope.

5.4 Building An Optimization Script

Figure 12 illustrates the typical structure of a POET transformation script for optimizing a
known input code, shown in Figure 5. This script serves as a structural guideline for automati-
cally applying parameterized compiler transformations to generate efficient implemen-
tations of an input code on varying architectures. To alternatively support domain-specific
code generation and translation, see the POET script in Figure 2.

The optimization script in Figure 12 starts by including the POET opt library, which
supports the large collection of source-to-source compiler transformations invoked by the script. It then declares all the trace handles which will be used to keep track of various input code fragments as they go through different transformations (line 2 of Figure 12). In particular, these trace handles are used inside the parsing annotations embedded within the input code in Figure 5 so that after parsing the input code using the `input` command (line 3 of Figure 12), all trace handles already contain correct values and can be properly inserted inside the parsing result contained in the `gemm` variable.

Note that when a tuple of trace handles are declared together, as illustrated at line 2 of Figure 12, they are assumed to be related, and their ordering in the declaration is assumed to be the same ordering that they should appear in a pre-order traversal of an AST. Subsequently, they can be inserted into the AST using a single `INSERT` operation, as illustrated by line 14 of Figure 12. Only related trace handles should be declared together in a single declaration, and unrelated trace handles should be separately declared.

Lines 4-12 of Figure 12 define all the transformations that can be applied to optimize the input code. In particular, lines 4-6 declare command-line parameters which will be used to extensively reconfigure program transformations, e.g., the `mu` and `nu` parameters are used to control the loop unroll&jam factors. Lines 6-12 define each transformation to the input code as a delayed invocation of a xform routine from the POET `opt` library, with each invocation parameterized by a number of command-line parameters declared at lines 4-6. Note that each transformation uses the pre-declared `trace handles` as input parameters so that it always operates on the correct code fragments no matter how many other transformations have already been applied, as all previous transformations have updated the trace handles properly after modifying the AST.

Lines 13-26 apply all the predefined transformations to the input code. In particular, line 14 inserts all the trace handles declared together with `gemm` to be embedded within the AST contained in `gemm`. Then, all the delayed transformations defined at lines 6-12 are applied one after another using the `APPLY` operator. Since all transformations operate on the trace handles independently, their composition is straightforward and the transformation ordering can be flexibly adjusted by simply swapping the `APPLY` invocations. The collection of delayed transformations can also be accumulated into a list and dynamically reordered using the `PERMUTE` operator (see Table 5) if necessary [55].

6 Use Case Studies

POET has been used to support two main software development needs: optimizing the performance of high performance computing applications so that efficient implementations can be automatically generated for varying architectures, and building source-to-source translators to support the code generation of various domain-specific languages. The following subsections discuss our experiences of the various use cases.

6.1 Programmable Compiler Optimization For Auto-Tuning

A primary design goal of the POET language is to enable programmable control and parameterization of compiler optimizations so that they can be automatically tuned to achieve
superior efficiency on varying architectures. Figure 13 shows the targeting optimization environment we are building using POET. In particular, an optimizing compiler, e.g., the analysis engine in Figure 13 which is based on the ROSE C/C++/Fortran source-to-source compiler [46], performs advanced optimization analysis to identify profitable program transformations and then produce output in POET so that architecture-sensitive optimizations are extensively parameterized. This POET output can then be ported to different machines together with the user application, where local POET transformation engines empirically reconfigure the parameterized optimizations until satisfactory performance is achieved. Computational specialists can modify the POET scripts to directly control the auto-generated compiler transformations and to add new domain-specific optimizations if necessary. Regular developers can use POET to obtain optimization feedbacks from compilers.

The following first summarizes our experiences in using POET to support the optimization environment in Figure 13 and then discusses the effectiveness of our optimization approach.

6.1.1 Using POET To Support Compiler Optimizations

As shown in Figure 13, POET can be used by developers to explicitly apply parameterized transformations to their code for the purpose of achieving portable high performance on varying architectures. Our previous work has manually developed POET optimization scripts for three dense linear algebra kernels and have achieved comparable performance as that achieved by manually written assembly code in the well-known ATLAS library [57]. A portion of the POET script for optimizing the matrix-matrix multiplication kernel is shown in Figure 12, where the input file is shown in Figure 5. We also used POET to manually optimize several SPEC95 benchmarks and studied the interactions between parallelization granularity and cache reuse [43]. Our recent work has extended the ROSE optimizing compiler [56] to automatically produce parameterized POET scripts for optimizing the three dense matrix kernels [53]. We have additionally developed an empirical search engine [28] which can automatically explore the configuration space of POET optimization scripts for varying architectures. The search engine is discussed in more detail in Section 6.2.2.

While using POET to optimize both performance-critical library kernels and full scientific applications, we have designed the language to be easy to use with an especially strong programming support for composing long sequences of parameterized compiler transformations without requiring any intermediate program analysis steps. The technical aspects of automatically generating the parameterized sequence of transformations are presented in [53] and are beyond the scope of this paper, which focuses on using POET to support such optimiza-
Parallelization and memory hierarchcy optimization

- **ParallelizeLoop**(x)**:** Parallelize the outermost loop using OpenMP in input code x
- **DistributeLoops**(n, x)**:** Distribute input code x so that fragments in n end up in separate components
- **FuseLoops**(n, p, x)**:** Fuse disjoint loops in n into a single one; then use it to replace fragment p in input code x
- **BlockLoops**(n,x)**:** Block the loops nested outside of fragment n but inside input code x
- **PermuteLoops**(n,x)**:** Permute the loops nested outside of fragment n but inside input code x
- **CopyRepl**(v, a, d, x)**:** Use buffer v to copy and replace memory referenced by a at loop iterations d in input code x
- **CleanupBlockedNests**(x)**:** Cleanup the blocked loop nests via loop splitting in input code x

Scalar and register performance optimization

- **UnrollLoops**(n,x)**:** Unroll the loops nested outside of fragment n and inside input code x
- **UnrollJam**(n,x)**:** Unroll the loops outside of fragment n and inside input code x; Jam the unrolled loops inside n
- **ScalarRepl**(v, a, d, x)**:** Use scalars named v to replace memory referenced by a at loop iterations d in input code x
- **FiniteDiff**(v,e,d,x)**:** Use loop induction variables v to reduce the cost of evaluating expression e + d in input code x
- **VectorizeCode**(v,n,x)**:** Apply SSE Vectorization to loop n inside input code x based on vector register assignment v
- **Prefetch**(a,n,i,x)**:** Prefetch memory address a with increment i at each iteration of loop n in input code x

High-level to low-level code translation

- **ArrayAccess2PtrRef**(x)**:** Convert all array references in input code x to pointer references
- **TransformThreeAddress**(x)**:** Transform input code x into three address code
- **TransformTwoAddress**(x)**:** Transform input code x into two address code

<table>
<thead>
<tr>
<th>Transformation Routines Supported by the POET Opt Library</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ParallelizeLoop</strong>(x)<strong>:</strong> Parallelize the outermost loop using OpenMP in input code x</td>
</tr>
<tr>
<td><strong>DistributeLoops</strong>(n, x)<strong>:</strong> Distribute input code x so that fragments in n end up in separate components</td>
</tr>
<tr>
<td><strong>FuseLoops</strong>(n, p, x)<strong>:</strong> Fuse disjoint loops in n into a single one; then use it to replace fragment p in input code x</td>
</tr>
<tr>
<td><strong>BlockLoops</strong>(n,x)<strong>:</strong> Block the loops nested outside of fragment n but inside input code x</td>
</tr>
<tr>
<td><strong>PermuteLoops</strong>(n,x)<strong>:</strong> Permute the loops nested outside of fragment n but inside input code x</td>
</tr>
<tr>
<td><strong>CopyRepl</strong>(v, a, d, x)<strong>:</strong> Use buffer v to copy and replace memory referenced by a at loop iterations d in input code x</td>
</tr>
<tr>
<td><strong>CleanupBlockedNests</strong>(x)<strong>:</strong> Cleanup the blocked loop nests via loop splitting in input code x</td>
</tr>
<tr>
<td><strong>UnrollLoops</strong>(n,x)<strong>:</strong> Unroll the loops nested outside of fragment n and inside input code x</td>
</tr>
<tr>
<td><strong>UnrollJam</strong>(n,x)<strong>:</strong> Unroll the loops outside of fragment n and inside input code x; Jam the unrolled loops inside n</td>
</tr>
<tr>
<td><strong>ScalarRepl</strong>(v, a, d, x)<strong>:</strong> Use scalars named v to replace memory referenced by a at loop iterations d in input code x</td>
</tr>
<tr>
<td><strong>FiniteDiff</strong>(v,e,d,x)<strong>:</strong> Use loop induction variables v to reduce the cost of evaluating expression e + d in input code x</td>
</tr>
<tr>
<td><strong>VectorizeCode</strong>(v,n,x)<strong>:</strong> Apply SSE Vectorization to loop n inside input code x based on vector register assignment v</td>
</tr>
<tr>
<td><strong>Prefetch</strong>(a,n,i,x)<strong>:</strong> Prefetch memory address a with increment i at each iteration of loop n in input code x</td>
</tr>
</tbody>
</table>

Table 7: Selected transformation routines supported by the POET opt library

...
optimize programs in C, C++, Fortran, or any other arbitrary language of interest, as long as an appropriate syntax description file has been created for the language.

**Partial parsing of the input code** While POET requires the syntax of input/output languages to be explicitly defined using code templates, only code fragments which are targets of program analysis or transformation need to be parsed. The other fragments which are not targets of optimization can simply be saved as lists of strings without being parsed into structured code templates. For example, when optimizing several of the SPEC95 Fortran benchmarks, we used POET to parse only a few loop nests that have been annotated similarly as Figure 5. The rest of the benchmarks are simply saved as strings. This support allows us to define POET syntax descriptions only for subsets of languages such as C, C++, and Fortran, without going through the time-consuming process of supporting the whole language. Being able to partially parse input code also significantly reduces the time required to parse an input program, therefore reducing the overhead of using POET to apply program transformations.

**6.1.2 Effectiveness Of The POET Optimization Approach**

As modern machines and applications both evolve to become increasingly complex and dynamic, it has become exceedingly difficult for compilers to accurately predict the behavior of applications running on different platforms. POET is provided to support the control of performance optimizations outside of the compilers and serves to support the auto-tuning of architecture-sensitive optimizations for portable high performance. In particular, it allows high performance computing specialists to directly control the transformation of their code while utilizing existing optimization capacities within compilers. It provides an interface to understand and effectively interact with optimizing compilers, and its built-in support for parameterization and tracing of optimized code makes it ideal for developers to easily port their code to different architectures.

When using POET to apply parameterized compiler transformations to an input program, the correctness of optimization depends on two factors: whether the xform routines are correctly implemented in the POET opt library, and whether the routines are invoked correctly in the input-specific optimization script. If either the library or the optimization script has errors, the optimized code may be incorrect. An optimizing compiler can ensure the correctness of its auto-generated POET scripts via conservative program analysis. Additional testing can be used to verify that the optimized code is working properly. Within our POET transformation engine in Figure 13, each optimized code is first tested for correctness before its performance is measured and used to guide the empirical tuning of optimization configurations. We have used POET to support the automatic generation of testing and timing drivers for individual routines, discussed in Section 6.2.3.

**6.2 Ad-hoc Language Translation and Code Generation**

Besides supporting performance optimizations, POET has also been used as an interpreted compiler writing language to support ad-hoc language translation and code generation of various domain-specific languages. The following first presents some example use cases and then discusses the effectiveness of using POET for this purpose.
6.2.1 Example 1: Translating Between Equivalent Languages

Using POET to translate between the syntaxes of two equivalent languages is extremely simple due to the special parsing and unparsing support inherently associated with POET code templates. In particular, each POET input command can explicitly specify a syntax description file to parse an input code. After parsing, the syntax descriptions are detached from the internal representation (i.e., the AST) so that a different syntax description can be used in the future to unpars the AST. Figure 14 shows a POET script for translating a set of C programs to Fortran. The script starts by declaring two command-line parameters so that users can dynamically specify input and output files of the translator. It then uses an input command to read the input file using C syntax. Finally, it uses an output command to unpars the input C code using Fortran syntax.

In essence, using POET to translate one language to another merely requires mapping both languages to a common set of code template concepts (e.g., loops, nests, if-conditionals). A subset of syntax descriptions for parsing the C language is shown in Figure 3, which are translated to the code templates in Figure 15 to support the C to Fortran syntax translation. Note that the code template parameters in Figure 15 do not need to be associated with parsing specifiers because these code templates are used for unparsing existing ASTs instead of building new ASTs from parsing.

6.2.2 Example 2: Building A POET Empirical Search Engine

When used to support auto-tuning of performance optimizations, the POET transformation engine in Figure 13 relies on a separate empirical search engine to automatically determine values of the command-line parameters used to control transformations to an input code. These parameters must be extracted from the POET optimization script and translated to acceptable input of a search algorithm to support the auto-tuning of POET optimizations. Our existing work has standardized the format of declaring configuration parameters for various optimizations supported by the POET opt library so that these optimizations can be automatically reconfigured by a separate empirical search engine [28].
Figure 16: Syntax specification for parsing standardized POET optimization parameters

```xml
<code CommandList parse=LIST(ParamDecl, "\n")/>
<code IntList parse=LIST(INT, " ")/>
<code ParamDecl parse=(param)>
  @param@
</code>
<code NumOfThreads parse=(parName:Name, loopNest:Name, defval:INT)>
  @parName@ type=1.._ default=@defval@ message="number of threads to parallelize loop @loopNest@"
</code>
<code BlockDimList parse=(spec:LIST("INT"," ")) rebuild=((spec:STRING)?1:LEN(spec))> @spec@ </code>
<code BlockFactor parse=(parName:Name, loopNest:Name, dim:BlockDimList, defval:IntList)>
  @dim@ @parName@ N 8 128 8 @defval@
</code>
```

Figure 17: Syntax specification for the input of a search engine

Figure 16 shows the list of syntax descriptions used to extract the optimization parameters from an arbitrary POET script. A key strategy here is that only the recognizable parameter declarations are parsed, while other components of the POET script are simply read as strings and then thrown away. In particular, lines 2-3 of Figure 16 specify that the top-level code template for parsing is a list, where each list component is parsed using either the `ParamDecl` or the `ThrowAway` code template. Line 4 specifies that each `ThrowAway` code template can be matched to anything, and their objects are replaced with empty strings after parsing (thus thrown away). Therefore, the result of parsing an arbitrary POET script would be the list of declarations that can be matched to the `ParamDecl` code template.

Figure 17 shows a collection of syntax descriptions used to translate the POET optimization parameters to the acceptable input format of a generic search algorithm. In practice, a different syntax description file could be developed for each alternative search algorithm, and a command-line parameter can be used in the POET translator to control which syntax file to use to output the search space descriptions.

### 6.2.3 Example 3: Automatic Generation of Timing and Testing Drivers

Large scientific applications often critically depend on a few computationally intensive routines that are either invoked numerous times by the application and/or include a significant number of loop iterations. These routines are often chosen as the target of automatic performance tuning, where differently optimized code are generated and experimentally evaluated.
to find superior performance. However, independent tuning of individual routines requires
a tester that can verify the correctness of differently optimized code and a timer that can
invoke the routine with an appropriate execution environment and accurately report the
performance of each invocation. Our existing work has developed POET translators to
automatically generate these testing and timing drivers based on user-provided interface
specifications for each routine of interest [35].

Figure 18 shows an example interface specification for the matrix multiplication routine in
Figure 5. The specification contains three components: a declaration of the routine at line 1
to specify all the routine parameters and return values, a driver description at lines 2-7 to
specify how to allocate, initialize, and control the cache states of each routine parameter,
and an optional formula to specify how to compute the MFLOPS (*millions of floating point
operations per second*). For example, line 3 of Figure 18 specifies that the three integer
parameters, \( M \), \( N \), and \( K \), should be initialized with environmental macros with default
value 72; lines 4-6 specify that the three matrices, \( A \), \( B \), and \( C \), should be allocated with
their appropriate sizes, initialized with pseudo-randomly generated data, aligned to a 16 byte
boundary, and flushed between timings. To generate a tester for the routine, a reference
routine implementation, e.g., Figure 5, needs to be defined so that the result of invoking the
differently optimized code can be compared with that of the reference implementation.

```c
1: routine=void dgemm_test(const int M, const int N, const int K, const double alpha, const double* A,
const int lda, const double* B, const int ldb, const double beta, double* C, const int ldc);
2: init={
3:   M=Macro(MS,72); N=Macro(NS,72); K=Macro(KS,72); lda=MS; ldb=KS; ldc=MS; alpha=1; beta=1;
4:   A=Matrix(double, M, K, RANDOM, flush|align(16));
5:   B=Matrix(double, K, N, RANDOM, flush|align(16));
6:   C=Matrix(double, M, N, RANDOM, flush|align(16));
7: };
8: flop="2*M*N*K+M*N";

Figure 18: Interface specification for the routine in Figure 5

Figure 19: Interface specification for the routine in Figure 5
Figure 19 shows some of the code templates used in automatically generating testers and timers in the C language from interface specifications illustrated by Figure 18. Specifically, three example code templates are used to specify how to allocate a single-dimensional array, a 2-dimensional array, and how to initialize a recently-allocated single-dimensional array respectively. The key strategy here is to use domain-specific concepts such as buffer allocation and initialization to define the structure of the auto-generated code instead of using lower-level concepts such as C/Fortran expressions and statements. These domain-specific concepts significantly reduce the complexity of generating the desired code from high-level specifications illustrated by Figure 18.

6.2.4 Example 4: A Finite-State-Machine-based Programming Language

Besides using POET to support the code generation of small ad-hoc languages such as the routine interface specification language discussed in Section 6.2.3, we have also used POET to support a more sophisticated programming language called iFSM [54], designed to collectively specify and verify the behavior notations and implementation strategies of object-oriented software. A key contribution of iFSM is a concise mapping from the runtime behavioral model of arbitrary C++/Java classes, expressed using finite state machines, to the internal implementations of these classes, expressed in terms of managing a collection of variables using an implementation specification language. Figure 20 shows the work flow of our framework for supporting the iFSM language, where we have used POET to implement the iFSM transformation engine, which automatically translates iFSM specifications to C++/Java class implementations and to the input language of a model checker, NuSMV [15].

Details of the iFSM language is beyond the scope of this paper. For this language, POET plays the role of implementing a prototype compiler to support the type checking, code generation, and verification of iFSM specifications. The process of using POET to implement these components are not fundamentally different from writing a typical compiler or language interpreter, except that developers can benefit from the built-in support for flexible parsing/unparsing, pattern matching, and program transformation. Our POET translator for iFSM can be configured via command-line parameters to dynamically produce output in C++, Java, or the input language of the NuSMV model checker, thus allowing variations of software implementations to be manufactured on demand based on different feature requirements.

6.2.5 Supporting Language Translation and Code Generation

POET is essentially an interpreted compiler writing language that can be used to quickly implement and conveniently support the type checking, program analysis, and transformation needs of various other programming languages. In particular, the following language features of POET can significantly improve the productivity of developers when building specialized translators or automatically generating code from domain-specific languages.
- **Easy construction of parsers and unparsers.** The POET input command can be invoked to dynamically parse an arbitrary programming language based on external syntax descriptions and automatically construct an internal representation of the input code. The internal representation can then be unparsed using similar syntax descriptions.

- **Partially parsing an input language.** As shown in Figure 16, POET can be used to easily parse embedded languages by simply throwing away unrecognized portions of the input code. This feature allows the parsing support for large and complex programming languages, e.g., C, Fortran, C++, Java, to be built in an incremental fashion.

- **Supporting domain-specific concepts.** As illustrated in Figure 19 and discussed in Section 6.2.3, when used in unparsing only, POET code templates can be used to directly associate high-level domain-specific concepts with parameterized complex lower-level implementations, significantly simplifying the task of generating low-level code from high-level domain-specific languages.

- **Mixing and correlating concepts from different languages.** Multiple programming languages can be freely mixed inside a single POET script through different input and output commands. These languages can share common concepts such as expressions, assignments, statements, and loops, so that a single code template can appear in multiple languages with different syntax definitions. This multi-lingual support makes it trivial to translate between different but equivalent languages that share a large number of common code templates.

- **Pattern matching and transformation support.** The built-in pattern matching and AST transformation operations, shown in Tables 4 and 5, make it much easier to write POET code to support simple program analysis (e.g., type checking) and transformation for ad-hoc domain-specific languages than using a general-purpose programming language such as C/C++/Java. Further, a large number of command-line parameters can be easily defined in a POET translator to satisfy different feature requirements.

## 7 Related Work

Existing research has developed many program transformation tools which use generalized compiler technology to assist software design, construction, or maintenance. These tools have been used to analyze, modify, reshape, and optimize existing code, including re-documenting code, re-implementing code, reverse engineering, changing APIs, porting to new platforms, rearranging system structure, etc [5, 9, 25]. A number of these translation systems can automatically generate programs from formal specifications [7, 8, 22, 26, 38] such as system design model [47], mathematical formulations [8, 21], reflection of metadata and code [20], design patterns [13] and dataflow graphs [39]. Several general-purpose transformation languages and systems have been developed [6, 12, 19, 30] and some have been widely adopted [6, 12].

Previous research on transformation-based software development mostly rely on pattern-based transformation rules coupled with application strategies [6, 12, 22, 26, 30, 38]. Although these rules are convenient to use and easy to learn, they are limited in their capability to express arbitrary program transformations. POET is a more advanced transformation language
in that developers can define their own transformations using compound data structures, arbitrary control flow, and recursive functions. A focus of the language design is on combining program transformation with empirical optimization technology to ensure portable high performance of the generated code. POET is a compile-time program transformation language and does not address runtime code generation as performed by various multistage languages and systems [10, 18, 27, 31].

Automatic generation of efficient implementations for special-purpose algorithms has been highly successful in a number of problem domains, including signal transform [23, 42], language translation [33], linear algebra [4, 51], device drivers [49], graph processing [29], among others. This body of research uses domain-specific specifications to define the computational problem and applies aggressive optimizations to improve algorithm implementation efficiency. Empirical tuning is often applied to automatically find the best implementations [4, 23, 42, 51]. POET can be used to automatically generate highly efficient algorithm implementations from compact problem specifications and can be used as the language of choice in developing such domain-specific empirical tuning frameworks. However, as a transformation language, POET target automatic program transformation for general-purpose applications beyond those supported by existing domain-specific frameworks.

In recent years, empirical tuning of application performance has become a de facto approach that both developers and optimizing compilers adopt for high performance computing [11, 23, 36, 40, 45, 51, 52]. POET supports existing iterative compilation frameworks [24, 32, 41, 48] through the extensive parameterization of program transformations. In particular, it can serve as a transformation engine that collaborate with different search and modeling techniques [14, 44, 50, 58]. Our previous work have used POET to empirically tune the performance of several linear algebra kernels and SPEC benchmarks [43, 55, 57].

POET scripts can be manually written by developers or automatically generated by an optimizing compiler, so that developers can have programmable control over the optimization decisions of compilers. Similar to POET, various annotation languages such as OpenMP [16] and the X language [17] also aim at supporting programmable control of compiler optimizations. These languages serve as a programming interface for developers to provide additional inputs to the optimizing compiler. In contract, POET is designed as an output language of compilers so that the optimization decisions by compilers can be easily modified or extended by developers when necessary.

8 Conclusions

This paper presents POET, an interpreted program transformation language designed for supporting programmable control of compiler optimizations for automatic performance tuning and for supporting the ad-hoc translation and code generation of arbitrary domain-specific languages. We present the key design and implementation decisions of the language and show that it can be the language of choice to satisfy many software development and optimization needs in practice. The POET language implementation and its manual can be freely downloaded at http://www.cs.utsa.edu/~qingyi/POET.
References


