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 reduce the difficulty 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 implementations using languages such as C/C++/Java, systematic program transformations need to be applied to improve the performance of existing C/C++/Java code, and compilers are required to translate C/C++/Java code 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 [66] and FFTW [29], have been built to automatically generate efficient implementations of key computational kernels on

*This research is funded by NSF through grant CCF-0747357 and CCF-0833203, and DOE through grants DE-SC0001770
a wide variety of computing platforms. These existing infrastructures, 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 [42]) can be used to provide infrastructure support for general-purpose program analysis and transformation, they are dedicated to only a few popular programming languages. There is a lack of infrastructure support for convenient parsing/unparsing of ad-hoc domain-specific notations and systematic application of structured program transformations. As a result the development cost for specialized code generation and optimization frameworks are prohibitive and have limited their wide-spread use. Reducing such overhead could significantly improve the availability of domain-specific code generators and optimization frameworks 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 includes 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. A POET script needs 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 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 be used for various purposes and play many different roles. 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 so that their configurations can be empirically tuned [72]. It provides developers with fine-grained parameterization and programmable control of compiler optimizations so that computational specialists can selectively apply these optimizations as well as conveniently define their own customized algorithm-specific optimizations [74].
• 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 [43], to automatically produce object-oriented C++/Java implementations from a finite-state-machine-based behavior modeling language [71], 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 [59].

This paper focuses on the key design and implementation decisions of the POET language to support the above use cases. Sections 2 and 3 first summarize the main design objectives and core concepts. Sections 4, 5 and 6 then present details of the language 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 7 presents use case studies. Finally, Sections 8 and 9 present related work and conclusions.

2 Design Objectives

POET focuses on supporting two main software development needs: (1) parameterizing compiler optimizations and making them readily available to developers for programmable control and performance tuning on varying architectures, and (2) significantly reducing the development cost of source-to-source program transformation, ad-hoc language translation, and domain-specific code generation. Each use case is discussed in detail in the following.

2.1 Programmable Compiler Optimization For Empirical Tuning

As modern hardware and software both evolve to become increasingly complex and dynamic, it has become exceedingly difficult for compilers to accurately predict the behavior of applications on different platforms. POET is provided to support programmable control of optimizations outside the compilers and empirical-tuning of optimizations for portable high performance. It allows computational specialists to directly control the optimization of their code while utilizing existing capabilities within compilers, by providing an interface for developers to understand and interact with optimizing compilers.

Figure 2 shows the targeting optimization environment we are building using POET. In particular, an optimizing compiler, e.g., the ROSE analysis engine [58] in Figure 2, 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 control the auto-generated compiler transformations and to add new optimizations if necessary. Regular developers can use POET to obtain optimization feedback from compilers.

The technical aspects of using a compiler to automatically generate parameterized POET scripts are presented in [69] and are beyond the scope of this paper, which focuses on using POET to support such an optimization environment with the following language features.

**Ability to dynamically support arbitrary programming languages** POET is language neutral and uses syntax specifications defined in external files to dynamically process different input and output languages. It has been used to support a wide variety of different programming languages such as C, C++, Fortran, Java. 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.

**Selective transformation of the input code** POET can be used to selectively parse only a subset of the input code fragments which are targets of program analysis or transformation. The other fragments can simply be saved as lists of strings with minimal processing overhead. Being able to partially parse input code allows developers to define POET syntax descriptions only for small subsets of programming languages such as C, C++, and Fortran, while maintaining their ability to support large-scale full applications in these languages.

**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 transformations** Each POET script can specify a large number of command-line parameters to dynamically reconfigure its behavior. A single script therefore can be used to produce a wide variety of different output, effectively allowing different software implementations be manufactured on demand based on varying feature requirements.

**Composition and Tracing of Transformations** Each POET script may apply a long sequence of different transformations to an input code, with each transformation controlled by command-line parameters and can be optionally turned off. Dramatically different code therefore may be produced as the result of varying transformation configurations. Without automatic tracing support, the complexity of combining different transformation configurations can quickly become exponential and out-of-hand. POET provides dedicated language support to automatically trace the modification of various code fragments as the input code...
Parallelization and memory hierarchy optimization

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParallelizeLoop(x)</td>
<td>Parallelize the outermost loop using OpenMP in input code x</td>
</tr>
<tr>
<td>DistributeLoops(n, x)</td>
<td>Distribute input code x so that fragments in n end up in separate components</td>
</tr>
<tr>
<td>FuseLoops(n, p, x)</td>
<td>Fuse disjoint loops in n into a single one; then use it to replace fragment p in input code x</td>
</tr>
<tr>
<td>BlockLoops(n, x)</td>
<td>Block the loops nested outside of fragment n but inside input code x</td>
</tr>
<tr>
<td>PermuteLoops(n, x)</td>
<td>Permute the loops nested outside of fragment n but inside input code x</td>
</tr>
<tr>
<td>SkewLoops(n1, n2, x)</td>
<td>Use the outer loop n1 to skew the inner loop n2 within input code x</td>
</tr>
<tr>
<td>CopyRepl(v, a, d, x)</td>
<td>Use buffer v to copy and replace memory referenced by a at loop iterations d in input code x</td>
</tr>
<tr>
<td>CleanupBlockedNests(x)</td>
<td>Cleanup the blocked loop nests via loop splitting in input code x</td>
</tr>
</tbody>
</table>

Scalar and register performance optimization

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

High-level to low-level code translation

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayAccess2PtrRef(x)</td>
<td>Convert all array references in input code x to pointer references</td>
</tr>
<tr>
<td>TransformThreeAddress(x)</td>
<td>Transform input code x into three address code</td>
</tr>
<tr>
<td>TransformTwoAddress(x)</td>
<td>Transform input code x into two address code</td>
</tr>
</tbody>
</table>

### Table 1: Selected transformation routines supported by the POET opt library

goes through different transformations (for more details, see Section 6.2). The tracing support makes the composition and parameterization of different transformations extremely flexible, where ordering of transformations can be easily adjusted or even dynamically tuned [72].

#### The POET optimization library

We have used POET to implement a large number of advanced compiler transformations, shown in Table 1, and have provided these transformations as xform routines in the POET opt library to support performance optimization. These routines can be invoked by arbitrary POET scripts and serve as the foundation for developers to build additional sophisticated optimizations.

### 2.2 Ad-hoc Language Translation and Code Generation

POET is essentially an interpreted compiler writing language which can be used to significantly improve the productivity of developers when building ad-hoc language translators or domain-specific code generators, by providing the following language features.

#### Easy construction of parsers and unparsers

POET can be used 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. The process is different from the conventional parser generator approach in that it allows the syntax descriptions to be provided dynamically as input data together with the input code, and that internal representations of the input code are automatically constructed without requiring extra work from the developers. POET can also be used to partially parse an input language by simply throwing away unrecognized portions of the input code. This feature allows the parsing support for large and complex languages, e.g., C, Fortran, C++, Java, to be built in an incremental fashion.

#### Supporting domain-specific concepts

POET provides a collection of code templates (defined in Section 4.1) which 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. 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 concrete syntax definitions. This multi-lingual support makes it trivial to translate between languages that support similar concepts with minor differences in syntax, e.g., a significant subset of C++ and Java.

**Flexibility and ease of use**  A POET program can include an arbitrary number of different files which can communicate with each other via a set of explicitly declared global variables. All variables can dynamically hold arbitrary types of values. A large collection of built-in operators are provided to easily construct, analyze, and modify internal representations of different programming languages. Command-line parameters can be declared to easily parameterize each POET file for different feature requirements.

### 2.3 Users Of The POET Language

Two groups of users could benefit from our design of the POET language: program transformation experts such as compiler writers, high performance computing specialists, and domain-specific language designers who use POET directly to achieve portable high performance on modern architectures (via specialized optimization scripts) or to support their domain-specific languages; and casual users such as application developers or domain scientists who use the POET-generated high-performance computational kernels as libraries or leverage POET-supported domain-specific systems to achieve varying goals. The discussion of the POET language in this paper targets the first group who use POET as an implementation language to support their performance optimization or language translation needs. The second group can simply use the POET-supported systems (built by the first group of POET users) without knowing about the existence of POET. Note that when computational specialists exert programmable control over how their applications are optimized, they can invoke an optimizing compiler to automatically generate the POET scripts before making modifications, without writing POET scripts from scratch.

### 3 Overview of the POET Language

This section presents the core concepts supported by POET to achieve its design goals. These core concepts are summarized in Table 2, and their uses demonstrated in Figure 3.

#### 3.1 The Type System

As shown in Table 2, POET supports two types of atomic values: integers and strings. The boolean value `false` is represented using integer 0, and `true` can be represented using any of 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.
Table 2: Overview of the POET language

<table>
<thead>
<tr>
<th>Types of values</th>
<th>int (e.g., 1, 20, -3), string (e.g., “abc”, “132”)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 atomic types</td>
<td>list, tuple, associative map, code template, xform handle</td>
</tr>
<tr>
<td>2 compound types</td>
<td></td>
</tr>
<tr>
<td>3 e₁ e₂ ... eₙ</td>
<td>A list of n elements e₁, e₂, ..., eₙ</td>
</tr>
<tr>
<td>4 e₁, e₂, ..., eₙ</td>
<td>A tuple of n elements e₁, e₂, ..., eₙ</td>
</tr>
<tr>
<td>5 MAP{f₁=&gt;t₁, ..., fₙ=&gt;tₙ}</td>
<td>An associative map of n entries which map f₁ to t₁, ..., fₙ to tₙ respectively</td>
</tr>
<tr>
<td>6 c # (p₁, ..., pₙ)</td>
<td>A code template object of type c with p₁, ..., pₙ as values of its parameters</td>
</tr>
<tr>
<td>7 f[v₁=p₁, ..., vₙ=pₙ]</td>
<td>A xform handle f with p₁, ..., pₙ as values for optional parameters v₁, ..., vₙ</td>
</tr>
<tr>
<td>Operating on different types of values</td>
<td></td>
</tr>
<tr>
<td>8 +, -, *, /, %, &lt;, &lt;=, &gt;, &gt;=, ==, !=</td>
<td>Integer arithmetics and comparison</td>
</tr>
<tr>
<td>9 !, &amp;&amp;,</td>
<td></td>
</tr>
<tr>
<td>10 ==, !=</td>
<td>Equality comparison between arbitrary types of values</td>
</tr>
<tr>
<td>11 a &amp; b</td>
<td>Concatenate two values a and b into a single string</td>
</tr>
<tr>
<td>12 SPLIT(p,a)</td>
<td>Split string a into substrings separated by p; if p is an int, split a at location p</td>
</tr>
<tr>
<td>13 a :: b</td>
<td>Prepend value a in front of list b s.t. a becomes the first element of the new list</td>
</tr>
<tr>
<td>14 HEAD(l), car(l)</td>
<td>The first element of a list l</td>
</tr>
<tr>
<td>15 TAIL(l), cdr(l)</td>
<td>The tail behind the first element of a list l; returns “” if l is not a list</td>
</tr>
<tr>
<td>16 a[b] where a is a tuple</td>
<td>The bth element of a tuple a</td>
</tr>
<tr>
<td>17 a[b] where a is a map</td>
<td>The value mapped to entry b in an associative map a</td>
</tr>
<tr>
<td>18 a[c.d] where c is a code template</td>
<td>The value stored in parameter d of a code template object a, which has type c</td>
</tr>
<tr>
<td>19 LEN(a) where a is a string</td>
<td>The number of characters in string a</td>
</tr>
<tr>
<td>20 LEN(a) where a is not a string</td>
<td>The number of entries in the list, tuple, or map; returns 1 otherwise</td>
</tr>
<tr>
<td>Variable assignment and control flow</td>
<td></td>
</tr>
<tr>
<td>21 a = b</td>
<td>Modify a local or static variable a to have value b; return b as result of evaluation</td>
</tr>
<tr>
<td>22 a[i] = b</td>
<td>Modify associate map a so that entry i is mapped to value b; return b as result</td>
</tr>
<tr>
<td>23 (a₁, ..., aₘ) = (b₁, ..., bₘ)</td>
<td>Modify a₁, ..., aₘ to have values b₁, ..., bₘ respectively; return the b tuple</td>
</tr>
<tr>
<td>24 a₁; a₂; ...; aₘ</td>
<td>Evaluate expressions a₁ a₂ ... am in order; return the result of evaluating am</td>
</tr>
<tr>
<td>25 RETURN a</td>
<td>Evaluate expression a and then return it as result of the current xform invocation</td>
</tr>
<tr>
<td>26 if (a) { b }</td>
<td>If conditional, returns b as result based on whether a is TRUE or FALSE</td>
</tr>
<tr>
<td>27 switch(a) { case b₁; c₁; ...; default: cₙ }</td>
<td>Similar to the switch conditional in C, returns value of the first successful branch</td>
</tr>
<tr>
<td>28 for (e₁; e₂; e₃) { b }</td>
<td>Equivalent to the for loop in C; always return empty string</td>
</tr>
<tr>
<td>29 BREAK, CONTINUE</td>
<td>Equivalent to the break and continue statements in C; used only in loops</td>
</tr>
<tr>
<td>Global type/variable declarations and commands</td>
<td></td>
</tr>
<tr>
<td>30 &lt;define a b /&gt;</td>
<td>Declare a global macro variable a and assign b as its value</td>
</tr>
<tr>
<td>31 &lt;trace a₁,...,aₘ /&gt;</td>
<td>Declare a list of related trace handles a₁,...,aₘ</td>
</tr>
<tr>
<td>32 &lt;parameter p type= default=default&gt;</td>
<td>Declare a command-line parameter p which has type t and default value v; Its value is built using parsing specifier r, and its meaning is defined in string d.</td>
</tr>
<tr>
<td>33 &lt;input cond=c from=f syntax=s to=t /&gt;</td>
<td>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>34 &lt;eval s₁,...,sm /&gt;</td>
<td>Evaluate the group of expressions/statements s₁, ..., sm</td>
</tr>
<tr>
<td>35 &lt;output from=t to=f syntax=s cond=c /&gt;</td>
<td>If expression c evaluates to true, unpars the expression t to file f using syntax descriptions defined in file s.</td>
</tr>
</tbody>
</table>

the other integers. Two notations, TRUE and FALSE, are provided to denote integers 1 and 0 respectively. Additionally, the following compound types are supported within POET.

- **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 “<=" b) produces a list with three elements, a, “<="”, and b. Lists can be dynamically extended using the :: operator at line 13 of Table 2, and are used to group sequentially-accessed elements.

- **Tuples.** A POET tuple is a finite number of elements composed in a predetermined order and is constructed by separating individual elements with commas. For example, (“i”, 0, “m”, 1) constructs a tuple with four elements, “i”, 0, “m”, and 1. A tuple cannot be dynamically extended and is used to group a statically-known number of values, e.g., the parameters of a function call.
3.2 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 scopes are restricted within the bodies of individual code templates or xform routines. For example, at lines 2-13 of Figure 3, \(i, \text{start}, \text{stop}, \text{step}\) are local variables of the code template `Loop`, and \(\text{list.result}\) and \(p\text{list}\) 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 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 explicit declaration. For example, at line 20 of Figure 3, both `backward` and `succ` are file-static variables, which are used to store temporary results across different components of the same file.

• **Global variables**, whose scopes span across all the given POET files being evaluated. Each global variable must be explicitly declared before used as one of the three categories: `macros` (e.g., the `OPT_STMT` variable declared at line 14 of Figure 3), `command-line parameters` (e.g., `inputFile`, `inputLang`, and `outputFile` declared at lines 15-17 of Figure 3), and `trace handles` (e.g., `inputCode` declared at line 18 of Figure 3). More details of these variables are discussed in Section 3.3.

Since only global variables need to be explicitly declared, all the undeclared names are treated as local or static variables, based on the scopes of their appearances. In particular, all names inside a code template or xform routine body are considered local variables unless an explicit prefix, e.g., `GLOBAL`, `CODE`, or `XFORM`, is used to qualify the name. An example of such qualified names is shown at line 14 of Figure 3. Both local and static variables can be freely modified within their scopes using assignments, shown at lines 21 and 23 of Table 2.

Note that global variables in POET serve various special purposes, and as a result they cannot be simply modified using regular assignments. In particular, global `macros` are used to reconfigure behavior of the POET interpreter (see Section 4.4) and can be modified only through the `define` command illustrated at line 14 of Figure 3; `command-line parameters` can be modified only through command-line options; `trace handles` are used to keep track of various fragments of the input code and can be modified only through special-purpose operators (see Sections 6.2). Also note that POET does not allow any portion of a compound data structure such as a tuple, list, or code template object, to be modified, unless `trace handles` have been inserted inside these data structures, discussed in Sections 6.2.

### 3.3 Components of A POET Script

Figure 3 shows the typical structure of a POET script, which includes a sequence of `include` directives (line 1), type declarations (lines 2-13), global variable declarations (lines 14-18), and executable commands (lines 19-21). The `include` directives 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 3, `comments` can appear anywhere in POET and must be enclosed either 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 are used to construct pointer-based data structures and internal representations of different languages,
and \textit{xform routines}, which are global functions used to implement various program analysis and transformation algorithms. In Figure 3, 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. 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 default value. An example invocation of the \textit{ReverseList} routine is illustrated at line 20 of Figure 3.

Each POET script must explicitly declare all the global variables it needs to use. For example, line 14 of Figure 3 declares a \textit{macro} named \textit{OPT\_STMT} and assigns the code template type \textit{Loop} as its value. Lines 15-17 declare three \textit{command-line parameters} named \textit{inputFile}, \textit{inputLang} and \textit{outputFile}, whose values can be redefined via command-line options. Line 18 declares a \textit{trace handle} named \textit{inputCode}, which is used to keep track of transformations to the input code. Details of using trace handles are discussed in Section 6.2.

POET supports three types of global commands, \textit{input}, \textit{eval}, and \textit{output}, illustrated at lines 19-21 of Figure 3. In particular, the \textit{input command} at line 19 is used to parse a file named by variable \textit{inputFile} using syntax descriptions contained in a file named by \textit{inputLang}. The parsing result is then converted into an internal representation and stored to variable \textit{inputCode}. The \textit{eval} command at line 20 specifies a sequence of expressions and statements to evaluate. The \textit{Output Command} at line 21 is used to write the transformed internal representation (i.e., \textit{inputCode}) to an external file named by \textit{outputFile}.

All POET expressions and statements must be embedded inside an \textit{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 2, 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.

## 4 Dynamically Parsing Arbitrary Languages

A key language feature of POET is the ability to dynamically parse an arbitrary programming language using a single \textit{input command}, where both the concrete syntax and the internal representation of the input language are collectively specified using a collection of \textit{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 \textit{input command}. This approach is more flexible than the conventional parser generation approach [40], where the auto-generated parser is specialized to work for a single predefined input language, and developers must manually construct an internal representation of the input code via syntax-directed translation. When using POET to parse an input code, the construction of the AST is fully automated. Further, developers can dynamically select and mix different input/output languages, easily unify different languages with a single interface, and invoke generic analysis and transformation routines that apply to all 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
Components of a code template declaration and their meanings

- **Template body**: Concrete syntax of the code template for parsing and unparsing.
- **Parse**: Template parameters which specify data fields of the corresponding internal representation.
- **Parse=pl**: Use pl as the alternative concrete syntax to substitute the template body for parsing.
- **Output=pl**: Use pl as the alternative concrete syntax for unparsing of the code template.
- **Lookahead=n**: Examine the n leading input tokens when using the code template for parsing (by default, n=1).
- **Rebuild=e**: Use expression e as the alternative return result after successful parsing using the code template.
- **V=INHERIT**: Use local variable v to save the previous parsing result before using the code template for parsing.
- **Allow c to be used in place of the code template when matching against the input code**

<table>
<thead>
<tr>
<th>Type specifiers for tokens and compound data objects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INT</strong>: The integer type, which includes all integer values</td>
</tr>
<tr>
<td><strong>STRING</strong>: The string type, which includes all string values</td>
</tr>
<tr>
<td><strong>ID</strong>: The identifier type, which includes all string values that can be used as identifiers in POET</td>
</tr>
<tr>
<td><strong>CODE</strong>: The code template type, which includes all code template objects</td>
</tr>
<tr>
<td><strong>XFORM</strong>: The xform handle type, which include all xform routine handles</td>
</tr>
<tr>
<td><strong>TUPLE</strong>: The tuple type, which includes all POET tuples</td>
</tr>
<tr>
<td><strong>MAP</strong>: The associative map type, which includes all POET associative maps</td>
</tr>
<tr>
<td><strong>ANY</strong>: The ANY type, which includes all values supported by POET</td>
</tr>
<tr>
<td><strong>lb .. ub</strong>: The range type, which includes all integers ( \geq lb ) and ( \leq ub )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parsing specifiers which define how to match concrete syntax specifications with leading input tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A constant value</strong></td>
</tr>
<tr>
<td><strong>A token type specifier</strong></td>
</tr>
<tr>
<td><strong>˜p</strong></td>
</tr>
<tr>
<td><strong>A code template name</strong></td>
</tr>
<tr>
<td><strong>EXP</strong></td>
</tr>
<tr>
<td><strong>V = p</strong></td>
</tr>
<tr>
<td><strong>P ...</strong></td>
</tr>
<tr>
<td><strong>LIST(p,s)</strong></td>
</tr>
<tr>
<td><strong>TUPLE(p)</strong></td>
</tr>
<tr>
<td><strong>P1 P2 ... pm</strong></td>
</tr>
<tr>
<td>**P1</td>
</tr>
</tbody>
</table>

Table 3: POET support for specifying syntax of arbitrary languages

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 4.1 presents how to use POET code templates to collectively specify the syntax and internal representation of an arbitrary language. Section 4.2 presents annotations that can be inserted within an input code to provide additional information for parsing. Section 4.3 presents our dynamic parsing algorithm. Section 4.4 presents POET macros that can be used to dynamically modify behavior of the dynamic parser.

### 4.1 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 3 shows the meaning of various code template components as they are used in parsing, unparsing, and AST construction.
Figure 4 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 $a$ is then used to substitute the original BNF symbol $t$ in the template body. For example, the $\texttt{Ctrl}$ symbol at line 1 of Figure 4(a) is translated to template parameter $\texttt{ctrl:CODE.Ctrl}$ in (b), where $\texttt{CODE.Ctrl}$ declares $\texttt{Ctrl}$ as a new POET code template name that will be defined later, and 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 its body and parameters to recursively define the concrete syntax specified by a BNF production. Table 3 shows the different formats of parsing specifiers supported by POET.

After parsing, an internal representation of the input code is automatically constructed, where corresponding code template objects are created to represent the internal structure of
Figure 5: The AST built after parsing line 6 of Figure 6 using code templates in Figure 4.

The input code. 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 4(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 5 shows the resulting AST built after using the code templates in Figure 4 to parse an input code fragment at line 6 of Figure 6.

The default AST construction for each code template can be reconfigured using the `rebuild` attribute shown in Table 3. For example, line 5 of Figure 4(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 attribute evaluation during AST construction. For example, line 6 of Figure 4(b) specifies that the previous code template object constructed (i.e., the true-branch of an if-conditional) should be saved in the `ifNest` field 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 concerned by the parsing process.

4.2 Annotating the Input Code

The POET dynamic parser accepts annotations embedded inside an input code and uses the additional information to guide the parsing of various code fragments. As illustrated by Figure 6, 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 variable to save the parsing result.

- Single-line annotations, each of which applies to a single line of program source and
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;  
    for (j = 0; j < N; j += 1)  
    {  
        for (i = 0; i < M; i += 1)  
        {  
            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 6: An example POET input code with embedded annotations

has the format => T, where T is a parsing specifier defined in Table 3. For example, line 5 of Figure 6 indicates that the source code should be parsed using the Stmt code template, and the parsing result should be stored in the 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 6 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 variables. Similarly, the other multi-line annotations in Figure 6 define values of the variables gemm (lines 1-17), nest2 (lines 8-15), and nest1 (lines 11-14).

4.3 The Parsing Algorithm

Figure 7 shows our dynamic parsing algorithm implemented within the POET interpreter. 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 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 7 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.

Step (1) of the algorithm first examines the leading token (tok1) from the input stream and
Figure 7: The dynamic recursive descent parsing algorithm in POET

processes it if it is an input annotation which associates a parsing specifier with a fragment of the input code. Each annotation is categorized as either single-line (illustrated at line 5 of Figure 6), where a complete input code fragment is annotated, or multi-line (illustrated at lines 6, 8, and 11 of Figure 6), where only the start of a fragment is annotated. The algorithm processes each single-line annotation simply by recursively invoking itself to parse the annotated code fragment. 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 tok1 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 tok1 followed by the rest of the input stream. In particular, the algorithm independently 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, `tok1` is removed from `tokens`) if and only if the value of `goal` matches that of `tok1`.

(2.2) `goal` is the name of a code template. If `tok1` is already an object of the given code template (the result of processing input annotations), the object is returned as result after removing `tok1` 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 `tok1` 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 3. 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.

(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 `tok1` to predictively determine which specifier to use. If any of the alternative specifiers can potentially match `tok1` 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.

4.4 Modifying The Default Parsing Behavior

POET provides three categories of built-in macros, shown in Table 4, 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 8 illustrates how to redefine these POET macros to support programming languages such as C/Fortran. In particular, the `TOKEN` and `KEYWORDS` macros at lines 1-2 of
### Table 4: Macros that can be used to reconfigure the default behavior of POET

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

Figure 8 are used to reconfigure the POET tokenizer. The PARSE macro at line 3 defines 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 EXP_BASE macro can be extended to include additional code templates as components and can recursively invoke the EXP parsing specifier if necessary².

### 5 Analyzing the Input Code

The POET language currently places more emphasis on supporting program transformations, discussed in Section 6, than supporting sophisticated program analysis such as iterative data-flow analysis, dependence analysis, and pointer aliasing analysis commonly implemented in full-blown optimizing compilers [45]. Being an interpreted transformation language, POET is not intended as a language of choice for implementing complex program analysis algorithms. 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 5 and discussed in Sections 5.1 and 5.2 respectively.

---

²Note that left-recursion is not allowed, as required by all top-down predictive parsers
5.1 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 9 which uses pattern matching to recursively check the type consistency of an simple expression. Table 5 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 9 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 9, 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 9 includes two
**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 3)
VAR | Trace handles which can be embedded within POET expressions
a 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
$\text{CLEAR } v$ | All values; variable $v$ is assigned with the value after matching
$c \neq p$ | 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
$p_1 \odot p_2$ | All lists of $n$ elements which match the pattern specifiers $p_1, p_2, ..., p_n$ respectively
p1 op p2 | All lists with the first element matching pattern $p_1$ and rest of the list matching pattern $p_2$
$p_1 :: p_2$ | All lists with the first element matching pattern $p_1$ and rest of the list matching pattern $p_2$
$p_1 \mid p_2 \mid ... \mid p_n$ | All values that can match one of the pattern specifiers $p_1, p_2, ..., p_n$

**Pattern matching operators**

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

**AST traversal operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreach $a : b : c$ { $d$ }</td>
<td>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.</td>
</tr>
<tr>
<td>foreach $r(a : b : c)$ { $d$ }</td>
<td>Same as the foreach operator, except that values within $a$ are traversed in reverse order</td>
</tr>
</tbody>
</table>

### Table 5: POET support for pattern matching and AST traversal

```
1: <xform TypeCheckExp pars=(symTable, exp)>
2: switch(exp)
3: {
4:  case Bop#("+"|"-"|"*"|"/"|"%", exp1, exp2):
5:      type1 = TypeCheckExp(symTable, exp1);
6:      type2 = TypeCheckExp(symTable, exp2);
7:      if (type1 : CODE.IntType && type2 : CODE.IntType) returnType=IntType;
8:      else if (type1 : CODE.FloatType && type2 : CODE.FloatType) returnType=FloatType;
9:      error("Type checking error: " exp);
10:     symTable[exp] = returnType; <<* saving the type of exp in symbol table
11:     returnType
12:  case STRING: (symTable[exp])
13:    case INT : IntType
14:  }
15: </xform>
```

Figure 9: Example: type checking for simple expressions

uninitialized variables, $exp_1$ and $exp_2$, so if the pattern matching succeeds, $exp_1$ and $exp_2$ 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 local and static variables.

### 5.2 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 5, both operators collectively apply pattern matching to the entire AST representation of an input computation.

As example, the `xform` routine in Figure 10 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 10 traverses the input code to find all loop nests, each of which 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 10 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).

5.3 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 9. Figure 11 presents an implementation of control-flow analysis.

The executable commands of the POET script in Figure 11 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 BuildBasicBlocks routine, defined in Figure 10, to identify all basic blocks (i.e., single-entry-single-exit sequences of statements) and then invoking the BuildCFG routine, defined in Figure 11, 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 10, the POET script for identifying basic blocks starts by declaring a code template type BasicBlock (at lines 1-3) to store the analysis result. Lines 4-21 then define
the 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 11, line 1 defines a code template type GraphEdge to support the construction of the control-flow graph. Lines 2-23 then define the BuildCFG routine which takes the input code together with the collection of identified basic blocks and returns a tuple of two components: the list of control flow edges to connect the basic blocks, and the last basic block encountered from traversing the input code.

As illustrated by the BuildBasicBlock routine in Figure 10 and by the BuildCFG routine in Figure 11, POET provides two ways to traverse an input AST: using the foreach or foreach_r operators, and using recursive invocations of AST visiting functions. While the foreach and 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 the implementation of arbitrary complex divide-and-conquer algorithms.

As shown in both Figures 10 and 11, code templates in POET can be used to easily build complex data structures, e.g., via the BasicBlock and GraphEdge data types. However, to avoid infinite recursion when traversing ASTs built from 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
### Table 6: 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;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 + 2 + 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;,Bop#(&quot;-&quot;,&quot;b&quot;,&quot;c&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>1</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((1 2 3), (&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 7: Examples of invoking transformation operators

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 can be implemented using either the built-in compound type `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 versus 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.

### 6 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, Sections 6.1, 6.2, and 6.3 discuss how to effectively use various POET built-in support, shown in Table 6, to build sophisticated program transformations. Section 6.4 illustrates how to build the top-level transformation scripts for optimizing known input programs.
Figure 12: Examples: modifying the AST

6.1 Modifying the AST

POET provides four built-in operators, REPLACE, REBUILD, DUPLICATE, and PERMUTE, shown in Table 6 and illustrated in Table 7, to support the replacement, simplification, replication, and permutation of various code fragments within an input AST. Figure 12 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 6.2 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 7. For example, the REPLACE invocation at line 13 of Figure 12 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 12 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.

The \texttt{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 \texttt{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 \texttt{rebuild} attribute (see Table 3), the \texttt{rebuild} expression is invoked, and the rebuilding result is used to substitute the original code template object. For example, entry 4 of Table 7 shows that when applied to a symbolic expression composed of constant numbers, the \texttt{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 \texttt{rebuild} attribute defined within the \texttt{Bop} code template, a type defined in our POET \textit{opt} library.

The \texttt{DUPLICATE} and \texttt{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 1.

### 6.2 Tracing of Transformed Code

POET uses a special concept called \textit{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 \textit{xform routines} even if the routines cannot directly access them through their names. In particular, \textit{xform routines} can invoke the built-in transformation operators shown in Table 6 to effectively keep all embedded trace handles up-to-date. POET dedicates six built-in operators, \texttt{INSERT}, \texttt{ERASE}, \texttt{COPY}, \texttt{TRACE}, \texttt{SAVE}, and \texttt{RESTORE}, shown in Table 6, 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, illustrated by lines 15-25 of Figure 13 and discussed in Section 6.4.

All trace handles must be explicitly declared at the global scope before used, and they must be inserted inside an input AST using the \texttt{INSERT} operator to keep track of the transformed code. 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 \texttt{gemm}, \texttt{gemmDecl}, \texttt{gemmBody}, \texttt{nest3}, \texttt{nest2}, and \texttt{nest1} in Figure 6, 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 \texttt{INSERT} operator, illustrated at line 14 of Figure 13.

Since embedded trace handles can be modified by the transformation operators shown in Table 6, 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 \texttt{EraseTraceHandle} routine at lines 2-7 of Figure 12 illustrates how to properly remove all nested trace handles from an input AST by invoking the \texttt{ERASE} operator. In particular, \texttt{ERASE(handlevalue, handlevalue)} at line 3 peels off the outermost trace handle contained in \texttt{handlevalue} by returning its value, and \texttt{ERASE(handlevalue, exp)} at line 3 removes the trace handle contained in \texttt{handlevalue} from \texttt{exp} and returns the new
1: include opt.pl
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 {
6: UnrollJam[(factor=(nu mu);trace=gemmBody(nest1,nest3));];
9: A_ScalarRepl=DELAY {
10: 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]));
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 13: A POET script that optimizes the input code shown in Figure 6

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 13, 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 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.

### 6.3 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 a variable, the delayed expression can be later evaluated simply by invoking the APPLY operator with the variable as parameter. For example, the DELAY operator is used at lines 6-12 of Figure 13 to save all the potential input code transformations into a list of static variables. Lines 15-25 then apply these transformations by invoking the
APPLY operator using the corresponding variables as parameters. 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.

6.4 Building An Optimization Script

Figure 13 illustrates the typical structure of a POET transformation script for optimizing a known input code, shown in Figure 6. This script serves as a structural guideline for automatically applying parameterized compiler transformations to generate efficient implementations of an input code on varying architectures.

The optimization script in Figure 13 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 13). In particular, these trace handles are used inside the parsing annotations embedded in the input code in Figure 6 so that after parsing the input code using the input command (line 3 of Figure 13), 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 13, 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 13. 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 13 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. Lines 6-12 define each input code transformation 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 at line 2 to be embedded inside 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 6) if necessary [72].
7 Use Case Studies

To demonstrate the effectiveness of POET in supporting its design objectives, the following summarizes our experiences in using it to support a number of uses cases both in enabling computational kernels to automatically achieve portable high performance and in supporting ad-hoc language translation and code generation for domain-specific languages.

7.1 Using POET To Support Compiler Optimizations

As shown in Figure 2, POET can be used by developers to control how to optimize their applications to achieve 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 [74]. A portion of the POET script for optimizing the matrix-matrix multiplication kernel is shown in Figure 13, where the input file is shown in Figure 6. We also used POET to manually optimize several SPEC95 benchmarks and studied the interactions between parallelization granularity and cache reuse [54]. Our recent work has extended the ROSE optimizing compiler [73] to automatically produce parameterized POET scripts [69]. We have additionally developed an empirical search engine [59] which can automatically explore the configuration space of POET optimization scripts for varying architectures. The search engine is discussed in more detail in Section 7.3.

When using POET to apply parameterized compiler transformations to an input program, the correctness of optimization depends on two factors: whether the program transformations are correctly implemented in the POET optimization library, and whether the transformation routines are invoked correctly in the input-specific POET optimization script. If either the library or the optimization script has errors, the optimized code may be incorrect. An optimizing compiler (i.e., our analysis engine) can ensure the correctness of its auto-generated POET scripts via conservative program analysis. For user supplied POET scripts, additional testing can be used to verify that the optimized code is working properly. Within our POET transformation engine in Figure 2, 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 7.4.

7.2 Translating Between Equivalent Languages

Using POET to translate two equivalent languages is made simple by 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 AST representation so that a different syntax description can be used in the future to unpars the AST. Figure 14 shows a POET script for translating 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 templates (e.g., loops, nests, if-conditionals). A subset of the syntax descriptions for parsing C is shown in Figure 4, which are redefined with alternative Fortran syntax in Figure 15 to support C to Fortran translation.

### 7.3 Building A POET Empirical Search Engine

When used to support auto-tuning of performance optimizations, the POET transformation engine in Figure 2 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 a 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 [59].

Figure 16 shows a list of syntax descriptions used to extract standardized 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.
7.4 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 [43].

Figure 18 shows an example interface specification for the matrix multiplication routine in Figure 6. 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 6, needs to be defined so that the result of invoking the
Figure 18: Interface specification for the routine in Figure 6

Figure 19: Interface specification for the routine in Figure 6

differently optimized code can be compared with that of the reference implementation.

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 allo-
cation 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.

7.5 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 7.4, we have also used POET
to support a more sophisticated programming language called iFSM [71], designed to col-
lectively 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 be-
havioral 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 [19].

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.

8 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 [6,11,31]. A number of these translation systems can automatically generate programs from formal specifications [9,10,28,32,46] such as system design model [60], mathematical formulations [10,27], reflection of metadata and code [26], design patterns [16] and data flow graphs [48]. Several general-purpose transformation languages and systems have been developed [8,15,25,36] and some have been widely adopted [8,15].

Previous research on transformation-based software development mostly rely on pattern-based transformation rules coupled with application strategies [8,15,28,32,36,46]. Although these rules are convenient to use and easy to learn, they are limited in supporting arbitrary program transformations. In POET, developers can define arbitrary transformations using compound data structures, flexible control flows, and recursive functions. A focus of the language design is to combine program transformation with empirical tuning 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 [12,24,33,37].

Automatic generation of efficient implementations of special-purpose algorithms has been highly successful in a number of problem domains, including signal transform [29,53], language translation [40], linear algebra [5,13,66], device drivers [62], graph processing [35], among others. This body of research uses domain-specific specifications to define the computational problem and applies aggressive optimizations to improve algorithm implemen-
tation efficiency. Empirical tuning is often used to automatically achieve portable high performance [5, 13, 29, 53, 66]. 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. On the other hand, as a transformation language, POET target automatic program transformation for general-purpose applications beyond those supported by existing domain-specific frameworks. Our previous work has used POET to empirically tune the performance of several linear algebra kernels and SPEC benchmarks [54, 72, 74].

Empirical performance tuning has been increasingly adopted by optimizing compilers in recent years [30, 38, 49, 50, 56, 61]. POET supports existing iterative compilation frameworks by providing a transformation engine which enables collective parameterization of advanced compiler optimizations. Previous research has studied the parameterization of a number of compiler optimizations, including loop blocking, unrolling [38, 50, 61], software pipelining [47], and loop fusion [57]. The work by Cohen, et al. [20] used the polyhedral model to parameterize the composition of loop transformations applicable to a code fragment. Our work is different from the work by Cohen, et al. in that we parameterize the configuration of each individual transformation instead of parameterizing the overall combined optimization space. Our research focuses on composing these parameterized transformations in a well-coordinated fashion without intermediate program analysis support. Our POET transformation engine can be easily extended to collaborate with different independent search and modeling techniques [18, 55, 65, 75] in auto-tuning.

Existing compiler research has developed a large collection of compiler optimization techniques for improving the performance of scientific applications [17, 20, 39, 44, 51, 63, 68]. POET can be used as an alternative output language of these optimizations. Note that POET focuses on supporting the collective parameterization and composition of individual program transformations, e.g., various loop transformations, after the safety of these transformations is already determined by sophisticated optimization analysis, such as those by the Polyhedral framework [14] or other frameworks [4, 14, 41, 52, 67] based on integer programming tools. POET currently does not directly support any sophisticated program analysis, although extensions can be made in the future to support various analysis abstractions. The loop optimization framework within the ROSE compiler is based on an optimization technique called dependence hoisting [70], which does not use integer programming.

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. Our existing work has extended the ROSE compiler [73] to automatically produce parameterized POET scripts as output [69]. Other source-to-source optimization frameworks, such as the Polyhedral framework [14], the Paralax infrastructure [64], the Cetus compiler [22], the Open64 compiler [7], among others, can be similarly extended to use POET as an alternative optimization output language.

Similar to POET, various annotation languages such as OpenMP [21] and the X language [23] also aim at supporting programmable control of compiler optimizations. The work by Hall et al. [34] allows developers to provide a sequence of loop transformation Recipes to guide transformations performed by an optimizing compiler. The X language [23] uses C/C++ pragma to guide the application of a pre-defined collection of compiler optimizations. These languages serve as a programming interface for developers to provide
additional inputs to the optimizing compiler. In contrast, 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.

9 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


