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 CCF0747357 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 unparse the transformation result. The script can be extensively parameterized and reconfigured via command-line options when invoking the transformation engine.

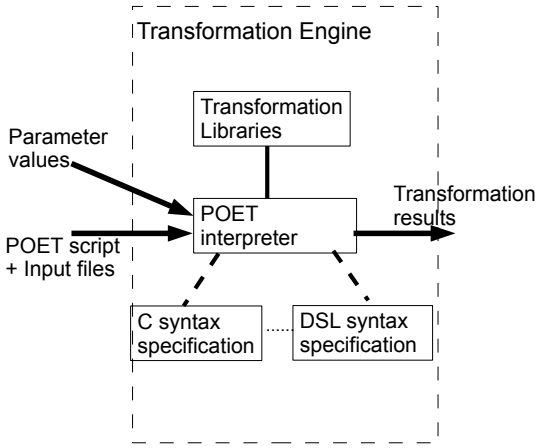

Figure 1: POET transformation engine

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

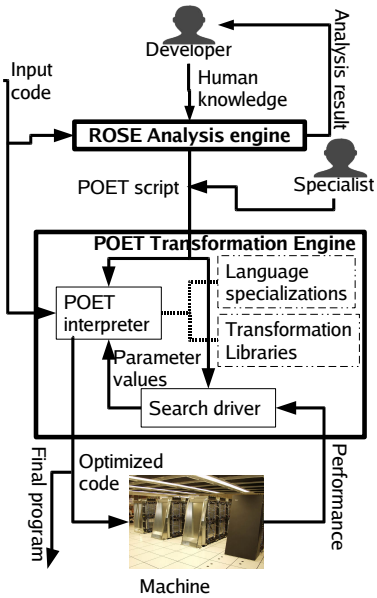

Figure 2: Optimization Environment

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	
ParallelizeLoop(x)	Parallelize the outermost loop using OpenMP in input code x
DistributeLoops(n, x)	Distribute input code x so that fragments in n end up in separate components
FuseLoops(n, p, x)	Fuse disjoint loops in n into a single one; then use it to replace fragment p in input code x
BlockLoops(n,x)	Block the loops nested outside of fragment n but inside input code x
PermuteLoops(n,x)	Permute the loops nested outside of fragment n but inside input code x
SkewLoops(n1,n2,x)	Use the outer loop $n1$ to skew the inner loop $n2$ within input code x
CopyRepl(v, a, d, x)	Use buffer v to copy and replace memory referenced by a at loop iterations d in input code x
CleanupBlockedNests(x)	Cleanup the blocked loop nests via loop splitting in input code x
Scalar and register performance optimization	
UnrollLoops(n,x)	Unroll the loops nested outside of fragment n and inside input code x
UnrollJam(n,x)	Unroll the loops outside of fragment n and inside input code x; Jam the unrolled loops inside n
ScalarRepl(v, a, d, x)	Use scalars named v to replace memory referenced by a at loop iterations d in input code x
FiniteDiff(v,e,d, x)	Use loop induction variables v to reduce the cost of evaluating expression $e + d$ in input code x
VectorizeCode(v,n,x)	Apply SSE Vectorization to loop n inside input code x based on vector register assignment v
Prefetch(a,n,i,x)	Prefetch memory address a with increment i at each iteration of loop n in input code x
High-level to low-level code translation	
ArrayAccess2PtrRef(x)	Convert all array references in input code x to pointer references
TransformThreeAddress(x)	Transform input code x into three address code
TransformTwoAddress(x)	Transform input code x into two address code

Table 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[1]. The boolean value $false$ is represented using integer 0, and $true$ can be represented using any of

[1]POET does not support floating point values under the assumption that program transformation does not need floating point evaluation. The language may be extended in the future if the need arises.

	Types of values	
1	atomic types	int (e.g., 1, 20, -3), string (e.g., “abc”, “132”)
2	compound types	list, tuple, associative map, code template, xform handle
	Compound type construction	
3	e_1 e_2 ... e_n	A list of n elements e_1, e_2, ..., e_n
4	e_1, e_2, ..., e_n	A tuple of n elements e_1, e_2, ..., e_n
5	MAP{f_1=>t_1, ..., f_n=>t_n}	An associative map of n entries which map f_1 to t_1, ..., f_n to t_n respectively
6	c # (p_1, ..., p_n)	A code template object of type c with p_1, ..., p_n as values of its parameters
7	f [v_1=p_1; ...; v_n=p_n]	A xform handle f with p_1, ..., p_n as values for optional parameters v_1, ..., v_n
	Operating on different types of values	
8	+,−,*,/,%,<,<=,>,>=,==,!=	Integer arithmetics and comparison
9	!, &&, ||	Boolean operators
10	==, !=	Equality comparison between arbitrary types of values
11	$a \wedge b$	Concatenate two values a and b into a single string
12	SPLIT(p,a)	Split string a into substrings separated by p; if p is an int, split a at location p
13	a :: b	Prepend value a in front of list b s.t. a becomes the first element of the new list
14	HEAD(l), car(l)	The first element of a list l
15	TAIL(l), cdr(l)	The tail behind the first element of a list l; returns “” if l is not a list
16	a[b] where a is a tuple	The bth element of a tuple a
17	a[b] where a is a map	The value mapped to entry b in an associative map a
18	a[c.d] where c is a code template	The value stored in parameter d of a code template object a, which has type c
19	LEN(a) where a is a string	The number of characters in string a
20	LEN(a) where a is not a string	The number of entries in the list, tuple, or map; returns 1 otherwise
	Variable assignment and control flow	
21	$a = b$	Modify a local or static variable a to have value b; return b as result of evaluation
22	$a[i] = b$	Modify associate map a so that entry i is mapped to value b; return b as result
23	$(a1, ..., am) = (b1, ..., bm)$	Modify $a1, ..., am$ to have values $b1, ..., bm$ respectively; return the b tuple
24	$a1$; $a2$; ... ; am	Evaluate expressions $a1$ $a2$ ... am in order; return the result of evaluating am
25	RETURN a	Evaluate expression a and then return it as result of the current *xform* invocation
26	if (a) { b } [else {c}]	If conditional, returns b or c as result based on whether a is TRUE or FALSE
27	switch(a){case b_1:c_1 ... default:c_n}	Similar to the *switch* conditional in C, returns value of the first successful branch
28	for ($e1$; $e2$; $e3$) { b }	Equivalent to the *for* loop in C; always return empty string
29	BREAK, CONTINUE	Equivalent to the *break* and *continue* statements in C; used only in loops
	Global type/variable declarations and commands	
30	<define a b />	Declare a global macro variable a and assign b as its value
31	<trace a1,...,am />	Declare a list of related trace handles $a1$,...,am
32	<parameter p type=t default=v parse=r message=d/>	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.
33	<input cond=c from=f syntax=s to=t />	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
34	<eval s1,...,sm />	Evaluate the group of expressions/statements $s1$, ..., sm
35	<output from=t to=f syntax=s cond=c />	If expression c evaluates to true, unparse the expression t to file f using syntax descriptions defined in file s.

Table 2: Overview of the POET language

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.

```
1: include utils.incl

2: <*The code template type for all loops supported by the POET optimization library *>
3: <code Loop pars=(i:ID, start:EXP, stop:EXP, step:EXP) >
4: for (@i@=@start@; @i@<@stop@; @i@+=@step@)
5: </code>
6: <xform ReverseList pars=(list) prepend=""> <<* a xform routine which reverses the input list
7:   result = HEAD(list) :: prepend;
8:   for (p_list = TAIL(list); p_list != ""; p_list = TAIL(p_list))
9:    {
10:     result = HEAD(p_list) :: result;
11:  }
12:  result
13: </xform>

14:<define OPT_STMT CODE.Loop />
15:<parameter inputFile message="input file name"/>
16:<parameter inputLang default="" message="file name for input language syntax" />
17:<parameter outputFile  default="" message="file name for output" />
18:<trace inputCode/>

19:<input cond=(inputLang!="") from=(inputFile) syntax=(inputLang) to=inputCode/>
20:<eval  backward = ReverseList[prepend="Reversed\n"](inputCode);
          succ = XFORM.AnalzeOrTransformCode(inputCode);  />
21:<output cond=(succ) to=(outputFile) syntax=(inputLang) from=inputCode/>
```

Figure 3: An example illustrating the overall structure of a POET file

- *Associative Maps.* POET *maps* are used to associate pairs of arbitrary values and are constructed by invoking the *MAP* operator at line 5 of Table 2. For example, *MAP*{*3=> "abc"*} builds a map that associates 3 with "abc". The content of associative maps can be dynamically modified using assignments, shown at line 22 of Table 2.
- *Code Templates.* Each POET code template is a distinct user-defined data type and is constructed using the # operator at line 6 of Table 2. For example, given the *Loop* code template declaration at line 3 of Figure 3, *Loop#*("*i*", 0, "*N*", 1) builds an object of the code template *Loop* with ("*i*", 0, "*N*", 1) as values for the template parameters. POET code templates are used to construct arbitrarily linked data structures (e.g., trees) and to support the dynamic parsing and internal representation of arbitrary languages. For more details, see section 4.
- *Xform Handles.* Each POET xform handle is a reference to a global POET *xform routine*, which are equivalent to global functions in C, and is constructed by following the name of the *routine* with an optional list of pre-configurations to set up future invocations of the routine, shown at line 7 of Table 2. Each *xform handle* can be invoked by simply following it with a tuple of actual parameters. More details of POET xform routines are provided in Section 3.3.

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*,*start*,*stop*,*step* are local variables of the code template *Loop*, and *list*,*result*, and *p_list* are local variables of the *xform* routine *ReverseList*. Local variables are introduced by declaring them as parameters or simply using them in the body of a code template or xform routine.
- *Static variables*, whose 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 *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 *Loop* which has four parameters (data fields) named *i*, *start*, *stop*, and *step* respectively. Lines 7-14 define a *xform routine* named *ReverseList* which has a single input parameter named *list* and an optional parameter named *prepend*, which has empty string as default value. An example invocation of the *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 *macro* named *OPT_STMT* and assigns the code template type *Loop* as its value. Lines 15-17 declare three *command-line parameters* named *inputFile*, *inputLang* and *outputFile*, whose values can be redefined via command-line options. Line 18 declares a *trace handle* named *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, *input*, *eval*, and *output*, illustrated at lines 19-21 of Figure 3. In particular, the *input command* at line 19 is used to parse a file named by variable *inputFile* using syntax descriptions contained in a file named by *inputLang*. The parsing result is then converted into an internal representation and stored to variable *inputCode*. The *eval command* at line 20 specifies a sequence of expressions and statements to evaluate. The *Output Command* at line 21 is used to write the transformed internal representation (i.e., *inputCode*) to an external file named by *outputFile*.

All POET expressions and statements must be embedded inside an *eval* command or the body of a code template or *xform* routine. Most POET expressions are *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 *input* command, where both the concrete syntax and the internal representation of the input language are collectively specified using a collection of *code templates* defined in an external file. These code template specifications are interpreted and matched against the input code in a top-down recursive descent fashion at runtime, and an abstract syntax tree (AST) representation of the input code is automatically constructed as result of the *input* command. This approach is more flexible than the conventional parser generation 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
pars=(v_1:p_1,...,v_m:p_m)	Template parameters which specify data fields of the corresponding internal representation
parse=p1	Use $p1$ as the alternative concrete syntax to substitute the template body for parsing
output=p1	Use $p1$ 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
match=c	Allow c to be used in place of the code template when matching against the input code
Type specifiers for tokens and compound data objects	
INT	The integer type, which includes all integer values
STRING	The string type, which includes all string values
ID	The identifier type, which includes all string values that can be used as identifiers in POET
CODE	The code template type, which includes all code template objects
XFORM	The xform handle type, which include all xform routine handles
TUPLE	The tuple type, which includes all POET tuples
MAP	The associative map type, which includes all POET associative maps
-	The *ANY* type, which includes all values supported by POET
lb .. ub	The *range* type, which includes all integers >= *lb* and <= *ub*
Parsing specifiers which define how to match concrete syntax specifications with leading input tokens	
a constant value	Match the first token with the given value, e.g. 3, 5, 137, "abc", "de3f7"
a token type specifier	Match the first token with the type specifier
~p	Match the first token with anything but the parsing specifier p
a code template name	Parse leading tokens using the corresponding code template
EXP	Parse leading tokens as an expression, using the built-in expression parser within POET
v = p	Parse the leading tokens using parsing specifier p, then save the parsing result to variable v
p ...	Parse the leading tokens as a list of 0 or more components, each parsed using parsing specifier p
p	Parse the leading tokens as a list of 1 or more components, each parsed using parsing specifier p
LIST(p,s)	Parse the leading tokens as a list of 0 or more parsing specifier p separated by a constant string s
TUPLE(p)	Parse the leading input tokens using parsing specifier p, then return a tuple as result
p1 p2 ... pm	Parse the leading input tokens as m consecutive parsing specifiers p1, ..., pm
p1 | p2 | ... | pm	Parse the leading input tokens using one of the m alternative specifiers p1,...,pm

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.

```
1: Nest : Ctrl SingleStmt
2: Ctrl : If | While | For | Else
3: If : "if" "(" exp ")"
4: While : "while" "(" exp ")"
5: For : "for" "(" exp ";" exp ";" exp ")"
6: Else : "else"
7: SingleStmt : EmptyStmt|Break|Continue|Return|StmtBlock|SwitchStmt|Nest|VarDeclStmt|Comment|ExpStmt
8: StmtBlock : "{" StmtList "}"
9: StmtList :  | SingleStmt StmtList
```

(a) Syntax specification using BNF

```
1: <code Nest pars=(ctrl : CODE.Ctrl, body : CODE.SingleStmt) >
   @ctrl@
     @body@
   </code>
2: <code Ctrl parse=CODE.If|CODE.While|CODE.For|CODE.Else match=CODE.Loop|CODE.If|CODE.While|CODE.For|CODE.Else />
3: <code If pars=(condition:EXP) >
   if (@condition@)
   </code>
4: <code While pars=(condition:EXP) >
   while (@condition@)
   </code>
5: <code For pars=(init:EXP|"",test:EXP|"",incr:EXP|"")  rebuild=(RebuildLoop(init,test,incr)) >
   for (@init@; @test@; @incr@)
   </code>
6: <code Else ifNest=INHERIT>  else  </code>
7: <code SingleStmt parse=CODE.EmptyStmt|CODE.Break|CODE.Continue|CODE.Return|CODE.StmtBlock|CODE.SwitchStmt|
                          CODE.Nest|CODE.StaticDecl|CODE.VarDeclStmt|Stmt|CODE.Comment|CODE.ExpStmt/>
8: <code StmtBlock pars=(stmts:CODEStmtList) >
   {
      @stmts@
   }
   </code>
9: <code StmtList parse=LIST(SingleStmt,"\n")/>
```

(b) Syntax specification using code templates

Figure 4: Syntax specifications for a subset of the C language

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 ...> β </code>, where the template name corresponds to the left-hand non-terminal A, and the template body corresponds to the right-hand side β. A template parameter in the format of *a:t* is created for each non-constant symbol t within β, 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 *Ctrl* symbol at line 1 of Figure 4(a) is translated to template parameter *ctrl:CODE.Ctrl* in (b), where *CODE.Ctrl* declares *Ctrl* as a new POET code template name that will be defined later, 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

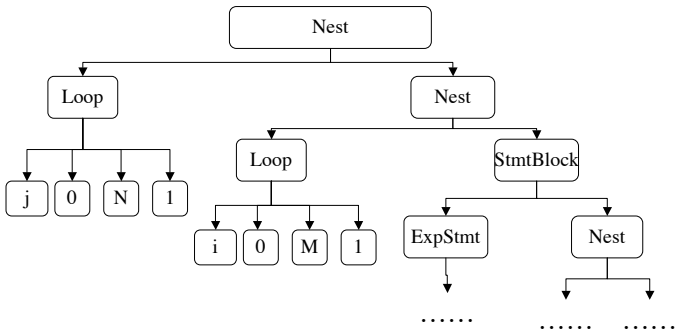

Figure 5: The AST built after parsing line 6 of Figure 6 using code templates in Figure 4

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

```
1: /*@ BEGIN(gemm=FunctionDecl) @*/
2: 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)
3:  {
4:
5:    int i,j,l;                                                  //@=>gemmDecl=Stmt
6:    for (j = 0; j < N; j += 1)                                  //@ BEGIN(gemmBody=Nest) BEGIN(nest3=Nest)
7:    {
8:       for (i = 0; i < M; i += 1)                               //@ BEGIN(nest2=Nest)
9:       {
10:         C[j*ldc+i] = beta * C[j*ldc+i];
11:         for (l = 0; l < K; l +=1)                            //@ BEGIN(nest1=Nest)
12:         {
13:            C[j*ldc+i] += alpha * A[l*lda+i]*B[j*ldb+l];
14:         }
15:     }
16:   }
17: }
```

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

```
parse(tokens,goal,inherit)
    tokens: the input token stream;
    goal: the parsing specifier to match;
    inherit: the previous parsing result;
(1) /*extract the first token from the input stream and
    and preprocess input annotations */
    tok1 = first_token(tokens);
    if (is_singleline_annotation(tok1)) then
        input1 = new_input_stream(annot_input(tok1));
        tok1=parse(input1,annot_goal(tok1),inherit);
        if (input1 is not empty) then
            error("incorrect annotation", tok1);
        tokens = prepend(tok1, forward(tokens));
    else if (is_multiline_annotation(tok1))
        tokens = prepend(annot_input(tok1), forward(tokens));
        tok1 = parse(tokens, annot_goal(tok1), inherit);
        tokens = prepend(tok1, tokens);
(2) /* Match parsing specifier against the input */
(2.1) if (goal is a constant value) then
        if (goal==empty) return goal;
        else if (goal ≠ tok1) then parse_error(tokens,goal);
        else forward(tokens); return tok1;
(2.2) else if (goal is a code template name) then
        if (match_type_succ(tok1,goal)) then
            forward(tokens); return tok1;
        syntax = codeTmpl_parseAttr_or_body(goal);
        push_symbol_table(goal);
        parse(tokens,syntax,inherit);
        res = build_codeTmpl_obj(goal,inherit);
        pop_symbol_table(goal);
        return res;
(2.3) else if (goal is a template parameter) then
        res=parse(tokens,var_constr(goal),inherit);
        set_variable_value(goal, res);
        return res;
(2.4) else if (goal is a built-in type specifier)
        if (match_type_succ(tok1, goal)) then
            forward(tokens); return tok1;
        else parse_error(tokens,goal);
(2.5) /* goal is a built-in operator */
    else if (goal is an assignment v = p) then
        res=parse(tokens,p,inherit);
        set_variable_value(v, res);
        return res;
    else if (goal is EXP) then
        return parse_exp(tokens);
    else if (goal is a list operator LIST(p, s)) then
        return parse_list(tokens, p, s);
    else if (goal is a tuple operator TUPLE(p)) then
        return parse_tuple(tokens, p);
(2.6) else if (goal is a list of parsing specifiers)
        res = empty;
        for (each list component elem in goal) do
            curRes = inherit = parse(tokens, elem, inherit);
            res = append_list(res, curRes);
        return res;
(2.7) else if (goal is the alternative | operator) then
    for (each alternative alt ∈ goal) do
        if (match_lookahead(alt, tok1)) then
            return parse(tokens,alt,inherit);
    parse_error(tokens,goal);
```

* *new_input_stream*(*input*): create a new input token stream from a list of tokens *input*;
prepend(*tok*1, *tokens*): prepend a list of tokens in *tok*1 at the start of the input token stream *tokens*;
forward(*tokens*): remove the first token from the input stream *tokens*;
match_type_succ(*tok*1, *goal*): match AST *tok*1 against *goal*, which is INT,STRING,ID,or a code template name;
push_symbol_table(*goal*): create a new symbol table to store values of local variables for code template *goal*;
pop_symbol_table(*goal*): remove the mostly recently created new symbol table for code template *goal*;
build_codeTmpl_obj(*goal*, *inherit*): create a new object of code template *goal* based on values in its symbol table;
match_lookahead(*alt*, *tok*1): return whether *tok*1 matches the first token of the parsing specifier *alt*.

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 *tok*1 could be a single input token (e.g., a string or an integer) or a code template object which is the result of processing an input annotation. Step (2) of the algorithm then continues by matching the top-level parsing specifier *goal* with *tok*1 followed by the rest of the input stream. In particular, the algorithm 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, *tok*1 is removed from *tokens*) if and only if the value of *goal* matches that of *tok*1.

(2.2) *goal* is the name of a code template. If *tok*1 is already an object of the given code template (the result of processing input annotations), the object is returned as result after removing *tok*1 from the input stream; otherwise, the input stream is matched against the syntax definition (i.e., the *parse* attribute or template body) of the given code template, and if the matching is successful, an object of the given code template is created based on values of the template parameters saved during the parsing process (using a temporary symbol table created before parsing).

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

(2.4) *goal* is a token type specifier such as *INT, STRING, ID*. Here *tok*1 is compared with the given type specifier and returned as the parsing result if the matching succeeds.

(2.5) *goal* is a built-in operator such as *assignment, EXP, LIST, and TUPLE*, shown in Table 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 *tok*1 to predictively determine which specifier to use. If any of the alternative specifiers can potentially match *tok*1 as the first input token, the specifier is used to recursively invoke the *parse* algorithm, and the parsing result is returned as the result of the whole parsing process; otherwise (none of the matching succeeds), an error is reported.

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

Macros for reconfiguring behavior of the *input* command	
TOKEN	Reconfigure the POET internal tokenizer to treat a list of parsing specifiers as single tokens
KEYWORDS	Reconfigure POET dynamic parser to treat a list of string literals as keywords of the input language
PREP	Reconfigure POET dynamic parser to invoke a *xform handle* to filter the input tokens before parsing
BACKTRACK	Reconfigure POET dynamic parser to enable/disable backtracking during parsing
PARSE	Reconfigure POET dynamic parser to use a given parsing specifier as goal to parse all input code
Macro for reconfiguring behavior of the *output* command	
UNPARSE	Reconfigure the POET unparser to invoke a given *xform handle* to post-process (reformat) output tokens
Macros for reconfiguring the internal expression parser (i.e., the support of the *EXP* parsing specifier)	
EXP_BASE	Use a given parsing specifier for base cases of expressions
EXP_BOP	Accept a given list of binary operators (in increasing order of precedence) within expressions
EXP_UOP	Accept a given list of unary operators within expressions
PARSE_CALL	Use a given code template as the internal representation of a function call when parsing expressions
PARSE_ARRAY	Use a given code template as the internal representation of an array access operation
PARSE_BOP	Use a given code template as the internal representation of all binary operators
PARSE_UOP	Use a given code template as the internal representation of all unary operators
BUILD_BOP	Invoke a given xform handle to rebuild internal representations of binary operators
BUILD_UOP	Invoke a given xform handle to rebuild internal representations of unary operations

Table 4: Macros that can be used to reconfigure the default behavior of POET

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 necessary[2].

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.

[2]Note that left-recursion is not allowed, as required by all top-down predictive parsers

```
1: <*Convert every pair of  "+""+" into a single "++" token, every pair of  "-" "-" into "--", and so forth*>
   <define TOKEN (("+" "+") ("-" "-") ("=""=") ("<""=") (">""=") ("!""=")("+""=") ("-""=") ("&""&") ("|""|")
               ("-"">") ("*""/") CODE.FLOAT CODE.Char CODE.String)/>
2: <*Treat "case","for" etc. as reserved words so that they cannot be matched to the STRING specifier*>
    <define KEYWORDS ("case" "for" "if" "while" "float")/>
3: <*Use the code template DeclStmtList as the start non-terminal (top-level parsing specifier) for parsing*>
   <define PARSE CODE.DeclStmtList/>
4: <define BACKTRACK FALSE/>  <<* disable backtracking when parsing C/Fortran code
5: <*Invoke the ParseLine routine to preprocess each line of the input code (used to parse Fortran code)*>
   <define PREP XFORM.ParseLine[comment_col=6;text_len=70] />
6: <*invoke the UnparseLine routine to post-process the token stream before output tokens to external files
    (Used to place tokens at the proper columns in Fortran)*>
   <define UNPARSE XFORM.UnparseLine/>
7: <*The base case of an expression can be an integer, a float, a string, a char, or a variable reference*>
   <define EXP_BASE INT|CODE.FLOAT|CODE.String|CODE.Char|CODE.VarRef />
8: <*Binary operators in C in increasing order of precedence. All operators are treated left-associative*>
    <define EXP_BOP ( ("=" "+=" "-=" "*=" "/=" "%=") ("&" "|") ("&&" "||") ("==" ">=" "<=" "!=" ">" "<")
                        ("+" "-") ("*" "%" "/") ("." "->")) />
9: <*Unary operators in C; All operators use prefix notation*>
    <define EXP_UOP ("++" "*" "&" "~" "!" "+" "-" "new")/>
10:<*Use code templates FunctionCall and ArrayAccess to build internal representations for function calls
     and array references respectively*>
   <define PARSE_CALL CODE.FunctionCall/>
   <define PARSE_ARRAY CODE.ArrayAccess/>
11:<*Use code templates Bop and Uop to build internal representations for binary and unary operators*>
   <define PARSE_BOP CODE.Bop/>
   <define PARSE_UOP CODE.Uop/>
12:<*Invoke xform routines BuildBop and BuildUop to rebuild binary and unary operators*>
   <define BUILD_BOP XFORM.BuildBop/>
   <define BUILD_UOP XFORM.BuildUop/>
```

Figure 8: Example: using macros to reconfigure the default behavior of POET

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 *type*1 and *type*2 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
(CLEAR v)	All values; variable v is assigned with the value after matching
v = p	All values that can match pattern specifier p; variable v is assigned with the value after matching
c # p	All objects of code template c with parameter values matching pattern specifier p
(p1, p2, ..., pn)	All tuples of n elements which match the pattern specifiers $p1$, $p2$, ..., pn respectively
(p1 p2 ... pn)	All lists of n elements which match the pattern specifiers $p1$, $p2$, ..., pn respectively
p1 :: p2	All lists with the first element matching pattern $p1$ and rest of the list matching pattern $p2$
p1 op p2	All expressions built using the given binary *op* (e.g., +,-,*,) to combine patterns $p1$ and $p2$
p1 | p2 | ... | pn	All values that can match one of the pattern specifiers $p1$, $p2$, ..., pn
Pattern matching operators	
a : b	Return whether value a matches the pattern specifier b
switch (a) { case b1: ... case bn: ... default: ...}	Match value a against pattern specifiers $b1$, ..., bn in turn, evaluate the matching branch; if all matches fail, evaluate the *default* branch.
AST traversal operators	
foreach(a : b : c) { d }	Traverse and match all values embedded within a against pattern specifier b; for each value x that can successfully match b, evaluate expressions d and then c; if c evaluates to true, skip the inside of x and continue; otherwise, continue traversing inside x to find more matches.
foreach_r(a : b : c) { d }	Same as the *foreach* operator, except that values within a are traversed in reverse order

Table 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:           else 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:  }
</xform>
```

Figure 9: Example: type checking for simple expressions

uninitialized variables, $exp1$ and $exp2$, so if the pattern matching succeeds, $exp1$ and $exp2$ will be assigned with the second and third parameters of the *Bop* code template object respectively. Therefore the pattern matching operations can be used not only for dynamic type checking, but also for initializing and assigning values to 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

```
1: <code BasicBlock pars=(label, stmts)>
2: @label@ : @stmts@
3: </code>

4: <xform BuildBasicBlocks pars=(input)>
5:    blocks = MAP{};
6:    foreach (input : (cur=Nest#(CLEAR loop, CLEAR body)) : FALSE)
7:      { blocks[cur] = blocks[body]=1; }
8:    start=""; curBlock = ""; count=0;
9:    foreach (input :  (cur=_) : FALSE) {
10:     if (blocks[cur] == 1) {
11:        if (curBlock != "") {
12:           blocks[start]=BasicBlock#(count, ReverseList(curBlock));
13:           count = count + 1; curBlock = "";
14:        }
15:        start=cur;
16:     }
17:     if (cur : Loop|ExpStmt) { curBlock = cur :: curBlock; }
18:  }
19:  if (curBlock!="")  blocks[start] = BasicBlock#(count+1, curBlock);
20:  blocks
21:</xform>
```

Figure 10: Example: identifying basic blocks for a simple loop-based language

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

```
1: <code GraphEdge pars=(from,to) >  "@(from)@"->"@(to)@"  </code>

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

Figure 11: Example: control-flow analysis for a simple loop-based language

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

Transformation Operators	
REPLACE(c1,c2,e)	Replace all occurrences of $c1$ with $c2$ in AST e
REPLACE(((o1,r1)...(om,rm)), e)	Traverse AST e in pre-order to locate and replace each o_i (i=1,...,m) with r_i
REBUILD(e)	Rebuild AST e by invoking the *rebuild* attribute of each code template object inside e
DUPLICATE(c1,c2,e)	Replicates AST e with copies, each copy replacing $c1$ by a different component in $c2$
PERMUTE((i1,i2,...,im),e)	Reorder list e s.t. the jth (j=1,...,m) element is located at i_j in the result
Tracing operators	
INSERT (x, e)	Insert all the trace handles rooted at x to be embedded within AST e
ERASE(x, e)	Remove all the occurrences of trace handle x from the input AST e
COPY(e)	Remove all trace handles in AST e and return the result
TRACE((x1,...,xm), e)	Treat variables $x1, ..., xm$ as trace handles during the evaluation of expression e
SAVE (v1,v2,...,vm)	Save the current values of trace handles v1,v2,...,vm to be restored later
RESTORE (v1, v2, ..., vm)	Restore the previous values saved for the trace handles $v1, ..., vm$
Evaluation operators	
DELAY {e}	Delay the evaluation of expression e until later
APPLY (e)	Force the evaluation of a delayed expression e

Table 6: Built-in POET operators for support program transformation

Index	Expressions	Evaluation result
1	REPLACE("x","y", SPLIT("","x*x-2"))	"y" "*" "y" "-" 2
2	REPLACE((("a",1) ("b",2) ("c",3)), SPLIT("","a+b-c"))	1 "+" 2 "+" 3
3	REPLACE((("a",1) ("b",2) ("c",3)), Bop#("+","a",Bop#("-","b","c")))	Bop#("+",1,Bop#("-",2,3))
4	REBUILD(Bop#("+", 0, Bop#("-",2,3)))	-1
5	REBUILD(Bop#("+", 1, Bop#("-",a,3)))	Bop#("-", a, 2)
6	DUPLICATE("var", (1 2 3), Stmt#"var")	Stmt#1 Stmt#2 Stmt#3
7	PERMUTE((3 2 1), ("a" "b" "c"))	("c" "b" "a")

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.

```
1: <*Erase handle from exp. Return the erased handle and modified exp*>
2: <xform EraseTraceHandle pars=(handle, exp)>
3:    for (handlevalue=handle; handlevalue:VAR; handlevalue=ERASE(handlevalue,handlevalue)){
4:       exp = ERASE(handlevalue, exp); <<* handlevalue is another trace handle
5:    }
6:    (handlevalue, exp)
7: </xform>

8:  <* Replace handle with newvalue within trace; if trace is empty, modify handle to contain newvalue*>
9: <xform ModifyTraceHandle pars=(handle, newvalue) trace="">
10:  if (trace : VAR || trace != "") {
11:    (handlevalue, trace) = EraseTraceHandle(handle, trace);
12:    REPLACE(handlevalue, newvalue, trace)
13:  }
14:  else {
15:   (handlevalue, newvalue) = EraseTraceHandle(handle, newvalue);
16:   REPLACE(handlevalue, newvalue,handle)
17:  }
18:</xform>

   <* Append a new type declaration, (type vars) at the end of the input decl *>
19:<xform AppendDecl pars=(type, vars, decl)>
20:    ndecl=decl;
21:    for (p_vars=vars; (cur = HEAD(p_vars)) ;  p_vars = cdr p_vars) {
22:       foreach (cur : (p = STRING|Name|ArrayAccess) : TRUE)
23:           ndecl = ndecl :: VarDeclStmt#(type,p);
24:    }
25:    ModifyTraceHandle(decl,ndecl)
26:</xform>

   <* Replace all occurrences of array references with equivalent pointer references *>
27:<xform ArrayAccess2PtrRef pars=(input) >
28:   repl="";
29:   foreach_r (input : (d = ArrayAccess#(CLEAR arr,CLEAR sub)) : FALSE) {
30:      cur = (d, VALUE#(Bop#("+",arr,sub))) ;
31:      repl = cur :: repl;
32:   }
33:  REPLACE(repl, input)
34:</xform>
```

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 *REBUILD* operator is invoked to simplify an input AST after it has been recently modified, e.g., with some fragments replaced with empty strings. In particular, when invoking *REBUILD* on an input AST, all the code template objects within the AST are traversed in post-order, and if a code template object has a pre-defined *rebuild* attribute (see Table 3), the *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 *REBUILD* operator can be used to evaluate the expression and return the evaluation result. Of course, the effectiveness of the expression evaluation depends on details of the *rebuild* attribute defined within the *Bop* code template, a type defined in our POET *opt* library.

The *DUPLICATE* and *PERMUTE* operators are provided to support special needs of replicating and reordering fragments of an input code. They are used to implement the loop unrolling and interchange transformations shown in Table 1.

6.2 Tracing of Transformed Code

POET uses a special concept called *trace handles* to automatically keep track of various fragments of an input code as they go through different transformations. These trace handles can be embedded inside an input AST and therefore be modified within *xform routines* even if the routines cannot directly access them through their names. In particular, *xform routines* can invoke the built-in transformation operators shown in Table 6 to effectively keep all embedded trace handles up-to-date. POET dedicates six built-in operators, *INSERT, ERASE, COPY, TRACE, SAVE,* and *RESTORE*, shown in Table 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 *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 *gemm*, *gemmDecl*, *gemmBody*, *nest*3, *nest*2, and *nest*1 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 *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 *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 *ERASE* operator. In particular, *ERASE(handlevalue, handlevalue)* at line 3 peels off the outermost trace handle contained in *handlevalue* by returning its value, and *ERASE(handlevalue, exp)* at line 3 removes the trace handle contained in *handlevalue* from *exp* and returns the new

```
1: include opt.pi
2: <trace gemm,gemmDecl,gemmBody,nest3,nest2,nest1/>
3: <input to=gemm syntax="Cfront.code" from="gemm.c" />

4: < parameter mu type=1..MB default=4
   message="Unroll and Jam factors for nest3" />
5: < parameter nu type=1..NB default=1
   message="Unroll and Jam factors for nest2" />
6: ......
7: <eval nest3_UnrollJam = DELAY {
8:        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:/>
```

(a) Transformation definitions

```
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/>
```

(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 unparse 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 unparse the input C code using Fortran syntax.

```
1: <parameter inputFile default="" message="input file name" />
2: <parameter outputFile default="" message="output file name" />

3: <input from=inputFile syntax="Cfront.code" to=inputCode/>
4: <output to=outputFile syntax="C2F.code" from=inputCode/>
```

Figure 14: POET script for translating C syntax to Fortran

```
<code Nest pars=(ctrl , body:StmtList) >
@ctrl@
   @body@
@(switch(ctrl)
{
 case Loop|While: "enddo"
 case If : "endif"
})@
</code>

<code Loop pars=(i,start, stop, step)>
do @i@=@start@,@(stop-1)@,@step@
</code>
<code While pars=(condition) >
while (@condition@) do
</code>
<code If pars=(condition) >
if (@condition@) then
</code>
```

Figure 15: Example code template definitions in C2F.code

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.

```
1: include utils.incl
2: <define PARSE CODE.CommandList/>
3: <code CommandList parse=LIST(CODE.ParamDecl|CODE.ThrowAway, "\n")/>
4: <code ThrowAway pars=(anything:_) rebuild=""> @anything@ </code>
5: <code ParamDecl pars=(param:CODE.NumOfThreads|CODE.BlockFactor|...))>
    <parameter @param@/>
   </code>
6: <code NumOfThreads pars=(parName:Name, loopNest:Name, defval:INT)>
   @parName@ type=1.._ default=@defval@ message="number of threads to parallelize loop @loopNest@"
   </code>
8: <code BlockDimList pars=(spec:LIST("INT"," ")) rebuild=((spec:STRING)?1:LEN(spec))> @spec@ </code>
9: <code BlockFactor pars=(parName:Name, loopNest:Name, dim:BlockDimList, defval:LIST(INT," "))>
   @parName@ type=(@dim@) default=(@defval@) message="Blocking factor for loop nest @loopNest@"
   </code>
10:   ......
```

Figure 16: Syntax specification for parsing standardized POET optimization parameters

```
<code CommandList parse=LIST(ParamDecl, "\n")/>
<code IntList parse=LIST(INT," ")/>
<code ParamDecl pars=(param)>
  @param@
  </code>
<code NumOfThreads pars=(parName:Name,loopNest:Name, defval:INT)>
  1 @parName@ N 1 16 1 @defval@
  </code>
<code BlockFactor pars=(parName:Name, loopNest:Name, dim:INT, defval:IntList)>
  @dim@ @parName@ N 8 128 8 @defval@
  </code>
```

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

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

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

Figure 18: Interface specification for the routine in Figure 6

```
<code StaticBufferAllocate pars=(type,name,size,align,nrep)>
@name@_size=@TimerAlignSize#(size,align)@; @ (if (nrep > 1) { @
@name@_rep=CacheSZ / @name@_size + 1; @})@
</code>

<code Static2DBufferAllocate pars=(type,name,size,size2,align,nrep)>
@name@_size=@TimerAlignSize#(size,align)@; @ (if (nrep > 1) { @
@name@_rep=CacheSZ / @name@_size + 1; @})@
@name@_size2=@TimerAlignSize#(size2,align)@;
</code>

<code TimerBufferInitialize pars=(name, nrep, value, valueIncr)>
@(ivar=PT_ivar#0; "")
@for (@ivar@=0; @ivar@<@name@_size @((nrep>1)? ("*" name "_rep"):"")@; ++@ivar@)
{
  @name@_buf[@ivar@] = @value@; @ ((valueIncr=="")?"":(@
  @valueIncr))@
}
@name@ = @name@_buf;
</code>
```

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 allocation and initialization to define the structure of the auto-generated code instead of using lower-level concepts such as C/Fortran expressions and statements. These domain-specific concepts significantly reduce the complexity of generating the desired code from high-level specifications illustrated by Figure 18.

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 collectively specify and verify the behavior notations and implementation strategies of object-oriented software. A key contribution of iFSM is a concise mapping from the runtime behavioral model of arbitrary C++/Java classes, expressed using finite state machines, to the internal implementations of these classes, expressed in terms of managing a collection of variables using an implementation specification language. Figure 20 shows the work flow

of our framework for supporting the iFSM language, where we have used POET to implement the iFSM transformation engine, which automatically translates iFSM specifications to C++/Java class implementations and to the input language of a model checker, NuSMV [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.

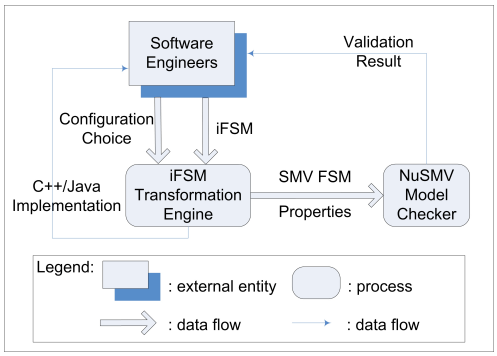

Figure 20: The iFSM framework

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 contract, POET is designed as an output language of compilers so that the optimization decisions by compilers can be easily modified or extended by developers when necessary.

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

[1] Implementing UML statechart diagrams. www.PathfinderMDA.com.

[2] Metamill 4.0. www.metamill.com.

[3] Umodel. Altova Inc. http://www.altova.com/.

[4] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, Santa Fe, New Mexico, May 2000.

[5] N. Ahmed, N. Mateev, K. Pingali, and P. Stodghill. A framework for sparse matrix code synthesis from high-level specifications. In *Supercomputing*, pages 74–74, 2000.

[6] R. Akers, I. Baxter, M. Mehlich, B. Ellis, and K. Luecke. Re-engineering c++ component models via automatic program transformation. In *Twelfth Working Conference on Reverse Engineering*. IEEE, 2005.

[7] J. N. Amaral, C. Barton, A. C. Macdonell, and M. Mcnaughton. Using the sgi pro64 open source compiler infra-structure for teaching and research, 2001.

[8] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[9] R. Balzer, N. Goldman, and D. Wile. On the transformational implementation approach to programming. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 337–344, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[10] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations-computer-aided, intuition-guided programming. *IEEE Trans. Softw. Eng.*, 15(2):165–180, 1989.

[11] I. D. Baxter. Using transformation systems for software maintenance and reengineering. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 739–740, Washington, DC, USA, 2001. IEEE Computer Society.

[12] O. Beckmann, A. Houghton, M. Mellor, and P. H. J. Kelly. Run-time code generation in C++ as a foundation for domain-specific optimisation. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume LNCS 3016, pages 291–306, Internationnal Seminar, Dagstuhl Castle, Germany, March 23–28, 2003, Revised Papers, 2004.

[13] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proc. the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM Press.

[14] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[15] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008.

[16] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Syst. J.*, 35(2):151–171, 1996.

[17] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[18] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, March 2005.

[19] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[20] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 151–160, New York, NY, USA, 2005. ACM.

[21] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1), Jan-Mar 1998.

[22] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *IEEE Computer*, 42:36–42, 2009.

[23] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In *LCPC*, October 2005.

[24] D. R. Engler, W. Hsieh, and M. Kaashoek. 'C: A language for high-level, efficient, and machine-independent code generation. In *POPL*, pages 131–144, 1996.

[25] M. Erwig and D. Ren. A rule-based language for programming software updates. *SIGPLAN Not.*, 37(12):88–97, 2002.

[26] M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 275–284, New York, NY, USA, 2006. ACM Press.

[27] M. S. Feather. A system for assisting program transformation. *ACM Trans. Program. Lang. Syst.*, 4(1):1–20, 1982.

[28] P. Freeman. A conceptual analysis of the draco approach to constructing software systems. *IEEE Trans. Softw. Eng.*, 13(7):830–844, 1987.

[29] M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, page 1381, 1998.

[30] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A pratical method for quickly evaluating program optimizations. In *HiPEAC*, November 2005.

[31] Y. Futamura, Z. Konishi, and R. Glück. Wsdfu: program transformation system based on generalized partial computation. *The essence of computation: complexity, analysis, transformation*, pages 358–378, 2002.

[32] D. Garlan, L. Cai, and R. L. Nord. A transformational approach to generating application-specific environments. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 68–77, New York, NY, USA, 1992. ACM Press.

[33] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[34] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan. Loop transformation recipes for code generation and auto-tuning. In *LCPC*, October 2009.

[35] S.-C. Han, F. Franchetti, and M. Püschel. Program generation for the all-pairs shortest path problem. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation*

techniques, pages 222–232, New York, NY, USA, 2006. ACM Press.

[36] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with safegen. In *Generative Programming and Component Engineering*, 2005.

[37] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, 1993.

[38] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijsho. Iterative compilation in program optimization. In *Compilers for Parallel Computers*, pages 35–44, 2000.

[39] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, Apr. 1991.

[40] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc.* O'Reilly & Associates, 1992.

[41] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[42] S. M. and Q. D. A source-to-source architecture for user-defined optimizations. In *Joint Modular Languages Conference held in conjunction with EuroPar'03*, Austria, Aug. 2003.

[43] J. Magee, Q. Yi, and R. C. Whaley. Automated timer generation for empirical tuning. In *The 4th Workshop on Statistical and Machine learning approaches to ARchitecture and compilaTion*, Pisa,Italy., Jan. 2010.

[44] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[45] S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann, San Francisco, Aug 1997.

[46] J. Neighbors. Software construction using components, 1980.

[47] M. O'Boyle, N. Motogelwa, and P. Knijnenburg. Feedback assisted iterative compilation. In *Languages and Compilers for Parallel Computing*, 2000.

[48] H. Oh and S. Ha. Efficient code synthesis from extended dataflow graphs for multimedia applications. In *Design Automation Conference*, 2002.

[49] Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *Proc. Parallel Architectures and Compilation Techniques*, 2006.

[50] G. Pike and P. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *SC*, Baltimore, MD, USA, November 2002.

[51] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 144–156, Washington, DC, USA, 2007. IEEE Computer Society.

[52] W. Pugh. Uniform techniques for loop optimization. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, June 1991.

[53] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *IEEE special issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[54] A. Qasem, J. Guo, F. Rahman, and Q. Yi. Exposing tunable parameters in multi-threaded numerical code. In *7th IFIP International Conference on Network and Parallel Computing*, Zhengzhou, China, Sept. 2010.

[55] A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th ACM International Conference on SuperComputing (ICS06)*, June 2006.

[56] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. In *Proceedings of the LACSI Symposium*, Los Alamos, NM, 2004. Los Alamos Computer Science Institute.

[57] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing*, 36(2):183–196, 2006.

[58] D. Quinlan, M. Schordan, Q. Yi, and B. de Supinski. Semantic-driven parallelization of loops operating on user-defined containers. In *16th Annual Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Oct. 2003.

[59] S. F. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *HIPEAC:High-Performance and Embedded Architectures and Compilers*, Heraklion, Greece, Jan 2011.

[60] I. Sander and A. Jantsch. Transformation based communication and clock domain refinement for system design. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 281–286, New York, NY, USA, 2002. ACM Press.

[61] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, San Jose, CA, USA, March 2005.

[62] S. A. Thibault, R. Marlet, and C. Consel. Domain-specific languages: From design to implementation application to video device drivers generation. *IEEE Trans. Softw. Eng.*, 25(3):363–377, 1999.

[63] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[64] H. Vandierendonck, S. Rul, and K. De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 389–400, New York, NY, USA, 2010. ACM.

[65] R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for automatic performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, 2004.

[66] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.

[67] William Pugh and Evan Rosser. Iteration Space Slicing For Locality. In *LCPC 99*, July 1999.

[68] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing*, Reno, Nov. 1989.

[69] Q. Yi. Automated programmable control and parameterization of compiler optimizations. In *CGO: ACM/IEEE International Symposium on Code Generation and Optimization*, Apr. 2011.

[70] Q. Yi, K. Kennedy, and V. Adve. Transforming complex loop nests for locality. *The Journal Of Supercomputing*, 27, 2004.

[71] Q. Yi, J. Niu, and A. R. Marneni. Collective specification and verification of behavioral models and object-oriented implementations. Technical Report CS-TR-2010-011, Computer Science, University of Texas at San Antonio, 2010.

[72] Q. Yi and A. Qasem. Exploring the optimization space of dense linear algebra kernels. In *The 21th International Workshop on Languages and Compilers for Parallel Computing*, Edmonton, Alberta, Canada, Aug. 2008.

[73] Q. Yi and D. Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *The 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, Sep 2004.

[74] Q. Yi and C. Whaley. Automated transformation for performance-critical kernels. In *ACM SIGPLAN Symposium on Library-Centric Software Design*, Oct. 2007.

[75] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. A comparison of empirical and model-driven optimization. *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.