Context-Sensitive Domain-Independent Algorithm Composition and Selection [*]

Troy A. Johnson Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-2035
troyj@purdue.edu eigenman@purdue.edu

Abstract

Progressing beyond the productivity of present-day languages appears to require using domain-specific knowledge. Domain-specific languages and libraries (DSLs) proliferate, but most optimizations and language features have limited portability because each language's semantics are related closely to its domain. We explain how any DSL compiler can use a *domain-independent* AI planner to implement algorithm composition *as a language feature*. Our notion of composition addresses a common DSL problem: good library designers tend to minimize redundancy by including only fundamental procedures that users must chain together into call sequences. Novice users are confounded by not knowing an appropriate sequence to achieve their goal. Composition allows the programmer to define and call an abstract algorithm (AA) like a procedure. The compiler replaces an AA call with a sequence of library calls, while considering the calling context. Because AI planners compute a sequence of operations to reach a goal state, the compiler can implement composition by analyzing the calling context to provide the planner's initial state. Nevertheless, mapping composition onto planning is not straightforward because applying planning to software requires extensions to classical planning, and procedure specifications may be incomplete when expressed in a planning language. Compositions may not be provably correct, so our approach mitigates semantic incompleteness with unobtrusive programmer-compiler interaction. This tradeoff is key to making composition a practical and natural feature of otherwise imperative languages, whose users eschew complex logical specifications. Compositions satisfying an AA may not be equal in performance, memory usage, or precision and require selection of a preferred solution. We examine language design and implementation issues, and we perform a case study on the BioPerl bioinformatics library.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.3.4 [*Programming Languages*]: Processors—compilers, optimization; I.2.8 [*Artificial Intelligence*]: Problem Solving and Search—plan formation

General Terms Algorithms, Languages, Performance

Keywords domain-specific languages, algorithm composition, algorithm selection, automated planning, bioinformatics

1. Introduction

Successively higher-level programming languages have increased programmer productivity [25] throughout the last several decades. Higher-level languages increase the abstraction level [27] such that less code is necessary to achieve a goal. The abstraction level directly impacts software development and maintenance costs, including the number of programming errors [38], because experience shows that they are roughly proportional [8] to the amount of code written. Progressing beyond the productivity of present-day languages appears to require using domain-specific knowledge to increase expressivity at the expense of generality. Creating much higher-level languages without domain knowledge is unrealistic [40] because allowing a programmer to express a statement using significantly less code requires a context under which the compiler can infer the omitted information. Domain-specific languages and libraries (DSLs) proliferate [24] as the more viable option for satisfying the demand for greater productivity. Generally, very high-level languages are thought to perform poorly [24, 25] because compilers must bridge wide semantic gaps. Compiler optimizations and powerful language features are therefore crucial [32, 33], but often have limited portability because each language's semantics are related closely to its domain.

We explain how a DSL compiler can use a *domain-independent* AI planner [11, 13, 14, 37, 49] to implement algorithm composition *as a language feature*. Composition addresses a common DSL problem: a good library designer tends to minimize redundancy and encourage reuse [27] by including only fundamental procedures that users must chain together into call sequences. In general, for each pair of useful library procedures A and B, the designer is reluctant to include a redundant procedure C that simply calls A then B. C may be convenient, but including only C prevents A and B from being reused separately. Including all three, for all such procedures (i.e., closure of the set of library procedures), greatly increases library size and hence complexity. For example, BioPerl [44] is a DSL for bioinformatics. A common operation is to download a protein sequence from a remote database and then write it to disk in a particular format so that it can be studied locally. The operation requires six calls[1] to the BioPerl library, shown in Figure 1; we use it as one example of many instances where novice users are confounded by not knowing an appropriate call sequence to achieve their goal [18]. The problem is neither poor library

[*] This work was supported, in part, by the National Science Foundation under Grants No. 0103582-EIA, 0429535-CCF, and by a U.S. Department of Education GAANN fellowship for computational science.

[1] `http://www.bioperl.org/wiki/HOWTO:Beginners`

design nor poor documentation: each call in the sequence is useful by itself, there is abundant documentation, BioPerl is popular, and it has been developed actively since 1995. Biologists can describe *what* they want to happen (e.g., "the result of querying the remote database should end up on my local hard disk"), but sometimes do not know *how* to achieve it. This situation is not limited to BioPerl; others have recognized that "most users lack the [programming] expertise to properly identify and compose the routines appropriate to their application [45]" and that "a common scenario is that the programmer knows what type of object he needs, but does not know how to write the code to get the object [29]." Furthermore, the best call sequence varies depending on context, so time invested learning one sequence may not pay off elsewhere.

```
Query q = bio_db_query_genbank_new(''nucleotide'',
   ''Arabidopsis[ORGN] AND topoisomerase[TITL]
     AND 0:3000[SLEN]'');
DB db = bio_db_genbank_new();
Stream stream = get_stream_by_query(db, q);
SeqIO seqio = bio_seqio_new(''>sequence.fasta'',
                             ''fasta'');
Seq seq = next_seq(stream);
write_seq(seqio, seq);
```

Figure 1. A sequence of BioPerl library calls: the library is object-oriented and used Perl syntax, but here we flatten the code and modify the syntax for a clearer presentation. The programmer must know the order of calls, six procedure names, five data types (inference can be used, but the programmer must match variables to parameters), and adapt the sequence to different contexts.

Composition allows the programmer to define an abstract algorithm (AA) by an *unordered* set of properties and then call it like a procedure; the compiler replaces each call with an *ordered* sequence of library calls, while considering the calling context. AI planners compute a sequence of operations (a plan) to reach a goal state. Similarly, a program is a sequence of operations to compute a result. The compiler can implement composition by providing the planner's initial state and merging the planner's solution into the program. The calling context is used to compute the initial state, making the solution context-sensitive. Composition differs from logic-program synthesis [30] because it combines procedures rather than primitive arithmetic operations, and it uses abstract semantic specifications that are practical to write. We permit procedure specifications to be incomplete, which allows our approach to apply beyond the small number of domains whose procedures can be formally specified. Incomplete specifications do not provide all information to distinguish among procedures that have similar effects. A reality of using specifications is that some will be unintentionally or unavoidably incomplete [40]. Instead of requiring programmers to write complete specifications, we embrace incompleteness as a strength. In exchange, the planner may find multiple solutions – some that do what the application programmer intends and some that do not. We address this problem with unobtrusive programmer-compiler interaction, such that the programmer makes the final selection without being overburdened. This tradeoff is key to making composition a practical and natural feature of otherwise imperative languages, whose users eschew complex logical specifications. For example, the following simple statements (logical predicates) describe the effects of certain BioPerl procedures:

- query_result(result, db, query) - *result* is the outcome of sending *query* to the database *db*
- contains(filename, data) - file *filename* contains *data*
- in_format(filename, format) - file *filename* is in format *format*

Do not confuse the predicates with procedure calls. In our approach, statements like these form part of a glossary that is written by the library programmer. Whereas the set of procedure names is a set of operations within the domain, this glossary is a set of properties of domain data. *The terms do not need a formal definition.* The library programmer uses the glossary to specify procedures, and then provides the glossary to the application programmers so that they may define AAs. For example:

```
algorithm save_query_result_locally
  (db_name, query_string, filename, format) =>
{ query_result(result, db_name, query_string),
  contains(filename, result),
  in_format(filename, format) }
```

The specification approximates the English description of the application programmer's goal by specifying what must be true after the AA call. The order of predicates within the braces does not matter, and `result` is a keyword providing a named return value. The AA is easier to write than Figure 1 because the programmer does not need to know the order of calls, the names of the procedures, or the types of intermediate variables. Writing an AA is different than writing a clause in a logic programming language because the predicates do not have definitions within the program. Each AA is written once and can be called many times. For example:

```
Seq seq = save_query_result_locally(''nucleotide'',
   ''Arabidopsis[ORGN] AND topoisomerase[TITL]
     AND 0:3000[SLEN]'', ''>sequence.fasta'',
   ''fasta'');
```

The compiler substitutes the code in Figure 1, found by the planner, for this AA call. The planner tailors its solution for the calling context. For example, if a database object already exists, then the plan does not include the second call in Figure 1, which creates a database object. The AA is semantically incomplete because it does not imply the use of GenBank instead of various other biological databases (e.g., SwissProt, GenPept, EMBL, SeqHound, RefSeq). We return to this example in Section 6.

Selection is necessary, in our approach, primarily to filter out undesirable compositions that arise from incompleteness; although novice programmers are not able to list every valid composition, they may be able to choose from a list of likely solutions. Selection also, in the more traditional sense [39], chooses among semantically-equivalent sequences that are not equal in performance, memory usage, or precision. Because automated selection methods have been studied extensively, we focus on composition and suggest a simple selection method. The library programmer can provide formulas that each estimate a procedure's utility in terms of some metric. The application programmer specifies as a compiler option the most important metric (e.g., time) for comparison. The compiler selects the best composition by evaluating the formulas, either automatically, or interactively by prompting for typical values of variables. The interaction is kept unobtrusive by caching the responses for future compilations and by a compiler option controlling to what extent decisions are made autonomously.

Our approach is unique; although some compilers have used theorem provers to validate optimizations [28, 36], our compiler is the first to use an AI planner to generate code. Software composition typically is performed as a meta-operation above the programming language level, using separate composition languages [23] or workflow diagrams [26, 45], whereas we propose that the programmer initiate it with inline code similar to a function call. Other algorithm-selection and automatic-specialization systems require at least one procedure that implements the most general (i.e., non-specialized) case of each algorithm [42, 47]. Our system allows AAs that have no single-call implementation and supports one-to-many replacement. Existing systems that retrieve software components by semantic matching [52] do not try to satisfy a match using multiple components, and other compilers [20] require explicit directives to perform optimizations.

Scope of Paper (i) Our system is not intended to prove that the library programmer's specifications are correct. Correctness is a separate issue from completeness; an incorrect specification says that a procedure does something that it does not do. The library programmer, as an expert in the library's domain, is assumed to provide correct specifications and is at fault for any incorrect specification, just as they would be for a bug in the library's code. (ii) Our system is not intended to automatically compose an entire application; although full automation probably is impossible [40], partial automation achieves partial benefits.

Our main contributions are:

- We design a language, `DIPACS`[2], and its compiler to demonstrate how context-sensitive composition of domain-specific library procedures can be implemented as a language feature.
- From a compiler perspective, `DIPACS` is the first to use an AI planner to replace the call of a programmer-defined abstract algorithm (AA) with a sequence of library calls, while considering the calling context. We explain how to overcome several challenges of mapping composition onto planning.
- From a planning perspective, our work is a novel application of planning that is directly comparable to only [16]. We incorporate many advanced features, such as object creation and returning multiple plans, into the same planner. New application areas motivate research in planning theory.
- We support incomplete, abstract procedure specifications that can be written for many domains. We use programmer-compiler interaction to clarify ambiguity. A decision cache and a notion of trust-levels keeps interaction unobtrusive.

Section 2 discusses related work, Section 3 introduces domain-independent planning, and Section 4 explains how we map composition onto a planning problem. We use simple examples throughout Section 5 to explain our programming language and to prepare the reader for the larger BioPerl case study, to which we return in Section 6. Section 7 concludes and provides a glimpse of future work.

2. Related Work

Jungloids [29] compose call sequences from the data types of an input (τ_{in}) and an output (τ_{out}) without considering the semantic relationship between them. The approach is similar to ours *only* in that it suggests a sequence, obtains the programmer's approval, and inserts the sequence into the program; there are numerous differences. A jungloid graph has an edge from τ_a to τ_b if there is a procedure (or typecast) that has a parameter of type τ_a and a return value of type τ_b. Call sequences are discovered by searching the graph for the k-shortest paths from τ_{in} to τ_{out}. The case of synthesizing an output from multiple inputs is addressed by performing a separate search from each type of variable accessible from the call site. The authors of [29] do not require a planner to find a sequence because they do not attempt to satisfy any semantic relationship, nor do they attempt to bind procedure parameters during their search. They find a potentially huge number of call sequences (consider querying a string manipulation library for $\tau_{in} = string$, $\tau_{out} = string$), many of which have an inappropriate semantic relationship, and then use a heuristic to order the sequences from most useful to least useful. A jungloid call sequence is inserted permanently by the Eclipse IDE; if a library is upgraded or procedures are otherwise added to or removed from the program, or if the surrounding context changes, then the sequence may no longer be appropriate. By contrast, `DIPACS` attempts to rule out semantically inappropriate sequences during planning, does not require an advanced editor like Eclipse, and can adapt to changes in context or library upgrades because the sequence is inserted during compilation instead of in the original source code.

Broadway [20] uses library annotations by a domain expert to perform source-to-source transformations. Instead of specifying the semantics of procedures, the annotations provide data-flow information that may be difficult to infer using static compiler analysis. Advanced annotations provide explicit directions to the compiler for determining when a library call should be inlined, replaced with a specialized call, or eliminated altogether. One problem with Broadway is that specialization is indicated with explicit directives; adding a new procedure to a library requires the domain expert to review the annotations for all existing procedures to determine if any could benefit by directives referring to the new procedure. `DIPACS` does not encounter that problem because there are no explicit optimization directives. The planner discovers specializations on its own by using the initial state and the semantic specifications of procedures, which do not explicitly refer to other procedures. Instead, they describe each procedure's effects using a consistent domain vocabulary, which serves as a level of indirection. Adding a new procedure does not require modifying existing specifications unless the vocabulary changes, which should be rare.

Speckle [47] makes specifications part of the programming language such that the compiler can use them to perform optimizations. Speckle is a statically-typed subset of the `CLU` language that uses "primitive, automated theorem-proving technology [47]" to determine if the preconditions of a specialized procedure are satisfied such that it may replace the call of a general procedure. Speckle contains many features of Larch [19], a tool for formally specifying computer programs. Instead of Larch, `DIPACS` uses a planner, which provides greater flexibility in replacement, but means that specifications are not as detailed, so we cannot prove correctness. We believe this tradeoff is adequate; in practice, specifications that seem "complete enough" are used instead of attempting to write more complex specifications. Speckle, by design, does not allow a programmer to directly call a specialized procedure because doing so might circumvent the compiler's proof mechanism. By contrast, `DIPACS` treats an AA call as an explicit request that the compiler assist the programmer; if a procedure is called directly, then the programmer retains all responsibility.

Domain-independent planners [11, 13, 14, 37, 49] gradually have added abilities of general-purpose programming languages, but remain planning languages with extra features [35]. A programmer would use such a language to solve only planning problems. By contrast, `DIPACS` is a mostly imperative language where the planner has a supporting role. Our approach is similar to planning for grid computing [6], but integrated with a programming language. Golden's `DPADL` [16] is the planning language most similar to `DIPACS`. Golden makes planning usable by software developers by loosely basing its syntax on Java, whereas `DIPACS` is a programming language whose compiler happens to use a planner to implement one language feature.

Automatic programming "allows a computationally naive user to describe problems using the natural terms and concepts of a domain with informality, imprecision, and omission of details [3]." A goal of automatic programming [2, 3, 4, 5] is to write a program by writing a specification that is more abstract and concise than, for example, C or Fortran code. The term "semi-automatic" is more accurate because there is interaction between the programmer and the system after writing the specification. One of the myths [40] about automatic programming is that interaction can be eliminated completely. By focusing automation on parts of an application, we are intentionally less ambitious; most code is written in an imperative style, allowing for incremental progress towards a full spec-

[2] Domain-Independent Planned Algorithm Composition and Selection

ification. It is important to allow for both programming styles in a realistic development environment, lest difficulty specifying one part of an application impede development elsewhere. Likewise, "it is not acceptable for the compiler to ignore specifications until all parts of the program are specified in full because this may never happen [47, p.17]."

Program synthesis [30] is one of several approaches [5] to automatic programming. Synthesis uses a theorem prover to generate a constructive proof from which a program is extracted. Programs often are as simple as swapping the values of two variables, but more complicated syntheses are possible [45]. Traditional synthesis is not practical in a real programming environment because "synthesized programs are often wantonly wasteful of time and space [30]." By contrast, algorithm composition combines procedures written in an imperative language by a knowledgeable library programmer; therefore, each part of the composed algorithm is not "wantonly wasteful." Certain compositions of library procedures may be less efficient than others, but they can be compared using formulas bundled with the library or by consulting the application programmer. Synthesis requires that primitive operations have complete semantics, whereas our approach does not. We use a planner, similar to [9], but use procedures for the primitive operations and use an interactive process to deal with incomplete semantics. It is reasonable for synthesis to require complete semantics when most components have complexity similar to machine instructions, but it is not reasonable to expect all larger components (e.g., procedures, applications) to have complete semantics.

Prolog and other logic (declarative) languages allow programmers to write goal-oriented programs, but cannot *directly* provide the behavior we seek for several reasons. First, a Prolog interpreter responds to queries by providing a yes or no answer, or by enumerating values that satisfy the goal. For example, the query `reverse([1,2,3],X)` asks if there is such a list X and the Prolog interpreter replies that yes there is one such list, `X = [3,2,1]`. The interpreter is not able to answer queries like `X([1,2,3],[3,2,1])`, which asks how that relationship could be achieved; the desired answer would be `X = reverse`. Second, the interpreter does not show the list of steps that it takes to produce its answers. If print statements or debug traces are used to reveal the steps without modifying the interpreter, then unsuccessful search paths are also printed. Third, Prolog Horn clauses are not sufficient for AA and procedure specifications. Horn clauses are of the form $head \vdash body_1, \ldots, body_n$ where the "head" of the clause is implied by the conjunction of terms in the right-hand "body." Although an AA can be written in this fashion, with the AA's name on the left and its effects on the right, writing procedure specifications in the same way does not allow the interpreter to find a solution. A procedure's effects would need to be on the left-hand side, which does not permit multiple heads. Splitting a multi-head clause into multiple clauses does not work because that causes plans to include multiple calls to the same procedure when only one call is needed. Finally, glossary terms are not defined within our programs, whereas every term in a Prolog program appears as a head somewhere. Therefore, instead of using a Prolog interpreter directly to plan compositions, one would use the Prolog language to write a planner, as can be done using any Turing-complete language. The use of Prolog is then an implementation choice, and we choose not to use it.

Algorithm recognition and replacement [34] are similar to algorithm composition and selection because both affect a program at the algorithmic level. Algorithm recognition seeks to understand multiple statements of imperative code (e.g., C, Fortran) before replacing them with a better implementation. By contrast, for composition and selection, the abstract algorithm written by the programmer cannot be executed directly. Algorithm recognition and replacement have not been very successful. The primary roadblock is that algorithm equivalence testing is a variation of the graph isomorphism problem, for which neither a polynomial-time algorithm nor an NP-completeness proof is known. In practice, recognizers are slightly less ambitious and make simplifications to reach the subgraph isomorphism problem, which is known to be NP-complete. Still, the heuristic approaches [31, 34, 50] to solving this problem are complicated and have not been implemented widely in commercial compilers. We avoid the recognition problem with explicit semantics and concentrate on replacement.

Adaptive algorithm selection [51] dynamically replaces computation patterns, such as reductions, with different implementations. It derives cost expressions for various implementations based on actual runs and predicts which implementations will be most efficient in the future. The `STAPL` [46] library internally performs algorithm selection. The Pythia [22, 48] systems recommend algorithms for solving scientific problems.

Commercial Software: Mathematica employs a proprietary algorithm selection system[3] to provide faster and more precise results to users. Although their math library is extensive, the programmer cannot expand the automated selection process to include their own functions. `MATLAB` commonly is used as an advanced prototyping and programming environment for scientific computing. It provides many application-oriented modules that can be viewed as a higher-level, domain-specific programming environment. Recent work [32, 33] focuses on high-level compiler optimizations.

3. Domain-Independent Planning Background

Figure 2 shows a conceptual model of domain-independent planning [14, p.5]. A domain-independent planner uses an initial state, a goal state, and operators to find a sequence of actions (instantiated operators) that, if executed by the plan user, will transform the world from the initial state to the goal state. The physical world is composed of a static set of *objects* with various properties and relationships that are changed by the actions; likewise, the planner manipulates a symbolic representation of that world where operators affect a corresponding symbolic state. For domain-*dependent* planners, the set of operators is implied (i.e., the operators are hard-coded into the planner) and not provided as input. The advantage of a domain-independent planner is that it can be used for many different types of problems without rewriting the planner's code.

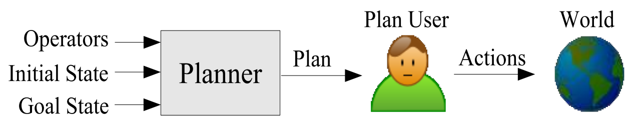


Figure 2. A conceptual model of domain-independent planning

The set of operators defines a state-transition system. Operators are applied to states, beginning with the initial state, to reach successor states. An operator may have a *precondition* that determines if it can be applied to a particular state. A precondition is a logical statement that must be true about the objects the operator will affect. If the current state satisfies the precondition, then the operator can be applied to the state. Applying an operator to a state α results in a new state β where the properties of β are a function of α and the operator's *effects*. The planner's job is to find a path through the state-transition system from the initial state to the goal state.

The state-transition systems considered by planners are typically so large that it is not feasible to enumerate all states and transitions. A planner must search intelligently, using reasonable

[3] http://www.wolfram.com/technology/guide/algorithmselection.html

amounts of time and memory, to find a solution. A naive depth-first search requires excessive backtracking and may be thwarted by infinite branches of the search space; breadth-first search requires large amounts of memory when the branching factor is high. Using an appropriate search method, the planner should output a solution and terminate, if a solution exists. When no solution exists, an output of “no solution” is helpful, but many planners will not terminate. Fortunately, for most planning applications with appropriate search methods, a solution is returned quickly if it exists; if the planner runs for an unusually long time, then it is normally assumed that there is no solution.

The first planning system [11] was created because traditional theorem provers were too inefficient at solving planning problems. In addition to specifying the effects of an operator, it was often necessary to specify what the operator did *not* affect in order to solve a problem using a theorem prover – a difficulty known as the frame problem. A planner separates the theorem-proving aspect of the problem from the search aspect, such that separate strategies can be used for each, yielding a more efficient approach.

There are several possible representation schemes that are equally powerful in terms of problem solving [14, p.19]. We use a variation of the classical representation where preconditions and effects are characterized by logical predicates. Figure 3 shows the canonical planning example, block stacking. The action `move` changes the location of a block that has nothing on top of it.

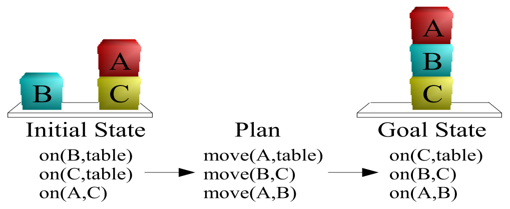


Figure 3. A planning example: a planner uses the initial state, the goal state, and operator descriptions (not shown) to find a plan that will arrange the blocks according to the goal.

4. Planned Algorithm Composition

Using a planner to form plans that affect program values instead of physical objects is unusual. A straightforward mapping of composition onto planning presents challenges to the library programmer, the compiler, and the planner. This section provides an overview of the mapping and then identifies four challenges that we overcome using a combination of compiler techniques (e.g., live-variable analysis, SSA [10]) and extensions to classical planning.

4.1 Mapping Composition onto a Planning Problem

We map algorithm composition onto a planning problem as in Figure 4. When the compiler encounters an AA call in the application code (e.g., `save_query_result_locally` from Section 1), it invokes a planner to find sequences of library calls (e.g., Figure 1) that achieve the AA’s effects. The compiler uses the calling context to determine the initial state for each planning problem. The library procedures and the procedures in the application code are the operators available to the planner.

Due to the large search spaces of these problems, which we describe in Section 5.4, planning techniques are more appropriate than simple breadth-first or depth-first searches. Nevertheless, existing planners are not helpful because planning research tends to focus on scalability without using advanced features such as axioms, equality, object creation, and returning multiple plans. We require these features, as we explain below.

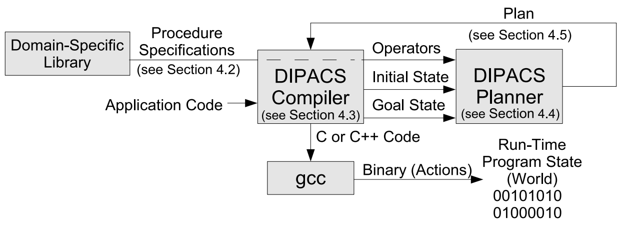


Figure 4. Using a planner for algorithm composition: the compiler creates the planner’s input in a format similar to PDDL [13], merges the resulting plan into the program at the call site of the AA, and generates a binary. The compiler becomes the plan user from Figure 2. Executing the binary causes the plan to affect the program’s run-time state, which is essentially the world from Figure 2.

4.2 Ontological Engineering

The first challenge of the approach in Figure 4 is for the library programmer to specify appropriately the behavior of the library procedures and for the application programmer to specify the desired effects of AAs. In Sections 1 and 2, we observe that attempts to specify precise semantics have not had significant success. Such approaches often require programmers to become experts in logic programming and have proven feasible only on small code kernels. Also, we observe that precise, formal semantics are often unnecessary for adequately describing software behavior because the most common form of specification is English comments.

In our approach, the library programmer defines a glossary of abstractions for describing the preconditions and effects of library routines. Choosing adequate abstractions is called ontological engineering – the process of deciding on a vocabulary for a domain and how to best represent information. Ontological engineering is a complex issue [17], and a full discussion is outside the scope of this paper. We assume that the library programmer knows the preconditions and effects for the individual procedures, and writes the specifications after choosing the abstractions; we do not attempt to solve the problem of automatically determining the preconditions and effects, which may be possible for low-level abstractions [1, 12, 15].

As a running example, we consider sort procedures, which are well-known and concise while still demonstrating important concepts; the larger BioPerl case study is covered in Section 6. Suppose that a library procedure P returns an array that is `sorted`. If the application programmer writes an AA that must produce a `sorted` result, then the planner might choose P as a step of the plan. To do so, the planner *does not need to know the precise semantics of the term sorted*. A formal specification involves a definition like (1).

$$\forall i \in \{0 \ldots result.length - 2\} \;\; result[i] \leq result[i+1] \qquad (1)$$

Using such a low abstraction level results in lengthy specifications that are cumbersome to write and a greater chance that there is more than one way to write the same specification. We believe that such a definition is not as programmer-friendly as the commonly understood term `sorted`. The English definition in the glossary can clarify any ambiguities (e.g., increasing versus decreasing order, sorting alphabetically versus numerically). Using a high abstraction level, such as `sorted(result)`, avoids the above problems. A disadvantage appears to be that `sorted` is not defined such that it can be proven or validated, but our approach requires neither.

In addition to `sorted(result)`, `permutation(result, input)` is a useful abstraction because otherwise any sorted result could be returned, even one that was not related to the input. Furthermore, it is useful for the planner to know that a permutation is reflexive, symmetric, and transitive. For example, if the input array was already sorted (i.e., `sorted(input)`), then it could not satisfy the goal `sorted(result)` ∧ `permutation(result, input)`, unless `permutation(input, input)` was known to be true via the reflexive property. As another example, transitive properties are useful to transfer semantic information from input to output through a series of intermediate operations. Axioms supply these properties to a planner. Many planners, particularly those that do not maintain a current state (i.e., plan-space planners), do not support axioms [14]. As we discuss in Section 5.4, our planner is a state-space planner that supports axioms. We expect the library programmer to use keywords to indicate which predicates are reflexive, symmetric, and transitive, but do not expect them to write general axioms. General axioms can introduce synonyms into the glossary, which confuse both application programmers and planners. An important relationship that cannot be expressed with axioms is equality; i.e., that two objects have the same properties and are interchangeable. An assignment statement introduces that relationship. Traditional planners do not need to deal with assignment, but our planner supports equality.

A related issue is the frame problem: whether and how to specify what a procedure *does not* do [7]. For example, the programmer should not have to specify that a sort procedure does not overwrite unrelated variables, calculate π, or format a disk. In general, the truth value of an unspecified property is unknown. Because we permit incomplete specifications, it is convenient to assume a default value of false [14, p.47], while remembering that it may be true.

Expressive Power Planning languages (e.g., [13]) have limited expressiveness. Because we do not require complete specifications, such languages are sufficient to specify the effects of procedures in real libraries such as BioPerl (see Section 6). Note that providing a theoretical bound on the power of our specifications is difficult for the same reason; how complete a procedure's specification must be to place it in the class of procedures that can be usefully specified is subjective. Any specifications that allow the compiler to suggest or select compositions that please the application programmer while not providing an annoying number of irrelevant compositions would be considered useful.

4.3 Determining the Initial and Goal States

The second challenge of implementing the mapping in Figure 4 is that the compiler must determine the initial state and clarify the goal state for each AA call in the program.

Initial State First, the compiler must determine which variables are in scope because the "physical" objects available to the planner are the values accessible from the call site. Next, all-paths flow analysis is used to discover the properties of the values, as in Figure 5. The compiler determines if a value has a semantic property as a result of the effects of a previous call, if two variables are equal, and if a variable has a particular value. The list of objects and their properties are provided to the planner as the initial state.

Goal State The AA's effects are the goal state of the plan. The goal is to create new values with certain properties. In planning terms, the goal is existential because it is satisfied by any variable containing a value with those properties (i.e., the name of the value does not matter). Because AAs are not typed, the compiler must restrict the existential to the type of variables receiving the return values of the AA call. For example, if an integer array receives the result of `sort(input)`, the goal is: ∃ `int[] result` such that `sorted(result)` ∧ `permutation(result, input)`.

4.4 Object Creation

The third challenge of implementing the mapping in Figure 4 is for the planner to use the operators because procedures behave differently than traditional planning operators. In classical planning,

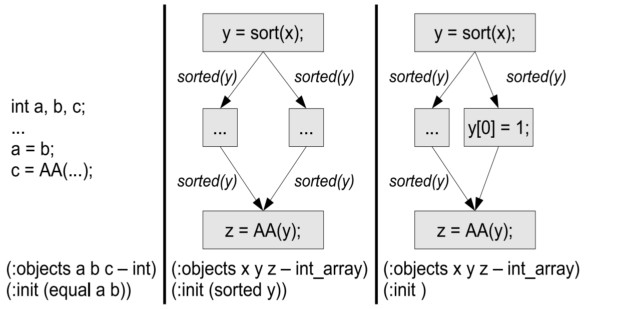


Figure 5. Calculating the initial state: in each of the three examples, we show the initial objects and state for the AA call. The `sorted` property comes from the effects of `sort`. An assignment to a variable along any path wipes out its properties.

the world consists of a static set of objects. Operators modify the properties of and relations among those objects, but never create or destroy objects; the initial state, goal state, and all intermediate states contain the same objects. Permitting object creation makes it more difficult to ensure that a planner will terminate because it dynamically changes the size of the search space.

A static set of objects does not model the state of most programs. A program's run-time state consists of the values of variables. Procedure calls often use different variables for input and output, such that some are read-only, while others are overwritten. Intermediate steps require temporary values not present at the beginning or end of a computation. Requiring all procedures to use the same variables for input and output is not practical. In fact, a purely destructive language corresponds to a restricted Turing machine that permits writes to only storage locations that already contain values. Such a machine never uses an amount of storage greater than the size of its input and is known as a linear bounded automata, which is computationally less powerful than a Turing machine [21].

Therefore, extending classical planners to the programming world requires object creation (also observed by [16]). Object creation can be added in two ways. The first method simulates creation by starting with a large number of extra objects that are blank; the objects are untyped and have no properties. When a new object is needed, one of these extra objects is given a type and some properties. The advantage of this method is that it maintains a static set of objects and thus fits the classical paradigm. The disadvantages are analogous to using a fixed-size data structure when a dynamically-sized structure is more appropriate. For example, when using a table of length N, one could choose N very large so that most problems fit; however, the disadvantages are wasting memory, having to search a large table even when solving small problems, and sometimes failing to choose a large enough N.

We employ the second method, which breaks the classical planning assumption by creating new objects on demand. Breaking the assumption does not raise termination issues because our planner does not create redundant objects; if a value exists that satisfies an existential goal, then the plan uses that value instead of calling a procedure to create a new value to satisfy it. Return values are assigned unique names, similar to how a compiler assigns temporary values unique names for intermediate code. We also use a renaming scheme for modified values, such that we can specify relationships between a variable's previous and current values:

$$\text{x} \rightarrow \text{x@1} \quad , \quad \text{x@1} \rightarrow \text{x@2} \quad , \quad \dots$$

When x is modified by a procedure, we create x@1, which is read as "x after the first modification." A before-after relation p would be written $p(x, x@1)$. These techniques are very similar to static single-assignment (SSA) [10], but inside a planner.

4.5 Merging the Plan into the Program

The fourth challenge of implementing the mapping in Figure 4 is to insert the plan into the call site. Although this challenge is not as difficult as the others, it must be handled correctly. It is possible that a destructive plan is found, but only a nondestructive version is allowed in the calling context, or vice versa. If a destructive call sequence is inserted where the application programmer expects a nondestructive sequence, then they will be surprised to find their data overwritten; likewise, if a nondestructive sequence is inserted where a destructive sequence is expected, they will wonder why their data did not change. In Section 5.2, we explain a language-design choice to always define an AA as if it were nondestructive, although some implementations of it may be destructive. The compiler knows whether it can use a nondestructive or a destructive implementation of an AA by using live-variable analysis. For example, assume that `sort` is an AA.

```
int[] a, b, c;  /* three integer arrays */
...
a = sort(a); /* request a destructive plan */
c = sort(b); /* request a nondestructive plan */
...
... = a;
... = b;
```

The first assignment permits a destructive plan because a is killed. If only a nondestructive plan is available, then the compiler uses it and adds an assignment on the end. Either is permissible:

```
destructive_sort(a);          /* option 1 */
a = nondestructive_sort(a);   /* option 2 */
```

The assignment to c does not permit b to be modified because it is used later. If only a destructive plan is available, then the compiler prepends code to copy the arguments, changes the planned call sequence to apply to the copies, and then copies the result into the output variables. For example, a nondestructive sort can be created from a destructive sort:

```
int[] t = b;
destructive_sort(t);
c = t;
```

If c is not used by the plan, then one assignment can be eliminated. Other situations may permit more elimination [41].

```
int[] c = b;
destructive_sort(c);
```

Note that it is not normally advisable for the application programmer to permanently replace an AA call with the planner's solution because the surrounding context may change or new procedures may become available, which can invalidate the solution or make it less optimal. Keeping AAs allows programs to evolve as libraries improve.

5. Language Design and Implementation

We have discussed the general principles behind our approach. In this section, we apply those principles in a language of our own design. We considered modifying an existing domain-specific language, but we wanted to be able to experiment with examples from multiple domains. We also considered modifying a compiler for an existing general-purpose language because they are often used to call domain-specific libraries. That approach became less attractive after considering that we would need to force specifications into an existing grammar and deal with many language features that were not directly related to our work. Therefore, we created `DIPACS` to try our ideas. We are not proposing wide acceptance of the language; it is a proof of concept only. We believe that our techniques could be integrated into existing languages, and certainly into new languages that considered all of the design issues up front.

We chose a traditional imperative programming model because many DSLs perform mathematical computations (e.g., linear algebra, game physics, Fourier transforms). These libraries often are written in C, Fortran, or even assembly language because they are intended to be reused many times, and their authors want complete control over the implementation to emphasize efficiency. Matching the programming model of these libraries makes it easier to test our ideas. Other kinds of libraries (e.g., object-oriented libraries written in C++ or Java) could benefit from our techniques, but we consider it future work. `DIPACS` is an integrated specification and programming language, which avoids the problems that occur with separate languages, as in Broadway [20]. It is convenient for programmers to define and use AAs from within their source code instead of using a separate file, which avoids extra writing and consistency problems. The integrated language enables the compiler to allow program constants and "sizeof" calculations in specifications. Separate specifications are still necessary when libraries are not distributed with their source code. In that case, we suggest the compiler of the library emit a specification-only header file and binary, both based on integrated source code.

5.1 Library Procedures and Their Informal Specifications

The general form of a procedure definition is

```
procedure ([type return_value_name]+)
name([type parameter_name]*)
  [<= { precondition* }] [=> { effect* }]
  [metric_name formula]*
{ [executable_statement]* }
```

Multiple return values are allowed and each must be given a name, unless there is only one return value, in which case parentheses are omitted and the name defaults to `result`. Parameters are typed. Preconditions specify n-ary relations on the parameters that must be true to call the procedure, and they may not refer to return values. Effects inform the compiler of relations that are true immediately after calling the procedure, and they may refer to return values. An effect can be a predicate, an equality relationship, a universal quantifier, or a conditional effect – all of which have appeared in planning languages. The `forall` keyword quantifies over all objects of a specific type that are in scope when the procedure is called. The `when` keyword indicates a conditional effect; its condition must be true for the effect of its body to apply. The return values themselves contain the implied effect of creating new objects. If used, metric names must be declared previously in a metric declaration; typical metric names are `time` and `space`, representing performance and memory usage. A formula must be a single mathematical expression containing constants and parameter names that predicts the utility of the procedure in terms of the metric. The expressions are provided by the library programmer, and may include calls to typical math functions and "sizeof" calculations. The intent is for the formulas to be evaluated at compile time to provide some guidance in comparing alternative call sequences; another way would be to heuristically order the sequences from shortest to longest [29]. The executable statements are similar to C or Java. A nondestructive and a destructive sort procedure are shown below. Note that pass-by-value is the default.

```
metric time, space;
axiom reflexive symmetric transitive permutation;

procedure int[]
nondestructive_sort(int[] array)
  => { sorted(result), permutation(result, array) }
  time pow(array.length, 2)
  space 2 * array.length
{ /* implementation */ }
```

```
procedure void
destructive_sort(int[]& array)
  => { sorted(array@), permutation(array@, array) }
  time 3 * pow(array.length, 2)
  space array.length
{ /* implementation */ }
```

Limited axioms are supported as described in Section 4.2. For the destructive procedure, the return type is `void` and the array is passed by reference so that the procedure can sort it in place. The @ operator, as in Section 4.4, refers to a parameter's modified, post-call version. The procedures have different time and space formulas, such that the first is faster, but the second is more space-efficient. The formulas are only crude estimates and do not consider the number of inversions, cache behavior, etc. The compiler can evaluate the formulas and select an implementation based on compiler flags to optimize for time or space, but may need to prompt the programmer or use profiling information to obtain the typical size of the array. More advanced selection techniques are possible, but beyond the scope of this paper.

5.2 Defining Abstract Algorithms (AAs)

The general form of an AA definition is

```
algorithm [(return_value_name*)]
name([parameter_name]*) [=> { effect* }]
```

An AA has a name and parameters, similar to a procedure, but instead of a body it has only effects. For now, these effects can only be predicates. A composition satisfies an AA if it causes all of the predicates to be true; it may cause other predicates to be true. Preconditions do not appear in the algorithm definition because they would specify under what circumstances the algorithm may be applied, instead of what it does. Parameters are not typed for exactly the same reason that preconditions are omitted: a parameter type can be thought of as a precondition (e.g., a unary predicate `int(x)` to indicate that `x` is an integer).

An AA always is described as a nondestructive operation, regardless of whether most implementations of it would be destructive. Nondestructive specifications are the default for two reasons: (i) the specification is simpler because there is no confusion over whether a variable name refers to a value before or after it is modified, and (ii) a nondestructive call more easily blends into a calling context (see Section 4.5). The name of another AA can be used as shorthand to include its effects. For example, here are the AA definitions for `sort` and `stable_sort`.

```
algorithm sort(x)
  => { sorted(result), permutation(result, x) }

algorithm stable_sort(y)
  => { sort(y), stable(result, y) }
```

The effects of `sort` are shared by `stable_sort`. Note that the predicates `permutation` and `stable` illustrate the convenience of the abstract approach; neither the application programmer nor the library programmer had to figure out how to formally define the predicates using logic. Although formally defining a `sorted` predicate is not very difficult, both `permutation` and `stable` are more complicated.

5.3 Calling AAs and Generating the Planner's Input

Suppose that the following procedures are available:

```
axiom reflexive symmetric transitive permutation;

procedure void insertion_sort(int[]& array)
  => { sorted(array@),
       permutation(array@, array) }
{ /* implementation */ }

procedure int[] build_max_heap(int[] array)
  => { max_heap(result),
       permutation(result, array) }
{ /* implementation */ }

procedure int[] sort_heap(int[] array)
  <= { max_heap(array) }
  => { sorted(result),
       permutation(result, array) }
{ /* implementation */ }
```

Furthermore, suppose that the programmer makes this AA call:

```
int[] output_array = sort(input_array);
```

The call can be implemented as a composition of `build_max_heap` and `sort_heap`, or by copying the `input_array` to the `output_array` and running `insertion_sort` on the `output_array`. The compiler generates PDDL [13] code to use the planner to find the compositions. PDDL is the planning language used for competitions within the planning community. For illustration, we show the code below.

```
(define (domain isort)
 (:axiom ; reflexive
  :vars (?x)
  :context ()
  :implies (permutation ?x ?x))
 (:axiom ; symmetric
  :vars (?x ?y)
  :context (permutation ?x ?y)
  :implies (permutation ?y ?x))
 (:axiom ; transitive
  :vars (?x ?y ?z)
  :context (and (permutation ?x ?y)
                (permutation ?y ?z))
  :implies (permutation ?x ?z))
 (:action insertion_sort
  :parameters (?array - int_array)
  :creates (?array@ - int_array)
  :effect (and (sorted ?array@)
               (permutation ?array@ ?array)))
 (:action build_max_heap
  :parameters (?array - int_array)
  :creates (?result - int_array)
  :effect (and (max_heap ?result)
               (permutation ?result ?array)))
 (:action sort_heap
  :parameters (?array - int_array)
  :precondition (max_heap ?array)
  :creates (?result - int_array)
  :effect (and (sorted ?result)
               (permutation ?result ?array))))

(define (problem sort)
 (:objects input_array - int_array)
 (:init ) ;null initial state
 (:goal (exists (?result - int_array)
          (and (sorted ?result)
               (permutation ?result input_array)))))
```

PDDL uses a Lisp-like syntax, with a question mark to denote a planning variable that can be bound to an object. The `creates` clause is not standard PDDL. When the planner applies an operator (an action in PDDL), the variables in the `creates` clause are bound to unique names (e.g., tmp0, tmp1, input_array@1) and added to the list of known objects, as in Section 4.4. In this example, the initial state is empty, so the planner finds two plans. Written in an imperative syntax, they are:

```
insertion_sort(input_array);
bind ?result to input_array@1

tmp1 = build_max_heap(input_array);
tmp0 = sort_heap(tmp1);
bind ?result to tmp0
```

```
solve_problem(domain_def, problem_def)
  parse definitions
  $\Pi \leftarrow$ **all_values** backward_search(...)

backward_search($s_0$, goal, operators,
                initial_objects, axioms)
 let loop ($s \leftarrow s_0$, $g \leftarrow goal$,
           $\Gamma \leftarrow$ null, ; previous goals
           $\pi \leftarrow$ null, ; the plan
           objects $\leftarrow$ initial_objects)
   if $g \in \Gamma$ then **fail** ; avoid $\infty$ paths
   $s \leftarrow$ apply_axioms($s$, objects, axioms)
   $s \leftarrow$ apply_equalities($s$, initial_objects)
   if $s$ satisfies $g$ then return $\pi$.bindings
   $R \leftarrow$ relevant_set($s$, operators, objects)
   if $R$ is null then **fail**
   $a \leftarrow$ **a_member_of**($R$)
   if $a$ has unbound parameters then
     subgoal_plans $\leftarrow$ **all_values**
      loop($s$, precondition[$a$], $g.\Gamma$, null, objects)
     if subgoal_plans is null then **fail**
     $sp \leftarrow$ **a_member_of**(subgoal_plans)
     $s$, objects $\leftarrow$ apply_plan($sp$, $s$, objects)
     $\pi \leftarrow \pi.sp$
     bind action $a$ using the binding in $sp$
   $s$, objects $\leftarrow$ apply_action($a$, $s$, objects)
   $g$' $\leftarrow$ unsatisfied_part($g$, $s$)
   loop($s$, $g$', $g.\Gamma$, $\pi.a$, objects)
```

Figure 6. Pseudocode for the planner: the dot operator (.) indicates list concatenation and a ⟵ indicates assignment. `a_member_of` is a "nondeterministic [43]" choice point to which the `fail` statements return. `all_values` safely catches the top-level `fail` and collects the results in a list.

The compiler converts the first plan to be nondestructive and eliminates temporary variables in the second plan to produce:

```
output_array = input_array;
insertion_sort(output_array);

output_array =
   sort_heap(build_max_heap(input_array));
```

If `input_array` already was sorted due to the effects of an earlier call, then the initial state is `(:init (sorted input_array))` and the planner finds what is effectively a null plan:

```
bind ?result to input_array
```

This plan contains no calls, so the compiler can eliminate the AA call and replace it with an array assignment. Therefore, *the same AA call can result in different plans depending on the calling context, which makes our approach context-sensitive.*

5.4 The Planner

Our planner is written in Guile[4], which implements the R5RS Scheme standard and provides an interpreter to C code through a library. Instead of using `exec` to start a new process each time it needs the planner, the compiler calls a Guile procedure that initiates the planner, directly passes information through the call, and receives a list of plans as a return value. We implement a backtracking mechanism using continuations similar to [43] to provide "nondeterministic" choice points, which make it easier to write a planner.

Although, by planning standards, the call sequences that we seek are short (frequently fewer than ten steps), the branching factor of the search space is very high. It is common to have libraries with tens to hundreds of procedures (operators), each with numerous parameters, called from contexts with tens to hundreds of values (objects) in scope. At each step of the plan, there are many possible actions because for each operator there are many ways to bind its parameters to the values. Consider a modest example where the library has 128 procedures with 2 parameters each, there are 8 live values at the AA call site, and we are trying to find a sequence of 4 calls. To simplify the math, we will assume that any live value can be bound to any parameter (i.e., that there is only a single data type). The following formula yields the number of potential plans:

$$(128 * 8^2)^4 = (2^7 * 2^6)^4 = 2^{(13*4)} = 2^{52} \approx 4.5 \text{ x } 10^{15} \quad (2)$$

Therefore, an exhaustive search for each AA call is not feasible. Our initial planner used a forward planning algorithm that began at the initial state and tried to reach the goal state. Unfortunately, using a forward planning algorithm in a search space with a high branching factor requires a very good heuristic to guide the search. Most planners are concerned with finding a single solution and so their heuristics can focus on identifying the path that is most likely to succeed. Because we are interested in returning multiple solutions, a good heuristic must identify multiple likely paths. We had great difficulty with the forward approach, so we switched to a backward planning algorithm that turned out to be more successful. Although searching backwards from the goal towards the initial state is counterintuitive, it can be more efficient than a forward search; a forward search tends to try many operators that are irrelevant to the goal and may backtrack more frequently than a backward search, which can more accurately determine the set of relevant operators. Both strategies are common in modern planners and we do not claim that a backward planner is necessary.

High-level pseudocode for our planner is shown in Figure 6. The main procedure is `solve-problem`, which accepts a definition of the problem and its domain. The definitions are parsed, and the extracted information is passed to `backward-search`, which performs the planning. `backward-search` is a heavily-modified version of the algorithm normally associated with early `STRIPS` [11] planners. We add axioms, equality, object creation, and a check for previously-considered subgoals to avoid infinite recursion.

The planner finds an operator that is relevant to satisfying the goal, and then recursively attempts to satisfy its precondition. An action is relevant if it helps satisfy an existential goal by creating an object of the desired type, or if it helps satisfy a regular goal by making one of the conjuncts true. The relevant set must include all relevant actions, otherwise the planner will fail to find all solutions. The set may safely include actions that are later discovered to be irrelevant because the check for repeated goals will detect that an action did nothing useful; however, including many irrelevant actions will reduce performance.

Normal `STRIPS` planners have two limitations [14, p.76] that do not affect our implementation. The first limitation is that they attempt to satisfy only subgoals relevant to the precondition of the most recently considered operator. If an operator has negative effects (i.e., causes a true property of a value to become false), then that artificial ordering can cause "deleted-condition interactions [14, p.77]," where a previously satisfied subgoal is undone. Because we rename modified values similar to SSA, as discussed in Section 4.4, there are no negative effects; there is simply a new value created that lacks properties of the original value. The second limitation is that `STRIPS` planners do not backtrack once they commit to applying an operator. Our implementation backtracks over applied operators because we wish to return multiple plans.

6. Composition Using the BioPerl DSL

In Section 1, we used an example from the BioPerl library without explaining its implementation; we return to that case study.

[4] `http://www.gnu.org/software/guile`

6.1 The BioPerl Domain

We show specifications for only the procedures relevant to the AAs that we discuss in this paper. We consulted the BioPerl documentation[5] to learn the purpose of each procedure. Although we are not biologists, we were able to understand enough of the documentation to specify the procedures shown below. After we read the documentation, writing these specifications took less than a half hour; specifying the entire library certainly would require more time and a greater understanding of the biology domain. We wrote the AAs *after* writing the procedure specifications because that is the order in which it would occur in practice.

```
procedure Query
bio_db_query_genbank_new(char[] db_name,
  char[] query)
  => { result.db == db_name,
       result.query == query };

procedure Stream
get_stream_by_query(DB db, Query q)
  => { stream_for_query(result, q) };

procedure SeqIO
bio_seqio_new(char[] filename, char[] format)
  => { result.file == filename,
       result.format == format };

procedure Seq next_seq(Stream stream)
  => { forall (Query q)
       when (stream_for_query(stream, q))
       { query_result(result, q.db, q.query) } };

procedure void write_seq(SeqIO io, Seq s)
  => { in_format(io.file, io.format),
       contains(io.file, s) };

procedure DB bio_db_genbank_new();

procedure Seq bio_seq_new(char[] id, char[] seq)
  => { result.id == id, result.seq == seq };

procedure Factory
bio_tools_run_standaloneblast_new(char[] program,
  char[] db_name)
  => { result.program == program,
       result.db_name = db_name };

procedure Report blastall(Factory blast, Seq s)
  => { report_for_blasting(result,
        blast.program, blast.db_name, s) };
```

All glossary terms are defined in Section 1, except `stream_for_query` and `report_for_blasting`. The former means that a stream supplies the results of a particular database query, and the latter means that a report is from "blasting[6]" a particular protein sequence. These predicates, along with structure fields, propagate semantic information throughout plans. For planning variables that represent structure fields, the planner only substitutes object names for the part before the dot (.) operator. None of the procedures have explicit preconditions, but the parameter types provide implicit preconditions. A **==** denotes equality. These specifications are translated by the compiler into PDDL, which we omit for brevity.

6.2 A Jungloid-like Abstract Algorithm

AAs can be used to implement Jungloids [29]. Suppose that the application programmer knows only the type of object that they desire. They can generate a large list of plans by defining a "dummy" AA that has no effects and then calling it:

```
algorithm dummy();

Seq s = dummy();
```

That code causes the compiler to produce a planning problem that has very few constraints:

```
(define (problem dummy)
 (:objects ...) ; whatever variables are in scope
 (:init ) ; null
 (:goal (exists (?result - Seq))))
```

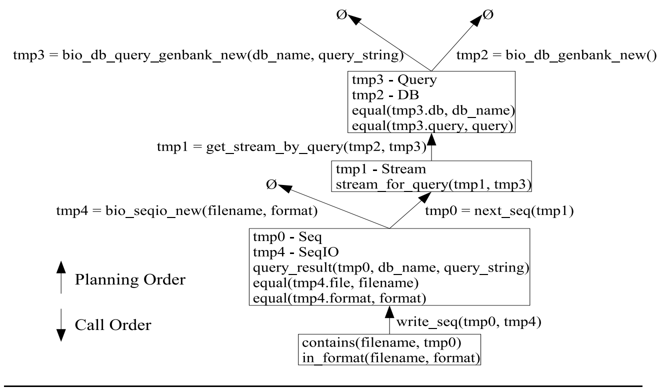
To solve the problem, the planner must find a way of creating a Seq object. Because it does not matter how the object is created, many plans are found. If the list of plans is too large from which to conveniently select a plan, then the programmer can add further constraints and recompile, as in the next example.

6.3 AA: save_query_result_locally

In Section 1, we introduced an AA called `save_query_result_locally` and showed an example of calling it. For that example call, the compiler generates the following problem:

```
(define (problem save_query_result_locally)
 (:objects db_name query_string
          filename format - char_array)
 (:init )
 (:goal (exists (?result - Seq)
   (and (query_result ?result db_name query_string)
        (contains filename ?result)
        (in_format filename format)))))
```

The problem and the domain above are given to the planner. The planner finds the plan in Figure 7 using a backward search. Nevertheless, the semantics of the BioPerl example are not completely specified. The procedures `bio_db_genbank_new` and `bio_db_query_genbank_new` are from the Bio::DB::GenBank and Bio::DB::Query::GenBank Perl modules. Nothing in the AA indicates that GenBank should have been used. There are several alternative databases: SwissProt, GenPept, EMBL, SeqHound, or RefSeq. If these modules are made available to the compiler, then there will be alternative ways of implementing the AA by creating different DB and Query objects. Presumably, the application programmer knows which database they want to use, and can respond to a compiler prompt to select a specific plan.



Ø
Ø
tmp3 = bio_db_query_genbank_new(db_name, query_string)
tmp2 = bio_db_genbank_new()
tmp3 - Query
tmp2 - DB
equal(tmp3.db, db_name)
equal(tmp3.query, query)
tmp1 = get_stream_by_query(tmp2, tmp3)
Ø
tmp1 - Stream
stream_for_query(tmp1, tmp3)
tmp4 = bio_seqio_new(filename, format)
tmp0 = next_seq(tmp1)
tmp0 - Seq
tmp4 - SeqIO
query_result(tmp0, db_name, query_string)
equal(tmp4.file, filename)
equal(tmp4.format, format)
Planning Order
Call Order
write_seq(tmp0, tmp4)
contains(filename, tmp0)
in_format(filename, format)

Figure 7. The state-space for save_query_result_locally: irrelevant actions are not shown, and each box shows only what predicates and objects are added in that state. The operators are considered relevant by the planner in the direction of the arrows, but they are applied in the opposite direction to yield Figure 1. Note the order in which temporary objects are created.

[5] `http://doc.bioperl.org`

[6] BLAST = Basic Local Alignment Search Tool

6.4 AA: blast_sequence

We can write another AA for using BLAST.

```
algorithm (report, seq)
blast_sequence(program, db_name, id, sequence)
  => { report_for_blasting(report, program,
         db_name, seq),
       seq.id == id, seq.seq == sequence };
```

If the AA is called and its return values are assigned to variables of type Report and Seq, then we have the following problem:

```
(define (problem blast_sequence)
 (:objects program db_name id sequence - char_array)
 (:init )
 (:goal
  (exists (?report - Report ?seq - Seq)
   (and (report_for_blasting ?report program
          db_name ?seq)
        (equal ?seq.id id)
        (equal ?seq.seq sequence)))))
```

The plan found is:

```
tmp2 = bio_seq_new(id, sequence);
tmp1 = bio_tools_run_standaloneblast_new(program,
  db_name);
tmp0 = blastall(tmp1, tmp2);
bind ?report to tmp0, ?seq to tmp2
```

6.5 Unobtrusive Interactive Compilation and Optimizations

Trust Levels The compiler can provide an option to customize the amount of interaction. We call the degrees of interaction "trust levels" because less interaction implies that the programmer trusts the compiler to make acceptable choices. *Level 0, Very Low:* At this level the compiler never selects a plan without consulting the programmer, even if there is only one possible plan. *Level 1, Low:* If there is only one possible plan, then the compiler will not consult the programmer. *Level 2, Medium:* The compiler will attempt to choose among multiple plans on its own. If the choice is not clear, then the compiler will consult the programmer. *Level 3, High:* At this level there is no interaction. The compiler always selects a plan, even if this means choosing one at random when there is insufficient information.

Decision Cache A decision cache stores the programmer's choices to provide the illusion of a normal compilation process. If during a subsequent compilation the programmer would be presented with a question that has been answered previously, then the compiler uses the answer in the cache and does not consult the programmer. Command-line arguments may be used to clear the cache either before or after compiling, or to ignore the cache. The decision cache is indexed with the question. For example, a particular planning problem may result in three potential solutions, from which the programmer selects the second. During a subsequent compilation, the same planning problem is encountered, but instead of asking the programmer to choose among the three potential solutions, the compiler automatically uses the second solution.

Performance Planning introduces compile-time overhead, but no run-time overhead. The compile-time overhead is expected to be less than the time the programmer saves by not having to learn every detail of the library. Although we have not done studies with actual programmers, we show in Table 1 that the overhead per AA call is small and conclude that it is not likely to be a significant fraction of the total compile time. The planner is interpreted and could be optimized further by implementing it in a compiled language. If performance does become an issue, a solution cache can store the planner's output for a given problem. If the same problem is encountered again, then the compiler can use the plans in the cache instead of executing the planner. Plans would need to be evicted from the cache if they used procedures no longer available to the compiler, and the cache would need to be cleared to take advantage of new procedures that have become available.

Table 1. Performance of our planner on various problems; these numbers were collected on a 1.6GHz AMD Athlon processor under Linux by taking the arithmetic mean of three runs. For the run of the dummy AA, four char[] variables were available at the call site. The timing resolution was 10ms.

domain	abstract algorithm	create	plans	time(ms)
bioperl	dummy	Seq	32	300
bioperl	save_query_result_locally	Seq	1	3270
isort	sort	int[]	2	800
isort	sort	float[]	0	30

7. Conclusions and Future Work

We have shown that a novel language feature for DSLs can be implemented by having the compiler use an AI planner. We have demonstrated this feature using a real case of composition from the BioPerl library and explained how the compiler and planner work together. The case showed that incomplete specifications are useful. We advocate that developers of new languages and libraries should support composition. Our planner is fully-implemented and supports the compositions shown in this paper plus several others. Our compiler development is in progress; it recognizes AA calls, starts the planner, and prompts the programmer to make choices, but the flow analysis is not yet finished. Future work will examine the interplay of composition with other language features, error handling, plans with branches (conditional planning) and loops, the applicability of our technique across more domains, and the experiences of programmers using composition.

Acknowledgments

We thank Michael Gribskov and Ying Li for a brief but informative discussion concerning the use of software libraries in the biological sciences. We thank Bob Givan for explaining how our work differs from traditional planning. We thank all of our colleagues at Purdue – particularly Sam Midkiff, T. N. Vijaykumar, Ethan Schuchman, Long Fei, Eddie Pettis, and Ayon Basumallik – for improving our presentation of an interdisciplinary topic.

References

[1] T. Ball et al. Automatic Predicate Abstraction of C Programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2001.

[2] R. Balzer. A 15-year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering*, 11(11):1257–1268, November 1985.

[3] D. R. Barstow. Domain-Specific Automatic Programming. *IEEE Transactions on Software Engineering*, 11(11):1321–1336, November 1985.

[4] D. Batory. The Road to Utopia: A Future for Generative Programming. In C. Lengauer et al., editors, *Domain-Specific Program Generation (Dagstuhl)*, pages 1–18, 2004.

[5] A. W. Biermann. Approaches to Automatic Programming. *Advances in Computers*, 15:1–63, 1976.

[6] J. Blythe et al. The Role of Planning in Grid Computing. In *Proceedings of the International Conference on Automated Planning and Scheduling*, June 2003.

[7] A. Borgida, J. Mylopoulos, and R. Reiter. On the Frame Problem in Procedure Specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[8] L. C. Briand, T. Langley, and I. Wieczorek. A Replicated Assessment and Comparison of Common Software Cost Modeling Techniques. In *Proceedings of the International Conference on Software Engineering*, pages 377–386, 2000.

[9] J. R. Buchanan and D. C. Luckham. On Automating the Construction of Programs. Technical report, Stanford Artificial Intelligence Project, 1974.

[10] R. Cytron et al. Efficiently Computing Static Single Assignment Form and the Program Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[11] R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[12] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 2002.

[13] M. Ghallab et al. PDDL – the Planning Domain Definition Language, Version 1.2. Technical Report DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[14] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning Theory and Practice*. Morgan Kaufmann Publishers, 2004.

[15] R. Givan. Inferring Program Specifications in Polynomial-Time. In *Proceedings of the Static Analysis Symposium*, 1996.

[16] K. Golden. A Domain Description Language for Data Processing. In *Proceedings of the International Conference on Automated Planning and Scheduling, Workshop on the Future of PDDL*, 2003.

[17] A. Gomez-Perez, O. Corcho, and M. Fernandez-Lopez. *Ontological Engineering: with Examples from the Areas of Knowledge Management, e-Commerce and the Semantic Web*. Springer, 2004.

[18] M. Gribskov. President, International Society for Computational Biology; Professor of Biological Sciences and Computer Science, Purdue University. Personal communication, September 2005.

[19] J. V. Guttag and J. Homing. Larch: Languages and Tools for Formal Specification. *Texts and Monographs in Computer Science*, 1993.

[20] S. Z. Guyer and C. Lin. Broadway: A Compiler for Exploiting the Domain-Specific Semantics of Software Libraries. *Proceedings of the IEEE*, 93(2):342–357, February 2005.

[21] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.

[22] E. N. Houstis et al. PYTHIA-II: a Knowledge/Database System for Managing Performance Data and Recommending Scientific Software. *ACM Transactions on Mathematical Software*, 26(2):227–253, 2000.

[23] N. Jefferson and S. Riddle. Towards a Formal Semantics of a Composition Language. In *Proceedings of the Workshop on Composition Languages at ECOOP*, 2003.

[24] K. Kennedy et al. Telescoping Languages: A System for Automatic Generation of Domain Languages. *Proceedings of the IEEE*, 93(3):387–408, 2005.

[25] K. Kennedy, C. Koelbel, and R. Schreiber. Defining and Measuring the Productivity of Programming Languages. *International Journal of High Performance Computing Applications*, 18(4):441–448, 2004.

[26] J. Kim, M. Sparagen, and Y. Gil. An Intelligent Assistant for Interactive Workflow Composition. In *Proceedings of the International Conference on Intelligent User Interfaces*, pages 125–131, 2004.

[27] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[28] S. Lerner, T. Millstein, and C. Chambers. Automatically Proving the Correctness of Compiler Optimizations. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2003.

[29] D. Mandelin et al. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2005.

[30] Z. Manna and R. Waldinger. Fundamentals of Deductive Program Synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, August 1992.

[31] B. D. Martino, G. Iannello, and H. P. Zima. An Automated Algorithmic Recognition Technique to Support Parallel Software Development. In *Proceedings of the International Workshop on Parallel and Distributed Software Engineering*, May 1997.

[32] V. Menon and K. Pingali. A Case for Source-Level Transformations in MATLAB. In *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages 53–65, October 1999.

[33] V. Menon and K. Pingali. High-Level Semantic Optimization of Numerical Codes. In *International Conference on Supercomputing*, pages 434–443, June 1999.

[34] R. Metzger and Z. Wen. *Automatic Algorithm Recognition and Replacement*. MIT Press, 2000.

[35] D. S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. The SHOP planning system. *AI Magazine*, Fall 2001.

[36] G. C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2000.

[37] E. P. D. Pednault. ADL and the State-Transition Model of Action. *Journal of Logic and Computation*, 4(5):467–512, 1994.

[38] E. S. Raymond. *The Art of Unix Programming*, chapter 8: Minilanguages. Addison-Wesley, 2004.

[39] J. R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976.

[40] C. Rich and R. C. Waters. Automatic Programming: Myths and Prospects. *IEEE Computer*, 21(8):40–51, August 1988.

[41] P. Schnorf, M. Ganapathi, and J. L. Hennessy. Compile-Time Copy Elimination. *Software Practice and Experience*, 23(11):1175–1200, November 1993.

[42] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic Program Specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25(4):452–499, 2003.

[43] J. M. Siskind and D. A. McAllester. Screamer: A Portable Efficient Implementation of Nondeterministic Common Lisp. Technical Report IRCS-93-03, Institute for Research in Cognitive Science, University of Pennsylvania, 1993.

[44] J. E. Stajich et al. The Bioperl Toolkit: Perl Modules for the Life Sciences. *Genome Research*, 12(10):1611–1618, October 2002.

[45] M. Stickel et al. Deductive Composition of Astronomical Software from Subroutine Libraries. In *Proceedings of the International Conference on Automated Deduction*, pages 341–355, June 1994.

[46] N. Thomas et al. A Framework for Adaptive Algorithm Selection in STAPL. In *Proceedings of the Symposium on the Principles and Practice of Parallel Programming*, June 2005.

[47] M. T. Vandevoorde. *Exploiting Specifications to Improve Program Performance*. PhD thesis, MIT, 1994.

[48] S. Weerawarana et al. PYTHIA: a Knowledge-Based System to Select Scientific Algorithms. *ACM Transactions on Mathematical Software*, 22(4):447–468, 1996.

[49] D. S. Weld. Recent Advances in AI Planning. *AI Magazine*, pages 93–123, Summer 1999.

[50] S. Woods and Q. Yang. The Program Understanding Problem: Analysis and a Heuristic Approach. In *Proceedings of the International Conference on Software Engineering*, pages 6–15, 1996.

[51] H. Yu, D. Zhang, and L. Rauchwerger. An Adaptive Algorithm Selection Framework. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2004.

[52] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.