Problem Classification using Program Checking

Christian S. Collberg
Department of Computer Science
University of Arizona
Tucson, AZ
collberg@cs.arizona.edu

Todd A. Proebsting
Microsoft Research
One Microsoft Way
Redmond, WA
toddpro@microsoft.com

Abstract

We describe AλgoVista, a web-based search engine that assists computer scientists find algorithms and implementations that solve specific problems. AλgoVista also allows algorithm designers to advertise their results in a forum accessible to programmers and theoreticians alike.

AλgoVista is not keyword based. Rather, users provide *input*$\Rightarrow$*output* samples that describe the behavior of their needed algorithm. This *query-by-example* requires no knowledge of specialized terminology—the user only needs an ability to formalize her problem.

AλgoVista's search mechanism is based on a novel application of *program checking*, a technique developed as an alternative to program verification and testing.

AλgoVista operates at `http://algovista.com`.

1 Background

Frequently, working software developers encounter a problem with which they are unfamiliar, but which—they suspect— has probably been previously studied. Just as frequently, algorithm developers work on problems that they suspect have practical applications.

Unfortunately, the programmer with a problem in search of a solution and the theoretician with a solution in search of an application are unlikely to connect across the geographical and linguistic chasms that often separate the two. In many organizations working programmers do not have easy access to a theoretician, and, when they do, they often find communication difficult.

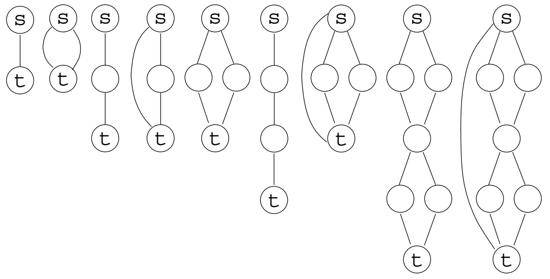
In this paper we will describe AλgoVista, a web-based, interactive, searchable, and extensible database of problems and algorithms designed to bring together applied and theoretical computer scientists. Programmers can query AλgoVista to look for relevant theoretical results, and theoretical computer scientists can extend AλgoVista with problem solutions.

AλgoVista relies on a novel application of a *program* (or *result*) *checking*. Program checking has been developed by Manuel Blum and others [3–5,10,16,17,19] as an alternative to program verification and testing. Program checking extends programs with *checkers* that verify the correctness of the results they compute.

1.1 Two Motivating Episodes

To motivate the need for specialized search engines for computer scientists, we will consider two concrete episodes from the experience of the authors.

Working on the design of graph-coloring register allocation algorithms, Todd showed his theoretician colleague Sampath Kannan the following graphs:

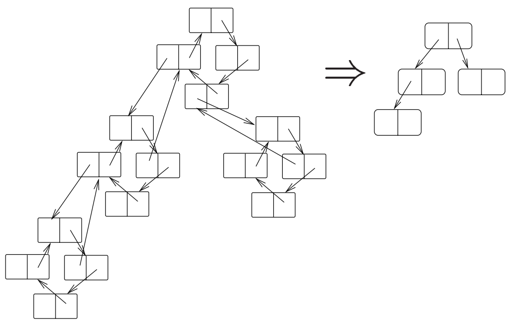


"Do these graphs mean anything to you?" Todd asked.

"Sure," Prof. Kannan replied, "they're series-parallel graphs."

This was the beginning of a collaboration which resulted in a paper in the *Journal of Algorithms* [13].

In a similar episode, Christian showed his theoretician colleague Clark Thomborson the following graph-transformation:



"Do you know what I am doing here?" Christian asked.

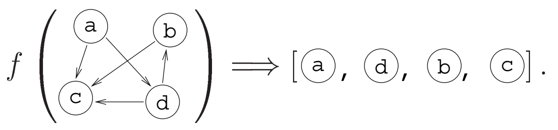
"Sure," Prof. Thomborson soon replied, "you're shrinking the biconnected components of the underlying (undirected) graph."

This result became an important part of a joint paper on software watermarking [7].

It's important to note that, while in both these episodes the authors (who consider themselves "theory-challenged") had a pretty good grasp of the problem they were working on, they lacked knowledge of the relevant terminology. Hence, standard keyword-based search techniques would not have been of much assistance. In these episodes, the theoretical computer scientist provided the crucial problem classification that allowed the authors to conduct further bibliographical searches themselves.

1.2 Interacting with AλgoVista

AλgoVista is an online database that stores and codifies problems, algorithms, and combinatorial structures developed within the Computer Science theory community. An applied computer scientist will typically interact with AλgoVista by providing *input⇒output* samples. AλgoVista will then search its database looking for problems that map *input* to *output*. As a concrete example, consider the following query:



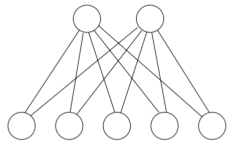
This query asks:

> "Suppose that from the linked structure on the left of the ⇒ I compute the list of nodes to the right. What function f am I then computing?"

AλgoVista might then respond with:

> "This looks like a *topological sort* of a *directed acyclic graph*. You can read more about topological sorting at `http://hissa.ncsl.nist.gov/~black/CRCDict/HTML/topologcsort.html`. A Java implementation can be found at `http://www.math.grin.edu/~rebelsky/Courses/152/97F/Outlines/outline.49.html`".

AλgoVista is also able to classify some simple combinatorial structures. Given the following query



AλgoVista might respond with:

> "This looks like a *complete bipartite graph*. You can read more about this structure at `http://www.treasure-troves.com/math/CompleteBipartiteGraph.html`."

1.3 Organization

The remainder of this paper is organized as follows. Section 2 introduces *program checking* and describes how *checklets* (program checkers in AλgoVista) are used as the basic entries in AλgoVista's database. Section 3 presents the overall architecture of AλgoVista and discusses relevant security issues. Section 4 describes the design of the AλgoVista query language and type system. Section 5 introduces *query transformations* that the system uses to bridge any potential semantic gap between user queries and checklets. Section 6 describes how advanced type analysis can speed up searching. Section 7 evaluates the performance of the search algorithms. Section 8 discusses related work, and Section 9, finally, summarizes our results.

2 Program Checking

AλgoVista can be seen as a novel application of *program checking*, an idea popularized by Manuel Blum and his students. The idea behind program checking is simply this. Suppose we are concerned about the correctness of a procedure P in a program we are writing. We intend for P to compute a function f, but we are not convinced it does so. We have three choices:

1. We can attempt to *prove* that $P \equiv f$ over the entire domain of P.
2. We can test that $P(x) = f(x)$, where x is drawn from a reasonable domain of test data.
3. We can include a *result checker* C_f^P with the program. For every actual input x given to P, the result checker checks that $P(x) = f(x)$.

We normally require C_f^P and P to be independent of each other; i.e. they should be programmed using very different algorithms. We also want the checker to be *efficient*. To ensure that these conditions are met, it is generally expected that a result checker C_f^P should be asymptotically faster than the program P that it checks. That is, we expect that if P runs in time T then C_f^P should run in time $o(T)$.

2.1 Checklets: Result Checkers in AλgoVista

The AλgoVista database consists of a collection of result checkers which we call *checklets*. A checklet typically takes a user query *input⇒output* as input and either *accepts* or *rejects*. If the checklet accepts a query, it also returns a description of the problem it checks for.

Figure 1 shows some simple checklets. Figure 1 (b), is a particularly interesting checklet for topological sorting. Any acyclic graph will typically have more than one topological order. It is therefore not possible for the checklet to simply run a topological sorting procedure on the input graph and compare the resulting list of nodes with the output list given in the query. Rather, the checklet must, as shown in Figure 1 (b), first check that every node in the input graph occurs in the output node list, and then check that if node f comes before

Figure 1: Some simple checklets.

(a) A *sorting* checklet. Its speed depends on how fast we can compare two multisets for equality. If the elements are small enough we can use bucket sort in $\mathcal{O}(n)$ time. Otherwise, we can use a hashing scheme that runs in time proportional to the size of the hash table.

```
checklet sorting (int[] input ⇒ int[] output)
   if length(input) ≠ length(output) then reject
   for i←1 to length(output)-1 do
      if output[i] > output[i+1] then reject
   if the multisets input and output do not contain the same elements then reject
   accept http://hissa.ncsl.nist.gov/~black/CRCDict/termsArea.html#sort
```

(b) A *topological sorting* checklet.

```
checklet topologicalSort (Digraph inGraph ⇒ Node[] outNodeList)
   if the nodes of inGraph ≠ outNodeList then reject
   for (f,t) ← the edges of inGraph do
      if index of f in outNodeList > index of t in outNodeList then reject
   accept http://hissa.ncsl.nist.gov/~black/CRCDict/HTML/topologcsort.html
```

node t in the output list then there is no path $t \rightsquigarrow f$ in the input graph.

AλgoVista currently contains some ninety problem descriptions, some of which are listed in Table 1.

2.2 Examples

We will next examine two examples of what AλgoVista can do.

Example 1: Suppose Bob is trying to write a program that identifies the locations for a new franchise service. Given a set of potential locations, he wants the program to compute the largest subset of those locations such that no two locations are close enough to compete with each other. It is trivial for him to compute which pairs of locations would compete, but he does not know how to compute the feasible subset. He starts by trying to come up with an example of how his program should work:

- If there are three locations a, b, c and a competes with b and c, then the best franchise locations are b and c.

If Bob is unable to come up with his own algorithm for this problem he might turn to one of the search-engines on the web. But, which keywords should he use? Or, Bob could consult one of the algorithm repositories on the web, such as `http://www.cs.sunysb.edu/~algorith/`, which is organized hierarchically by category. But, in which category does this problem fall? Or, he could enter the example he has come up with into AλgoVista at `algovista.com`:

```
[a--b,a--c]==>[c,b]
```

This query expresses:

"If the input to my program is two relationships, one between `a` and `b` and one between `a` and `c`, then the output is the collection `[b,c]`."

Another way of thinking about this query is that the input is a graph of three nodes `a`, `b`, and `c`, and edges `a-b` and `a-c`, but it is not necessary for Bob to know about graphs. AλgoVista returns to Bob a link directly to `http://www.cs.sunysb.edu/~algorith/files/independent-set.shtml` which contains a description of the Maximal Independent Set problem. From this site there are links to implementations of this problem.

Example 2: Suppose Bob is writing a simple DNA sequence pattern matcher. He knows that given two sequences $\langle a, a, t, g, g, g, c, t\rangle$ and $\langle c, a, t, g, g\rangle$, the matcher should return the match $\langle a, t, g, g\rangle$, so he enters the query

```
([a,a,t,g,g,g,c,t],[c,a,t,g,g]) ==> [a,t,g,g]
```

into AλgoVista which (within seconds) returns the link `http://evo.apm.tuwien.ac.at/AlgDesignManual/BOOK/BOOK5/NODE208.HTM#SECTION03178000000000000000` to a description of the longest common subsequence problem.

2.3 Checklet Construction

Much research has gone into the search for efficient result checkers for many classes of problems. In some cases, efficient result checkers are easy to construct. For example, let $P(x)$ return a factor of the composite integer x. This is generally thought to be a computationally difficult problem. However, checking the correctness of a result returned by P is trivial; it only requires one division. On the other hand, let $P(x)$ return a least-cost traveling salesman tour of the weighted graph x. Checking that a given tour is actually a minimum-cost tour is as expensive as finding the tour itself.

Table 1: Partial list of problem and graph descriptions found in AλgoVista.

Eulerian graph	Maximal independent set	Transitive closure
Longest common subsequence	Matching	Clique problem
Independent set	Proper edge coloring	Permutation
Perfect matching	Euler cycle	Spanning Tree
AVL Tree	Biconnected Graph	Undirected Graph
Complete graph	Connected graph	Single destination shortest path
All pairs shortest path	Single pair shortest path	Strongly connected Graph
Single source shortest path	Bipartite Graph	Combination
Maximum bipartite matching	Clique	Least common multiple
Directed Acyclic Graph	Maximum consecutive subsequence	Hamiltonian cycle
Articulation points		

In some cases it may be difficult to construct checklets which run in an acceptable amount of time. This is particularly true of NP-hard problems for which it would seem to be impossible to find polynomial time result checking algorithms. In these cases we may have to use *spot-checking* [10], a recent development in result checking, to check hard problems probabilistically.

In spite of these problems, checklets are typically very simple to write, for the following reasons:

1. Checklets do not actually have to compute the result that the problem they are checking for does, they only have to check that the output is a valid result of the input. For example, the sorting checklet does not have to sort its input, just check that the output array is sorted.

2. The queries that users will submit to AλgoVista are almost always extremely short. Therefore, checklets do not have to worry about being efficient; the simplest, most straight-forward algorithm will often be adequate.

3. Checklets can often "cheat." For example, a sorting checklet should not only check that the output array is sorted, it should also check that the output array contains the same elements as the input array. This second step is moderately hard if the input array contains duplicates. A relaxed checklet could omit this step, with the result that it would *accept* slightly more often than it should. This will most likely not be a problem since – like all search engines – we expect AλgoVista to sometimes return false positives.

Checklets that operate on floating-point numbers present their own set of problems concerning floating-point equality. For example, which, if any, of the queries ⌜2.0 ⇒ 1.4142135623⌝, ⌜2.00000 ⇒ 1.4140⌝, and ⌜2.0 ⇒ 1.0⌝ should a *floating-point square root* checklet accept? Checklets can, of course, define their own equality primitives, but AλgoVista provides a default heuristic that works well in most situations: floating-point comparisons are done in the *minimum* precision of any floating-point number in the input query.

3 System Overview

A typical user will search AλgoVista by submitting a query through the AλgoVista web page, where it is matched against the checklets in the checklet database. The output from any accepting checklet is transferred back to the client and presented to the user.

To extend the database with new problem classifications, a user downloads a checklet template, modifies and tests it, and uploads the new checklet into the server where it is added to the checklet database. AλgoVista is the first search engine on the web to allow arbitrary users to upload executable code into its database. In [8] we address a number of related security issues.

The basic AλgoVista search algorithm is very simple:

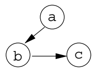
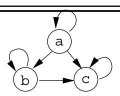
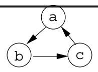
```
function search (query)
    $q$ ← parse(query)
    responses ← {}
    for every combination of query
        transformations $\mathcal{T}_1(\mathcal{T}_2(\cdots))$ do
        $q'$ ← $\mathcal{T}_1(\mathcal{T}_2(\cdots q \cdots))$
        for every checklet $c$ in the database do
            if $c$ accepts $q'$ with response $r$ then
                responses ← responses ∪ {$r$}
    return responses
```

The algorithm is essentially an exhaustive search: a query is submitted to every checklet in the database, and the response of every accepting checklet is returned. In Section 5 we show that a query may also undergo a set of *representation transformations* prior to being submitted. These transformations try to compensate for the fact that user queries and checklets may use different data representations for the same problem. In Section 6 we explore more sophisticated algorithms that speed up search times significantly.

4 The Query Language

The primitives of QL, the AλgoVista query language, include integers, floats, booleans, lists, tuples, atoms, and links. Links are (directed and undirected) edges between atoms that are used to build up linked structures such as graphs and trees. Special syntax was provided for these structures since we anticipate that many

Figure 2: Example QL queries.

#	QL query	Query explanation
①	`[a->b,b->c] ==> [a->a,a->b,a->c,b->b,b->c,c->c]` **Query result:** ⟨`Transitive closure`⟩	What function maps  to  ?
②	`[a->b,b->c,c->a]` **Query result:** ⟨`Strongly connected graph`⟩	What kind of graph is this:  ?
③	`([3,7],[5,1,6]) ==> [5,1,6,3,7]` **Query result:** ⟨`List append`⟩	What function maps the lists `[3,7]` and `[5,1,6]` to the list `[5,1,6,3,7]`?
④	`[a->c,a->d,b->c,d->c,d->b] ==> [a,d,b,c]` **Query result:** ⟨`Topological sort`⟩	List of edges representation. Node set is implied.
⑤	`[[0,0,1,1],[0,0,1,0],[0,0,0,0],[0,1,1,0]] ==> [1,4,2,3]` **Query result:** ⟨`Topological sort`⟩	Adjacency matrix representation.
⑥	`[a->[c,d],b->[c],c->[],d->[c,b]] ==> [a,d,b,c]` **Query result:** ⟨`Topological sort`⟩	List of neighbors representation. Node set is implied.

AλgoVista users will be wanting to classify graph structures and problems on graphs.

The following grammar shows the concrete syntax of the query language:

```
S      →  int | float | bool |
          S `==>´ S |
          atom [ `/´S ] |
          atom `->´ [ `/´S ] atom |
          atom `--´ [ `/´S ] atom |
          `[´[ S { `,´S } ] `]´ |
          `(´ S `,´ S `)´
bool   →  `true´ | `false´
atom   →  `a´...`z´
int    →  `0´...`9´ { `0´...`9´ }
float  →  int `.´int
```

⌜S `==>` S⌝ maps inputs to outputs, ⌜`(` S `,` S `)`⌝ represents a pair of elements, and ⌜`[`S { `,`S } `]`⌝ represents a list of elements. Atoms, ⌜`atom` [`/`S]⌝, are one-letter identifiers that are used to represent nodes of linked structures such as graphs and trees. They can carry optional node data. Links between nodes can be directed ⌜`atom ->` [`/`S] `atom`⌝, or undirected ⌜`atom --` [`/`S] `atom`⌝, and can also carry edge data.

Figure 2 gives some example queries. In the query in Figure 2 ①, a directed graph is mapped to a directed graph. The query in Figure 2 ② asks AλgoVista to classify a particular graph, which turns out to be a strongly connected directed graph.

Figure 2 ③, finally, shows a query that maps a pair of vectors to a vector:

⌜`([3,7],[5,1,6])==>[5,1,6,3,7]`⌝.

AλgoVista returns the result ⟨`List append`⟩ since `append([5,1,6],[3,7])=[5,1,6,3,7]`. To arrive at this result AλgoVista first swapped the input pair using a *query transformation*. We discuss this further in Section 5.

5 Query Transformations

Early on in the design of AλgoVista we realized that there is often a representational gap between a user's query and the checklet that is designed to match this query. For example, there are any number of reasonable ways for a user to express a topological sorting query, including representing the input graph as a *list of edges*, an *adjacency matrix*, or a *list of neighbors*. These queries are shown in Figure 2 ④–⑥. The corresponding topological sorting checklet, on the other hand, might expect the input graph only in a matrix form.

AλgoVista provides a set of *query transformations* that will automatically mutate queries between common representations. For example, given the topological sorting query in Figure 2 ④, AλgoVista would automatically produce the queries in Figure 2 ④–⑥, all of which would be matched against the checklets in the checklet database. Figure 3 lists some of the transformations currently in use by the search engine. See [8] for a complete listing.

Transformation `Int2Float` in Figure 3 promotes a integer to a float. Transformation `FlipPair` swaps the elements of a pair. `List2VectorB` and many other transformations are concerned with transforming between different representations of various linked structures.

Consider a user who wants to submit a query con-

Figure 3: Query transformations. $\emptyset$ represents `Null`. Greek letters are type variables. Examples are in the format *signature*:*query*.

`Int2Float:Int`$\Rightarrow$`Float`
Description: Convert an integer to a float.
Example: `Int:3`$\Rightarrow$`Float:3.0`
`Int2Bool:Int`$\Rightarrow$`Bool`
Description: Convert 0/1 to false/true.
Condition: The integer must be 0 or 1.
Example: `Int:0`$\Rightarrow$`Bool:false`
`FlipPair:Pair(`α`,`β`)`$\Rightarrow$`Pair(`β`,`α`)`
Description: Swap the elements in a pair.
Example: `Pair(Int,Float):(1,2.3)`$\Rightarrow$`Pair(Float,Int):(2.3,1)`
`Vector2Pair:Vector(`α`)`$\Rightarrow$`Pair(`α`,`α`)`
Description: Convert a vector to a pair.
Condition: The vector must contain exactly 2 elements.
Example: `Vector(Int):[1,2]`$\Rightarrow$`Pair(Int,Int):(1,2)`
`Vector2VectorPair:Vector(`α`)`$\Rightarrow$`Pair(Vector(`$\emptyset$`),Vector(`β`))`
Description: Convert a vector of edges to a pair of vectors of nodes and edges.
Example: `Vector(DEdge(`$\emptyset$`)):[a->b,c->d]`$\Rightarrow$`Pair(Vector(`$\emptyset$`),Vector(`$\emptyset$`)):([a,b,c,d],[a->b,c->d])`
`List2VectorB:List(`$\emptyset$`,`α`)`$\Rightarrow$`Vector(`α`)`
Description: Convert a linked list to a vector.
Example: `List(`$\emptyset$`,Int):([a,b,c,d],[a->/1 b,b->/2 c,c->/3 d])`$\Rightarrow$`Vector(Int):[1,2,3]`

taining the list of integers ⌜`[5,1]`⌝. There are several ways to represent this list, and our user decides on a linked-list representation with edge-weights:

$$\begin{tikzcd} a \arrow[r, "5"] & b \arrow[r, "1"] & c \end{tikzcd}.$$

In our query language this would be expressed as

⌜`[a->/5b,b->/1c]`⌝.

However, a particular checklet might expect the list of integers to be given in a vector representation:

⌜`[5,1]`⌝.

Figure 4 shows how AλgoVista's transformation engine would mutate the original linked representation to the vector representation (using the transformation `List2VectorB`), a form which would be acceptable to the checklet.

As show in Figure 4, further transformations will mutate the original query into pairs of integers, booleans, and reals. The basic AλgoVista search algorithm in Section 2 would hand off all twelve mutated queries to all the checklets in the checklet database.

6 Query Optimization

In Section 3 we described a straight-forward algorithm that employs exhaustive search to submit every possible mutation of a query to every checklet in the checklet database. Obviously, with dozens of transformations and maybe hundreds of checklets this procedure will be prohibitively expensive.

In this section we will examine a more sophisticated search algorithm that explores the fact that queries, checklets, and transformations are all *typed*. The type system corresponds almost one-to-one to the concrete syntax given in Section 4. The following type assignments map concrete syntax into types:

$$\begin{array}{lcl}
\mathcal{T}[\![\texttt{int}]\!] & = & \texttt{Int} \\
\mathcal{T}[\![\texttt{float}]\!] & = & \texttt{Float} \\
\mathcal{T}[\![\texttt{true}]\!] & = & \texttt{Bool} \\
\mathcal{T}[\![\texttt{false}]\!] & = & \texttt{Bool} \\
\mathcal{T}[\![S_1\ \grave{}\texttt{==>}'\ S_2]\!] & = & \texttt{Map}(\mathcal{T}[\![S_1]\!],\mathcal{T}[\![S_2]\!]) \\
\mathcal{T}[\![\grave{}\texttt{(}'\ S_1\ \grave{}\texttt{,}'\ S_2\ \grave{}\texttt{)}']\!] & = & \texttt{Pair}(\mathcal{T}[\![S_1]\!],\mathcal{T}[\![S_2]\!]) \\
\mathcal{T}[\![\grave{}\texttt{[}'[\ S_1\ \{\ \grave{}\texttt{,}'S_2\ \}\ ]\ \grave{}\texttt{]}']\!] & = & \texttt{if}\ \mathcal{T}[\![S_1]\!]=\mathcal{T}[\![S_2]\!] \\
 & & \texttt{then Vector}(\mathcal{T}[\![S_1]\!]) \\
 & & \texttt{else}\ \bot \\
\mathcal{T}[\![\texttt{atom}/S]\!] & = & \texttt{Node}(\mathcal{T}[\![S]\!]) \\
\mathcal{T}[\![\texttt{atom}\ \grave{}\texttt{->}'/S\ \texttt{atom}]\!] & = & \texttt{DEdge}(\mathcal{T}[\![S]\!]) \\
\mathcal{T}[\![\texttt{atom}\ \grave{}\texttt{--}'/S\ \texttt{atom}]\!] & = & \texttt{UEdge}(\mathcal{T}[\![S]\!])
\end{array}$$

For example, the query ⌜`([1,2],[3,4])==>[4,6]`⌝ has the type

⌜`Map(Pair(Vector(Int),Vector(Int)),Vector(Int))`⌝

and the query ⌜`([a/3,b/2,c/1],[a->b,a->c])`⌝ has the type

⌜`Pair(Vector(Node(Int)),Vector(DEdge(Null)))`⌝.

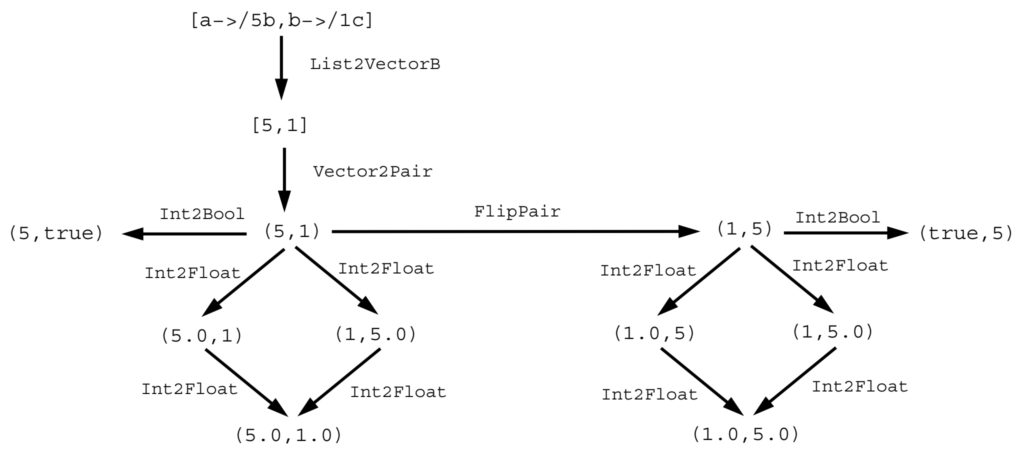
To see how type-analysis can help us speed up the search, consider a situation where we have two checklets

```
   FloatExp:  Map(Pair(Float,Int),Float)
   FloatAdd:  Map(Pair(Float,Float),Float)
```

where `FloatExp` checks for real exponentiation and `FloatAdd` checks for real addition, and two transformations

   `Int2Float:` `Int`$\Rightarrow$`Float`
   `FlipPair:` `Pair(`α`,`β`)`$\Rightarrow$`Pair(`β`,`α`)`

Figure 4: Query transformation example.



where `Int2Float` promotes an integer to a real and `FlipPair` commutes a pair.

Suppose the input query is ⌜`(2.0,2)==>4.0`⌝. This input has a signature of `Map(Pair(Float,Int),Float)`, and therefore can be tested immediately against the `FloatExp` checklet. Similarly, by applying the `Int2Float` transformation, the query can be transformed into ⌜`(2.0,2.0)==>4.0`⌝, which matches the signature of `FloatAdd`, and therefore can be submitted to that checklet.

It is trivial to determine that the query ⌜`true==>1`⌝ (which has the type `Map(Bool,Int)`) cannot match any of our (current) checklets, regardless of which transformations are applied. Still, the algorithm in Section 2 would apply all possible combinations of transformations to ⌜`true==>1`⌝ and submit any generated query mutation to every checklet in the database.

We will next show how precomputing *viable* transformations can speed up searching by eliminating any such useless transformations.

6.1 Fast Checking by Precomputation

Whenever a new checklet is added to the database, AλgoVista generates a new search procedure $\mathcal{S}_{\mathcal{T},\mathcal{C}}$ automatically. This procedure is hardcoded to handle exactly the set of transformations $\mathcal{T}$ which are available in the transformation database, and the set of checklets $\mathcal{C}$ which are currently available in the checklet database. $\mathcal{S}_{\mathcal{T},\mathcal{C}}$ is constructed such that given an input query q whose type is $\mathcal{T}[\![q]\!]$, $\mathcal{S}_{\mathcal{T},\mathcal{C}}$ will apply exactly those combinations of transformations to q that will result in *viable* mutated queries. A query is *viable* if it is correctly typed for checking by at least one checklet.

In other words, AλgoVista's optimized search procedure $\mathcal{S}_{\mathcal{T},\mathcal{C}}$ will never perform a useless transformation, one that could not possibly lead to a mutated query correctly typed for some checklet.

In order to apply transformations and to test checklets efficiently, AλgoVista determines the signature of an input query upon its arrival. Given the query's signature, AλgoVista knows exactly which, if any, checklets to test, and which, if any, transformations to apply. Furthermore, AλgoVista knows the exact signature of each newly-generated query because it knows the input query signature and how the transformation will transform the signature. (For example, AλgoVista knows that applying the `FlipPair` transform to `Map(Pair(Float,Int),Float)` will yield `Map(Pair(Int,Float),Float)`.) This observation yields a very simple, but highly optimized architecture for AλgoVista to apply transformations and test checklets based on signatures, in which there is one function per signature responsible for all the operations that affect queries of that signature. Each function has three parts: verifying the originality of the query, testing all matching checklets, and generating isomorphic queries by applying transformations. All generated queries are simply handed off to the function that handles their signature.

For the given checklets and transformations above, the function that handles the signature `Map(Pair(Float,Int),Float)` would look like this:

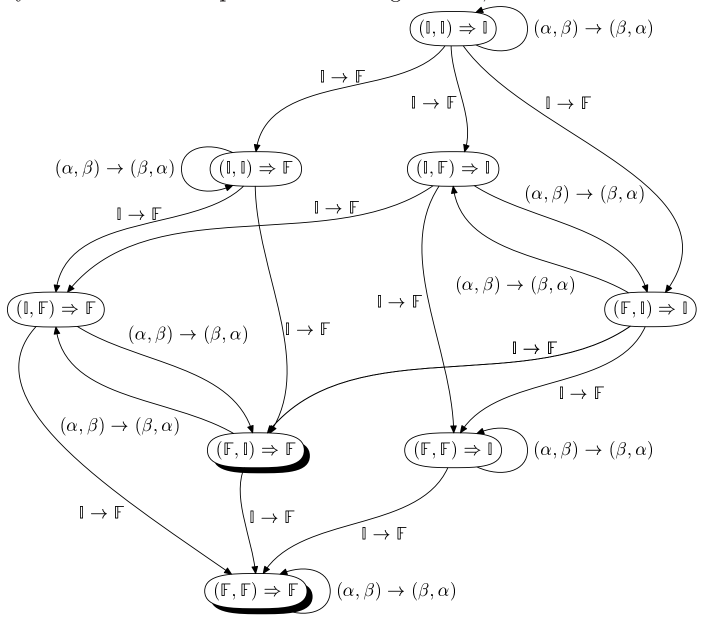
```
set FI_F_AlreadySeen;
function FI_F(query $Q$) {
    if $Q$ in FI_F_AlreadySeen then return;
    insert $Q$ into FI_F_AlreadySeen;

    Check if the FloatExp-query accepts $Q$;

    Apply Int2Float (whose signature is
    Map(Int,Float)) to $Q$, yielding $Q'$ (whose
    signature is Map(Pair(Float,Float),Float));

    Call FF_F($Q'$);
}
```

Figure 5: A query signature graph. The two transformations `Int2Float` and `FlipPair` are represented by $\mathbb{I} \rightarrow \mathbb{F}$ and $(\alpha, \beta) \rightarrow (\beta, \alpha)$, respectively. Shaded nodes represent *viable* signatures, those that have associated checklets.



$(\mathbb{I},\mathbb{I}) \Rightarrow \mathbb{I}$
$(\mathbb{I},\mathbb{I}) \Rightarrow \mathbb{F}$
$(\mathbb{I},\mathbb{F}) \Rightarrow \mathbb{I}$
$(\mathbb{I},\mathbb{F}) \Rightarrow \mathbb{F}$
$(\mathbb{F},\mathbb{I}) \Rightarrow \mathbb{I}$
$(\mathbb{F},\mathbb{I}) \Rightarrow \mathbb{F}$
$(\mathbb{F},\mathbb{F}) \Rightarrow \mathbb{I}$
$(\mathbb{F},\mathbb{F}) \Rightarrow \mathbb{F}$
$\mathbb{I} \rightarrow \mathbb{F}$
$(\alpha, \beta) \rightarrow (\beta, \alpha)$

The set `FI_F_AlreadySeen` prevents the same query mutation from being produced more than once, as in this example:

$$\begin{array}{lll} \texttt{(1,2)==>3} & \overset{\text{FlipPair}}{\Rightarrow} & \texttt{(2,1)==>3} \\ & \overset{\text{FlipPair}}{\Rightarrow} & \texttt{(1,2)==>3} \\ & \overset{\text{FlipPair}}{\Rightarrow} & \texttt{(2,1)==>3} \\ & \Rightarrow & \cdots \end{array}$$

The only non-trivial aspect of the generated function is knowing which transformations can be applied to a given signature, and *where*. For instance, given the query signature, `Map(Pair(Pair(Int,Float),Pair(Float,Int)))`, it is possible to apply the `FlipPair` transformation at any of the three `Pair`s in the query—even the nested ones.

In addition to the signature-specific functions, it is also necessary to generate a large decision tree that determines the signature of the original query before that query is dispatched to the appropriate function.

6.2 The Query Signature Graph

Figure 5 is a graphical representation of the functions that would be generated for the checklets and transformations in our running example. The nodes depict the signature-bound functions and the edges show transformations from one signature to another. The shaded nodes are those nodes that have associated checklets.

To construct this query signature graph we start with those signatures accepted by checklets—they are trivially acceptable. Then, for all of those signatures, we apply the *inverted* transformations wherever possible. I.e., at each step of this process we determine those signatures that are one transformation away from the given acceptable signature. By repeatedly applying these inverted transformations, all acceptable query transformations can be discovered and the graph can be constructed.

There is, however, one unfortunate complication to this architecture. Consider the following example:

$$\begin{array}{ll} \multicolumn{2}{l}{\texttt{[a->b,b->c]}} \\ \overset{\text{Vector2VectorPair}}{\Rightarrow} & \texttt{([a,b,c],[a->b,b->c])} \\ \overset{\text{Vector2VectorPair}}{\Rightarrow} & \texttt{([a,b,c],([a,b,c],[a->b,b->c]))} \\ \overset{\text{Vector2VectorPair}}{\Rightarrow} & \texttt{([a,b,c],([a,b,c],([a,b,c],} \\ & \qquad\qquad\qquad\qquad \texttt{[a->b,b->c])))} \\ \Rightarrow & \cdots \end{array}$$

In this particular example, the query ⌜`[a->b,b->c]`⌝ (representing a linked list $\langle a, b, c\rangle$) is transformed into ⌜`([a,b,c],[a->b,b->c])`⌝. This is the standard AλgoVista representation of a linked structure, a pair of a node-list and an edge-list. However, the `Vector2VectorPair` transformation can be re-applied to the edge-list in the transformed query, *ad infinitum.*

As it turns out, with any sufficiently rich set of transformations, it is always possible to generate an infinite number of signatures.

To avoid this problem, and to bound the number of signatures, we put a limit on the number of transformations that will be applied to any query. Typical values for this limit is four to six. This would seem to limit the usefulness of AλgoVista, but in practice this is not so. First of all, the exhaustive search algorithm in Section 2 is still available to those users who are willing to trade a somewhat longer response-time for a more complete response. Secondly, very deep chains of transformations will often mutate a query beyond recognition, resulting in spurious query results that have little meaning to the user.

With our current database of 95 checklets, with 28 unique signatures, and 23 transformations, AλgoVista can accept queries with 9828 different signatures.

The generation of the decision tree and all of the signature-specific functions is done automatically by a small Icon program [12].

7 Evaluation

Table 2 shows the search times for some typical queries. The times were collected by running each query four times and averaging the wall clock times of the last three runs. The reason for discarding the first measurement is that Java start-up times are quite significant and unpredictable. Furthermore, in web applications such as this one, programs are typically pre-loaded into (a large) primary memory and queries are fielded without any disk accesses.

The five columns of Table 2 show the query, the average wall clock times for the query using the exhaustive and the precomputed search, and the average wall clock times for generating all mutated queries using the exhaustive and the precomputed algorithms. In other words, the last two columns do not include the execution times of the checklets, just the time it takes to generate the transformed queries that would be submitted to the checklets.

Looking at Table 2 it is clear, as would be expected, that in most cases the precomputed search algorithm is superior to the exhaustive algorithm. However, it should be stressed that the comparison is inherently unfair. The exhaustive algorithm, although slower, will sometimes report results that the precomputed algorithm will overlook. The reason is that the precomputed algorithm limits the number of transformations that can be applied to a query, while the exhaustive one does not.

We expect that as the system grows with more checklets and query transformations, the performance of the precomputed search algorithm will greatly exceed that of the exhaustive algorithm. The reason is that the execution time of the exhaustive algorithm for a query Q is

$$\mathcal{O}(\texttt{\#mutations}(Q) \times \texttt{\#checklets})$$

while the execution time of the precomputed search algorithm is

$$\mathcal{O}(\texttt{\#viable\_mutations}(Q)),$$

where we expect

$$\texttt{\#viable\_mutations}(Q) \ll \texttt{\#mutations}(Q).$$

8 Related Work

A number of web sites, for example the *CRC Dictionary* [2] and the *Encyclopedia of Mathematics* [20], already provide encyclopedic information on algorithms, data structures, and mathematical results. Like all encyclopedias, however, they are of no use to someone unfamiliar with the terminology of the field they are investigating.

More relevant to the present research is *Sloane's On-Line Encyclopedia of Integer Sequences* [18]. This search service allows users to look up number sequences without knowing their name. For example, if a user entered the sequence $\ulcorner 1, 2, 3, 5, 8, 13, 21, 34 \urcorner$, the server would respond with "Fibonacci numbers." It is interesting to note that, although many of the entries in the database include a program or formula to generate the sequences, these programs do not seem to be used in searching the database. Similar search services are *Plouffe's Inverter* [15] where one can look up real numbers and the *Encyclopedia of Combinatorial Structures* [14].

Inductive Logic Programming (ILP) [1] is a branch of Machine Learning. One application of ILP has been the automatic synthesis of programs from examples and counter-examples. For example, given a language of list-manipulation primitives (`car`, `cdr`, `cons`, and `null`) and a set of examples

```
append([],[],[]).
append([1],[2],[1,2]).
append([1,2],[3,4],[1,2,3,4]).
```

an ILP system might synthesize the following Prolog-program for the `append` predicate:

```
append(A, B, B) :- null(A).
append(A,B,C) :-   car(A, X), cdr(A, Y),
                   append(Y, B, C1),
                   cons(X, C1, C).
```

Obviously, this application of ILP is far more ambitious than AλgoVista. While both ILP and AλgoVista produce programs from *input*⇒*output* examples, ILP *synthesizes* them while AλgoVista just retrieves them from its database. The ILP approach is, of course, very attractive (we would all like to have our programs written for us!), but has proven not to be particularly useful in practice. For example, in order to synthesize *Quicksort* from an input of sorting examples, a typical ILP system would first have to be taught *Partition* from a set of examples that split an array in two halves around a pivot element:

```
partition(3,[],[],[]).
partition(5,[6],[],[6]).
partition(7,[6],[6],[]).
partition(5,[6,3,7,9,1],[3,1],[6,7,9]).
```

AλgoVista is essentially a *reverse definition* dictionary for Computer Science terminology. Rather than looking up a term to find its definition (as one would in a normal dictionary), a reverse definition dictionary allows you to look up the term given its definition or an example. The *DUDEN* [6] series of pictorial dic-

Table 2: Timing measurements. Times are in seconds. Anomalous measurements are due to rounding errors and inadequate timer resolution. The measurements were collected on a lightly loaded Sun Ultra 10 workstation with a 333 MHz UltraSPARC-IIi CPU and 256 MB of main memory, running AλgoVista on Sun JDK 1.2.1.

	Search		Mutations	
Query	Exhaustive	Precomputed	Exhaustive	Precomputed
⌜`[1,3]==>2`⌝	0.41	0.37	0.12	0.39
⌜`(1,2)==>3`⌝	0.41	0.44	0.12	0.46
⌜`([a,b,c,d],[a->b,b->c,c->d,d->a])`⌝	0.79	0.32	0.17	0.31
⌜`([a,b,c,d],[a->b,b->c,c->d])==>[a,b,c,d]`⌝	2.2	0.55	0.83	0.54
⌜`[a->b,b->c,c->d]==>[a,b,c,d]`⌝	0.16	0.44	0.04	0.37
⌜`[a->b,b->c]==>[a,b]`⌝	0.69	0.39	0.13	0.42
⌜`([a,b,c,d],[a->/2b,b->/2c,c->/3d])==>3`⌝	2.1	0.45	0.78	0.49
⌜`[a->/1b,b->/2c,c->/3d]==>6`⌝	0.17	0.47	0.04	0.36
⌜`([1,2,3],[4,5,6])==>[1,2,3,4,5,6]`⌝	0.41	0.45	0.09	0.42
⌜`[6,5,4,3,2,1]==>[1,2,3,4,5,6]`⌝	0.05	0.34	0.01	0.34

tionaries is one example: to find out what that strange stringed musical instrument with a hand-crank and keys is called, you scan the *musical instruments* pages until you find the matching picture of the *hurdy-gurdy*. Another example is *The Describer's Dictionary* [11] where one can look up ⌜mixture of gypsum or limestone with sand and water and sometimes hair used primarily for walls and ceilings⌝ to find that this concoction is called *plaster*.

9 Summary

AλgoVista provides a unique resource to computer scientists to enable them to discover descriptions and implementations of algorithms without knowing theoretical nomenclature. AλgoVista is a web-based search engine that accepts *input⇒output* pairs as input and finds algorithms that match that behavior. This Query-By-Example mechanism relieves users of the burden of knowing terminology outside their domain of expertise. AλgoVista is extensible—algorithm designers may upload their algorithms into AλgoVista's database in the form of checklets that recognize acceptable input/output behavior.

AλgoVista is operational at `http://AlgoVista.com`.

The current implementation of AλgoVista provides several different search modes. Users can choose to search *comprensively* or *quickly*, using the exhaustive or precomputed search algorithms, respectively. Furthermore, searching can be done by *value* (the default search mode as described in this paper), by *signature*, or by *keyword*. Signature searching provides faster but less precise results by only matching the types of queries and checklets. Finally, AλgoVista also provides signature searching of the Java APIs.

It should be obvious that AλgoVista is not able to solve *all* programmers' problems *all* of the time. A programmer who is unable to abstract away from details of the problem at hand, formalizing it into one or two crisp examples will not be helped by AλgoVista. He will also not be helped by any other search tool or Computer Science text-book. Furthermore, a programmer who is not able to come up with these simple *input⇒output* samples for his problem also will not be able to generate test data for his finished program.

Acknowledgments: Will Evans pointed out the relationship between checklets and program checking. Dengfeng Gao implemented most of the checklets in the current database. Sean Davey implemented the Java API signature search algorithm. We thank them all.

References

[1] Francesco Bergadano and Daniele Gunetti. *Inductive Logic Programming – From Machine Learning to Software Engineering*. MIT Press, 1995. ISBN 0-262-02393-8.

[2] Paul E. Black. Algorithms, data structures, and problems – terms and definitions for the CRC dictionary of computer science, engineering and technology. `http://hissa.ncsl.nist.gov/~black/CRCDict`.

[3] Manuel Blum. Program checking. In Somenath Biswas and Kesav V. Nori, editors, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 560 of *LNCS*, pages 1–9, Berlin, Germany, December 1991. Springer.

[4] Manuel Blum. Program result checking: A new approach to making programs more reliable. In Svante Carlsson Andrzej Lingas, Rolf G. Karlsson, editor, *Automata, Languages and Programming, 20th International Colloquium*, volume 700 of *Lecture Notes in Computer Science*, pages 1–14, Lund, Sweden, 5–9 July 1993. Springer-Verlag.

[5] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, January 1995.

[6] Michael Clark and Bernadette Mohan. *The Oxford–DUDEN Pictorial English Dictionary*. Oxford University Press, 1995. ISBN 0-19-861311-3.

[7] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, San Antonio, TX, January 1999. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergTh%omborson99a/index.html`.

[8] Christian S. Collberg and Todd A. Proebsting. AλgoVista — A search engine for computer scientists. Technical Report 2000-01, 2000.

[9] Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole optimizations. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 111–116. ACM, ACM, 1984.

[10] Funda Ergün, Sampath Kannan, S. Ravi Kumar, Ronitt Rubinfeld, and Mahesh Vishwanathan. Spot-checkers. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 259–268, New York, May 23–26 1998. ACM Press.

[11] David Grambs. *The Describer's Dictionary*. W. W. Norton & Company, 1995. ISBN 0-393-31265-8.

[12] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 2 edition, 1990.

[13] Sampath Kannan and Todd A. Proebsting. Register allocation in structured programs. *Journal of Algorithms*, 29(2):223–237, November 1998.

[14] Stéphanie Petit. Encyclopedia of combinatorial structures. `http://algo.inria.fr/encyclopedia`.

[15] Simon Plouffe. Plouffe's inverter. `http://www.lacim.uqam.ca/pi`.

[16] Ronitt Rubinfeld. Batch checking with applications to linear functions. *INFORMATION PROCESSING LETTERS*, 42(2):77–80, May 1992.

[17] Ronitt Rubinfeld. Designing checkers for programs that run in parallel. *ALGORITHMICA*, 15(4):287–301, April 1996.

[18] Neil J. A. Sloane. Sloane's on-line encyclopedia of integer sequences. `http://www.research.att.com/~njas/sequences/index.html`.

[19] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[20] Eric Weisstein. Encyclopedia of mathematics. `http://www.treasure-troves.com/math`.