AλgoVista—A Search Engine for Computer Scientists

Christian S. Collberg
The University of Arizona

Todd A. Proebsting
Microsoft Research

January 27, 2000

University of Arizona Computer Science Technical Report
2000–01

Microsoft Research Technical Report
MSR–TR–2000–06

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

AλgoVista—A Search Engine for Computer Scientists

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

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

Abstract

We describe AλgoVista, a web-based search engine designed to allow applied computer scientists to classify problems and find algorithms and implementations that solve these problems. Unlike other search engines, AλgoVista is not keyword based. Rather, users provide a set of *input*$\Rightarrow$*output* samples that describe the behavior of the problem they wish to classify. This type of *query-by-example* requires no knowledge of specialized terminology, only an ability to formalize the problem.

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

1 Background

Frequently, working software developers will encounter a problem with which they are unfamiliar, but which – they suspect – has probably been treated by the Computer Science theory community. Just as frequently, theoretical computer scientists will be working on a problem which they suspect might have a practical application.

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 chasm 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. Working programmers can query AλgoVista to look for theoretical results that are relevant to their current application. Theoretical computer scientists can extend AλgoVista with problems with which they are familiar, or with references to new algorithms they have developed for these problems.

AλgoVista is based on a novel application of a technique known as *program* (or *result*) *checking*, developed over the last decade by Manuel Blum and others [4–6,10,16,17,19] as an alternative to program verification and testing. Program checking extends programs with *checkers* to allow them to 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 graphs in Figure 1(a).

"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 graph-transformation in Figure 1(b).

"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 [8].

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.

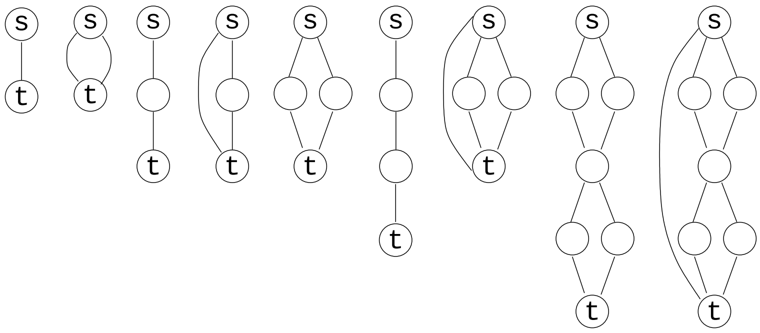
1.2 Interacting with AλgoVista

AλgoVista[1] is an online database that stores and codifies problems, algorithms, and combinatorial structures

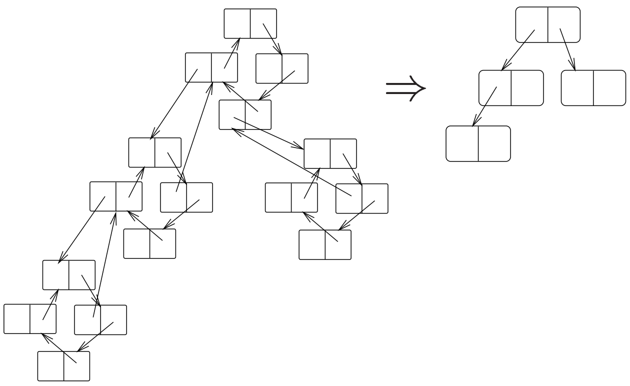
[1] AlgoVista.cs.arizona.edu.

Figure 1: Some motivating examples and queries.

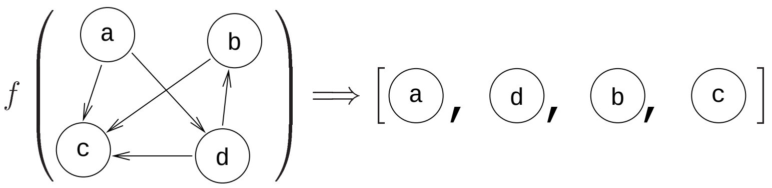
(a) Series parallel graphs.



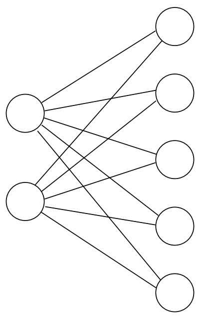
(b) Shrinking biconnected components.



(c) A topological sorting query



(d) Bipartite graph query.



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 query in Figure 1(c). 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 query in Figure 1(d), 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 discusses checklet design issues. Section 7 describes how advanced type analysis can speed up searching, and Section 8 evaluates the performance of the search algorithms. Section 9 discusses related work, and Section 10, 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're writing. We intend for P to compute a function f, but we're 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. Unfortunately, in spite of decades of research into program verification, it is still only feasible to prove the correctness of trivial programs.

2. We can test that $P(x) = f(x)$, where x is drawn from a reasonable domain of test data. The problem with testing is that the actual distribution of input data to the program is often either unknown or prohibitively large.

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 $o(T)$.

Much theoretical 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 seems to be as expensive as finding the tour itself.

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*$\Rightarrow$*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 2 shows some simple checklets. Simplest of all is the integer addition checklet `intAdd` of Figure 2 (a). Given the query $\ulcorner(15, 6) \Rightarrow 21\urcorner$ the checklet would accept and return the result "`http://www.cs.arizona.edu/~collberg/IntAdd.html`." Figures 2 (b) and (c) show a straightforward (slow) and a more complex (faster) implementation of a sorting checklet.

Figure 2 (d), finally, shows 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 2 (d), first check that every node in the input graph occurs in the output node list, and then check that if node f comes before node t in the output list then there is no path $t \rightsquigarrow f$ in the input graph.

In some cases it may be difficult to construct checklets which run in an acceptable length 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.

Writing checklets for floating point problems can also be challenging. For example, which, if any, of the queries $\ulcorner 2.0 \Rightarrow 1.4142135623\urcorner$, $\ulcorner 2.00000 \Rightarrow 1.4140\urcorner$, and $\ulcorner 2.0 \Rightarrow 1.0\urcorner$ should a *floating-point square root* checklet accept? In all cases, the right hand side *is* an approximation of $\sqrt{2}$, but just how accurate should the approximation be in order to be acceptable to the checklet? In our current implementation, floating-point comparisons are done in the *minimum* precision of any floating-point number in the input query. Hence, $\ulcorner 2.0 \Rightarrow 1.4142135623\urcorner$ will accept (since $1.4142135623^2 = 1.9999999997 \approx 2.0$ when comparing with a precision of one decimal digit), but $\ulcorner 2.00000 \Rightarrow 1.4140\urcorner$ will not (since $1.4140^2 = 1.999396 \not\approx 2.0000$ when comparing with four digits' precision).

Figure 2: Some simple checklets.

(a) An *integer addition* checklet.

```
checklet intAdd ((int a, int b) ⇒ int c)
   if (a+b)=c then
      accept http://www.cs.arizona.edu/~collberg/IntAdd.html
   else
      reject
```

(b) A slow *sorting* checklet.

```
checklet sorting1 (int[] input ⇒ int[] output)
   int[] tmp ← quicksort(input)
   if tmp = output then
      accept http://hissa.ncsl.nist.gov/~black/CRCDict/termsArea.html#sort
   else
      reject
```

(c) A faster *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 sorting2 (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 don't contain the same elements then
      reject
   accept http://hissa.ncsl.nist.gov/~black/CRCDict/termsArea.html#sort
```

(d) 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
```

3 System Overview

Points ①– ④ in Figure 3 show how a typical user will search AλgoVista. A query is submitted through the AλgoVista web page, transferred to the AλgoVista server, and matched against the checklets in the checklet database (the *checklet coop*). The output from any accepting checklet is transferred back to the client and presented to the user.

Figure 4 shows the basic AλgoVista search algorithm. The algorithm is very simple: 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 7 we explore more sophisticated algorithms that speed up search times significantly.

3.1 Extending the Database

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 coop. This is illustrated by points ⑤– ⑦ in Figure 3.

To the best of our knowledge, AλgoVista is the first search engine on the web to allow arbitrary users to upload executable code into its database. Obviously, there are a number of security issues that have to be addressed.

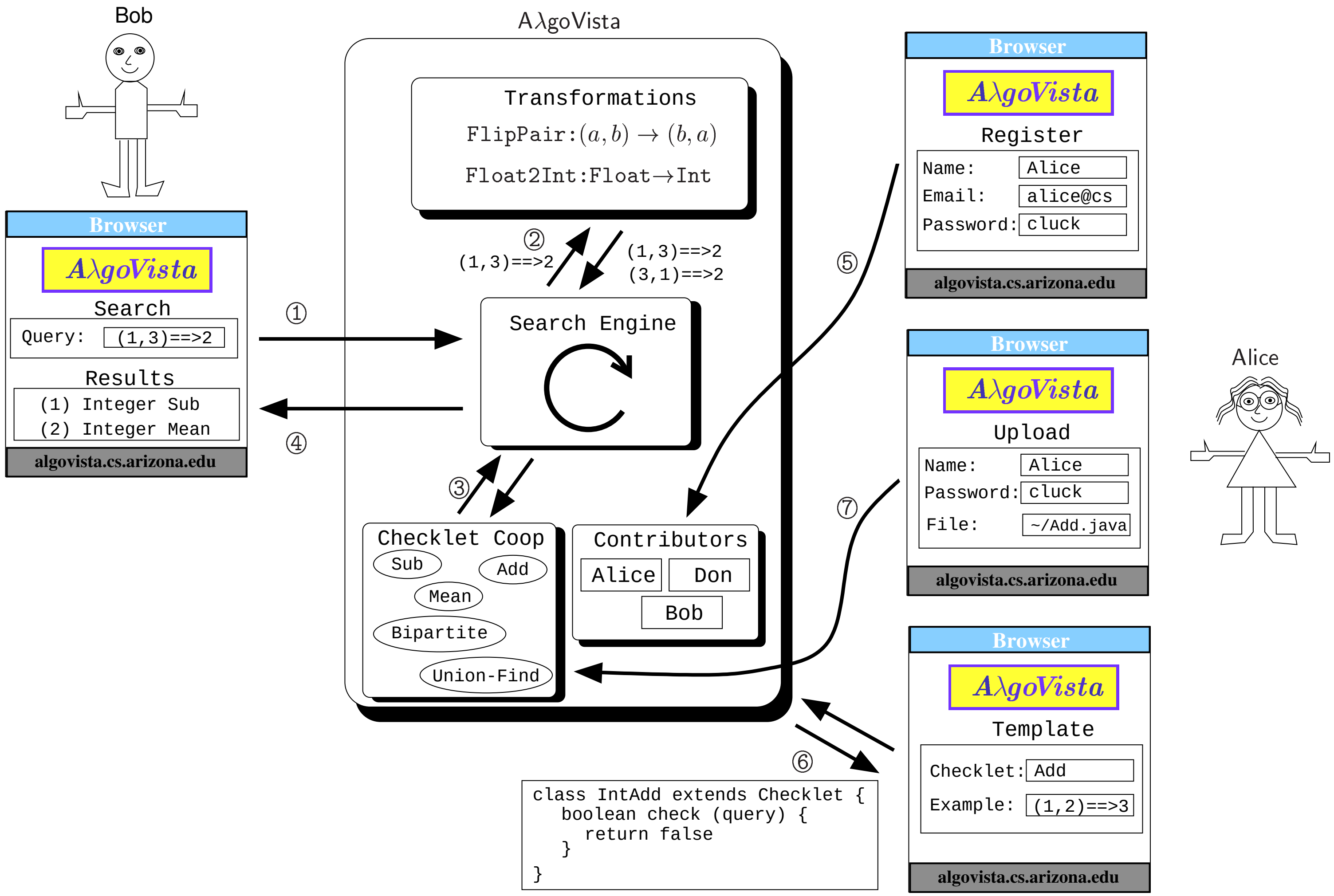


Figure 3: Alice, a theoretician, and Bob, a software developer, interact with AλgoVista. At ①, Bob submits a search query. At ②, the query is mutated into a set of two queries by the transformation database. At ③, the search engine matches these queries against the checklets in the checklet coop. The output of any accepting checklets is returned to Bob at ④. At ⑤, Alice registers as a potential contributor to AλgoVista. At ⑥, she submits an example query of a checklet **Add** she intends to write and receives a template checklet in return. At ⑦, she modifies the template, and uploads it to the checklet coop.

Figure 4: Exhaustive search algorithm.

```
function search (query)
   q ← parse(query)
   responses ← {}
   for every combination of query transformations T1(T2(···)) do
      q′ ← T1(T2(··· q ···))
      for every checklet c in the coop do
         if c accepts q′ with response r then
            responses ← responses ∪ {r}
   return responses
```

Figure 5: Evil and stupid checklets.

(a) A checklet that always accepts, returning a bogus URL.

```
checklet evil1 (any ⇒ any)
   accept http://www.quayle.org/
```

(b) A *denial-of-service* checklet that steals memory and/or CPU cycles.

```
checklet evil2 (any ⇒ any)
   while true do
      Node n = new Node
```

(c) A checklet that reads from or writes to the local file system.

```
checklet evil3 (any ⇒ any)
   exec "mail evil@spam.com < /etc/passwd; /bin/rm -R *"
```

(d) A *prime factorization* checklet that uses an extremely slow result checking algorithm, although a trivial fast one exists. The effect is identical to that of a *denial-of-service* attack.

```
checklet stupid1 (int composite ⇒ int[] factors)
   int[] primes = factor(composite)        // Known factorization algorithms are slow.
   if primes = factors then
      accept http://www.utm.edu/research/primes
   else
      reject
```

(e) A checklet providing a list of accepting examples.

```
checklet intAdd ((int a, int b) ⇒ int c)
   examples (0,0)==>0, (5,6)==>11
   if (a+b)=c then accept ···
```

Figure 5 shows some examples of hostile checklets. Figure 5 (a) shows an overly general checklet `evil1` that was uploaded in an attempt to promote someone's web site. Regardless of the input query, `evil1` will always accept and return a link to the bogus site. Checklet `evil2` in Figure 5 (b) launches a *denial-of-service* attack by stealing as many CPU cycles or as much memory as possible. Checklet `evil3` in Figure 5 (c) attempts to compromise the security of the AλgoVista server by reading from or writing to the local file system. Checklet `stupid1` in Figure 5 (d), finally, while not being outrightly hostile, uses an extremely slow result checking algorithm which results in effects similar to that of a denial-of-service attack.

AλgoVista checklets are written in Java and are executed with the same security privileges as an applet would. This allows us to rely on Java's built-in security features to prevent checklets from compromising the security of the AλgoVista server.

Denial-of-service attacks [14] are more difficult to deal with. While time-outs are used to stop checklets from stealing too many CPU cycles, as far as we are aware, Java does not provide the means to limit the dynamic memory allocation of a process.

It is unclear whether there are any strong technical means to prevent attacks by overly general checklets. The same problem plagues keyword search engines such as AltaVista: to promote their own web-pages unscrupulous users will "submit pages with numerous keywords, or with keywords unrelated to the real content of the page" [1]. Currently, we require every checklet to provide a list of *accepting examples*, as shown in Figure 5 (e). When a checklet is uploaded AλgoVista ensures that

a) the checklet accepts every one of the example queries it has provided, and

b) the checklet only accepts a *small fraction* of all the example queries provided for all other checklets in the coop.

While not foolproof, this policy provides a reasonable level of security.

4 The Query Language

To make the use of AλgoVista a pleasant experience, users must be able to easily formalize their queries. This necessitates the design of a natural and expressive query language. QL, the AλgoVista query language, is essentially a *domain-specific language* where the domain is very large: users need to be able to express any reasonable mapping between any reasonable literal data structures.

As is always the case with domain-specific languages, there is a tension between a minimalist syntax (LISP-like, for example) and a "kitchen-sink" syntax. The former has a few simple primitives whereas the latter has many complex primitives, one for each anticipated use. The minimalist syntax is easy to learn but combining the primitives into complex sentences (queries, in our case) can be cumbersome. The kitchen-sink syntax, on the other hand, has a steeper learning-curve, but common sentences (those anticipated by the language designers) are easier to express.[2]

QL is of "medium" complexity: while it has some kitchen-sink features, there are many data structures which cannot be expressed directly but have to be constructed by combining simple primitives. QL primitives 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 AλgoVista users will be wanting to classify graph structures and problems on graphs.

Figure 6(a) shows the syntax of QL, and Figure 6(b) gives some example queries. In the query in Figure 6(b) ① a pair of integers (1 and 2) is mapped to an integer (3). AλgoVista returns the result set ⟨Binary or, Integer add⟩ since $1 + 2 = 3$ and $1 \mid 2 = 3$. In the query in Figure 6(b) ② a directed graph is mapped to a directed graph. Each graph is represented, by convention, as a pair of a node-list and an edge-list. The query in Figure 6(b) ③ asks AλgoVista to classify a particular graph, which turns out to be a strongly connected directed graph.

Figure 6(b) ④, 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.

4.1 The AλgoVista Type System

In Section 7 we show how type analysis can speed up AλgoVista's search engine. The idea is to assign a type signature to every query, checklet, and query transformation, and only submit a query to a checklet if the signature of the query matches that of the checklet.

Figure 7(a) shows the AλgoVista type hierarchy. Only some of these types are directly expressible in QL. For example, even though AλgoVista has a *set* type, there is no concrete QL syntax for sets. Rather, as we will see in Section 5, query transformations are responsible for inferring a collection of possible types from a QL query, including that a vector ⌜`[1,2,3]`⌝ could represent the set $\{1, 2, 3\}$. This allows checklets to be very specific about what types of queries they will accept, and it allows AλgoVista users to be very non-specific in how they formulate their queries. For example, an unsophisticated user might issue the query

⌜`([1,2,3],[3,2,4])==>[1,2,3,4]`⌝

in the search for the *set union* operation. He is not required by the AλgoVista type system to explicitly state that the three operands are *sets*, since he may not even be familiar with this concept. Rather, he can simply represent the sets as *vectors*, AλgoVista's general term for "collections of objects".

The *set union checklet*, on the other hand, can specify explicitly that it will only accept queries that map two sets to a third set, i.e. that has the signature

```
Map(Pair(Set(Int),Set(Int)),Set(Int)).
```

Type-checking queries and checklets will prevent a query such as

⌜`([1,2,2],[2,3,4])==>[1,2,3,4]`⌝

from being submitted to the *set union checklet* since one of the vectors is not a set.

Figure 7(b) shows the mapping from QL queries to type signatures. Figure 7(c) gives some examples.

[2] Our terms "minimalist language" and "kitchen-sink language" are equivalent to the terms *union-language* and *intersection-language* coined by Davidson and Fraser [9].

Figure 6: AλgoVista's query language and example queries.

(a) A grammar for QL, AλgoVista's query language.

```
S      →  int | float | bool |             #  Primitive types.
          S `==>´ S |                      #  Map from input to output.
          atom [ `/´S ] |                  #  Node with optional node data S.
          atom `->´ [ `/´S ] atom |        #  Directed edge with optional edge data S.
          atom `--´ [ `/´S ] atom |        #  Undirected edge with optional edge data S.
          `[´[ S { `,´S } ] `]´ |          #  List of elements.
          `(´ S `,´ S `)´                  #  Tuple of two elements.
bool   →  `true´ | `false´
atom   →  `a´...`z´
digit  →  `0´...`9´
int    →  digit { digit }
float  →  int `.´int
```

(b) Example QL queries.

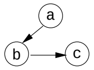
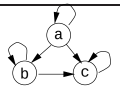
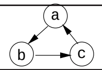
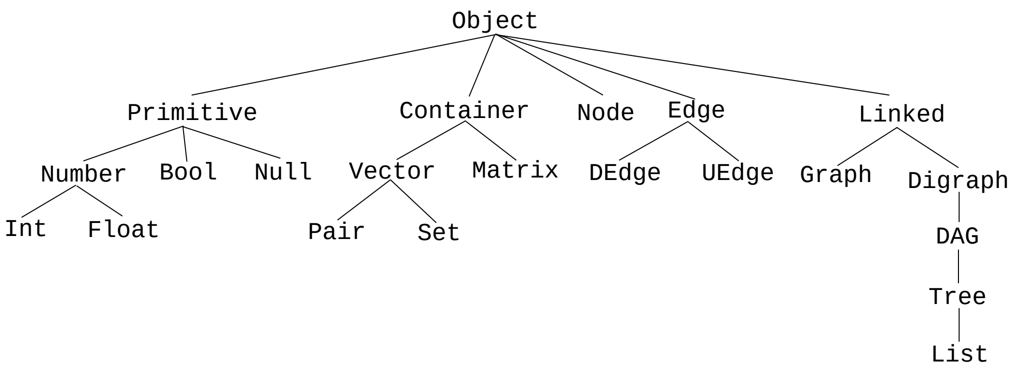
#	QL query	Query explanation	Query Result
①	`(1,2)==>3`	What function maps the two integers 1 and 2 to the integer 3?	⟨Binary or, Integer add⟩
②	`([a,b,c],[a->b,b->c])==> ([a,b,c],[a->a,a->b,a->c,b->b,b->c,c->c])`	What function maps  to  ?	⟨Transitive closure⟩
③	`([a,b,c],[a->b,b->c,c->a])`	What kind of graph is this:  ?	⟨Strongly connected graph⟩
④	`([3,7],[5,1,6]) ==> [5,1,6,3,7]`	What function maps the lists `[3,7]` and `[5,1,6]` to the list `[5,1,6,3,7]`?	⟨List append⟩
⑤	`([a,b,c,d],[a->c,a->d,b->c,d->c,d->b]) ==> [a,d,b,c]`	List of edges representation.	⟨Topological sort⟩
⑥	`[a->c,a->d,b->c,d->c,d->b] ==> [a,d,b,c]`	List of edges representation. Node set is implied.	⟨Topological sort⟩
⑦	`[[0,0,1,1],[0,0,1,0],[0,0,0,0],[0,1,1,0]] ==> [1,4,2,3]`	Adjacency matrix representation.	⟨Topological sort⟩
⑧	`([a,b,c,d],[a->[c,d],b->[c],c->[],d->[c,b]]) ==> [a,d,b,c]`	List of neighbors representation.	⟨Topological sort⟩
⑨	`[a->[c,d],b->[c],c->[],d->[c,b]] ==> [a,d,b,c]`	List of neighbors representation. Node set is implied.	⟨Topological sort⟩

Figure 7: AλgoVista's type System.

(a) QL's type hierarchy.



(b) Type assignment for QL.

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

(c) Examples of QL's type system.

QL query	Signature
1	Int
4.5	Float
true	Bool
[1,2,3]	Vector(Int)
(1,4.5)	Pair(Int,Float)
a	Node/Null
a/5	Node/Int
a->b	DEdge/Null
a->/5b	DEdge/Int
a--b	UEdge/Null
a--/5.5b	UEdge/Float
([a,b,c],[a->b,a->c])	Pair(Vector(Node(Null)),Vector(DEdge(Null)))
([a/3,b/2,c/1],[a->b,a->c])	Pair(Vector(Node(Int)),Vector(DEdge(Null)))
(1,2)==>3	Map(Pair(Int,Int),Int)
([1,2],[3,4])==>[4,6]	Map(Pair(Vector(Int),Vector(Int)),Vector(Int))

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 the topological sorting query in Figure 1(c), including representing the input graph as a *list of edges*, an *adjacency matrix*, or a *list of neighbors*. These queries are shown in Figure 6(b) ⑤– ⑨. The corresponding topological sorting checklet, on the other hand, might expect the input graph only in a matrix form.

This gap between query and checklet representation is probably the most contentious part of AλgoVista, and solving this problem is a major key to the success of the search engine. We have considered two ways of attacking the problem:

1. The first solution is the kitchen-sink approach to query language design that was alluded to in the previous section. The idea is to provide special syntax for *every* conceivable literal data structure, including graphs, trees, lists, polygons, points, line segments, planes, sets, bags, etc. The advantage of this approach is that the query language syntax will guide both checklets and queries to use the same representation. The disadvantages are (a) that it is difficult to know when the query language is complete, and (b) that the query language becomes large and difficult to learn.

2. The second approach is to provide a set of *query transformations* that will automatically mutate queries between common representations. For example, given the topological sorting query in Figure 6(b) ⑤, AλgoVista would automatically produce the queries in Figure 6(b) ⑤– ⑨, all of which would be matched against the checklets in the checklet coop.

The current implementation of AλgoVista uses the second approach. Figures 8 and 9 lists the transformations currently in use by the search engine.

Transformation $\mathcal{T}^B$ (`Float2IntFloor`) in Figure 8 transforms a float to an integer by truncation. Transformation $\mathcal{T}^F$ swaps the elements of a pair. Transformation $\mathcal{T}^H$ converts a vector to a set, provided the elements of the vector are unique. Transformations $\mathcal{T}^I$ through $\mathcal{T}^O$ are concerned with transforming a pair of vectors to various linked structures.

Figure 10 gives an elaborate query transformation example. A user query

⌜`[a->/5b,b->/9c]`⌝

is first transformed (using the `Vector2VectorPair` transformation) to

⌜`([a,b,c],[a->/5b,b->/9c])`⌝

by adding the list of nodes left out by the user. Through a series of analyses it is eventually determined that the query represents a linked list

⌜`List(∅,Int):([a,b,c],[a->/5b,b->/9c])`⌝,

which could also be represented by the vector

⌜`Vector(Int):[5,9]`⌝.

Since the vector is of size two, it could also represent a pair. The elements of this pair, finally, can be swapped and converted from integers to floats. The AλgoVista search engine would hand off any or all intermediate results of this string of transformations to checklets that have the appropriate signature.

6 Checklet Design

Extending the AλgoVista coop with a new checklet may seem like an involved process, but, fortunately, much has been automated. A typical upload involves the following steps:

1. Produce an example query, for example ⌜`(1,2)==>3`⌝ for the `IntAdd` checklet.

2. Submit the example query to AλgoVista's checklet template generator. Figure 11 shows the template produced from the query ⌜`(1,2)==>3`⌝.

3. Fill in the `Description()` method with a short description of the problem the checklet tests for.

4. Fill in the `References()` method with a list of hyper-references to on-line resources related to the problem.

5. Replace "`return false`" in the `Check()` method with the relevant program checking code that returns `true` if the checklet accepts a query, and `false` otherwise. The `Check()` method takes an `AlgoVista.CL.Object` as argument, essentially the abstract syntax tree of a parsed query. The template already contains code to unpack this representation. For example, in Figure 11 `intVal0` and `intVal1` contain the input part of the query, and `intVal2` the output part. Hence, in our example, "`return false`" would be replaced by "`return (intVal0+intVal1)==intVal2`"

6. Upload the new checklet to the AλgoVista server where it will be compiled, verified, and tested. AλgoVista will possibly return a list of compilation errors or security violations that have to be fixed before the checklet will be accepted into the coop.

The most difficult part is certainly constructing the actual program checking code. Inspiration can sometimes be had from the result checking literature [4–6,10, 16,17,19], but more often by examining existing checklets. AλgoVista supports this by making the source code of checklets available for perusal.

Figure 8: Query transformations (A). $\emptyset$ represents Null. Greek letters are type variables. Examples are in the format *signature*:*query*.

$\mathcal{T}^A$	Int2Float:Int⇒Float *Description:* Convert an integer to a float. *Example:* Int:3⇒Float:3.0
$\mathcal{T}^B$	Float2IntFloor:Float⇒Int *Description:* Round a real number to the nearest smaller integer. *Example:* Float:1.3⇒Int:1
$\mathcal{T}^C$	Float2IntCeil:Float⇒Int *Description:* Round a real number to the nearest larger integer. *Example:* Float:1.3⇒Int:1
$\mathcal{T}^D$	Int2Bool:Int⇒Bool *Description:* Convert 0/1 to false/true. *Condition:* The integer must be 0 or 1. *Example:* Int:0⇒Bool:false
$\mathcal{T}^E$	Bool2Int:Bool⇒Int *Description:* Convert false/true to 0/1. *Example:* Bool:true⇒Int:1
$\mathcal{T}^F$	FlipPair:Pair(α,β)⇒Pair(β,α) *Description:* Swap the elements in a pair. *Example:* Pair(Int,Float):(1,2.3)⇒Pair(Float,Int):(2.3,1)
$\mathcal{T}^G$	Vector2Pair:Vector(α)⇒Pair(α,α) *Description:* Convert a vector to a pair. *Condition:* The vector must contain exactly 2 elements. *Example:* Vector(Int):[1,2]⇒Pair(Int,Int):(1,2)
$\mathcal{T}^H$	Vector2Set:Vector(α)⇒Set(α) *Description:* Convert a vector to a set. *Condition:* The vector must contain no duplicate elements. *Example:* Vector(Int):[1,2,5,9]⇒Set(Int):{1,2,5,9}
$\mathcal{T}^I$	VectorPair2Linked:Pair(Vector(α),Vector(β))⇒Linked(α,β) *Description:* Convert a pair of vectors of nodes and edges to a linked structure. *Example:* Pair(Vector(Node(Int)),Vector(DEdge($\emptyset$))):([a/1,b/2],[a->b])⇒ Linked(Int,$\emptyset$):([a/1,b/2],[a->b])
$\mathcal{T}^J$	Vector2VectorPair:Vector(α)⇒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]⇒Pair(Vector($\emptyset$),Vector($\emptyset$)):([a,b,c,d],[a->b,c->d])
$\mathcal{T}^K$	Linked2Graph:Linked(α,β)⇒Graph(α,β) *Description:* Convert a linked structure to an undirected graph. *Condition:* The linked structure must be undirected. *Example:* Linked($\emptyset$,$\emptyset$):([a,b,c],[a--b,b--c,c--a])⇒Graph($\emptyset$,$\emptyset$):([a,b,c],[a--b,b--c,c--a])
$\mathcal{T}^L$	Linked2Digraph:Linked(α,β)⇒Digraph(α,β) *Description:* Convert a linked structure to a digraph. *Condition:* The linked structure must be directed. *Example:* Linked($\emptyset$,$\emptyset$):([a,b,c],[a->b,b->c,c->a])⇒Digraph($\emptyset$,$\emptyset$):([a,b,c],[a->b,b->c,c->a])
$\mathcal{T}^M$	Digraph2DAG:Digraph(α,β)⇒DAG(α,β) *Description:* Convert a digraph to a directed acyclic graph. *Condition:* The digraph must be acyclic. *Example:* Digraph($\emptyset$,$\emptyset$):([a,b,c],[a->b,b->c,a->c])⇒DAG($\emptyset$,$\emptyset$):([a,b,c],[a->b,b->c,a->c])
$\mathcal{T}^N$	DAG2Tree:DAG(α,β)⇒Tree(α,β) *Description:* Convert a directed acyclic graph to a tree. *Condition:* No node may have indegree > 1. Exactly one node must have indegree $= 0$. *Example:* DAG($\emptyset$,$\emptyset$):([a,b,c,d],[a->b,b->c,c->d])⇒Tree($\emptyset$,$\emptyset$):([a,b,c,d],[a->b,b->c,c->d])
$\mathcal{T}^O$	Tree2List:Tree(α,β)⇒List(α,β) *Description:* Convert a tree to a linked list. *Condition:* No node may have outdegree > 1. *Example:* Tree($\emptyset$,$\emptyset$):([a,b,c],[a->b,b->c])⇒List($\emptyset$,$\emptyset$):([a,b,c],[a->b,b->c])

Figure 9: Query transformations (B).

$\mathcal{T}^P$	`List2VectorA:List(`α`,`$\emptyset$`)`$\Rightarrow$`Vector(`α`)` *Description:* Convert a linked list to a vector. *Example:* `List(Int,`$\emptyset$`):([a/1,b/2,c/3],[a->b,b->c])`$\Rightarrow$`Vector(Int):[1,2,3]`
$\mathcal{T}^Q$	`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]`
$\mathcal{T}^R$	`Linked2MatrixA:Linked(`$\emptyset$`,`$\emptyset$`)`$\Rightarrow$`Matrix(Int)` *Description:* Convert a linked structure to an adjacency matrix representation. *Example:* `Linked(`$\emptyset$`,`$\emptyset$`):([a,b,c],[a->b,b->c,c->a])`$\Rightarrow$`Matrix(3,3,Int):[0,1,0; 0,0,1; 1,0,0]`
$\mathcal{T}^S$	`Linked2MatrixB:Linked(`$\emptyset$`,`α`)`$\Rightarrow$`Matrix(`α`)` *Description:* Convert a linked structure to an adjacency matrix representation. *Example:* `Linked(`$\emptyset$`,Int):([a,b],[a->/5b,b->/8b])`$\Rightarrow$`Matrix(2,2,Int):[0,5; 0,8]`
$\mathcal{T}^T$	`VectorOfVectors2Matrix:Vector(Vector(`α`))`$\Rightarrow$`Matrix(`α`)` *Description:* Convert a vector of vectors to a matrix representation. *Condition:* All vectors must be of the same length. *Example:* `Vector(Vector(Int)):[[1,2],[3,4]]`$\Rightarrow$`Matrix(2,2,Int):[1,2; 3,4]`

Figure 10: Query transformation example. The first line is the user's QL query. Subsequent lines show the query and its signature after a transformation.

```
Vector(DEdge(Int)):[a->/5b,b->/9c]
  =T^J=> Pair(Vector(Node(∅)),Vector(DEdge(Int))):([a,b,c],[a->/5b,b->/9c])
  =T^I=> Linked(∅,Int):([a,b,c],[a->/5b,b->/9c])
  =T^L=> Digraph(∅,Int):([a,b,c],[a->/5b,b->/9c])
  =T^M=> DAG(∅,Int):([a,b,c],[a->/5b,b->/9c])
  =T^N=> Tree(∅,Int):([a,b,c],[a->/5b,b->/9c])
  =T^O=> List(∅,Int):([a,b,c],[a->/5b,b->/9c])
  =T^Q=> Vector(Int):[5,9]
  =T^G=> Pair(Int,Int):(5,9)
  =T^F=> Pair(Int,Int):(9,5)
  =T^A=> Pair(Float,Int):(9.0,5)
  =T^A=> Pair(Float,Float):(9.0,5.0)
```

7 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 coop. 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*. 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⇒Float
FlipPair:  Pair(α,β)⇒Pair(β,α)
```

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 trans-

Figure 11: The checklet template generated automatically by AλgoVista from the example query ⌜(1,2)==>3⌝.

```
public class IntAdd implements AlgoVista.DataBase.Checklet {

   public String Description () {
      return "";
   }

   public String[] ProtoExamples () {
      String [] examples = {"(1,2)==>3"};
      return examples;
   }

   public String Signature () {
      return "Map(Pair(Int(),Int()),Int())";
   }

   public AlgoVista.DataBase.Reference[] References () {
      AlgoVista.DataBase.Reference [] examples = {
          new AlgoVista.DataBase.Reference("tag","link")
      };
      return examples;
   }

   public boolean Check (AlgoVista.CL.Object obj) throws Throwable {
      AlgoVista.CL.Map mapObject0 = (AlgoVista.CL.Map)obj;
      AlgoVista.CL.Object mapInput0 = mapObject0.GetInput();
      AlgoVista.CL.container.vector.Pair pairObject0 = (AlgoVista.CL.container.vector.Pair)mapInput0;
      AlgoVista.CL.Object pairFirst0 = pairObject0.GetFirst();
      AlgoVista.CL.primitive.number.Int intObject0 = (AlgoVista.CL.primitive.number.Int)pairFirst0;
      long intVal0 = intObject0.GetInt();
      AlgoVista.CL.Object pairSecond0 = pairObject0.GetSecond();
      AlgoVista.CL.primitive.number.Int intObject1 = (AlgoVista.CL.primitive.number.Int)pairSecond0;
      long intVal1 = intObject1.GetInt();
      AlgoVista.CL.Object mapOutput0 = mapObject0.GetOutput();
      AlgoVista.CL.primitive.number.Int intObject2 = (AlgoVista.CL.primitive.number.Int)mapOutput0;
      long intVal2 = intObject2.GetInt();
      return false;
   }
}
```

formed into ⌜(2.0,2.0)==>4.0⌝, which matches the signature of `FloatAdd`, and therefore can be submitted to that checklet.

It is a simple observation that a query ⌜true==>1⌝ (which has the type `Map(Bool,Int)`) can never match any of the checklets, regardless of which transformations are applied. Still, the algorithm in Section 3 would apply all possible combinations of transformations to ⌜true==>1⌝ and submit any generated query mutation to every checklet in the coop.

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

7.1 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 coop. $\mathcal{S}_{\mathcal{T},\mathcal{C}}$ 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)` is as follows:

```
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');
}
```

The `AlreadySeen`-set prevents the same query mutation from being produced more than once:

`(1,2)==>3`
$\stackrel{\mathcal{T}^{F}}{\Rightarrow}$ `(2,1)==>3`
$\stackrel{\mathcal{T}^{F}}{\Rightarrow}$ `(1,2)==>3`
$\stackrel{\mathcal{T}^{F}}{\Rightarrow}$ `(2,1)==>3`
$\Rightarrow \cdots$

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 four `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.

7.2 The Query Signature Graph

Figure 12 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. With a sufficiently rich set of transformations, it is possible to generate an infinite number of signatures:

`[a->b,b->c]`
$\stackrel{\mathcal{T}^{J}}{\Rightarrow}$ `([a,b,c],[a->b,b->c])`
$\stackrel{\mathcal{T}^{J}}{\Rightarrow}$ `([a,b,c],([a,b,c],[a->b,b->c]))`
$\stackrel{\mathcal{T}^{J}}{\Rightarrow}$ `([a,b,c],([a,b,c],([a,b,c],[a->b,b->c])))`
$\Rightarrow \cdots$

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. This limit is currently set to four. This would seem to limit the usefulness of AλgoVista, but in practice this is not so. First of all, the exhaustive search algorithm from Section 3 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].

8 Evaluation

The ultimate test for AλgoVista will be

a) whether theoreticians will be willing to extend the database with new problem specifications, and

b) whether the resulting database will actually provide useful information to practicing programmers.

Two secondary concerns are

c) whether security breaches can be prevented, and

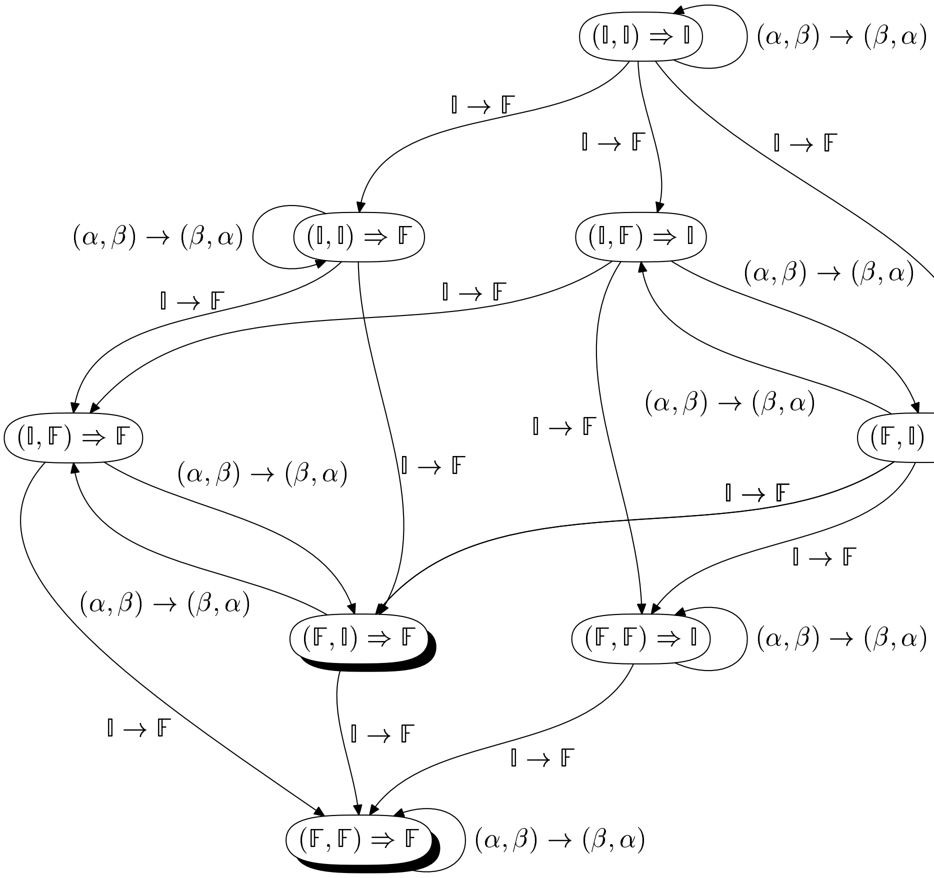
d) whether the performance of the search engine will be adequate to ensure reasonable response time.

Unfortunately, most of these questions remain unanswered at this time, since AλgoVista has yet to be fully deployed and so far the authors are its only users.

We can, however, give some preliminary timing measurements to evaluate the relative performance of the two search algorithms.

Table 1 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

Figure 12: Query signature graph. The two transformations `Int2Float` and `FlipPair` are represented by $\mathbb{I} \to \mathbb{F}$ and $(\alpha, \beta) \to (\beta, \alpha)$, respectively. Shaded nodes represent *viable* signatures, those that have associated checklets.



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 1 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 1 it is clear, as would be expected, the precomputed search algorithm is vastly 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.

In our current implementation we limit the precomputed algorithm to apply at most six transformations. The hard-coded programs generated by the algorithms in Section 7 are rather large (even limiting the search to four mutations yields roughly 1.2 million lines of code over 20000 Java classes) and current Java tools are only barely able to handle programs of this size.

9 Related Work

A number of web sites, for example the *CRC Dictionary* [3] 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

Table 1: 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.

	Search		Mutations	
Query	Exhaustive	Precomputed	Exhaustive	Precomputed
⌜`(1,2)==>3`⌝	3.25	0.22	3.10	0.21
⌜`[1,3]==>2`⌝	3.40	0.22	3.01	0.2
⌜`([a,b,c,d],[a->b,b->c,c->d,d->a])`⌝	2.38	0.03	1.78	0.03
⌜`([a,b,c,d],[a->b,b->c,c->d])==>[a,b,c,d]`⌝	11.51	0.21	9.89	0.22
⌜`([a,b,c,d],[a->/2b,b->/2c,c->/3d])==>3`⌝	13.39	0.13	12.53	0.12
⌜`[a->/1b,b->/2c,c->/3d]==>6`⌝	0.47	0.02	0.34	0.02
⌜`([1,2,3],[4,5,6])==>[1,2,3,4,5,6]`⌝	2.00	0.06	1.66	0.06
⌜`[6,5,4,3,2,1]==>[1,2,3,4,5,6]`⌝	0.13	0.01	0.07	0.01

service allows users to look up number sequences without knowing their name. For example, if a user entered the sequence ⌜$1, 2, 3, 5, 8, 13, 21, 34$⌝, 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. A similar search service is *Encyclopedia of Combinatorial Structures* [15].

Inductive Logic Programming (ILP) [2] 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]).
```

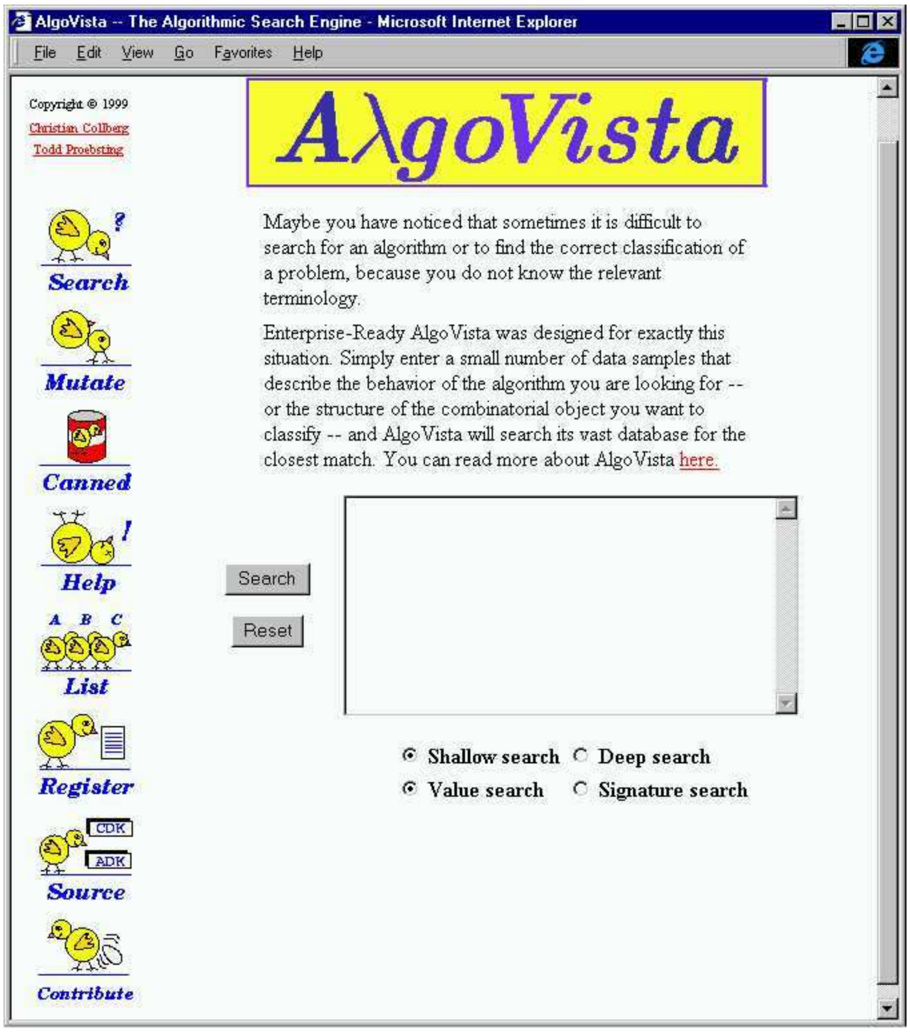
AλgoVista is essentially a *reverse definition* dictionary for Computer Science terminology. Rather than looking up a term to find its definition (as one would in a normal dictionary), a reverse definition dictionary allows you to look up the term given its definition or an example. The *DUDEN* [7] series of pictorial dictionaries 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*.

10 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.cs.arizona.edu/`. Figure 13 shows a snapshot of this web page.

Figure 13: Snapshot of the AλgoVista web page.



Acknowledgments: Will Evans pointed out the relationship between checklets and program checking. Dengfeng Gao implemented most of the checklets in the current database. We thank them both.

References

[1] AltaVista – Adding pages or URLs to the index. `http://www.altavista.com/cgi-bin/query?pg=addurl`.

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

[3] 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`.

[4] 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.

[5] 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.

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

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

[8] 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`.

[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] Zheng YL Leiwo J. A method to implement a denial of service protection base. In *INFORMATION SECURITY AND PRIVACY*, volume 1270 of *LNCS*, pages 90–101, Berlin, Germany, 1997. Springer.

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

[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`.