Assigning Function to CDS Through Qualified Query Answering: Beyond Alignment and Motifs

Terry Gaasterland **Natalia Maltsev** **Jorge Lobo** **Guo-Hua Chen**

Mathematics and Computer Science
Argonne National Laboratory
Argonne, IL 60439
gaasterland@mcs.anl.gov

Dept. of EECS
University of Illinois at Chicago
Chicago, Illinois 60607
jorge@eecs.uic.edu

Dept. of Chemistry
Illinois Institute of Technology
Chicago, IL
chen@mcs.anl.gov

Abstract

In this paper, we show how to use *qualitative query answering* to annotate CDS-to-function relationships with confidence in the score, confidence in the tool, and confidence in the decision about the function. The system, implemented in Prolog, provides users with a powerful tool to analyze large quantities of data that have been produced by multiple sequence analysis programs. Using qualified query answering techniques, users can easily change the criteria for how tools reinforce each other and for how numbers of occurrences of particular functions reinforce each other. They can also alter how different scores for different tools are categorized.

Introduction

When analyzing new DNA sequence data through available pairwise alignment and matching software, a series of issues arise: how does one compare the outputs from different software packages? how does one determine whether one package or another is more reliable? how can one use the results from one package to reinforce the results from another in a systematic way? And not least, how does one deal with the sheer volume of the output (e.g. for 300 sequences of length 100-300 nucleotides, the Blastx output is 25 megabytes of human-readable files).

In this paper, we show how to use *qualitative query answering* (GL94) to address these questions. Qualified query answering enables users to specify preferences and then receive answers according to those preferences. Specifically, it allows a user to specify a set of user constraints over the query domain and rank the constraints by annotating them according to preferences; accepts a query from a user and return a set of annotated answers; and allows the user to annotate queries and receive answers that satisfy the annotation.

A data collection system built by Gaasterland and Overbeek (GO94) sends contiguous DNA sequences out to a variety of software packages and parses the human-readable output into logical facts. Each fact represents a match between a section of an input query sequence and a section of a sequence in some database, e.g. GenBank, SwissProt, or the EMBL Nucleotide Databank, or a similarity between a query sequence and a motif pattern, linked in turn to local multiple sequence alignments of entries in sequence databases.

From the logical form of the output data, we use qualified query answering to merge the "opinions" from each different piece of software and make a qualified decision about what region of an input query sequence is a CDS and what its function is. The basic property of this scheme is that the criteria for making a judgement about the data are represented separately from the data itself and from rules for deriving new information from the data. This means that a set of rules for deriving a CDS to function mapping need be written only once. After that, it is straightforward for a user to change the combination criteria.

To show the power of qualified query answering for combining similarity hits against new DNA sequences, we have devised a scheme which adheres to the following combination considerations: (1) Within a sequence analysis tool, partition similarities by score; (2) Within a set of similarities involving a single protein, partition similarities by tool; (3) Let tools reinforce each other; (4) Look for families of proteins represented by multiple similarities against related proteins and partition similarities by confidence in family. These criteria allow weak similarities in one tool to be reinforced by other similarities in other tools. They also allow the occurrence of many different proteins from the same family to reinforce each other.

Once a basic set of rules are written for mapping similarity hits to CDS and function pairs, the criteria above are captured in a separate set of data, called user constraints. The user constraints, declarative statements that are easily changed, represent *goodness* of scores, *goodness* of tools, and *goodness* of families. In addition to the user constraints, a user provides a *lattice* hierarchy over the symbols assigned to each tool and to each category of score. The user can easily change these lattices to see how making one or another tool take precedence changes the outcome.

The next section discusses how the output from se-

quence analysis tools is represented through logical facts and a method to derive CDS-to-function relationships from those facts. Then a background section discusses qualified query answering and logic programming. The following sections show how to qualify CDS-to-function relationships with judgements about scores and tools in a way that allows different tools to reinforce each other. Finally, we show how the same approach is used to combine similarities against proteins into a judgement about a function.

Representing Sequence Analysis Data Through Logic

Available sequence analysis tools can be thought of as producing connections between the query sequence and sequences that appear in a variety of sequence databases (including versions of SwissProt (BB91; Bai93b), GDB (Pea91), and the EMBL Nucleic Acid Database (EMB93)). Those connections have a score associated with them. This functionality is clear in the Blast, Blaize, and Fasta families of sequence analysis tools (AGM+90; CC90; WL83). Each of these tools matches a section of a query sequence with sections of database sequences and assigns a score to that match. Blast, Blaize, and Fasta perform pairwise local sequence alignments. the functionality also applies to Blocks (HH93), which searches for Prosite motif patterns (Bai91) in a query sequence and associates that region of the query sequence with a multiple local sequence alignment — called a "block" — in which the block sequences each exhibit the prosite pattern in question. Blocks associates the query sequence with the best matching sequence in a "block."

Gaasterland and Overbeek (GO94) have built a system that runs sequences through a set of tools (Blocks, Blaize, Blastx, Blastn, Tblastn, and Fasta) and converts the output into logical facts that capture connections between query sequences and database sequences. Each fact has the following form:[1]

similarity([Contig, Fr, To],
[ProteinID, Fr_p, To_p],
Score, Tool).

This logical fact can be read as *There is a similarity between the input contig from DNA sequence location from Fr to To and ProteinID sequence location from Fr_p to To_p, with a score of Score using the tool Tool.*

For example, for a contiguous DNA sequence, say c030, in the region between 330 and 430, blastx associates it with SwissProt entry P04540 and blaize associates it with 'ARYB_MANSE':

similarity([c030,6,208],
[score(160),expect(0.0064),p(0.0064)],
[emb|M62622,51493,51695],blastn).
similarity([c030,689,1018],
[score(73),expect(1.3e-08),p(1.3e-8)],
[gb|X05182,865,536],tblastn).
similarity([c030,713,1024],
[score(99),per_match(4.3),pred_no(4.09e-5)],
[YM71_TRYBB,469,574],blaize).

With two straightforward rules, we have a declarative program that derives CDS/function pairs from the similarity facts for a sequence. The first rule invokes a search for an open reading frame (ORF) in a contig and for a similarity contained within that ORF:

hit(Contig, Fr, To, Protein) ←
orf(Contig, Fr, To),
similarity([Contig, Fr1, To1],
[Protein, Fr2, To2], Score, Tool),
within(Fr1, To1, Fr, To).

The *orf* relation can be read as *There is an Orf in sequence Contig from Fr to To.* The orf relation can be derived in many ways[2] The *within* relation is read as *the range Fr1-To1 is contained within the range Fr-To, and both ranges have the same direction (i.e. increasing or decreasing).*

Thus, the rule for a *hit* can be read as *there is a hit on Contig from Fr to To against protein Protein if there is an orf between Fr and To and a similarity with that protein within that region, using the tool Tool.*

A second rule derives a relation that relates a CDS to function:

cds_function(Contig, Fr, To, Fcn) ←
hit(Contig, Fr, To, Protein),
function_of_protein(Protein, Fcn).

The relation *function_of_protein* simply relates a protein to its function. For now, this relation ties enzyme proteins to their enzyme code (obtained from the EMBL Enzyme Database (Bai93a)) and other proteins to themselves. A refinement of this rule would not assume that a protein has a single function. Rather, it would ensure that the part of the protein associated with a given function is the part that is matches the CDS.

Using just these two rules and a collection of *similarity* facts, one can ask the following query about a particular contig, called say *c030*:

[1] Users who send a sequence to the system choose whether or not to use default parameter settings for matrices, genetic codes, fileters, and gap penalties; currently, the parameter information is not included in the similarity facts. However, the method described here does not preclude utilizing that information as well.

[2] We used a method devised by Overbeek which can be described simply as follows: *look for a stop codon and then look upstream (downstream on negative reading frames) for a start codon that is not preceded by a stop codon. If the upstream (downstream) search hits the end of the sequence, that is temporarily considered to be the "start" of the ORF.*

?- cds_function(c030, Fr, To, Fcn).

The result is a list of CDS-to-function relationships defined by *Fr*, *To* and the potential CDS *Fcn*.

However, although this is helpful in inspecting the data from all of the tools, it does not address the problem of how to combine the data from different tools. We must do more to accomplish the following: (1) allow various categorizations of scores for each tool to be used in determining how good a particular similarity is; (2) allow similarities from one tool to be preferred over similarities from another tool; (3) Allow multiple similarities from different tools to reinforce each other in a variety of ways; (4) Allow multiple hits against proteins in the same family to reinforce each other. Before showing how to accomplish these items, we first provide some background in the next section.

Qualified Query Answering

Much work has been done to explore methods to handle user preferences in databases (see (CCL90; CD89)), human computer interaction (see (AWS92)), user models (see (McC88; KF88)), and artificial intelligence (see (AP86; Pol90; Par87)). Cooperative answering systems try to enable users to receive answers that they are actually seeking rather than literal answers to the posed questions (see (Mot90; GM88; GGMN92)). Qualified query answering (GL94) presents a complementary approach that incorporates handling user preferences into the query answering procedure of a database. Once a declarative formalism for expressing user preferences and needs as a body of information separate from the database is defined, a query answering procedure then takes both preferences and data into account when providing answers.

The major advantages of qualified query answering are threefold. Preferences can be easily altered without touching the database. Users can ask for all answers either with or without the annotation of preference or for all answers that meet some level of preference. Preferences are captured by separate bodies of declarative information that can be changed independently. They are: (1) qualitative labels with an ordering expressed as an upper semi-lattice, (2) logical statements, and (3) a function for combining preferences.

The notions of *need* and *preference* are reflected through a lattice of values provided by the user. Lattice values are used together with logical statements to express preferences. As an illustration, consider a traveler, Kass, who wants to travel from Chicago to Amsterdam, preferably nonstop. If she has to make a stop, she would rather stop in Washington, where her boyfriend lives than in any other city. She absolutely does not want to stop in London. We can define a set of annotated user constraints that express Kass' restrictions:

nonstop_flight(A, B, Date, Flight):good.
direct_flight(A, B, Date, Flight):okay.
indirect_flight(A, B, Date, Flights):bad.
stopover(Flight, Airport):fine
← dc_airport(Airport).
stopover(Flights, Airport):terrible
← london_airport(Airport).

Consider the rule with the annotation *terrible*. The predicate *london_airport* in the body (to the right of the arrow) may be read as "Airport is located in London." The atom in the head (to the left of the arrow) may be read as "The flights in *Flights* involve a stopover in Airport." The entire constraint may be read as "A list of Flights that involves a stopover in Airport is terrible if the airport is a London airport." (See (Gaa92) for a discussion of natural language descriptions of constraints.) Furthermore, any answer that depends on a flight that stops over in a London airport should be annotated as terrible.

In this example, a set of five symbols {*terrible, bad, okay, good, fine*} reflects preference levels. Suppose the order *terrible* < *bad*, *bad* < *okay*, *okay* < *good*, *okay* < *fine* is assigned to the symbols; then a higher rank indicates a higher preference. Any upper semilattice of values may be used for ordering the symbols.

Now, when Kass asks the query "How can I travel to Amsterdam from Chicago on May 1?", expressed logically as, say, ← *travel(chicago, amsterdam, (may, 1, Time), TravelPlan)*, the search space of the query should be modified with the constraints so that nonstop flights are noted as *good*; direct flights through Washington as *fine*; flights through any other city, except London noted as *okay*, and so on. Alternatively, she may want to ask for flights that are *fine* or better. Then all answers below this level must be discharged.

Suppose the lattice contains only two values, say *unacceptable* and *acceptable* with the order *unacceptable* < *acceptable*. Let the user constraints on *direct_flight* and *nonstop_flight* be annotated with *acceptable* and the rest with *unacceptable*. In this case, the annotated user constraints reflect Kass' needs.

The method for handling user needs and preferences is summarized as follows: Once a user has provided a lattice of values and a set of user constraints annotated with the values, the constraints are automatically incorporated into a relational or deductive database through a series of syntactic transformations that produces an annotated deductive database. Query answering procedures for deductive databases are then used, with minor modifications, to obtain annotated answers to queries. In contrast with earlier work, the only burden on users is to express their preferences. The separation of preference declaration from data representation is achieved through the use of the theory of annotated logic programs, deductive databases, and the series of simple transformations that are invisible at the user level.

Now, we shall discuss annotation in logic programs.

following closely the notation in Kifer and Subrahmanian (KS92). An *annotated logic program* comprises a set of annotated clauses of the from:

$$A : \alpha \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n.$$

Intuitively, this may be read as "A is true with confidence (or quality) α if we can prove B_1 is true with confidence (or quality) β_1, ..., and if we can prove B_n is true with confidence (or quality) β_n." The βs on the right hand are usually combined in some way to produce the value α for the derived head atom. To illustrate this, we will modify the example above slightly. The following annotated rule says that a *stopover* has the annotation α if the airport on the right hand side has the annotation α:

stopover(Flights,Airport):$\alpha \leftarrow$ *airport(Airport):*α.

Suppose we have two airport facts: *airport(dc_national):fine airport(heathrow):terrible.* Then *stopover(Flights,heathrow)* would receive the annotation *terrible* and *stopover(Flights,dc_national)* would receive the annotation *fine.*

A and the Bs are atoms as usually defined in logic programs; α and the βs are *annotation terms.* $A : \alpha$ is the *head* of the annotated clause, and $B_1 : \beta_1, \ldots, B_n : \beta_n$ the *body.* The annotation terms are defined based upon an upper semi-lattice $\mathcal{T}$.

The lattice reflects the rankings of the user about the importance of states expressed in a user constraint. For example, consider the constraints from Section . They included a constraint about not stopping in London, a constraint about preferring direct flights over indirect flights, that is flights with a change of planes, and a constraint about preferring nonstop flights over direct flights. The user may assign the value *terrible* to the constraint about London and the value *bad* to the constraint about indirect flights, the value *okay* to the constraint about direct flights, and the value *good* to the constraint about nonstop flights. The bottom of the lattice is *terrible.* The lattice is completed with the top element *very good,* and the partial order is given by the transitive and reflexive closure of the following relation: *terrible* < *bad*, *bad* < *okay*, *okay* < *fine*, *okay* < *good*, *fine* < *very good*, *good* < *very good.*

We want to allow negation in the rules and facts of a deductive database. This can be easily done by extending the concepts of negation in normal logic programs to annotated programs. Such an extension based on the stable model semantics can be found in (GL94).

Definition 0.1 A *user constraint* υ is an annotated normal clause of the form:

$$A : \alpha \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n, \ \mathbf{not}C_n : \bot, \ldots, \ \mathbf{not}C_m : \bot.$$

where $A : \alpha$ is a c-annotated atom and the $B_i : \beta_i$ are c- or v-annotated atoms. ■

The user constraint υ can be interpreted as saying that if the antecedent of the implication, $(B_1 : \beta_1, \ldots, B_n : \beta_n, \ \mathbf{not}C_n : \bot, \ldots, \ \mathbf{not}C_m : \bot)$, is true then *at most* $A : c$ can be accepted to be true. Formally,

Definition 0.2 Let the annotated clause

$$A : c \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n, \ \mathbf{not}C_n : \bot, \ldots, \ \mathbf{not}C_m : \bot$$

be a user constraint υ and $I_{\mathcal{A}}$ an annotated interpretation. $I_{\mathcal{A}}$ satisfies υ iff for any ground instance $A' : c \leftarrow B_1' : \beta_1', \ldots, B_n' : \beta_n, \ \mathbf{not}C_n' : \bot, \ldots, \ \mathbf{not}C_m' : \bot$ of υ such that $(B_1' : \beta_1', \ldots, B_n' : \beta_n, \ \mathbf{not}C_n' : \bot, \ldots, \ \mathbf{not}C_m' : \bot)$ is satisfied in $I_{\mathcal{A}}$, $I_{\mathcal{A}}$ satisfies $A' : e$ only when $e \leq c$. ■

As a simplification to the user interface, we allow users to pair an annotation with the head of each constraint as follows:

$$A : c \leftarrow B_1, \ldots, B_n, \ \mathbf{not}C_n, \ldots, \ \mathbf{not}C_m.$$

To incorporate a set of user constraints into a logic program to produce an annotated logic program, two transformations are necessary. An initial transformation translates normal logic programs into annotated normal logic programs. Then a second transformation incorporates a set of user constraints, $\mathcal{U}$, into an annotated program, $\Pi_{\mathcal{A}}$.

For more information on annotated logic programs and the compilation procedures, the reader is referred to (GL94)

Qualifying similarities by score and tool

The first two requirements listed two sections ago specified that each answer be prioritized according to what tool was used to obtain it and according to the score within that tool. To achieve this using Qualified Query Answering, two sets of information is added to the program in the form of user constraints:

Partition by score for each tool For each tool, we add a set of user constraints of the form:

*similarity(_,_,Score,Tool):*S$\leftarrow$ *Tool=tool1,Score>N.*

where *tool1* is the name of a tool and N is a numerical cut-off level for *Score.*[3] S is a symbolic value from the lattice used for scores. For now, we use *strong, medium,* and *weak* as score symbols. An "_" denotes an argument of the predicate *similarity* that is not relevant in the user constraint.

Partition by tool For each tool, we add a set of user constraints of the form:

[3]This last expression, *Score* > N becomes a bit more complicated when *Score* actually consists of more than one number, as it does in the Blast family of tools. The actual implementation accommodates this complication.

similarity(_,_,_,Tool):T← *Tool=tool1.*

where *tool1* is the name of a tool and *T* is some symbolic value in the lattice used for tools.

In addition to the user constraints, a lattice for each set of symbols must also added to the program. For example, for the scores, we might impose a simple lattice in which *strong* > *medium* > *weak*.

Suppose that the symbols that have been assigned to each tool is the name of the tool itself. Then, for the tool lattice, we might impose something like the following for the set of tools that includes blaize, blocks, blastx, tblastn, and fasta:

Using a method called *semantic compilation*, the user constraints are compiled into the basic program, that is, into the two rules defined above, to produce the following new *annotated* program, in which *SCORE*, *TOOL*, *S* and *T* are annotation variables:

cds_function(Contig,Fr,To,Fcn):SCORE,TOOL←
  hit(Contig,Fr,To,Protein):SCORE,TOOL,
  function_of_protein(Protein,Fcn).

hit(Contig,Fr,To,Protein):SCORE,TOOL←
  orf(Contig,Fr,To),
  similarity([Contig,Fr1,To1],
    [Protein,Fr2,To2],
    Score,Tool):SCORE,TOOL,
  within(Fr1,To1,Fr,To).

Semantic compilation is also used to compile the the user constraints into the *similarity* data. With the user constraints above for score and tool, each similarity fact is transformed into the following annotated form:

similarity([Contig,Fr1,To1],
  [Protein,Fr2,To2],
  Score,Tool):SCORE,TOOL.

where the values for *SCORE* and *TOOL* are obtained by applying the user constraints for scores and tools to each *similarity* fact.

So, for example, the following similarity fact:

similarity(c030,[c030,354,383],
  [score(35),expect(5.0e+03),p(1.0)],
  [sp|P04540,566,575],blastx).

would be compiled into the following annotated fact:

similarity(c030,[c030,354,383],
  [score(35),expect(5.0e+03),p(1.0)],
  [sp|P04540,566,575],blastx):weak,blastx.

given that the expression *[score(35), expect(5.0e+03), p(1.0)]* evaluates into the annotation value *weak*.

With this new annotated version of the program, users can ask for *cds_function* relations that meet certain levels of score or tool or both. For example, to ask for *cds_function* relations with strong scores and a blastx level of tool a user would ask the following query:

?- cds_function(c030,Fr,To,Fuction):strong,blastx.

Users can also ask for each *cds_function* together with its qualitative annotation as follows:

?- cds_function(c030,Fr,To,Fcn):S,T.

From the similarity facts described above, the following answers are returned. (Notice that in the second and third cases, the names of the proteins have been retained as an indicator of the function; in the first and last cases, the SwissProt ID is used since there is no EC code for YM71_TRYBB — in the database that we are using):

cds_function(c030,350,404,'YM71_TRYBB'):weak,blastx.
cds_function(c030,2,208,emb|M62622|MISCCG):weak,blastn
cds_function(c030,968,1024,gb|X05182):medium,tblastn.
cds_function(c030,968,1024,'YM71_TRYBB'):medium,blaize

Allowing tools to reinforce each other

To allow tools to reinforce each other, we must specify a combination function for the tool symbols. One possible combination function is an enumeration of a variety of combination criteria as follows, where *f* is the combination function:

f(blocks,blastx) = blastx
f(blaize,blocks) = blastx
...

This function defines that a hit from *blaize* and from *blocks* should be regarded as having the same quality as a hit from blastx[4] and that a hit from *blocks* and from *blastx* should remain at the *blastx* level.

Another possible combination function is *least-upper-bound*, or LUB, within the lattice over the symbols in question. Using LUB over the lattice above, a similarity from blocks and a similarity from blaize would combine to have the quality *blastx*.

As described in the previous section, the user constraints and the combination function are compiled into the basic program. Now, the rule for *cds_function*

[4] Regardless of score, for now, for simplicity of presenting the approach.

is replaced with the following two rules (the compiled rule for *hit* remains the same as in the previous section):

cds_function(Contig,Fr,To,Fcn):SCORE,TOOL← *combo_hit(Contig,Fr,To,Protein)*:SCORE,TOOL, *function_of_protein(Protein,Fcn)*.

combo_hit(Contig,Fr,To,Protein):SCORE,TOOL ← *setof((S,T), (hit(Contig,Fr,To,Protein)*:S,T*), STs), combine_scores(STs,SCORE), combine_tools(STs,TOOL)*.

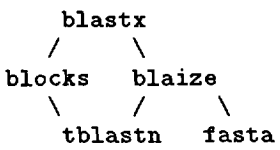
Careful inspection of this automatically generated program reveals that for a particular protein, a *cds_function* mapping takes the set of similarities between a particular ORF and that protein, applies the combination function for scores ond for tools, and returns a *cds_function* fact with a score and tool qualification.

Using this new program and the compiled data, the user can again ask for *cds_function* facts about contig *c*030 that are of a *blastx* quality with *strong* scores, but this time, the tool qualification reflects the combination of all similarities within an ORF. Again, this query looks like the following:

?- cds_function(c030,Fr,To,Fcn):strong,blastx.

As before, the user may also ask for *cds_function* relations qualified by their annotations.

Suppose the user wants to change how the tools combine to reinforce each other. Continuing to use the LUB combination function, the user would alter the tool lattice of symbols. So, for example, to allow *fasta* and *tblastn* similarities to combine to produce a similarity of a *blaize* level and to allow *blaize* and *blocks* similarities to combine to produce a similarity of a *blastx* level, the lattice is the following:



To change the cutoff levels for the scores, the user must alter the set of user constraints, and recompile the program.

Now, we shall turn to the issue of how to allow multiple hits against the same family reinforce each other. Again, we can achieve this by adding user constraints, a lattice, and a combination function to the program.

Allowing multiple similarities to reinforce each other

When analyzing matches of a query sequence against a database of known amino acid or DNA sequences, it is important to remember that a set of weak similarities against different proteins, or DNA sequence that codes for different proteins, may be meaningful if many of those proteins are related in function. The collection of weak hits may indicate that the query sequence region codes for a protein that has a similar function.

A single user constraint together with a combination function over the symbols specified by the user constraint captures enough knowledge to make the ranking of a *cds_function* relation higher when reinforced by multiple occurrences of proteins with similar function. The necessary user constraint simply annotates the *function_of_protein* relation used in the rule for *cds_function* above with the function itself:

function_of_protein(Protein,FCN):FCN.

The combination function must reflect how the user wants to combine a set of functions to rank a particular function. Recall that in an earlier analogous situation with the TOOL annotation, LUB was used for a combination function. If the user desired, they could use LUB here as well, but for this domain, it does not make intuitive sense. Instead, we shall illustrate the approach with a combination function that calculates the number of times that a particular protein function appears divided by the total number of appearances of protein functions.

More formally, for a particular ORF, the function for each protein that has a *similarity* relation with it is obtained. Then a list made up of each protein's function is obtained. Let the list have length L. Let the functions that appear in the list be denoted by $F_1, \cdots, F_n$. Let $count(F_i)$ be the number of times that F_i appears in the list, where $1 \leq i \leq n$. Then the annotation for F_i is

$$\frac{count(F_i)}{L}$$

This is just one possibility for a combination function. Users are free to use whatever combination function they wish to define.

The compilation of the original basic program with the user constraints and combination functions for *SCORE* (now abbreviated *SC*) and *TOOL* as well as *FCN* produces the following program:

cds_function(Contig,Fr,To,Fcn):FCN,SCORE,TOOL ← *combo_cds_fcn(Contig,Fr,To,Fcn)*:FCN,SCORE,TOOL.

combo_cds_fcn(Contig,Fr,To,Fcn):FCN,SCORE,TOOL ← *setof((F,S,T), (cds_function(Contig,Fr,To,Fcn)*:F,S,T*,FSTs), combine_functions(FSTs,FCN), combine_scores(FSTs,SC), combine_tools(FSTs,TOOL)*.

cds_function(Contig,Fr,To,Fcn):SCORE,TOOL ← *combo_hit(Contig,Fr,To,Protein)*:SCORE,TOOL,

function_of_protein(Protein,Fcn).

combo_hit(Contig,Fr,To,Protein):SCORE,TOOL ←
setof((S,T), (hit(Contig,Fr,To,Protein):S,T*), STs),*
combine_scores(STs,SC),
combine_tools(STs,TOOL).

hit(Contig,Fr.To,Protein):SCORE,TOOL —
orf(Contig,Fr,To),
similarity([Contig,Fr1,To1],
[Protein,Fr2,To2],Score,Tool):SCORE,TOOL,
within(Fr1,To1,Fr,To).

Recall that above we chose that both *combine_scores* and *combine_tools* use LUB, the least upper bound in the lattice, for the combination function. Note that either or both of these combination functions can be made more complex as the user wishes in order to reflect a different view of the data.

With this compiled program, a user can qualify queries with the following: (1) specification of a particular level of score; (2) specification of a particular level of tool; and (3) specification of a particular confidence in the cds to function mapping. Users can also ask queries that qualify each *cds_function* relation with its annotation scores.

Conclusion

The ability to annotate CDS-to-function relationships with confidence in the score, confidence in the tool, and confidence in the decision about the function provides users with a powerful tool to analyze large quantities of data that have been produced by sequence analysis programs. Using qualified query answering techniques, users can easily change the criteria for how tools reinforce each other and for how numbers of occurrences of particular functions reinforce each other. They can also alter how different scores for different tools are categorized. Without an automated method to deal with this data, the gap between the amount of known DNA sequence and the amount of interpreted DNA sequence will continue to increase. We have implemented the program described here and are in the midst of running it to evaluate the output from a set of sequence analysis software packages run on a set of DNA sequences for the *Mycoplasma capricolum* genome. The approach easily extends to new tools. To add a new tool, a set of user constraints defining score partitions and one user constraint that assigns a tool symbol to the tool must be added to the system. The tool symbol must also be added to the tool lattice.

For the *Mycoplasma capricolum* data, Chris Sander's group is also looking for hits among the contiguous *Mycoplasma capricolum* sequences by hand. They gather the data from the output of a wide array of analysis tools and then display it in a manner that is easy for a user to peruse visually (SS91; BOSgy). Our approach goes a large step further: it allows the user to build criteria for making judgements into the analysis system. With our approach, the same system that sends contiguous DNA sequences through each analysis package and parses the output files into Prolog facts can take the next step of analyzing the output and assigning function to CDSs within the sequences.

It is critical to validate the judgements about CDS-to-function relationships produced by our system with the judgements made by hand by Sander's group and by Maltsev at Argonne. So far, the results are consistent. Once we have worked through the *Mycoplasma capricolum* sequence data, we expect to have a tuned and validated system that will be useful for automatically assigning CDS-to-function through logical post-processing of output from sequence analysis tools.

Acknowledgments We thank Ross Overbeek and Pat Gillevet for their inspiration in this work and Zoran Budimlik for taking the time to implement the qualified query answering system. Terry Gaasterland was supported for this work by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38. The NSF partially supported Jorge Lobo for this work under grant #IRI-9210220.

References

S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

J. F. Allen and C. R. Perrault. Analyzing intention in utterances. In Barbara J. Grosz, Karen Sparck Jones, and Bonnie Lynn Weber, editors, *Readings in Natural Language Processing*, pages 441–458. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1986.

C. Ahlberg, C. Williamson, and B. Shneiderman. Dynamic queries for information exploration: An implementation and evaluation. In *Proc. of the ACM CHI '92*, pages 619-626, California, 1992.

A. Bairoch. Prosite: A dictionary of sites and patterns in proteins. *Nucleic Acids Research*, 19:2241–2245, 1991.

A. Bairoch. The enzyme data bank. *Nucleic Acids Research*, 21:3155–3156, 1993.

A. Bairoch. The swiss-prot protein sequence data bank: User manual. release 25, april 1993. (e-mail to *netserv@embl-heidelberg.dc get prot:userman.txt*)

A. Bairochand and B. Boeckmann. The swiss-prot protein sequence data bank. *Nucleic Acids Research*, 19:2247–2249, 1991.

P. Bork, C. Ouzounis, and C. Sander. From genome sequences to protein function. 1994 (submitted to Current Opinion in Structural Biology).

J.F. Collins and A. Coulson. Significance of protein sequence similarities. In R.F. Doolittle, editor, *Methods in Enzymology*, 183:474–486. Academic Press, 1990.

W. W. Chu, Qiming Chen, and Rei-Chi Lee. Cooperative Query Answering via Type Abstraction Hierarchy. In *Draft Proc. of the Intl. Working Conf. on Cooperative Knowledge Based Systems*, pages 67–68, U. of Keele, England, Oct. 1990.

F. Cuppens and R. Demolombe. How to Recognize Interesting Topics to Provide Cooperative Answering. *Information Systems*, 14(2):163–173, 1989.

EMBL. Embl data library: Nucleotide sequence database: User manual release 36, september 1993. (ftp to *ftp.embl-heidelberg.de* in */pub/databases/embl/doc*).

T. Gaasterland. *Cooperative Answers for Database Queries*. PhD thesis, U. of Maryland, Dept. of Computer Science, College Park, 1992.

T. Gaasterland, P. Godfrey, J. Minker, and L. Novik. A Cooperative Answering System. In Andrei Voronkov, editor, *Proc. of the Logic Programming and Automated Reasoning Conf.*, pages 101–120, Vol. 2, St. Petersburg, Russia, July 1992.

T. Gaasterland and J. Lobo. Qualified answers that reflect user needs and preferences. In *20th Intl. Conf. on Very Large Databases*, Santiago, Chile, 1994.

A. Gal and J. Minker. Informative and Cooperative Answers in Databases Using Integrity Constraints. In V. Dahl and P. Saint-Dizier, editors, *Natural Language Understanding and Logic Programming*, pages 277–300. North Holland, 1988.

T. Gaasterland and R. Overbeek. An automated system for gathering sequence analysis data from multiple tools. Technical report, 1994. In preparation.

S. Henikoff and J. Henikoff. Protein family classification based on searching a database of blocks (document: blockman.ps). (ftp to *sparky.fhcrc.org* in */blocks*).

R. Kass and T. Finin. Modeling the user in natural language systems. *Computational Linguistics*, 14(3):5–22, Sept. 1988.

Michael Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its appli cations. *Journal of Logic Programming*, 1992.

K. McCoy. Reasoning on a highlighted user model to respond to misconceptions. *Computational Linguistics*, 14:52–63, Sept. 1988.

A. Motro. FLEX: A Tolerant and Cooperative User Interface to Database. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):231–245, June 1990.

C. Paris. Combining discourse strategies to generate descriptions to users along a naive/expert spectrum. In *Proc. of IJCAI*, pages 626–632, Milan, Italy, 1987 Aug. 1987.

P. Pearson. The genome data base (gdb) – a human gene mapping repository. *Nucleic Acids Research*, 19:2237–2239, 1991.

M. E. Pollack. Plans as complex mental attitudes. In M.E. Pollack P.R. Cohen, J. Morgan, editor, *Intentions in Communication*, pages 77–103. MIT Press, 1990.

C. Sander and R. Schneider. Databases of homology-derived protein structures and the structural meaning of sequence alignment. *PROTEINS: Structure, Function, and Genetics*, 9:56–68, 1991.

W. Wilbur and D. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proc. Natl. Acad. Sci. U.S.A.*, 80:726–730, 1983.