Experiences using logic programming in bioinformatics

Chris Mungall

Abstract. Reverse engineering complex biological systems requires the integration of multiple different databases using detailed background knowledge. Logic programming can provide a means of both performing integrative queries and rule-based inference to account for implicit knowledge.
The Biological Logic Programming toolkit (Blipkit) was developed as a means of doing this kind of data integration. Implemented in SWI-Prolog, Blipkit has models of different aspects of life sciences data, including genes and gene sequences, RNA structures, evolutionary relationships, phenotypes and biological interactions. These can be combined to answer complex questions spanning multiple datasources. Blipkit also has means of integrating with and combining life sciences databases and ontologies.

1 Introduction

1.1 Background

The study of biological systems is progressing at an astonishing rate. The determination of the three billion bases of the reference DNA sequence of the human genome in 2001 was a landmark event in science, but accomplishment will be dwarfed with the advent of next-generation massively parallel sequencing technologies which allow the sequencing of genomes of individual organisms or cells on a truly massive scale. The scientific and medical potential is enormous. For medical purposes we would like to know the mechanisms by which individual chemical changes in DNA molecules combine with other forces to give rise to effects of clinical importance. However, a DNA sequence is not in itself sufficient to unlock this potential: sequence data must be analyzed in the context of other rich and complex data derived from a variety of life forms. How is the gene structured in the genome? How is it related to other genes, going back to common ancestors hundreds of millions of years ago? What structures do these genes encode, and how do these structures interact with other similar structures to give rise to a functioning organism – or in the case of deleterious genetic variation, the dysfunctioning of an organism?

This richness of data and the attendant challenges in analyzing it has lead to a new multi-disciplinary endeavour: bioinformatics, the application of computer science and informatics to solving biological problems. One of the biggest challenges in bioinformatics is *semantic multi-scale data integration*, automatically combining facts from a variety of heterogeneous sources using background knowledge to answer complex questions and yield new insights. Historically this

integration has been done in an ad-hoc fashion, using scripting languages to write disposable programs for gathering together data for specific purposes[19]. Recognizing the inherent problems with this approach, the community has been a move towards providing Application Programmer Interfaces (APIs) to data, both object-oriented and web services based[20]. These systems are typically successful for answering specific questions about a limited range of datatypes, but queries across databases cannot easily be combined. In addition, it is difficult to integrate knowledge and semantics into the query answering process[21].

Whereas a large portion of scientific programming comes down to "number crunching", many bioinformatics analyses come down to "symbol crunching". This can be an ideal substrate for logic programming (LP). LP has been successively applied to individual problems, programming "in the small", but there has been less in the way of using LP for integrative analyses and programming "in the large".

The Biological Logic Programming toolkit (Blipkit) was developed as an experiment in applying LP techniques to data integration and "programming in the large" in bioinformatics. It is a collection of prolog modules for integrating, modeling, querying and performing complex operations over diverse biological data. Also known as BioProlog, it comprises one of the Bio* projects under the aegis of the Open Bioinformatics Foundation (OBF), alongside BioPerl, BioJava, BioRuby and others. Each of these projects represents a community effort to provide a relatively comprehensive integrated library of code for researchers in the life sciences that takes advantages of the features of the host programming language. Most of these bioinformatics language libraries use an object-oriented approach, as is popular in software engineering today. Blipkit, written in SWI-Prolog[23], offers a radically different approach, and can be described as predicate-oriented as opposed to object-oriented. The assumption underpinning the development of Blipkit was that this would offer some unique advantages, especially when applied to complex multi-source data integration problems requiring the application of logical rules and inference.

This paper first describes the organizational principles of the library, then illustrates a subset of the domains covered, focusing on the biology of genomes. The development of this library has yielded some interesting lessons, both in bioinformatics and for the logic programming community at large. These are presented at the end of the paper.

2 A Biological Logic Programming Toolkit

2.1 Modular Organisation

Most software developers would agree that all non-trivial programming projects benefit from a modular design. Partitioning programs into modules helps make programs maintainable by ensuring a separation of concerns. Unfortunately, Prolog systems vary tremendously with respect to their module systems (if they provide them at all). This has not historically been a problem for "programming

in the small" style research projects, but it does hamper the adoption of Prolog for "programming in the large" style projects. The variability of module systems means that Blipkit is largely restricted to its host implementation, SWI-Prolog. The implications of this decision are discussed later on.

The modular organization of blipkit is as follows. Modules are organized into *packages*, with each package consisting of multiple modules and corresponding to a particular domain of the life sciences: for example, one package for representing genes, the other for representing the evolutionary history of those genes, and yet another for the interactions those genes participate in. There are also packages that are independent of the life-sciences per-se – for example, packages for representing metadata, or for working with relational databases. Table 1 shows the packages that comprise Blipkit.

Table 1. List of packages included in Blipkit

Package	Core Model	Description
metadata	**metadata_db**	annotations and metadata
ontol	**ontol_db**	ontologies
genomic	**genome_db** , **seqfeature_db**	DNA sequences and features (1D)
structure	**rna_db**	Secondary and tertiary structures (2D and 3D)
phylo	**phylo_db**	Phylogenetic trees and evolutionary history
pheno	**pheno_db**	Phenotypes and diseases
curation	**curation_db**	Statements about biology
sb	**sb_db** , **interaction_db**	Systems biology and interactions
sql		Relational databases
web		Accessing data and services over the web
serval		Web application framework
stats		Statistical calculations
blipcore		Miscellaneous, I/O

Models Each package is centered around one or more *models* of the domain in question[1]. Model modules by convention always have the **_db** suffix. For any given package there may be multiple models, each representing complementary overlapping views on that domain. Model modules consist of modeling predicates, each corresponding to some relationship or type within that domain. These predicates are split into two disjoint categories, extensional and intensional. Extensional predicates are intended to be loaded or asserted as unit clauses or *facts* (i.e head with no body), whilst intensional predicates have a body and are never asserted. From a database perspective these can be thought of as tables and views respectively. By convention, models generally adhere to the Datalog

[1] The term model is here used in the sense of a schema or data model, rather than of a stable model in disjunctive datalog

subset of prolog (i.e. no compound terms as arguments) in order to increase interoperability with relational databases and datalog systems.

Prolog has no inherent notion of intensional and extensional predicates – clauses are largely treated equally (although systems may distinguish between dynamic and compiled predicates, these do not strictly correspond). Blipkit has a metamodeling module **dbmeta** that each model uses to describe itself, primarily using the **extensional/1** directive/predicate to declare a predicate as being extensive. This makes it easier to perform options such as writing all facts in a model to a file. This can be thought of as partially analagous to the Data Description Language (DDL) in a relational schema.

In addition, both extensional and intensional predicates are extensively documented using the PlDoc system[24].

Utility modules A large chunk of bioinformatics is unfortunately devoted to prosaic plumbing exercises - there is a bewildering plethora of different data formats of varying degrees of formality, and these formats must be parsed in order to be able to integrate data from a variety of source. Often it is useful to be able to export model facts conforming to these formats.

Thus blipkit provides many different parsers, translators and writers to handle these different formats. These are divided into four categories: (i) XML-based formats are parsed using the SWI-Prolog SGML package, and then translated to model assertions using a prolog XSLT-like mapping specification (ii) Tabular formats are translated to prolog facts using a uniform syntactic translation, and then the facts are translated to model assertions using prolog (iii) For some formats for which a BNF grammar exists or can be created, Blipkit uses a DCG (Definite Clause Grammar) to parse the data (iv) for other formats we leverage parsers written in other languages such as perl, and write perl programs to generate prolog facts.

Uniform naming conventions help clarify large codebase. By convention, parsers are named **parser_*format*** . XML to model translation modules are named ***model*_xmlmap_*format*** .

There are also modules for importing and exporting facts from ontological representations and databases - these are discussed further on.

A general purpose module called **io** deals with input and output from these formats. Each blipkit installation also has a data source registry. This means that the programmer typically just has to call the **load_biosource/1** command, giving the name of the data source, and the Blipkit system will utilize the correct parser and populate the relevant model module with facts. The system will also take care of fetching remote data dumps across HTTP, maintaining a local cache. The SWI-Prolog **qcompile/1** predicate is immensely useful for making a fast-loading pre-compiled prolog database from a data source.

2.2 Ontologies and metadata

Regardless of the domain being represented, there are often common modeling predicates that can be reused in a number of contexts. For example, regardless

of whether the elements in our domain are representations of genes, chemical entities or human patients, these elements have shared attributes such as the *primary label* by which these elements are known, *alternate labels*, *identifiers* and so on. The **metadata_db** model consists of mostly extensional predicates such as **entity_label/2** , **entity_synonym/2** and so on.

Many Blipkit models are intended to work in concert with *ontologies*, computable representations of the types of entity in some domain. The Open Biological Ontologies (OBO) project organizes and stratifies the various different ontologies used in bioinformatics[17] into orthogonal domains, and the blipkit **ontol_db** model is geared towards these ontologies, and is based on the OBO language. This model includes extensional predicates such as **subclass/2** , which represents the *is_a* relationship[16] that can hold between two ontology classes. The intensional predicate **subclassT/2** is the transitive version of this predicate, defined recursively in the standard fashion and resolved through WAM-based backward chaining. The lack of tabling in SWI-Prolog can be problematic here; certain kinds of inferences will lead to cycles. Two alternate strategies are employed - one strategy is to write the facts to a file and use a different set of intensional predicates within a Prolog such as Yap or XSB. In general it is preferable to do the inferences directly from SWI-Prolog, so Blipkit also provides a forward chaining inference engine called **ontol_reasoner** . This iteratively applies rules to the existing database, asserting new facts until no new facts can be inferred. Expressivity is limited in that rules which produce infinite or prohibitively large databases are avoided.

One of the most popular ontologies in Bioinformatics is the Gene Ontology (GO)[3], which is used to assign functions and cellular locations to the products of genes. This adds semantics to gene databases, as it allows computable statements such as "this gene encodes a product that regulates the transcription of other genes" or "this gene encodes a product that produces the neurotransmitter dopamine ". The **curation_db** model is used to represent these assignments, together with associated provenance.

The **ontol_db** module is also used to implement Obol grammars[13]. These provide a configurable means of automatically translating between biologist-friendly textual descriptions and formal logical expressions using Definite Clause Grammars (DCG), and have proved invaluable for the GO[12].

For example, the following portion of a DCG can be used to relate the tokenization of the string "permeability of mitochondrial membrane" to a prolog expression that can be reasoned over:

```
phenotype( A that attribute_of( C ) ) --> physical_attribute(A), [of],object(C).
object( membrane that surrounds(C) ) --> relational_adjective(C), [membrane].
```

2.3 Genomes

Every living organism has a genome - a blueprint for life, carried by each cell in that organism, encoded as a discrete sequence of chemicals called bases arranged along a backbone of DNA (Deoxyribonucleic Acid). There are four types of bases

used, conventionally abbreviated as A, C, G and T, and these can be thought of as the symbols of a molecular alphabet life uses to feed instructions to the cellular machinery The genome sequence is passed on from generation to generation, with successive modifications and rearrangements giving rise to new phenotypes (organismal characteristics) and ultimately new species.

The layout of information along a genome sequence can be difficult to reverse engineer. The fundamental units are *genes*, genome subsequences which encode the molecular agents that give rise to myriad biological processes. In many organisms, these gene sequences are often split into subsequences called *exons*, interspersed with subsequences called *introns* (figure 1). The introns are later excised by the cellular machinery. Whilst this may seem wasteful, it serves many functions, one of which is to allow different combinations of exons of the same gene to come together in vary contexts.

The blipkit **genomic** package contains a number of modules for representing genome sequences and manipulating those representations. The architecture of a genome can be conceived of in terms of discrete intervals. The inter-relationships of these intervals can be formalized using the Allen Interval Calculus[2]. The **biorange** module contains predicates for determining the relationship between two intervals. This module extends the interval calculus as it deals with *reverse complementation* – every DNA molecule has two complementary strands going in reverse directions. A base on one strand is paired with a complementary base on the other strand.

This module also includes relationships between discontiguous sets of intervals, such as **interleaves/2** . These kinds of complex relationships are often observed in higher organisms - figure 1 shows two interleaved genes on opposite strands.

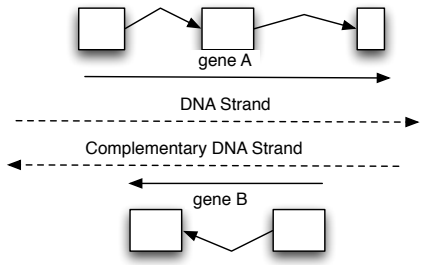

Fig. 1. Two genes, each denoted by an arrow, interleaved on opposite strands of DNA (dotted lines). When the gene is transcribed by the cell, the introns (bent lines) are excised and the exons (boxes) are joined together. Different combinations (called transcripts) are possible in different contexts (for example, the middle exon of gene A may be omitted).

There are many other types of features encoded along a genome sequence - in fact there is an entire ontology dedicated to these, the Sequence Ontology[9][10]. Other important feature types include various classes of regulatory region - these provide cellular context for the switching on and off of genes as part of complex regulatory networks. If genes are activated at the wrong time the consequences can be disastrous - one illustration of this is in the fruitfly version of the human PAX6 gene (implicated in eye conditions in humans). Inappropriate expression of this gene leads to the insect growing eyes in the wrong places such as on wings and legs.

Fast retrieval of features within a given range is a common operation. There are two implementations of this: the first is a pure prolog implementation, the second implementation is in C and uses the Nested Containment List argorithm[1]. The SWI-Prolog Foreign Language Interface (FLI) is used to wrap this such that it is seamlessly accessible via prolog.

The **biorange** module ignores the actual bases themselves and treats genomic features as intervals. The **bioseq** module has various predicates for performing operations on sequences, including excising subsequences and finding the complement of a sequence. These sequence operations are very standard for any kind of bioinformatics software library.

Neither of these modules are models, according to how blipkit partitions packages. The predicates do not lookup facts in the prolog database, they operate entirely on the arguments supplied as input.

There are two complementary models for representing genomic features: **genome_db** and **seqfeature_db** ; in addition there is a largely orthogonal model called **seqanalysis_db** . In addition, there is a module for performing operations on genomic intervals, **range** , a module for handling sequences, **bioseq** .

The **genome_db** model contains a *direct* prolog representation of the elements of a genome, with unary predicates corresponding to the major types of feature (as represented in the Sequence Ontology), as well as extensional n-ary predicates for representing the relationships between features. The collections of facts below corresponds to the upper part of figure 1

```
gene(geneA).
exon(geneA_exon1).
exon(geneA_exon2).
exon(geneA_exon3).
dna(dnaseq1).
transcript(geneA_transcriptX).
exon_transcript_order(geneA_exon1,geneA_transcriptX,0).
exon_transcript_order(geneA_exon2,geneA_transcriptX,1).
exon_transcript_order(geneA_exon3,geneA_transcriptX,2).
exon_dnaseq_pos(geneA_exon1,dnaseq1,1000,2000,1).
exon_dnaseq_pos(geneA_exon2,dnaseq1,3000,4000,1).
exon_dnaseq_pos(geneA_exon2,dnaseq1,6000,6500,1).
```

Note that there are no facts for the introns (gaps between exons). Given any two successive exons on a transcript, we can *infer* the existence of an intron

between them, as well as the position of that intron. The intensional predicate **intron/1** provides this inference, using a skolem term as intron identifier:

```
intron( intron(Seq,End,Beg,Str) ):-
        exon_transcript_order(X1,T,R1),
        exon_transcript_order(X2,T,R2),
        1 is R2-R1,
        exon_dnaseq_pos(X1,Seq,_Beg,End,Str),
        exon_dnaseq_pos(X2,Seq,Beg,_End,Str).
```

The arguments of the skolem term can be used in further intensional predicates for inferring the position of the inferred intons. A number of other feature types are inferred using intensional predicates (not shown here for brevity). This kind of inference may seem simple to a logic programming expert, but in fact it turns out to be tremendously useful for querying genome databases, where incomplete information is the norm. In bioinformatics, a little inferencing can go a long way.

One limitation with the current inference model is that the decision as to which feature types are extensional and which are intensional must be made a priori. Whilst it is more commonly the case (for example) that introns need to be inferred from exons, occasionally the situation is reversed. If we attempt to mutually define exons and introns in terms of one another, then it is difficult to use an untabled Prolog without the execution of queries getting caught in infinite loops.

A number of options are being explored here. One option is to use a Prolog which allows tabling, such as Yap or XSB. Currently this requires manual porting of code, as Blipkit is SWI-specific. So far there is an extension to the **genome_db** that works with Yap and allows for a greater variety of inferences, and allows for greater flexibility in what features are stated and which are inferred. However, at this time the additional modules of Blipkit have not been ported, which limits the potential for powerful integrative cross-domain queries.

Another option simultaneously being explored is to use an even more expressive logical modeling paradigm such as Answer Set Programming (ASP) and Disjunctive Datalog. There is a separate implementation of the **genome_db** model written for the DLV system, part of an extension of core Blipkit. Core DLV lacks the ability to have compound terms as predicate arguments, which means that the intensional predicate for intron shown above cannot be used. Fortunately there is an extension to DLV called DLV-complex which does allow for these more expressive rules. The use of DLV opens up other possibilities, such as disjunctions in the head of rules, and the ability to specific constraints on the model. This is extremely useful in the context of incomplete or incorrect genomic information (a common scenario). Of course, DLV is not a Prolog system so the rest of Blipkit cannot be used unless it too is converted.

In contrast to **genome_db** , the **seqfeature_db** model contains an *indirect* representation of the elements of a genome using a generic **feature/1** predicate, with the type of the feature represented using a binary **feature_type/2**

predicate, the second argument of which is the name of the type taken from the Sequence Ontology. In other words, types are *reified* (i.e. they are predicate *arguments* and are thus part of the domain of discourse), in contrast to **genome_db** , where the types are directly *realized* in the prolog model as predicates. There is a *bridge* module for converting between these two representations.

2.4 RNA Molecules

Despite the inherent 3-dimensionality of DNA molecules, their sequences are essentially linear and 1-dimensional. The cellular machinery transcribes genes encoded along the DNA molecule into RNA molecule transcripts which assume 3-dimensional configurations. These RNA molecules were once assumed to be passive intermediaries in between protein coding genes and protein molecules. However, more recently attention has been focused on RNA genes, as these molecules are shown to play critical roles in the cell, and have been implicated in a number of diseases. In fact it has even been hypothesized that the origin of all life was the "RNA world".

An RNA molecule, like a DNA molecule, can be modeled as a discrete sequence of bases. However, additional interactions between the bases become more significant.

The relationships between bases are modeled in **structure_db** using a hierarchy of binary relations such as **five_prime_to/2** and **paired_with/2** , derived from the RNA Ontology[4]. We can then define intensional predicates to infer the presence of higher level relationships and features such as bulges and loops.

Figure 2 shows an RNA pseudoknot, the existence of which can be inferred using the intensional predicate printed below.

```
pseudoknot(X):-
        stemloop(SL1),
        stemloop(SL2),
        stem(S2),
        loop(L1),
        has_part(SL2,S2),
        has_part(SL1,L1),
        part_of(S2,L1),
        mereological_union(SL1,SL2,X).
```

This is another area in which tabling is extremely useful. The RNA module of Blipkit works best with Yap Prolog.

2.5 Phylogenetics

A phylogenetic tree is a representation of the evolutionary relationships between either a collection of species, molecules or molecular sequences. Unlike a taxonomic tree, the branches or edges of a phylogenetic tree are labeled with distance quantities.

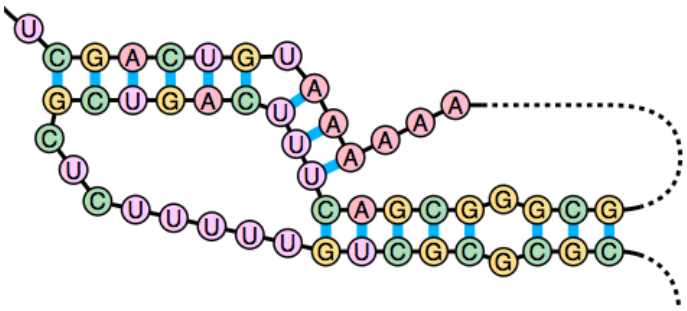

Fig. 2. RNA Pseudoknot

The blipkit **phylo_db** model is used for representing phylogenetic relationships. The extensional predicates correspond to the branches of the tree and their lengths, together with additional metadata about the nodes (for example, taxonomic identifiers). Intensional predicates are used for derived information, such as transitive ancestors and cumulative branch distances. There are additional intensional predicates for comparing two phylogenetic trees.

Gene or protein trees are more complex than organismal phylogenetic trees as both speciation and duplication events have to be taken into account. The latter happens when the genome of an organism (in whole or in portion) is duplicated within the cells of that organism, and passed on to its ancestors. This can be crudely thought of as initially comprising a "backup copy" of the genome, which then affords more freedom for other genes to vary and evolve. The **phylo_db** model includes intensional predicates for inferring the relationship between two genes based on whether the splitting event was a speciation or a duplication.

Two popular formats for exchanging phylogenetic trees are New Hampshire format and PhyloXML. Blipkit supplies a DCG for the former and an XML mapping for the latter. The Nexus format is more complex, and has a pre-existing Prolog parser that is being adapted for use with Blipkit.

One limitation of the **phylo_db** model is that the structure of the tree is modeled as extensional predicates. The trees are themselves inferred, and this inference could be performed in Prolog. One advantage of doing this in Prolog is to combine information about shared features encoded using ontologies[11]. For now, it is assumed that tree inference is performed outside blipkit, which justifies the use of extensional predicates.

2.6 Systems Biology and Biological Interactions

The study of genes in isolation is by definition reductive. Systems biology aims to take a more holistic approach, looking at genes in the context of the function of other genes in the cell, and thereby study the emergent properties that underly living systems. In practice this usually takes the form of studying biological pathways and the interactions between genes, gene products and chemical

entities within and between cells. The Blipkit **sb** package provides functionality for dealing with systems biology data.

Blipkit includes the **sb_db** model, strongly influenced by the SBML standard. SBML is an XML-based exchange format for representing cellular interactions in a quantitative fashion. SBML can be imported and exported by many simulation tools. SBML also includes the ability to annotate the model using OBO ontologies with RDF.

Like SBML, **sb_db** has a reaction-centric view, and thus has predicates **reaction_reactant/2** , **reaction_product/2** and **reaction_modifier/2** for representing the inputs, outputs and agents in a biochemical reaction event. In SBML terminology, these are all "species". SBML allows the expression of complex formula via embedded MathML. In **sb_db** these have their cognate representations as nested Herbrand terms.

At this time, there is no quantitive modeling built in to Blipkit. However, it is possible to perform simple qualitative modeling – for example, the **species_path/3** intensional predicate allows us to ask if there is a chain of reactions that allows species A to be produced from species B.

Another format commonly used is the RDF-based BioPAX. Data expressed in BioPAX can either be directly translated to the **sb_db** model using the provided Blipkit bridge module (which itself uses the SWI-Prolog RDF library). There is an alternative model which is a direct automated translation of the BioPAX schema (expressed in OWL) to Blipkit extensional predicates. The programmer has the option of choosing, depending on which model provides the best view for the questions they wish to ask.

Yet another model within the **sb** package is the **interaction_db** model. This provides a simplified view in terms of a binary **interacts_with** extensional predicate. Databases such as BioGRID[6] aggregate protein-protein interaction data from a variety of sources. This data can be extremely powerful when combined with other data types.

3 Integration with Relational Databases

Bioinformatics is an information science, and much of the relevant information is stored in relational databases. Sometimes these databases are supplied with an API (accessed via a specific programming language or as a web service) that allows for programs to access data. However, these APIs typically do not allow complex boolean expressive queries, as is possible with direct SQL queries. Fortunately, many of the important bioinformatics databases make their data available as database dumps, or have an open SQL port through which queries can be made remotely. For example, the ENSEMBL database system[5] has genomic annotation data for most sequenced genomes and is thus crucially important for bioinformatics analyses.

The pure subset of prolog is in part an extension of the relational model, and the methods for mapping prolog predicates to relational databases have been known for nearly twenty years[8]. Using the **sql_compiler** code written by

Christoph Draxler, it is possible to automatically translate complex prolog goals into SQL queries. This stands in contrast to the more commonly used imperative languages based on object-oriented principles, in which the impedance mismatch between the two formalisms is known to be problematic[7].

Draxler's **sql_compiler** code has been adapter to work with various Vendor-specific prologs such as Ciao, YAP and XSB. This has also been adapted to SWI-Prolog as part of the Blipkit library. The adaptation also introduces numerous novel extensions to the original code, including:

- **Database connectivity.** The original **sql_compiler** writes out SQL rather than making queries to the database. This functionality is still retained, but additional functionality is added using the SWI-Prolog ODBC library to connect directly to a database and translate the result sets back into prolog terms. In addition, prolog predicates can be bound to a database handle, such that prolog execution of the goals will call the database "behind the scenes". Similar capabilities are available in some of the other adaptations such as the one in Ciao prolog.
- **Query optimization.** The original **sql_compiler** can produce some inefficient SQL queries for complex goals. By adding additional metadata about the schema, particular unique key declarations, the Blipkit **sql_compiler** can rewrite queries to avoid redundant joins, resulting in faster queries.
- **Query rewriting based on prolog clauses.** In the original **sql_compiler** , all the terminal subgoals in the compiled goal must correspond to tables in the relational schema. The Blipkit **sql_compiler** will rewrite subgoals that correspond to prolog intensional predicates. This is in fact quite simple to do due to the introspectability of prolog, and it turns out to be a powerful feature, as the same prolog rules can be re-used in both in-memory contexts and relational database contexts. The one proviso is that the prolog rules are non-recursive pure prolog.
- **Mapping to SQL Functions.** Prolog functions such as **sub_atom/5** can be translated to SQL functions such as **substr** . This increases interoperability between the prolog code and SQL.

Blipkit includes both prolog metadata on a number of bioinformatics relational schemas such as Ensembl and Chado[14], and mapping modules that translate between the schema and models. These are shown in table 2.

Schema	Mapping	Model	Package
Chado	**seqfeature_sqlmap_chado**	**genome_db**	genomic
Ensembl-core	**genome_sqlmap_enscore**	**genome_db**	genomic
Ensembl-compara	**phylo_sqlmap_enscompara**	**phylo_db**	phylo
GODB	**homol_sqlmap_go**	**homol_db**	phylo
GODB	**ontol_sqlmap_go**	**ontol_db**	ontol

Table 2. Mappings between relational schemas and prolog models

The presence of mapping modules means that the same prolog model can be used regardless of the underyling database schema. For example, the **genome_db** model includes an intensional predicate called **gene_overlaps/2** to test if two genes overlap on the same DNA strand. This is defined as follows:

```
gene_overlaps(G1,G2):-
        gene_dnaseq_pos(G1,Seq,Beg1,End1,Str),
        gene_dnaseq_pos(G2,Seq,Beg2,End2,Str),
        End1 >= Beg2,
        Beg1 =< End2.
```

If blipkit is configured with this predicate bound at an Ensembl instance, then calls to **gene_overlaps/2** will be translated behind the scenes to SQL queries involving joins over multiple tables and executed against a remote Ensembl server, with the results bound non-deterministically to prolog variables. The exact same predicate definition can also be used with in-memory prolog databases.

Neither of the two main open source database vendors provide means of executing recursive SQL. To get round this limitation, Blipkit allows the use of the **ontol_db** module to do some semantic query expansion. The strategy is to perform the recursive part of the query first in Prolog, and then feed the results back in to the SQL query. This is all taken care of behind the scenes in the mapping module.

4 Integrating with XML and Web Services

Many biological databases and analysis programs are available as Web Services. The National Center for Biotechnology Information (NCBI) make a number of core databases and services available through eUtils[15].

The SWI-Prolog **http_client** library makes it simple to access these services programmatically. Blipkit contains a number of pre-written web wrapper modules for a variety of services. These include all the databases at NCBI, including sequence databases and the PubMed service. There are also wrappers for caBIG and Cancergrid.

In addition to these bioinformatics services, there are also wrappers for querying Google, Yahoo and Wikipedia. These wrappers also provide the ability to perform semantic query expansion using ontologies. For example, a search for "neurotransmitter" will be expanded to the various subtypes of neurotransmitter such as dopamine and serotonin.

Many web services return results in XML. In addition, many databases provided static XML dumps. SWI-Prolog provides the **sgml** module for parsing XML, which translates XML documents into nested Herbrand terms. Mapping code that translates between these XML terms and models can be quite verbose, so Blipkit has an XSLT-inspired library called **xml_transform** that can be used to quickly write mapping code. This illustrates how prolog is a suitable language for writing DSLs (Domain Specific Languages).

5 Integrating with Ontologies and the Semantic Web

The native ontology module in Blipkit is based around the OBO model, commonly used in Bioinformatics. Information is also increasingly available as OWL ontologies, or as RDF triples.

For RDF datasources, SWI-Prolog provides the **semweb** package[25] which allows for fast parsing of RDF sources. This is used for mapping some external datasets, such as those encoded using BioPAX, into Blipkit models.

Even though the Web Ontology Language OWL can be layered on RDF, it turns out that triples often provide a poor level of abstraction for complex class expressions. Here we use the Thea library, which exposes OWL axioms directly as Prolog predicates. Thea is being extended to handle OWL2[22].

In addition, Blipkit provides a parser for the Common Logic Interchange Format (CLIF), a lisp-like syntax that is used for encoding any collection of first order logic sentences.

6 Presentation layers

6.1 Visualization of graphs

Network-style graph views are common for visualizing complex biological data, such as interaction networks and ontological classification of genes. Blipkit provides two means of visualizing such data, the first through GraphViz, and the second through XPCE.

GraphViz takes an abstract specification of a graph and uses a layout algorithm to render it such that the nodes are placed as optimally as possible, minimizing edge-crossings. The specification is in the form of the dot language, a domain specific language for graphs. Blipkit includes a dot grammar, for generating dot files, given a prolog representation of a graph. In addition, graphs can be configured using simple configuration predicates, for example, edges of certain types can be colored or even nested.

One limitation of GraphViz is that the representation is static, the graph cannot be manipulated, unless the dot file is imported via another program. The alternate means of visualization graphs is via a bridge to XPCE, which allows nodes to be dragged, and allows the graphs to be embedded in larger XPCE applications. The main limitation here is that the XPCE canvas does not have a layout algorithm, placement of nodes is arbitrary, leading to unsightly edge crossings. One possible solution would be to use dot to perform the layout, and then use this to guide placement within XPCE.

6.2 The Serval web application framework

An increasing number of applications use the web browser as a platform. For prolog applications, this has the advantage that the end user does not need to install prolog on their machine.

SWI-Prolog includes the powerful **http** package which greatly simplifies the task of writing web applications. However, this package does not include any specific capabilities for translating HTML. The general paradigm is to generate XHTML terms using a DCG, which introduces extra syntax that can obscure the relationship between the code and output.

The paradigm used by most web application frameworks, such as the popular Ruby on Rails is to use a template language. This introduces an additional language, with possible language mismatch problems. Blipkit takes a different approach with the bundled Serval module, which uses a scheme-like functional-style language with prolog syntax to specify dynamic HTML.

To give a trivial example, the following piece of code will generate a nested HTML list (built with **ul** and **li** HTML elements) for a phylogenetic tree in the **phylo_db** model, by recursively traversing down the tree.

```
phylohtml(Node) =>
  if(phylonode_leaf(Node),
      then: b(Node),
      else:
       ul(li( phylohtml(Child) ) forall phylonode_parent(Child, Node))).
```

The serval framework allows for the specification of state machine like rules for determining how transitions are made from one web page to another. The above code can be extended to allow for a dynamic tree in which nodes can be expanded and collapsed by the user.

A number of Blipkit applications are available as web applications and web services. One such application is an ontology browser and visualization tool[2]

7 Composing models and complex queries

Whilst each Blipkit package is useful as a standalone piece of software, the real power comes from composing queries across multiple models and data sources. For example, using the **genome_db** model and the bridge to the Ensembl database we can query for all upstream regulatory regions for a given gene, or discoverer which genes make which proteins. Using the **ontol_db** module and an ontology of chemical entities we can find all the different kinds of neurotransmitter. Using both Gene Ontology annotations and pathway databases we can find all proteins that produces or transport those neurotransmitters. We can combine all these as follows:

```
entity_label(NT,neurotransmitter),
subclassT(Entity,NT),
(produces(Protein,Entity) ; transports(Protein,Entity)),
encodes(Gene,Protein),
regulates(Reg,Gene),
upstream_of(Reg,Gene)
```

[2] http://berkeleybop.org/obo/

If blipkit is configured such that the relevant predicates query the correct sources, then this will find the regulatory regions for genes whose expression determines the identity of different kinds of neurotransmitter secreting cells.

Some of the subgoals will query the in-memory prolog database using intensional predicates; others will be translated to relational queries. Unlike a relational query, the ordering of subgoals is important, as they are executed sequentially.

One possible future extension would be perform query optimization and reorder subgoals within a goal.

8 Discussion

8.1 Comparison with other Bioinformatics Toolkits

The Open Bioinformatic Foundation (OBF) is a non-profit organization that acts as an umbrella for the open source Bio* projects. The Bio* projects include BioPerl, BioJava, BioPython, BioRuby and BioSQL, each representing a community effort to provide a relatively comprehensive library of code for researchers in the life sciences that takes advantages of the features of the host programming language. The longest running and perhaps most comprehensive of these projects is BioPerl[18], as perl is a popular language due to it's string matching and manipulation capabilities. Blipkit, being part of the OBF, an alias for BioProlog. It thus makes sense to compare Blipkit/BioProlog with some of the other Bio* projects such as BioPerl.

The organization and architecture of Blipkit is similar to, and influenced by BioPerl. BioPerl is divided into sub-modules each dealing with a separate subdomain of biology. These are then further divided into object-oriented modeling classes, and additional classes for parsing and writing a variety of data formats. The libraries are intended to be used in similar ways, as tools for investigators to ask questions of complex data, and also as core components within larger infrastructures.

Of course, BioPerl is far more comprehensive due to a large critical mass of developers and contributors, especially in the domain of genomics. However, Blipkit offers many capabilities absent from BioPerl, such as modules for systems biology data. For the set of capabilities in the intersection between the two systems, the use of Prolog can offer a more concise declarative way to perform operations. In many cases, prolog can offer a faster execution time, which can be surprising to some people given the reputation of Prolog as an academic language.

8.2 Comparison with Semantic Web Technology

An alternative approach to semantic data integration is to use semantic web technology to query across multiple databases. Databases can be mapped to RDF triples, either dynamically, using a system such as D2RQ, or statically, by

building a native RDF triplestore. However, RDF on its own does not provide any of the "semantics" in the semantic web. For that there has to be some kind of inference, usually driven by ontologies and ontological formalisms such as RDFS or OWL.

RDFS on its own does not provide enough semantics for complex biological data. OWL is more expressive, but most OWL reasoners do not scale to large databases. Another issue is that the expressivity afforded by OWL and Description Logics in general may not be quite right for all bioinformatics applications.

For example, Description Logics are unable to represent accurately represent cyclic classes of structure, such as carbon rings, regulatory networks or RNA structures. For example, the following logic programming predicate can classify RNA tetraloops - chains of 4 bases, each end of which is connected to bases that are themselves paired.

```
tetraloop([B,C,D,E]) :-
  chain([A,B,C,D,E,F]),
  paired_with(A,F).
```

The equivalent is not possible as an OWL class expression. One possibility is to go beyond OWL and use the Semantic Web Rules Language (SWRL), which is comparable in expressive power to Datalog. One crucial difference is that all semantic web formalisms make the open world assumption, whereas the Datalog family of languages from the relational model up through the pure subset of prolog up to answer set programming makes the closed world assumption. The open world assumption can be useful when making inferences over the web, but it can prove to be a hindrance with bioinformatics since it can often be assumed that certain classes of data are complete.

8.3 Blipkit Issues and Limitations

The main limitation of Blipkit is the lack of expressivity that comes from the lack of predicate tabling. However, for a great many uses there are workarounds, so in practice this often does not prove to be an unsurmountable problem. However, some of these workarounds are inelegant, tabling would result in a smaller codebase and cleaner, more declarative code.

One of the problems with blipkit is that it is too big. Ideally it would consist entirely of biolog-specific code, as it is there are lots of general purpose modules that belong either in independent projects or even in some kind of prolog common library. This applies especially to the **sql_compiler** code.

8.4 Expressivity and suitability of Prolog

For the majority of the tasks within the scope of Blipkit, the expressivity provided by Prolog is mostly sufficient. The declarative nature of the language makes it well-suited to answering complex queries. In addition the programmability of Prolog means that fully fledged applications can be constructed entirely

in prolog, avoiding any impedance mismatches that often result from connecting together different technologies.

As mentioned, WAM type resolution can often be a limitation. In addition there are cases where it is useful to go beyond the expressivity of Prolog, for example, disjunctive datalog or nonmonotonic reasoning, as found in systems like DLV. It would be useful to have a more seamless way of reusing portions of the same programs across logic programming systems of varying expressivity. Currently this is impeded by a lack of standards in these areas.

Currently the types of inference offered by Blipkit are purely logical. Often the final step of semantic data integration is some kind of statistical or probabilistic modeling. In the future a number of options will be explored. One is tighter integration with libraries developed using the statistical programming language R, such as BioConductor. Another is the use of formalisms that allow a more seamless integration of logical and statistical programming. This includes bayesian CLP, inductive logic programming and bayesian probabilistic logical modeling, as provided in the PRISM system.

8.5 Portability

Different prolog implementations offer different advantages. Ideally the same library code could be reused across all implementations. Unfortunately Blipkit is tied to one Prolog system, SWI-Prolog.

Historically, the initial implementation of Blipkit was in XSB Prolog. XSB offers powerful deductive database capabilities, including tabling. The decision was made to switch to SWI-Prolog due to a number of factors such as the more extensive collection of libraries and development tools, an active mailing list, frequent releases and ease of installation on a number of different operating systems. The original goal was to maintain a compatibility layer, but this fell by the wayside, and the system quickly become SWI-specific.

SWI-Prolog proved to be an excellent environment, but the lack of tabling proved to be a hindrance. Certain parts of the code had to be rewritten to avoid cycles in the deduction chain, which resulted in less compact code. Some functionality is simply unavailable. Blipkit does include a module for performing memoization on calls to goals (misleadingly called the **tabling** module, in fact much less powerful than full tabling). This can increase efficiency in certain contexts, but predicates must still be written to avoid cycles. Blipkit also has a module for forward chaining, performing the full deductive closure, but this can only be used in certain contexts, and can entail a large upfront time-penalty.

Recent efforts to improve interoperability between Yap and SWI are extremely encouraging. Hopefully it will soon be possible to use Blipkit from within Yap, which will open the possibility of using tabling, available in Yap.

One of the main sources of incompatibility between prolog systems is difference in module systems. One possibility is to refactor Blipkit to use LogTalk as a means of providing encapsulation between different modules.

8.6 Large datasets and relational databases

Blipkit is in some sense a large modular multi-purpose deductive database schema. The core is largely Datalog, with additional utility layers in Prolog. Prolog and Relational Database Management Systems (RDBMSs) have many similarities, but this similarity is often under-exploited.

To say that RDBMSs are more prevalent would be a massive understatement. Yet RDBMs could benefit from the addition of logic programming features. The SQL99 standard allows for a very limited kind of deductive predicate, and this remains unimplemented in the two major open source RDBMSs. Programming with RDBMSs is frequently difficult: procedural SQL is unwieldy, and use of object oriented languages introduces an impedance mismatch. RDBMSs deserve a relational language, along the lines of Prolog. Prolog can also be a more "Agile" alternative to full fledged database systems – a prolog database can be located anywhere on the filesystem. In this respect Prolog shares some features with lightweight databases such as SQLLite.

On the other hand, prolog systems could benefit from having additional optional RDBMS like features. The lack of schemas or typing in Prolog is often a boon, but sometimes it would be useful to have optional metamodeling facilities. The programmer is of course free to roll their own (as in for example the Blipkit **dbmeta** metamodel) but it would be better to have this as a standard library.

Better ways of making seamless transitions between the two would also be welcome.

9 Conclusions

Integration of data across multiple databases remains a difficult problem, limiting the number of and scope of questions that researchers can ask. Despite having a critical mass of developers, the Blipkit library has slowly evolved to have a core set of models and utility modules that make it useful for a wide variety of powerful queries and data extraction operations. In particular, the ability to combine inference rules, relational databases and ontologies with a collection of pre-written mappings make it especially powerful.

Blipkit would not have been possible without the SWI-Prolog system. In particular, the comprehensive set of libraries offered by SWI-Prolog make it simple to develop full web applications and web services using Blipkit. The extension of Blipkit may require a hybrid solution, using different Prolog implementations and logic programming systems in general depending on the particular task at hand. The major barrier here is the lack of interoperability between logic programming systems.

Acknowledgements

Thanks to Stephen W Veitch for the development of the NCL library, Vangelis Vassiliadis for assistance with the Thea OWL library, and Jan Wielemaker for SWI-Prolog. Also many thanks to Nicola Leone and the DLV teams for assistance with the DLV system.

References

1. Alexander V Alekseyenko and Christopher J Lee. Nested containment list (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases. *Bioinformatics*, page btl647, 2007.
2. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.
3. M. Ashburner, C. A. Ball, JA Blake, H Butler, JM Cherry, J Corradi, K Dolinski, JT Eppig, M Harris, DP Hill, S Lewis, B Marshall, C Mungall, L Reiser, S Rhee, JE Richardson, J Richter, M Ringwald, GM Rubin, G Sherlock, and J Yoon. Creating the gene ontology resource: design and implementation. *Genome Res*, 11(8):1425–1433, Aug 2001.
4. Colin Batchelor, Thomas Bittner, Karen Eilbeck, Chris Mungall, Jane Richardson, Rob Knight, Jesse Stombaugh, Craig Zirbel, Eric Westhof, and Neocles Leontis. The rna ontology (rnao): An ontology for integrating rna sequence and structure data. In *Proceeeings of the First International Conference on Biomedical Ontology*, 2009.
5. E. Birney, T. D. Andrews, P. Bevan, M. Caccamo, Y. Chen, L. Clarke, G. Coates, J. Cuff, V. Curwen, T. Cutts, T. Down, E. Eyras, X. M. Fernandez-Suarez, P. Gane, B. Gibbins, J. Gilbert, M. Hammond, H. R. Hotz, V. Iyer, K. Jekosch, A. Kahari, A. Kasprzyk, D. Keefe, S. Keenan, H. Lehvaslaiho, G. McVicker, C. Melsopp, P. Meidl, E. Mongin, R. Pettett, S. Potter, G. Proctor, M. Rae, S. Searle, G. Slater, D. Smedley, J. Smith, W. Spooner, A. Stabenau, J. Stalker, R. Storey, A. Ureta-Vidal, K. C. Woodwark, G. Cameron, R. Durbin, A. Cox, T. Hubbard, and M. Clamp. An overview of ensembl. *Genome Res*, 14(5):925–8, 2004. 1088-9051 Journal Article Review Review, Tutorial.
6. Bobby-Joe Breitkreutz, Chris Stark, Teresa Reguly, Lorrie Boucher, Ashton Breitkreutz, Michael Livstone, Rose Oughtred, Daniel H. Lackner, Jrg Bhler, Valerie Wood, Kara Dolinski, and Mike Tyers. The biogrid interaction database: 2008 update. *Nucleic Acids Research*, 36:D637D640, 2008. PMC2238873.
7. W. R. Cook and A. H. Ibrahim. Integrating programming languages and databases: What is the problem. *ODBMS. ORG, Expert Article*, 2006.
8. C. Draxler. *Accessing Relational and Higher Databases Through Database Set Predicates*. PhD thesis, PhD thesis, Zurich University, 1991, 1991.
9. K. Eilbeck, S. E. Lewis, C. J. Mungall, M. D. Yandell, L. D. Stein, R. Durbin, and M. Ashburner. The sequence ontology: a tool for the unification of genome annotations. *Genome Biology*, 6(5), 2005.
10. Karen Eilbeck and Christopher Mungall. Evolution of the sequence ontology terms and relationships. In *Proceedings of the First International Conference on Biomedical Ontology*, 2009.
11. Paula M Mabee, Michael Ashburner, Quentin Cronk, Georgios V Gkoutos, Melissa Haendel, Erik Segerdell, Chris Mungall, and Monte Westerfield. Phenotype ontologies: the bridge between genomics and evolution. *Trends Ecol Evol*, Apr 2007.
12. Chris Mungall, Michael Bada, Tanya Berardini, Jennifer Deegan, Amelia Ireland, Midori Harris, David Hill, and Jane Lomax. Cross-product extensions of the gene ontology. In *Proceedings of the First International Conference on Biomedical Ontology*, 2009.
13. Christopher J. Mungall. Obol: Integrating language and meaning in bio-ontologies. *Comparative and Functional Genomics*, 5(7):509–520, 2004.

14. Christopher J. Mungall, David B. Emmert, and The FlyBase Consortium. A chado case study: an ontology-based modular schema for representing genome-associated biological information. *Bioinformatics*, 23(13):i337–346, 2007.
15. E. Sayers, D. Wheeler, and National Center for Biotechnology Information (US. *Building customized data pipelines using the entrez programming utilities (eutils)*. NCBI, 2004.
16. B. Smith, W. Ceusters, J. Kohler, A. Kumar, J. Lomax, C.J. Mungall, F. Neuhaus, A. Rector, and C. Rosse. Relations in biomedical ontologies. *Genome Biology*, 6(5), 2005.
17. Barry Smith, Michael Ashburner, Cornelius Rosse, Jonathan Bard, William Bug, Werner Ceusters, Louis J Goldberg, Karen Eilbeck, Amelia Ireland, Christopher J Mungall, The OBI Consortium, Neocles Leontis, Philippe Rocca-Serra, Alan Ruttenberg, Susanna-Assunta Sansone, Richard H Scheuermann, Nigam Shah, Patricia L Whetzel, and Suzanna Lewis. The obo foundry: coordinated evolution of ontologies to support biomedical data integration. *Nat Biotechnol*, 25(11):1251–1255, Nov 2007.
18. J. E. Stajich, D. Block, K. Boulez, S. E. Brenner, S. A. Chervitz, C. Dagdigian, G. Fuellen, J. G. Gilbert, I. Korf, H. Lapp, H. Lehvaslaiho, C. Matsalla, C. J. Mungall, B. I. Osborne, M. R. Pocock, P. Schattner, M. Senger, L. D. Stein, E. Stupka, M. D. Wilkinson, and E. Birney. The bioperl toolkit: Perl modules for the life sciences. *Genome Res*, 12(10):1611–8, 2002. 1088-9051 Journal Article.
19. L. Stein. How perl saved the human genome project. *The Perl Journal*, 1(0001), 1996.
20. L. Stein. Creating a bioinformatics nation. *Nature*, 417(6885):119–120, 2002.
21. L. D. Stein. Integrating biological databases. *Nature Reviews Genetics*, 4(5):337–345, 2003.
22. V. Vassiliadis and C.J. Mungall. Logic programming with owl2 using the thea prolog library. *In preparation*, 2009.
23. J. Wielemaker. An overview of the SWI-Prolog programming environment. In *13th International Workshop on Logic Programming Environments*, pages 1–16, 2003.
24. J. Wielemaker and A. Anjewierden. PlDoc: wiki style literate programming for prolog. In *Proceedings of the 17th Workshop on Logic-Based methods in Programming Environments*, page 1630, 2007.
25. J. Wielemaker, G. Schreiber, and B. Wielinga. Prolog-based infrastructure for RDF: scalability and performance. *Lecture notes in computer science*, pages 644–658, 2003.