

A language for querying trees

Ed Avis, ed@membled.com

November 5, 2002

Chapter 1

Abstract

Semistructured data is a mix of different data types, following some known pattern but not fully restricted to a fixed schema. For example the project guidelines for this report suggest section headings but allow some variation. We explain the data model of unordered trees, and ‘graphical’ links within trees to express relationships which can’t fit into a pure tree model. Associated with this data model is a logic specifying properties of trees. By asking which valuations for free variables cause a formula to match, we can query a tree and use the values returned to build a new tree.

SSD is interesting because many popular data models like XML, X.500 directories and even the Web itself are semistructured. We compare our data model with XML, and the logic-based query language with some XML query tools. The biggest difference is between ordered and unordered trees.

This project implements the query language for unordered trees with graphical links, building on earlier work implementing querying for trees without links. We present the data model and logic, with examples of why each new feature is useful. There are design choices in the definition of some parts of the logic and query language which we try to explore.

Chapter 2

Acknowledgements

My supervisor, Philippa Gardner, of course. The implementer of the TQL query tool, Giorgio Ghelli, explained its workings to me, and without the explanation I would never have been able to fully implement this project. Thanks to all those who helped with L^AT_EX arcana, especially kao98 for his bibliography style file.

Contents

1	Abstract	1
2	Acknowledgements	2
3	Introduction	6
3.1	Data model	6
3.2	Logic	8
3.3	Query language	8
4	Background	9
4.1	Motivation for semistructured data	10
4.2	Different semistructured data models	11
4.2.1	XML	11
4.2.2	X.500 directories	14
4.2.3	Filesystems	15
4.2.4	Ambients and info-terms	16
4.2.5	The graph data model	17
4.2.6	General trees with dangling pointers	17
5	Data model	18
5.1	Unordered trees	18
5.2	The need for graphical links	19
5.3	Representing this data on a computer	21
6	Logic	23
6.1	Unordered labelled trees, no recursion	23
6.1.1	Examples	24
6.2	Existential quantification	26
6.3	Adding recursion	27
6.3.1	Using \exists together with recursion	28
6.3.2	The \diamond operator	29
6.4	Equality tests	29
6.5	Extending the logic to graphical links	30
6.5.1	Example	30
6.6	Tree variables	31
6.7	Comparing variables of different kinds	32
6.8	Conclusion	32
7	Implementation of the logic	34

7.1	First attempt: mapping variables to sets of values	35
7.1.1	Representing a cofinite set of values	35
7.1.2	Mapping from variables to ValuePs	36
7.1.3	Unification	37
7.1.4	Inadequacy of the first attempt	38
7.2	Second attempt: a disjunction of SubstPs	38
7.2.1	Negation	39
7.2.2	Existential quantification	43
7.2.3	Summary	43
7.3	Final version: store (in)equality info too	43
7.3.1	Absorbing (in)equality info into sets of values	44
7.3.2	Storing a list of variable relationships	45
7.3.3	Representing a set of substitutions	47
7.3.4	Unification	47
7.3.5	Negation	50
7.3.6	Summary	50
7.4	Technical notes on the implementation	51
8	Query language	53
8.1	A simple query language	54
8.2	General <i>from/select</i> querying	56
8.3	Adding addresses and links	57
8.3.1	Copying subtrees containing links	59
8.4	Implementation of the query language	60
9	Summary of the implementation work	62
9.0.1	Haskell version	62
9.0.2	Small Perl version	62
10	Evaluation and future work	64
10.1	Efficiency of operator	64
10.2	Comparison with TQL	66
10.2.1	Path expressions	66
10.2.2	Data model differences	67
10.2.3	Fully general equality tests	67
10.2.4	Front end	68
10.2.5	Report	68
10.3	Selecting subtrees in a query	68
10.4	Comparison with XML	71
10.4.1	Ordering	71
10.4.2	Pointers	72
10.5	Alternate meaning of graphical links	72
10.6	Different data models	73
10.6.1	Converting the tree and graph logics into this form	75
10.6.2	Variable matching	76
10.6.3	Implementation	76
10.7	Alternate meanings of negation	77
10.7.1	Incompleteness of negation by failure	79
10.7.2	Implementation	79
10.8	Multiplicity	79

10.8.1 Conclusions	83
10.9 Alternate semantics for recursion	84
10.10Static checking of queries	85
10.11Allowing query results to be infinite	86
11 Conclusions	88
A Summary of the logic	90
B Testing	92
C Getting the code	94

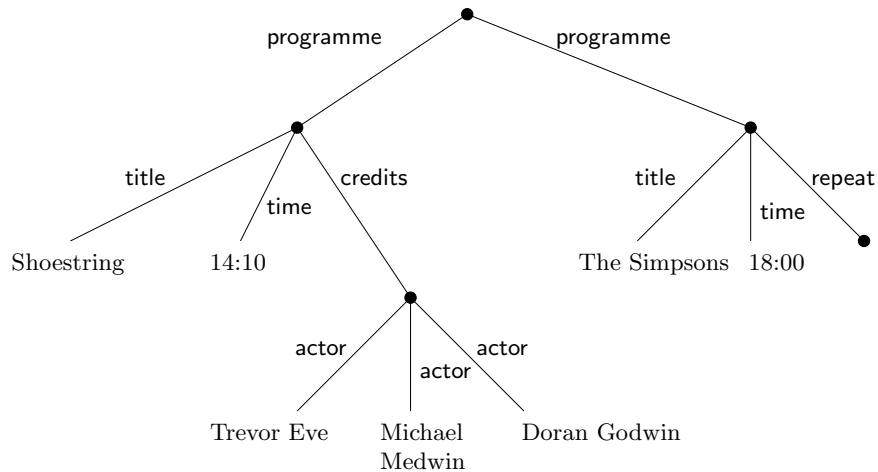
Chapter 3

Introduction

To represent semistructured data we adopt a data model of unordered trees, with ‘graphical links’ to represent relationships between trees. A logic, similar to the ambient logic, makes statements about these trees. Then a query language based on the logic allows us to query trees and build new trees as a result. In this section we give an overview of the data model and logic.

3.1 Data model

We are dealing with unordered trees with labelled branches and atomic data appearing where a branch can appear. The techniques for storing relational data are well-known, but for semistructured tree data things are still evolving and there are several different choices to make such as whether to have ordered trees or allow data only at the leaves. But this project is to implement querying for one particular data model. Here’s an example tree showing television listings, and its syntactic representation.



$\text{programme} \mapsto [\text{title} \mapsto [\text{Shoestring}] \mid \text{time} \mapsto [14:10]$

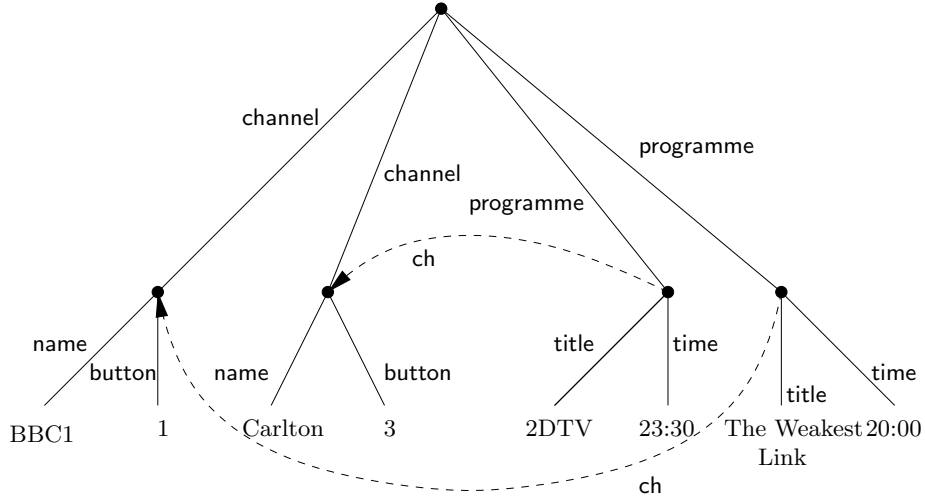
```

| credits ↦ [ actor ↦ [Trevor Eve] | actor ↦ [Michael Medwin]
| actor ↦ [Doran Godwin] ] ]
| programme ↦ [ title ↦ [The Simpsons] | time ↦ [18:00] | repeat ↦ [nil] ]

```

But what about cases where a simple tree approach is not sufficient? An example could be introducing channel information to the TV listings, so that each programme is associated with one channel, which has its own set of information like name and frequency. We don't want to include each channel as a subtree of each programme, because that would duplicate the information lots of times and make it hard to update. We want to link back up the tree to some channel information which is stored once.

The special feature of our data model compared to that used in [1] or [2] is that it allows links (also called graphical links or pointers) from one part of the tree to another. Each subtree can be given a unique address and as well as labelled arcs and atomic data, we also allow labelled links in the tree. If we wanted to have programmes referring to channel information, we might have:



```

channel ↦ x0 [ name ↦ [BBC1] | button ↦ [1] ]
| channel ↦ x1 [ name ↦ [Carlton] | button ↦ [3] ]
| programme ↦ [ title ↦ [2DTV] | time ↦ [23:30] | ch@x1 ]
| programme ↦ [ title ↦ [The Weakest Link] | time ↦ [20:00] | ch@x0 ]

```

In fact every subtree must have a unique address, though I have not written addresses other than $x0$ and $x1$ since they are not used. This means that every tree is unique so subtrees must be a set not a multiset. However there is no restriction on how many times an address may be pointed to by a graphical link, and it is possible to have 'dangling' links which point to an address not used elsewhere. You can think of the addresses like $x0$ as URLs, and the labels as the human-readable text associated with a link.

3.2 Logic

There is a logic to make statements about these trees, for example:

```
programme  $\mapsto$  [ title  $\mapsto$  [The Simpsons] | time  $\mapsto$  [18:00] | true ] | true
```

means ‘there is an episode of the Simpsons on at 18:00’, and

```
channel  $\mapsto$  x[ name  $\mapsto$  [BBC1] | true ]  
 $\wedge \neg$  ( programme  $\mapsto$  [ channel@x | time  $\mapsto$  [18:15] | true ] | true )
```

means ‘a channel BBC1 exists which never shows anything at 18:15’, using the variable **x**.

The logic is made from structural matching operators which mirror the structure of trees, variables, standard logical connectives and existential quantification. There is also a ‘somewhere’ modal operator, useful for searching a tree to arbitrary depth, and optionally some fixed-point operators to construct recursive formulæ. It is similar to modal logics used for other unordered-tree representations, such as [1] and [3].

The \mapsto structural connective in the logic matches the same connective in trees, as does |. But because | in trees is associative and commutative, there are many different ways to rearrange a tree for matching the | logical operator. $\varphi | \psi$ is true if there exists a way to split the tree such that one side satisfies φ and the other ψ .

3.3 Query language

It is possible to check whether a tree satisfies a formula (i.e., the tree is a model of that formula). And for a formula with unbound variables we can find all possible assignments of those variables that make the tree satisfy the formula. For example, taking the formula that says there is a Simpsons at 18:00 and replacing 18:00 with a tree variable **t**, we can find all times **t** when the programme is on. Using this set of substitutions the query language can construct a result:

```
from  $tv \models$  programme  $\mapsto$  [ title  $\mapsto$  [The Simpsons] | time  $\mapsto$  x[t] | true ] | true  
select time  $\mapsto$  x[t]
```

If more than one assignment for **x** and **t** is possible, multiple time \mapsto **x**[**t**] results are constructed and glued together with the | connective.

Chapter 4

Background

There are two main types of data storage, fully-structured and semistructured. A fully-structured system specifies in advance the schema of all data to be stored. An SQL relational database is one example of a fully-structured system; before adding rows to the database you must first create a table and give the data type associated with each column. It is not possible to insert a row which contains some extra data not mentioned in the table definition; at best you will have to modify the table to add a new column, at worst drop the table and recreate it with a different schema. (A schema is a set of rules that describes the allowable shape of a data structure. In the case of SQL it is the table definitions and integrity constraints. There are other database systems which are not relational, but still have a rigid schema.)

You can also think of fully-structured data as being like data structures in a strongly-typed programming language. In C, a `struct` gives a data type and name to all its fields, which lets the layout of the structure be determined at compile time. It is not possible at run time to add new fields to an instance of the structure, nor (without dangerous casting) to make an instance whose fields have a different type to that specified.

Now data that is completely unstructured would not be much use for automatic processing. There needs to be some pattern and some regularity so that computer programs (and humans too, for that matter) can traverse the data structures and find information of interest.

But for many applications having to specify the whole structure in advance is too limiting. For example you could create a data format for images that holds the image data, the author's name and the date of creation. Because these are the only fields possible, they can be stored at certain positions within the file. Suppose that the file format becomes widely adopted—but later someone realizes there ought to be provision for giving each image a title. There is no way to add this information to the format without breaking some existing programs which expect the previous fixed layout.

Suppose instead that the image format had been specified as a list of 'chunks' each with a name. A program reading the images would take note of some of

the chunks, while skipping those whose names it did not recognize. That would allow later additions to the format. But while new data can be added, there would still be some chunks (such as the image data itself) which are mandatory. The file format is open-ended, but still parts of it are forced to follow a given structure. (The PNG[4] image format follows a scheme similar to this, where the name of each chunk indicates whether it is essential to the display of the image, or can be skipped.)

The hypothetical image format is a very simple example of semistructured data. But in this project we are concerned with more general formalisms for SSD. Rather than ad hoc systems to introduce some flexibility into a particular application, is there a way to generalize the idea?

Most computer data is represented as a tree in some way or another. All widely-used imperative languages allow building data structures which contain other structures, or pointers to other structures. File storage on all popular operating systems is represented as a ‘directory tree’. Most models for semistructured data can also be thought of as trees. We give concrete examples in the next section.

This project uses one particular model for semistructured data, that of unordered trees with graphical links. By adding new branches to a tree it is possible to add new information to a tree or to include optional information. The example data in the introduction illustrated this.

4.1 Motivation for semistructured data

The main advantages of giving all schema information up-front are speed and the possibility of compile-time checking. A typical C program is certainly faster than the equivalent program written in a loosely-typed language such as Perl or many Lisp variants. Because the layout of a C data structure is known at compile time, different parts can be accessed at fixed offsets in memory—but in a language which allows new fields to be created on the fly, it’s usually necessary to look up strings to find whether a structure has a given field. And SQL statements can be typechecked to see if they are consistent with the database’s table definitions, which catches a large number of errors and not just typos. If it were possible to put data of mixed types into a single column, the database user would have more flexibility, but programming errors from giving the wrong column name could not be detected.

On the other hand, if the schema information is not fixed but extensible, it is often easier to make changes to the data format. This is particularly important when a large number of different groups are using the format, since you cannot predict in advance all the possible extensions which will be wanted. A semistructured format allows each group to add its own extensions without becoming incompatible with older software; the older software can simply skip over parts it does not understand. The example of the image format illustrated how this can be essential. A similar process happened with HTML where different vendors added their own extensions, while older web browsers were mostly still able to read the ‘enhanced’ web pages. It’s a matter of debate whether this was a Good

Thing, but at least the extended formats did not become completely unreadable by older software. You could make the argument that semistructured data formats are inherently more suitable for Internet use, since they allow variations to be made to the format without needing permission from a central authority, yet the variant formats need not be mutually incompatible.

Processing semistructured data is always going to be slower than a fixed, binary format, but perhaps not that much slower. As computers have become faster we've seen a move away from formats that suit the computer towards formats that suit the user. Already there is widespread adoption of XML even for applications where extensibility of the format is not necessary. Semistructured data is already widespread and its use is growing.

4.2 Different semistructured data models

Now we give some examples of different models for semistructured data. These are all trees, although they make different choices about whether branches are labelled, whether ordering is significant, and so on.

4.2.1 XML

XML[5] grew out of the SGML[6] markup language and defines a tree structure of 'elements' each of which can contain other elements, a list of attributes, and textual data. XML is really a mixture of several different models because it combines ordered and unordered information, and it can be used either as text-with-markup (as in HTML) or as a pure tree structure.

We assume some familiarity with XML and do not give a full description here, just noting a few important points. Firstly, that the ordering of subelements is important, so the following two documents are distinct:

```
<place>
  <name>Elbonia</name>
  <exists>no</exists>
</place>
<place>
  <exists>no</exists>
  <name>Elbonia</name>
</place>
```

However the ordering of attributes is not supposed to be important in XML, so these two documents are equivalent:

```
<a attr0="hello"
  attr1="there" />
<a attr1="there"
  attr0="hello" />
```

Thus XML mixes ordered and unordered data. But since the values of attributes can only be strings, whereas subelements are complete trees in their own right and normally used a lot more, we should consider XML an ordered-tree data model. Also note that multiplicity matters, for example these two are different:

```

<a>
  <b />
</a>

```

```

<a>
  <b />
  <b />
</a>

```

It is possible to distinguish the two `b` subelements and to refer to just one of them. There is no ambiguity because each subelement has an implicit position; instead of saying ‘the `b` element inside the `a` element’, you talk of the *first* or *second* `b` element. It is not just the name but the position which identifies a subelement within its parent. We’ll come to this again when discussing how to translate XML to our data model.

Defining the schema of XML files

Several schema proposals exist to define the allowable shape of an XML document. The oldest and most widespread is the Document Type Definition or DTD, defined as part of the original XML specification [5], which is a simple context-free grammar to say which elements may appear inside which other elements, and what attributes they may have. For example,

```

<!ELEMENT person (name, age?, hobby*)>
<!ATTLIST person date-entered CDATA #IMPLIED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT hobby (#PCDATA)>
<!ATTLIST hobby competence (beginner | expert) #IMPLIED>

```

states that `person` must contain one `name` element, at most one `age` element, and any number of `hobby` elements—and the three types of element must appear in that order. The three subelements of `person` in turn contain text (‘parsed character data’ in XMLspeak). As well as defining the allowable tree structure of the elements, this DTD specifies that `person` has an optional `date-entered` attribute which is text, and `hobby` has an optional `competence` attribute which can take the values ‘beginner’ or ‘expert’.

There are more things that can be specified with DTDs but there isn’t room for a complete listing here. One thing to note is that a DTD can completely constrain the order of subelements, so that the ordering in a given file no longer carries any meaning. In the above example the `name` *must* always appear before any `age` or `hobby`, so its relative ordering in the file read can be ignored.

Defining a DTD for an application of XML amounts to taking a semi-structured data format and constraining it so that it becomes closer to fully-structured. If you require documents to conform to a given DTD, then there is no provision for adding new elements; any new element would make the document invalid with respect to the DTD. However, most XML parsers do not validate, so often it is possible to extend the file format and retain compatibility despite the DTD disagreeing—this is what happened with proprietary HTML extensions, for example.

But if you know that all documents will conform to the DTD, then it is easier to do the parsing; for example FlexXML[7] can read a DTD and generate C parser code. The code is much more efficient than using a general-purpose XML-reading API like SAX[8]; but it will fail if it encounters any deviation from what the DTD specifies. This backs up the point that fully-structured data is faster for the machine to handle, but makes it more difficult to make changes to the format.

Links

One more feature of XML should be mentioned: the possibility to have references within the document. Syntactically, this is nothing very special: you just give one element an attribute specifying its ‘name’, and then from another element a different attribute can refer to that name. For example,

```
<a>
  <b id="42">This is something you might want to refer to.</b>
  <c likes="42">I refer to the element with id '42'.</c>
</a>
```

With most XML reading interfaces, it’s up to the application to keep track of the attributes and decide what meaning such a ‘reference’ has—although it would certainly be possible to build a library which automatically follows these links and returns a data structure with pointers to the linked-to elements. The fun part, if you can call it that, comes when defining the DTD:

```
<!ELEMENT a (b, c)>
<!ELEMENT b (#PCDATA)>
<!ATTLIST b id ID #REQUIRED>
<!ELEMENT c (#PCDATA)>
<!ATTLIST c likes IDREF #REQUIRED>
```

This enforces that the `likes` attribute must ‘point’ to a name defined elsewhere in the file. A validating parser or standalone validator such as `nsgmls`[9] will check this. However, it is not possible to specify what type of element `likes` refers to, if there is more than one element type with attributes of type `ID`.

It is not possible to make this kind of link from one document to another. The `IDREF` must point within the same file. This means it is not always possible to split an XML document into several smaller documents: XML is not quite a tree composed of independent subtrees.

Other proposals for describing the structure of XML have not been as widely adopted as DTDs. Two of the most prominent candidates are XML Schema[10] and Document Structure Description[11] (DSD).

Both of these allow finer control than DTD of the allowed textual content of elements and attributes; with a DTD you can say that an attribute must have one of a list of values but you cannot specify that it must be ‘a positive integer’, for example. A notable feature of DSD is that it isn’t context-free; the allowable

content of an element can be influenced by its ancestor elements. It also allows restriction of what elements can be pointed to by IDREFs.

The XDuce XML processing library[12] is also interesting because it answers some of the problems discussed above. It provides ‘regular expression types’ as an alternative to DTD—and a subtyping system to make the schema extensible, which solves the problem that extensions to the format make documents appear invalid. It provides a typesafe interface for reading XML documents (mapping document elements to and from programming language types) as FlexXML does, but the file reading will not give an error every time it encounters an extension to the known format. Most importantly, the language for specifying the shape of XML documents is a variant of the logic we use!

Then there are equally many query languages for XML, which search an XML document and extract information to build a new XML document, with names such as XQL, XML-QL and UnQL. These have differing degrees of theoretical ‘rigour’ but all can be thought of, in some sense, as matching a template against a tree and then instantiating a result template with the values found. An overview of the different XML schema languages and query languages is given in [13].

4.2.2 X.500 directories

X.500[14] is a standard for building a ‘directory’, a distributed hierarchical database most commonly used for storing personnel and organizational information (hence the name). (The LDAP[15] protocol is a way of accessing directories which shares the X.500 data model.) The data model is a tree of ‘directory entries’, each entry having some ‘attributes’. An entry may have zero or more children; there is a single root entry representing The World. Attributes have a name and a value, except that some attributes may be multivalued.

We are not concerned here with how this data is stored on multiple computers—in practice there is no computer holding the root entry. We just want to show the concepts.

Unlike XML or most other semistructured data models, there is no branch label from a parent to its child. Instead, you refer to the children of an entry by giving some of their attributes: the Relative Distinguished Name or RDN. For example, the RDN of a Person entry within a Department entity might consist of the two attributes GivenName and Surname. There is no ordering among the different child entries; only the RDN or other attributes can be used to identify them. A chain of RDNs leading from the root to an entry is called a Distinguished Name or DN.

Note the difference between identifying a subtree by its branch label, and identifying an X.500 entry by some of its attributes. If the attributes of an entry change then the DN will also change; whereas changing some of the data in a subtree will not affect its address or position relative to the root of the tree.

We won’t cover X.500 directories any further in this report; they’re just mentioned as an example and to show that unordered-tree data are used for some applications. The best reference to learn more is [16].

Our data model, which we will introduce later, uses unordered labelled trees. So in a sense it is halfway between X.500 and XML—it has labels leading to subtrees like XML does, but those subtrees are unordered like child entries in an X.500 directory.

4.2.3 Filesystems

There is one example of semistructured data which every computer user encounters all the time, although they may not realize it. A Unix-style filesystem is arranged as a tree, with filenames being labels of branches leading to subtrees, directories being nonleaf nodes and plain files being leaf nodes. There is a clear distinction between the data which may appear at the leaves of the tree (as file contents) and labels of branches (filenames, which are typically restricted to a certain maximum length). There are certain conventions which the filesystem tree must follow if the computer is to work, but usually adding extra bits to the tree will not break anything. Hence it is possible to install additional software or create new documents without adversely affecting the existing software.

It was not always like that—in the past, programs often worked with direct access to the disk and calculated their own sector positions for reading and writing. This could let access be more efficient, indeed direct disk access is still sometimes used for very speed-critical applications, but it made it harder to write new data to unused areas of the disk or to change the size of some files currently being used. For most applications it was a great improvement when the operating system could offer access to files by name, automatically handling the details of how to allocate space for new files or reuse space for old files. It was better even at the cost of slightly slower disk access for looking up a file name. This is another example of a tendency towards using semistructured data as computers have become faster.

Although operating systems provide ways to query the filesystem by listing all files in a directory, finding the amount of free space and so on, and although some provide rudimentary query tools like Unix's `find(1)`, there are not many applications which use the filesystem directly as a database. Normally most querying is done by reading data from the files—the leaves of the tree—and keeping that in some format suitable for querying. It would be inefficient to use a separate file to keep track of every small item of data, because typically small files waste a lot of space on disk. This may change as Oses add more database-like functionality to their filesystems to and improve the efficiency of small files.

It turns out that the Unix filesystem is actually a rather good match for the data model we adopt. The main difference is that a Unix directory cannot contain two files with the same name, whereas our trees can have two branches with the same label.

The similarity extends as far as graphical links: in Unix you can create symbolic links, or symlinks, which refer to another file by name. Although it is possible for the user to distinguish whether a file is contained in a directory, or simply referred to by a symlink in that directory, the intention is that most of the time, accessing a symlink to a file behaves in the same way as accessing the

file itself. Although the directory structure is a tree, symbolic links can be used to point sideways, up or down the tree—so that any program which traverses the directory structure must be careful to distinguish symlinks lest it get stuck in a cycle. When copying a directory structure, the symbolic links are copied but they continue to refer to the same destination files, which are not copied themselves. We return to this similarity later.

4.2.4 Ambients and info-terms

The ambient logic[3] is a modal logic for describing distributed computation or ‘ambients’. Because the structure of an ‘information term’ in the ambient logic is an unordered labelled tree, it turns out that the ambient logic can also be used to describe properties of semistructured data—an idea introduced in [17]. Further, it can be used to build a query language, as explained in [1]. The following explanation is condensed from that paper.

An information term is defined as

$$\begin{aligned}
 F ::= & \quad \text{nil} && \text{empty multiset} \\
 & m[F] && \text{labelled branch } lrl \\
 & F \mid F && \text{multiset union}
 \end{aligned}$$

The \mid operator is known as parallel composition and is analogous to parallel composition in process calculi such as CCS. But we are interested in using it to build trees from one or more labelled branches. It is symmetric and commutative, so you can think of it building an unordered multiset of branches composed in parallel.

There is only one kind of ‘data’ which can appear in these trees, the label m —and m always leads to a subtree, so the leaves of a tree must always be nil. This can make the data structures look slightly awkward, for example,

$$\text{emperor} \mapsto [\text{Augustus} \mapsto \text{nil}] \mid \text{emperor} \mapsto [\text{Nero} \mapsto \text{nil}]$$

Those extra nil trees at the leaves seem to make the data structure larger than it needs to be. By analogy to the Unix filesystem mentioned as an earlier example of semistructured data, an information term in this system is like a directory tree without any plain files—the leaves of the tree are always empty directories. On the other hand, information terms do have a very simple definition and it is useful to have only one place, m , in which data can appear. Most real-world examples of semistructured data have many more places to put data (for example XML can store attribute names, element names, attribute values and element content), which makes them often more natural for expressing real information, but complicates mathematical reasoning.

(The data model used in this project, to be described in the next section, is perhaps halfway between the ambient information terms and a richer, ‘messier’ framework for SSD such as XML. It has two places where data can appear, both as the label of a branch and separately at the leaves of the tree. It also differs from the ambient information trees in having links or pointers from one part of the tree to another. All these features will be introduced later.)

The query language for these information terms is based on the ambient logic, together with a way to find the values bound to free variables and construct a result tree from those values. The most interesting feature of the logic is the parallel composition $\varphi | \psi$, which matches a composition of trees $F_0 | F_1$ such that F_0 satisfies φ and F_1 satisfies ψ . Because $|$ is associative and commutative, the formula effectively tries all possible ways to split the tree into two halves—but the two halves must be disjoint. This makes the logic more powerful than first order logic for querying trees, because first order logic has no equivalent ‘sideways’ formula to take a part of an arbitrarily large number of branches in parallel.

Parallel composition is convenient to use for querying semistructured data, because it allows you to easily ignore details you’re not interested in. We will see this later on when describing the logic and query language implemented for this project.

In general, the query language described in this report has most of its features in common with the ‘Tree Query Language’ described in [1]; so much so that there is no need to give a complete summary of that paper here.

4.2.5 The graph data model

Although arbitrary graphs are not usually the obvious model to pick for semi-structured data, the graph logic defined in [18] has a lot in common with the ambient logic and the logic used in this project. It too has a way to break down a graph into separate parts and match each part against a separate formula.

4.2.6 General trees with dangling pointers

The layout of data structures in an imperative programming language can be considered as a forest of trees, with pointers between locations in those trees. To reason about imperative programs it’s useful to have a way to reason about such trees. [19] introduces some ideas for doing this; the logic used is similar to that in this project.

Chapter 5

Data model

We take it as given that the data should be represented as a tree or forest. Other kinds of data model (such as the relational model) have been studied enough. Further, we adopt unordered trees, meaning that subtrees are identified only by the labels of arcs pointing at them and there is no particular order among the subtrees of a node.

This contrasts with XML, which is (in general) an ordered tree model. In fact most computer-represented data structures tend to be ordered trees of some kind, since computer memory is inherently ordered. However there are some systems such as X.500 directories where the address of a subtree is determined only by labels, and there is no ordering among siblings. Later, we show how to map between ordered and unordered data models and compare our data model with some alternatives.

5.1 Unordered trees

First consider ‘plain’ trees without graphical links, defined as

$$\begin{array}{ll} M ::= & \text{data} \quad \text{leaf data} \\ & B \quad \text{branches to subtrees} \\ B ::= & \text{nil} \quad \text{empty} \\ & a \mapsto [M] \quad \text{branch with label } a \text{ to subtree } M \\ & B \mid B \quad \text{parallel composition} \end{array}$$

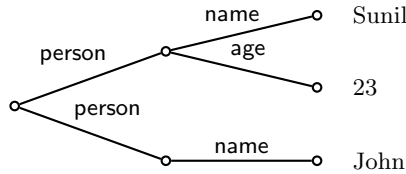
We have two types of data that can be stored: data at the leaves, and labels of branches to subtrees. Zero or more branches can be composed in parallel. We could make different choices, for example allowing ‘leaf data’ to be composed in parallel with branches or other leaf data. Choosing this particular data model is just a matter of taste, trying to pick a structure for trees that fits well with other models and users’ intuitions.

(The two productions in the grammar are because M represents the leaves of trees, while B is a set of branches leading to subtrees. Later on we adopt

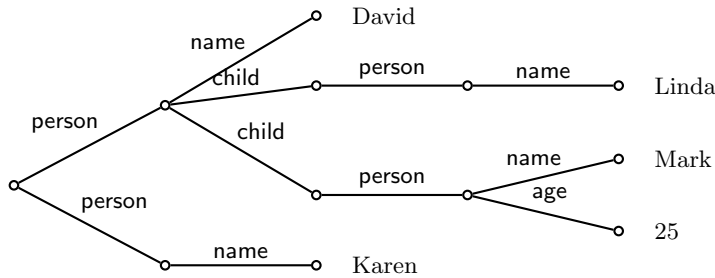
the convention T to refer to ‘trees’ which could be either a leaf, M , or a set of branches B .)

Because the trees are unordered, we have the structural congruence $\text{nil} | B \equiv B$, and commutativity and associativity of $|$, which makes the trees unordered. (But also note $B | B^{-1} \equiv B$, unless $B = \text{nil}$.) We assume this from now on, using \equiv as the standard test to see whether two trees are equal. Any tree logic will be defined using \equiv so it must respect structural congruence, by construction.

What data can we represent with these trees? An old example is personal information:



Here we can see two people, Sunil aged 23 and John of indeterminate age. It is clear how to add more people, or extra information such as ‘favourite-colour’. What about a family tree?



This is all fairly straightforward so far. You can see that there are some similarities with the earlier data structure, so a program looking for ‘person’ and ‘name’ labels might still work. But we have added some extra things under new labels—this is what semistructured data is all about. One thing to note is the decision to have multiple ‘child’ arcs each pointing to a ‘person’ subtree; alternatively we could have chosen to put all children under a single ‘children’ arc, or other arrangements besides. This is just a matter of taste.

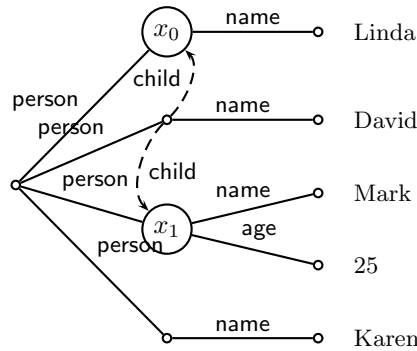
5.2 The need for graphical links

But consider now what happens if Mark and Karen beget offspring. Clearly the children should be referred to from both Mark’s and Karen’s subtree. But then the tree is no longer a tree but a DAG. Either that, or we end up duplicating information. Or what if we want to add arcs for parents as well as children? That would lead to a cycle and could never be expressed as a straightforward tree, not even if we try to unroll and copy subtrees.

At this point we could give up on trees and move to directed graphs as our data model (as in [18]). But trees are a natural fit for many concepts,

in particular those of ownership and location. Even the example above has a strong sense that a name ‘belongs’ to a single person, but a parent does not belong to the child in the same way. Can we use trees and containment for most things, while keeping the ability to have references back up the tree (or sideways) when needed?

This is what the idea of the graphical link is for. A graphical link is a labelled arrow from one tree node to another, without any restrictions in the data model about what points to what. They are called ‘graphical’ because they let you do the things you could do in an arbitrary graph. (Another possible name would be ‘pointers’; we discuss later the connection between graphical links and pointers in imperative programs.) Using graphical links, the family tree could be represented as:



Here some nodes have an address such as x_0 and the ‘child’ relation is a graphical link from one person’s tree to another. Now it’s clear how to add new children linked from two existing people, or how to add new relationships such as ‘friend’. You can add graphical links wherever you like without disturbing the tree structure of people themselves. Note that I have not nested one person inside another, it is no longer necessary when we have graphical links, and there is no real containment or ownership relation between people.

In fact the data model requires an address for every node, but if there are no links to that node then we need not bother to write the address. The new data model is defined as:

$M ::=$	data	leaf data
	B	branches to subtrees
$B ::=$	nil	empty
	$a \mapsto x[M]$	branch with label a to subtree M with address x
	$a@x$	graphical link with label a to address x
	$B B$	parallel composition

where each address x is globally unique but may be pointed to from several places. (It is possible to formalize this uniqueness requirement, but I’m sure common sense will be sufficient here.)

Writing a tree with globally unique addresses can be a pain and more importantly we may want to explicitly state that a certain subtree isn’t addressable.

There are two approaches to this: allow subtrees without addresses, or add a local name-hiding operator:

$$T ::= \dots \mid (\text{local } x) T$$

This would mean that the address x would not be visible outside that subtree, so it could not be pointed to by graphical links from outside but the name x could be reused without clashing. We could define an arc to a subtree without address to be equivalent to $(\text{local } x) a \mapsto x[T]$. Trees completely without addresses would reintroduce the question of set versus multiset—and so might the `local` operator, if you consider whether $(\text{local } x) a \mapsto x[T]$ is equal to $(\text{local } x) a \mapsto x[T]$. In this project we do not use `local` in the data model, and so we keep the requirement that every subtree must have a unique address.

5.3 Representing this data on a computer

Storing trees in computer memory is no problem, but the unorderedness (or if you prefer, structural congruence rules) is a little unusual. We just have to pick some arbitrary order among subtrees for storing them, and then make sure that any operations we define on the data type are unaffected by ordering.

In a functional language such as Haskell we can define a tree as either a piece of leaf data or a list of branches, provided that we do not write any code that depends on the order of the list, as above. The datatype definition is:

```
data Tree α ξ δ = Parr [BranchOrLink α ξ δ] | LeafData δ
data BranchOrLink α ξ δ = Branch α ξ (Tree α ξ δ) | Link α ξ
```

defined on three types, α for labels of branches, ξ for addresses of subtrees and δ for leaf data. In practice we normally use `String` for the labels and data and `Int` for the addresses. The branches are stored as a list, so there is more than one possible representation for the same tree. (Later, when we come to define the logic, you can see that the necessary operations can be implemented using three primitives, none of which is affected by the list order.)

Storing the branches-and-links as a list also means that finding the branch with a given label takes linear time in the length of the list. A possible optimization would be to use a data structure like a hash table or binary tree which has faster lookup.

It is not obvious how to handle addresses and graphical links, because Haskell does not have language support for pointers or references. The way I used, as seen above, is to just keep an address with each subtree. This is inefficient because to follow a graphical link you have to search through the whole tree looking for the corresponding address. But it matches what happens in the logic without special graphical link support (see later).

As mentioned in the introduction, one way to think of graphical links is as pointers in an imperative language, while subtrees are like embedding one data structure within another. This analogy is useful for implementation too! As well as a complete implementation of the query language in Haskell, I implemented a subset of it in Perl. Perl allows you to take the address of a data

structure and store pointers (called 'references') to it elsewhere. So instead of storing addresses as part of the data structure and searching from the top of the tree for the subtree with a given address, instead a graphical link is represented by a reference. This means that following a graphical link as part of running a query can be a much faster (constant-time) operation.

Chapter 6

Logic

The starting point of this project was the logic defined on unordered trees, based on the ambient logic [3]. It has formulæ to match the empty tree, a branch leading to a subtree, and a parallel composition of two trees. Along with a match for leaf data and standard logical connectives, this allows you to distinguish any two unordered trees, up to structural congruence.

Adding variables, recursion and variable hiding to the logic makes it powerful enough to use as a query language. Then we discuss how to extend the logic to take account of graphical links.

6.1 Unordered labelled trees, no recursion

To start with, we take the logic for plain unordered trees, without recursion, to be:

label expression	$\alpha ::=$	a	label
		\mathbf{a}	label variable
data expression	$\delta ::=$	d	leaf data
		\mathbf{d}	data variable
formulæ	$\varphi, \psi ::=$		
tree structure		\mathbf{nil}	empty
		$\alpha \mapsto [\varphi]$	tree branch
		δ	data
		$\varphi \mid \psi$	composition
classical logic		\mathbf{true}	true
		$\varphi \wedge \psi$	conjunction
		$\neg\varphi$	negation
equality tests		$\alpha_1 = \alpha_2$	label equality
		$\delta_1 = \delta_2$	data equality

So there are four cases to match the structure of a tree, and logical connec-

tives to combine these. We can define the meaning of the logic by associating each formula with a set of trees; then $T \models \varphi$ iff $T \in \llbracket \varphi \rrbracket$. Or more precisely, each formula under a given valuation ρ for label variables and data variables corresponds to a set of trees; then $T \models^\rho \varphi$ iff $T \in \llbracket \varphi \rrbracket_\rho$. Equivalently, we can define the meaning of a formula by defining \models directly:

$$\begin{aligned}
T \models^\rho \text{nil} &\Leftrightarrow T \equiv \text{nil} \\
T \models^\rho a \mapsto [\varphi] &\Leftrightarrow T \equiv \rho(a) \mapsto [T'] \wedge T' \models^\rho \varphi \\
T \models^\rho \delta &\Leftrightarrow T \equiv \rho(\delta) \\
T \models^\rho \varphi \mid \psi &\Leftrightarrow \exists T_1, T_2 \in \mathcal{T}. (T \equiv T_1 \mid T_2 \wedge T_1 \models^\rho \varphi \wedge T_2 \models^\rho \psi) \\
T \models^\rho \text{true} &\text{ always} \\
T \models^\rho \varphi \wedge \psi &\Leftrightarrow T \models^\rho \varphi \wedge T \models^\rho \psi \\
T \models^\rho \neg \varphi &\Leftrightarrow \neg(T \models^\rho \varphi) \\
T \models^\rho \alpha_1 = \alpha_2 &\Leftrightarrow \rho(\alpha_1) = \rho(\alpha_2) \\
T \models^\rho \delta_1 = \delta_2 &\Leftrightarrow \rho(\delta_1) = \rho(\delta_2)
\end{aligned}$$

The formula $\varphi \mid \psi$ operator checks if there exists one way to split the tree along its \mid parallel composition such that one half satisfies φ and the other satisfies ψ . It is defined in terms of \equiv , and for the structural congruence rules for \equiv we can see that there are many different ways to split a tree in two. Complexity results are not covered in this project report, but you can see there's an exponential number of different splittings to try. In general, a tree which is a parallel composition of n branches can be split 2^n different ways.

This logic is sound with respect to \equiv , in other words whenever $T \equiv T'$, there is no formula φ and substitution ρ such that $T \models^\rho \varphi$ but $T' \not\models^\rho \varphi$. Proof of this property follows from the definition of \models , all the cases are defined in terms of \equiv . More interestingly, we have that whenever two trees are not structurally congruent, there exists a formula which can distinguish between them.

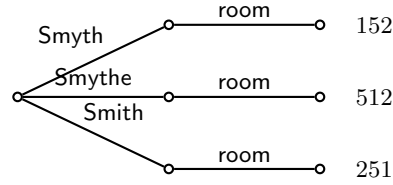
Knowing this, we can say that two formulæ are equal if and only if the same trees satisfy both. In other words

$$\varphi = \psi \Leftrightarrow T \models^\rho \varphi \Leftrightarrow T \models^\rho \psi, \text{ for all } \rho.$$

6.1.1 Examples

Here we show how this logic can be used for querying unordered trees without graphical links. The definition of the logic asks whether a formula φ under a substitution ρ matches a tree T , yes or no. But for building a query language we ask a different question: given φ and T , what is the set of possible ρ such that $T \models^\rho \varphi$? In the following sections this will be what we're really interested in, finding the possible substitutions ρ .

Take a tree representing people and their room numbers:



These example diagrams, by the way, are automatically generated from code, as is the syntactic representation of the tree:

$\text{Smyth} \mapsto [\text{room} \mapsto [152]] \mid \text{Smythe} \mapsto [\text{room} \mapsto [512]] \mid \text{Smith} \mapsto [\text{room} \mapsto [251]]$

The formula $\varphi = \text{Smith} \mapsto [\text{room} \mapsto [251]]$ states that the tree consists of a branch **Smith**, pointing to a branch **room**, pointing to the leaf data 251. It sounds like this is how to ask whether Smith has room 251. But in fact $T \not\models \varphi$. Why? Because the tree is not just that single branch; there are branches for other people in parallel.

To match a branch even though it may be in parallel with others, we use a formula like $a \mapsto [\psi] \mid \text{true}$. This means the tree can be split into two halves, one of which satisfies $a \mapsto [\psi]$ and the other satisfies **true**. Since any tree satisfies **true** what this means is that some part of the parallel composition satisfies $a \mapsto [\psi]$. The ‘**true**’ is a common idiom.

So to be fully general our formula specifying Smith’s room number should be $\varphi = \text{Smith} \mapsto [\text{room} \mapsto [251] \mid \text{true}] \mid \text{true}$. Now we have $T \models \varphi$, and φ will continue to work even if extra information such as telephone numbers is added to the tree.

But most often users do not ask questions like ‘I believe Smith’s room is 251, true or false?’. They are more likely to ask ‘what is Smith’s room?’ or ‘who has room 251?’. Perhaps even, ‘tell me all people and their room numbers’. This is where we want to write a formula with free variables and find which substitutions ρ make it match. To find Smith’s room number use $\varphi = \text{Smith} \mapsto [\text{room} \mapsto [\mathbf{d}] \mid \text{true}] \mid \text{true}$. Then we have $T \models^\rho \varphi$ if and only if $\rho(\mathbf{d}) = 251$. The result of the query is defined as the set of minimal ρ which make it match, minimal meaning that there are no useless variables included. The result is thus the single substitution $\{\mathbf{d} \mapsto 251\}$.

Because you can mention the same variable more than once, we can ask for the names of two people who share a room:

$\mathbf{a}_1 \mapsto [\text{room} \mapsto [\mathbf{d}] \mid \text{true}] \mid \mathbf{a}_2 \mapsto [\text{room} \mapsto [\mathbf{d}] \mid \text{true}] \mid \text{true}$

Here the data variable \mathbf{d} holds the room number, while the label variables \mathbf{a}_1 and \mathbf{a}_2 hold the names of the two people. As it happens the tree does not satisfy this formula under any substitution, so the result is the empty set.

You might ask, what about when $\mathbf{a}_1 = \mathbf{a}_2$? But the parallel composition chops the tree in two, so that if for example the ‘Smythe’ branch matches the $\mathbf{a}_1 \mapsto \dots$ part of the formula, only the remaining parts of the tree are available for matching the $\mathbf{a}_2 \mapsto \dots$ and **true** parts of the formula. So it is not possible for \mathbf{a}_2 and \mathbf{a}_1 to end up matching the same branch of the tree. We need not worry about how to bracket the splittings since \mid in formulæ is associative.

Compared with first-order logic for describing trees, the interesting feature of this logic is the ‘horizontal reasoning’ provided by the $|$ operator. We have the single step down the tree provided by $\alpha \mapsto \xi[\varphi]$, and also the ability to describe parts of a tree split sideways. First-order logic, where a predicate $Ref(t, \alpha, t')$ means t contains an arc labelled α to the subtree t' , has only the single step. Because first-order logic has general \exists and \forall , it is able to implement $|$, but here the parallel split is considered important enough to provide as a primitive—and the full power of first-order logic is not needed.

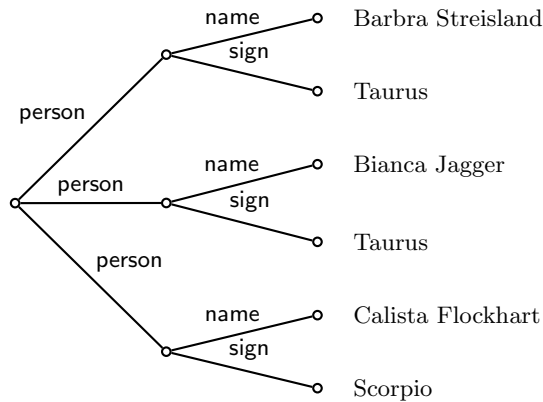
6.2 Existential quantification

We now add the \exists quantifier to the logic. \exists in a formula is used to ‘hide’ a variable so that it is no longer free—so that it no longer appears in the set of substitutions needed to make the tree satisfy the formula. Formally,

$$T \models^\rho \exists \mathbf{a}. \varphi \Leftrightarrow \exists \mathbf{a}. T \models^{\rho[\mathbf{a} \mapsto \mathbf{a}]} \varphi$$

where \mathbf{a} is any label, and $\rho[\mathbf{var} \mapsto \text{value}]$ is the standard function overriding. Similarly $\exists \mathbf{d}$ for hiding data variables. We can see from the definition that if a substitution S giving a value for \mathbf{a} makes a tree satisfy φ , then S' will make it satisfy $\exists \mathbf{a}. \varphi$, where S' is S without the valuation for \mathbf{a} .

As an example, suppose I have the tree:



and my supervisor asks me to find celebs who share the same star sign. ‘I don’t care about what that sign is’, she insists, ‘only that two different people share it’. Without existential quantification you could write a formula

```

person  $\mapsto$  [ name  $\mapsto$  [name0] | sign  $\mapsto$  [s] | true ]
| person  $\mapsto$  [ name  $\mapsto$  [name1] | sign  $\mapsto$  [s] | true ]
| true

```

which on the tree above, matches under the substitution

{**name**₀ \mapsto Barbra Streisland, **name**₁ \mapsto Bianca Jagger, **s** \mapsto Taurus}, and the same with **name**₀ and **name**₁ swapped around. That is the correct result of course, but it contains the extra information **s** which was not asked for. But wrapping the entire formula with $\exists \mathbf{s}$ makes it match under substitutions which need not mention that variable, so in a way it’s slightly cleaner.

(Note that in fact a substitution containing useless information, values for variables which are never used in the formula, can still cause a formula to match. When we give examples of substitutions necessary to make a tree satisfy a formula, we implicitly mean ‘and not containing any useless information’. Or if you prefer, each substitution is as small as it can be while still letting the formula match. It is also possible to have useless variables in the formula, as in $\text{age} \mapsto [\mathbf{a}] \vee \text{true}$, which will match any tree no matter what the value of \mathbf{a} . Such issues could take up a lot of space in the report, but I hope you agree that they are not very interesting and most of the time we quietly ignore them.)

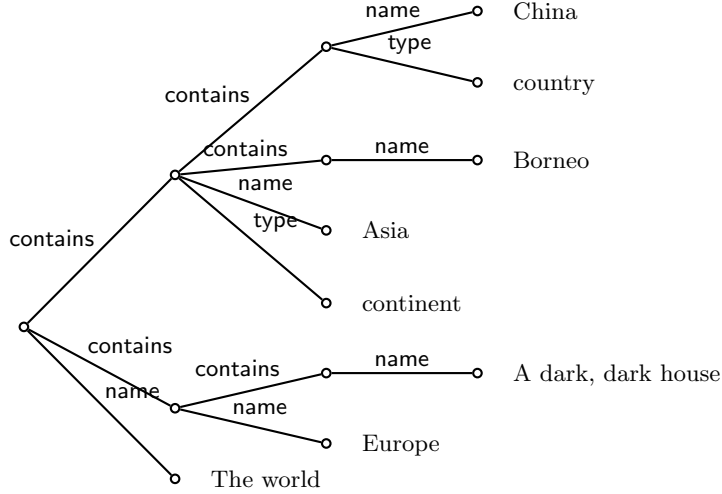
That was a trivial example of \exists , and if hiding variables for simple formulæ were the only application, it wouldn’t be worth bothering with. But we’ll see later that \exists is very useful when put together with recursion.

6.3 Adding recursion

So far, the logic is useful for trees where the depth and some structure is known in advance. But such data structures are really more like lists or sets than trees. If we expect to build a query language which is useful for arbitrary tree data, then we must be able to write a single formula which works on trees to arbitrary depth, and for that you need recursion.

It’s worth noting that many of the standard examples of tree-structured data (such as a bibliographic database, and all the examples given so far in this report) are not really trees in this sense because they do not require recursion to query them. That is, the data structures have a given depth which is known in advance. A ‘book’ entry in a bibliography contains ‘author’, ‘title’ and perhaps even subtrees inside those—but it does not contain another ‘book’ subtree. The family tree first used to introduce the data model was genuinely a tree structure, because it contained ‘person’ trees nested inside each other to arbitrary depth. But then we saw that the best expression of family structure is to use graphical links, so that ‘person’ trees are no longer nested inside each other and the tree once more has a fixed depth.

It is actually quite hard to think of a real-world example which is best expressed as a tree nested to *arbitrary* depth. This is good news, in a way, because it backs up the claim that the most natural way to represent real data is often to use graphical links rather than nesting. Consider a tree of places with ‘contains’ branches. That doesn’t have a fixed depth, since even a small place could contain some even smaller place; so to query it will need recursion. But again, places do not really form a tree, for example the Danube flows through several countries yet is contained in none of them. Again it looks as though graphical links (such as ‘overlaps’ or ‘borders’) are the best way to represent the information. But for the sake of an example, suppose that we decided to represent places just as a tree.



A formula with a free data variable \mathbf{d} , which matches the tree whenever $\rho(\mathbf{d})$ is the name of a place, would have to follow the ‘contains’ branches to arbitrary depth. To make this possible, we add recursive formulæ with two additions to the logic:

least fixed point	$\mu \mathbf{R}.$
recursive call	\mathbf{R}

Recursion has the property

$$T \models^{\rho} \mu \mathbf{R}.\varphi \Leftrightarrow T \models^{\rho} \varphi[\mu \mathbf{R}.\varphi/\mathbf{R}].$$

The formal definition of the μ recursion operator is as the least fixed point of the above equation with respect to \subseteq . This would mean for example that the semantics of $\mu \mathbf{R}.\mathbf{R}$ would be the empty set, in other words this formula matches no trees. [20] explains that this least fixed point is known to exist as long as \mathbf{R} appears only positively in the body of the formula; that is, a formula like $\mu \mathbf{R}.\neg \mathbf{R}$ is not allowed. The implementation in this project deviates from that definition a little; it does not have the restriction that \mathbf{R} appear only positively, but the evaluation of some recursive formulæ like $\mu \mathbf{R}.\mathbf{R}$ will fail to terminate. In Evaluation we discuss a possible alternative semantics which could remove the ‘appears only positively’ restriction while remaining computable (and agreeing with what was implemented).

Informally, any occurrence of \mathbf{R} within the formula φ is a recursive call. Using recursion, we can define the formula to find all place names as:

$$\mu \mathbf{R}.\left(\left(\text{name} \mapsto [\mathbf{d}] \vee \text{contains} \mapsto [\mathbf{R}]\right) \mid \text{true}\right)$$

This matches a ‘name’ branch, or matches a ‘contains’ branch and applies itself recursively to the subtree underneath.

6.3.1 Using \exists together with recursion

Previously, \exists didn’t do much, just removing a particular variable from the result set. But if we have recursion then being able to reuse the same name at different

recursive levels is very useful. The formula above to find all names of places didn't need to hide variables because it just followed the 'contains' arcs down the tree. But suppose we didn't know the exact structure of the world—it might have 'borders' or 'overlaps' or 'comprises' arcs, we don't know—and we want to search down the whole tree for everything under 'name'. This is part of what semistructured means: we don't have full knowledge of the schema used, but we know a few familiar things to look out for. Is it possible to write a query which, given any tree of any form, will walk down it and extract all the 'name's'?

With \exists and recursion we can express the query as

$$\mu\mathbf{R}.\left(\text{name} \mapsto \mathbf{d} \vee \exists \mathbf{a}.\mathbf{a} \mapsto [\mathbf{R}] \mid \text{true}\right)$$

using \mathbf{a} as a scratch variable for the label of any arc found. The \exists means that it is a fresh \mathbf{a} each time, so the recursive call can follow a different label to its parent. Compare this to the same query without \exists , which would be restricted to following sequences of arcs following a single label.

6.3.2 The \diamond operator

The above example shows the most common use of recursion: looking downwards through the tree for the same thing at each place. We can define an operator \diamond to help with this:

$$T \models^\rho \diamond\varphi \Leftrightarrow T \models^\rho \varphi \vee (T \equiv \mathbf{a} \mapsto [T'] \wedge T' \models^\rho \diamond\varphi, \text{ for some label } \mathbf{a}).$$

So this operator encapsulates the structure of trees and traversing them. \diamond is expressible in terms of recursion and \exists :

$$\diamond\varphi = \mu\mathbf{R}.\varphi \vee \exists \mathbf{a}.\mathbf{a} \mapsto [\mathbf{R}] \mid \text{true}$$

where \mathbf{R} is a new recursion variable not seen before. Note that while this definition of \diamond follows branches which are in parallel with other branches, it doesn't recurse along the \mid parallel composition by itself. In other words if $T \models \varphi$ it does not necessarily follow that $T \mid T' \models \diamond\varphi$. But you could use $\diamond(\text{true} \mid \varphi)$.

6.4 Equality tests

The explicit equality test $\alpha_1 = \alpha_2$ asserts that two label expressions have the same value:

$$T \models^\rho (\alpha_1 = \alpha_2) \Leftrightarrow \rho(\alpha_1) = \rho(\alpha_2)$$

In fact the label expressions could be constants, not variables, but let us assume that $\rho(\mathbf{a}) = \mathbf{a}$ for all constants 'a'. There are similar equality tests for leaf data.

The best example of how to use equality tests is finding two things which are *different*. For example, abbreviating $\neg(a = b)$ as $a \neq b$,

$$\begin{aligned} &(\text{attraction} \mapsto [\mathbf{a} \mapsto [\text{to be announced}] \mid \text{true}] \\ &\mid \text{attraction} \mapsto [\mathbf{a}' \mapsto [\text{to be announced}] \mid \text{true}]) \end{aligned}$$

| true)
 $\wedge \mathbf{a} \neq \mathbf{a}'$

will match a tree if and only if two different ‘attraction’ subtrees have two differently-labelled branches to the value ‘to be announced’.

Equality tests on leaf data and on tree variables are not so essential because general negation can be used for that. If you have a data variable \mathbf{d} and you want to match some leaf data that is not \mathbf{d} , you can write the formula $(\neg \mathbf{d})$. It is not yet clear whether there is always an equivalent shortcut for branch labels (and addresses, which we come to next); so it’s not known whether general equality tests add power to the logic. But they certainly add convenience, and they are in the definition of the logic so they should be implemented.

6.5 Extending the logic to graphical links

We now have a logic which is useful for making statements about plain unordered trees, and which could be used as the basis for a query language by finding all ρ such that $T \models^\rho \varphi$. But the point of this project is to extend the language to the new data model, trees with graphical links.

We can keep the essential ideas the same: we still have variables and constants, classical logic, recursion and name hiding. Only the formulæ to match against the structure of a tree need change. We already know what the structure of trees is, so as a first attempt why not just copy that structure in formulæ:

address expression	$\xi ::=$	x	address
		\mathbf{x}	address variable
tree structure		nil	match empty tree
		$\alpha \mapsto \xi[\varphi]$	match tree branch (address ξ)
		$\alpha @ \xi$	match graphical link to address ξ
		δ	data
		$\varphi \psi$	composition

Everything else is as before; we have just added address expressions (and address variables to ρ), included the address in matching a subtree, and added a case to match a graphical link. The satisfaction relation is as before, except:

$$T \models^\rho \alpha \mapsto \xi[\varphi] \Leftrightarrow T \equiv \rho(\alpha) \mapsto \rho(\xi)[T'] \wedge T' \models^\rho \varphi$$

$$T \models^\rho \alpha @ \xi \Leftrightarrow T \equiv \rho(\alpha) @ \rho(\xi)$$

For notational convenience we define $\alpha \mapsto [\varphi] = \exists \mathbf{x}. \alpha \mapsto \mathbf{x}[\varphi]$, so that you can continue to write formulæ without worrying about addresses if the addresses don’t interest you.

6.5.1 Example

Take the example given earlier of some people with the ‘child’ relation indicated by graphical links; syntactically this is written as

$$\begin{array}{l}
\text{person} \mapsto x_1 [\text{name} \mapsto [\text{Linda}]] \\
| \text{person} \mapsto [\text{child}@x_1 | \text{child}@x_0 | \text{name} \mapsto [\text{David}]] \\
| \text{person} \mapsto x_0 [\text{name} \mapsto [\text{Mark}] | \text{age} \mapsto [25]] \\
| \text{person} \mapsto [\text{name} \mapsto [\text{Karen}]]
\end{array}$$

Suppose you wanted to find everyone who has a sibling. To do this we must first find all parents who have more than one child and then follow the child links to get the details of the children. To find the addresses of the children use

$$\text{person} \mapsto [\text{child}@x | \text{child}@y | \text{true}] | \text{true}$$

which matches the above tree under the substitutions $\{\mathbf{x} \mapsto x_0, \mathbf{y} \mapsto x_1\}$ and $\{\mathbf{x} \mapsto x_1, \mathbf{y} \mapsto x_0\}$. Now to find the names of the children we must find the parts of the tree with those addresses. We know that the children's subtrees will be reachable via a 'person' arc from the root, so to find them:

$$\begin{array}{l}
\exists x. \exists y. \quad \text{person} \mapsto [\text{child}@x | \text{child}@y | \text{true}] \\
| \quad \text{person} \mapsto x[\text{name} \mapsto [\mathbf{m}] | \text{true}] \\
| \quad \text{person} \mapsto y[\text{name} \mapsto [\mathbf{n}] | \text{true}] \\
| \quad \text{true}
\end{array}$$

This matches with $\{\mathbf{m} \mapsto \text{Linda}, \mathbf{n} \mapsto \text{Mark}\}$ and the other way round. In writing the formula we made some assumptions, for example, the 'child' relation is irreflexive so the child's subtree will be different from the parents. In general if we wanted to search for subtrees with given addresses anywhere in the tree, not just immediately under a 'person' branch, we could use the \diamond operator:

$$\begin{array}{l}
\exists x. \exists y. \quad \text{person} \mapsto [\text{child}@x | \text{child}@y | \text{true}] | \text{true} \\
\wedge \quad \diamond \exists \mathbf{a}. \mathbf{a} \mapsto x[\text{name} \mapsto [\mathbf{m}] | \text{true}] \\
\wedge \quad \diamond \exists \mathbf{a}. \mathbf{a} \mapsto y[\text{name} \mapsto [\mathbf{n}] | \text{true}]
\end{array}$$

So in general 'following' a graphical link has two parts: binding the address pointed to to a variable, and then searching the whole tree for the subtree with that address. It would be nice to encapsulate this in a new derived formula, but this is not possible because a formula of the logic operates on a 'current tree' whereas the searching with \diamond needs to start from the top of the whole tree. We return to this subject in the Evaluation section.

6.6 Tree variables

The 'children' example also serves as motivation for another extension to the logic. What if we wanted to find not only the children's names but all their details? Given an address x , you could try writing

$$\diamond \exists \mathbf{a}. \mathbf{a} \mapsto x[\mathbf{branch} \mapsto [\mathbf{data}] | \text{true}]$$

to match all the branches underneath the subtree with address x . For the address x_0 , the children tree would satisfy this formula under the substitutions $\{\mathbf{branch} \mapsto \text{age}, \mathbf{data} \mapsto 25\}$ and $\{\mathbf{branch} \mapsto \text{name}, \mathbf{data} \mapsto \text{Mark}\}$, so it

might appear we can find out all information about the subtree this way. But what if the subtree at x_0 contains further branches to arbitrary depth? It is not possible to write a formula of the logic whose matching substitutions will find all the information under x_0 . Without recursion all formulæ are limited to fixed depth, as we have seen; and even with recursion there would be no way to indicate the precise tree structure in the substitutions returned.

In any case, even if it were possible to get by with lots of **branch** and **data** variables, it is awkward to get results in this form. What we really want is just to grab the whole subtree under x_0 and include it in the results. To do this we extend logical formulæ and substitutions ρ with tree variables \mathbf{t} , so that

$$T \models^{\rho} \mathbf{t} \Leftrightarrow \rho(\mathbf{t}) \equiv T.$$

This lets us write:

$$\diamond \exists \mathbf{a}. \mathbf{a} \mapsto x[\mathbf{t}]$$

to capture into \mathbf{t} the whole subtree at address x . Putting together the features discussed in the above sections, the formula to match people with a sibling is:

$$\begin{aligned} \exists \mathbf{x}. \exists \mathbf{y}. \quad & \text{person} \mapsto [\text{child@}\mathbf{x} \mid \text{child@}\mathbf{y} \mid \text{true}] \mid \text{true} \\ & \wedge \diamond \exists \mathbf{a}. \mathbf{a} \mapsto \mathbf{x}[\mathbf{t}] \\ & \wedge \diamond \exists \mathbf{a}. \mathbf{a} \mapsto \mathbf{y}[\mathbf{u}] \end{aligned}$$

which matches the ‘children’ tree under $\{\mathbf{t} \mapsto (\text{name} \mapsto [\text{Linda}]), \mathbf{u} \mapsto (\text{age} \mapsto [25] \mid \text{name} \mapsto [\text{Mark}])\}$, and the same with \mathbf{t} and \mathbf{u} swapped.

6.7 Comparing variables of different kinds

Note that there are no comparisons between variables of different kinds; no way to compare leaf data to a label, for example. In general, those different types might not be comparable—perhaps addresses are stored as integers or pointers, while the other data are strings. Even if labels and leaf data are of the same type, it may not always make much sense to compare them: returning to the Unix filesystem mentioned in the Background chapter, how often is it necessary to compare a file’s *name* with another file’s *contents*? But for some applications it might be rather important to compare labels and leaves, and it would be needed to get the best flexibility in the query language. For this project, comparisons between different kinds of variable were not implemented.

6.8 Conclusion

We have now introduced all the features of the logic. There are structural cases to match the different kinds of trees, including graphical links; these can contain variables which are bound to the data contained in the tree, and a tree variable matches the entire tree. Variable equality tests check that the values bound to two variables are the same. Formulæ can be recursive; this is necessary to query trees of arbitrary depth. Existential quantification is equivalent to picking a fresh variable name and is useful together with recursion.

The whole logic is presented in A. The next chapter explains how to handle some of the problems which come up dealing with this logic, in particular those caused by negation and variable equality tests.

Chapter 7

Implementation of the logic

We want to write a computer program which will take a formula and a tree, and return the set of substitutions which let the formula match the tree. The trouble is that for some formulæ and trees the set is infinite, and a computer has only finite memory. For example, take the query ‘return all the words \mathbf{n} which are not used as somebody’s name’,

$$\neg \diamond (\text{name} \mapsto [\mathbf{n}] \mid \text{true}).$$

It sounds reasonable, until you realize that \mathbf{n} can be bound to any data at all except for a finite number of strings which are seen in the tree underneath a `name` arc. That is an infinite set of substitutions.

The trouble occurs with introducing the new variable inside a negation; if we wrote a negation not containing variables:

$$\neg \diamond (\text{name} \mapsto [\text{Jones}])$$

or one where the variable’s possible values can be fixed to a finite set outside the negation:

$$\diamond (\text{name} \mapsto \mathbf{n} \mid \neg (\text{maiden-name} \mapsto \mathbf{n}))$$

then the result is finite.

It is possible to define sufficient syntactic conditions for a ‘safe’ formula, one which will not produce infinite results. But for this project we won’t do that, because there is a way to handle all queries correctly and cope with the infinite results. This is based on the technique explained in [21].

In this chapter we explain how to make a finite representation of the possibly-infinite set of results returned by matching a formula. Some operations need to be defined on this representation, in particular a ‘unification’ to implement \wedge and a ‘complement’ to implement \neg . The trick is to make sure all these can be done while still keeping the data structure finite in size.

The technique used in this project is loosely based on that in [21]—in fact, it’s based on a spoken explanation by that paper’s author, since the paper itself is opaque. The handling of equal/unequal relationships between variables is original, because [21] did not implement that.

We make three attempts at finding a representation for the sets of substitutions. The first attempt illustrates some general principles but is clearly inadequate because it cannot handle negation. The second attempt can deal with most of the logic, but doesn't have provision for equality tests between variables. It can support comparison of a variable to a value, but not an assertion that two variables are equal. However, if you were willing to forgo fully general equality tests then the second representation would be adequate. The final version is the most complex and supports all the features of the logic.

7.1 First attempt: mapping variables to sets of values

If you take a finite result and negate it, you get a result of the form: all values except a finite number. Such a cofinite set can be represented by storing all the values that are not in the set. The idea is that a substitution can map a variable to a value, but it can also map it to 'anything but' a set of values.

7.1.1 Representing a cofinite set of values

We define the data type

```
data ValueP  $\alpha$  = One  $\alpha$  | NotIn [ $\alpha$ ].
```

The meaning of this type is that `One x` signifies 'definitely x ', while `NotIn x s` signifies 'could be anything, but definitely not one of the x s'. Before defining substitutions we first give a notion of agreement on values. If two `ValueP`s both give some information, is there a new `ValueP` which combines the information from both? It is possible that the two contradict each other, so no agreement is possible.

To implement the function which tries to combine information from two `ValueP`s, use the standard `Maybe` type:

```
data Maybe  $\beta$  = Nothing | Just  $\beta$ 
```

We'll use this to return a value of type `Maybe (ValueP α)`, with `(Just x)` meaning that the two values could be combined into x , and `Nothing` meaning no agreement was possible. Later on, many of the operations to combine information from two different sources will use the `Maybe` type, because it's possible that the information will conflict and so the result should be `Nothing`.

The function `agree` takes two `ValueP`s to a new `ValueP` if they agree, and `Nothing` if they do not:

```
agree :: ValueP  $\alpha$  → ValueP  $\alpha$  → Maybe (ValueP  $\alpha$ )
```

agree (One x) (One y)		$x == y = \text{Just (One } x)$
		otherwise = Nothing
agree (One x) (NotIn l)		$x \notin l = \text{Just (One } x)$
		otherwise = Nothing
agree (NotIn l) (One x)	=	agree (One x) (NotIn l)
agree (NotIn l) (NotIn l')	=	Just (NotIn (union l l'))

The cases are explained as follows. If one argument says ‘definitely x ’, and the other says ‘definitely y ’, then they agree if and only if $x = y$. If one says ‘definitely x ’ and the other says ‘definitely not in l ’, then if $x \notin l$ they agree, otherwise they contradict each other. The order of the arguments does not matter, so the third case is the same as the second. Finally, the agreement of ‘not in l ’ and ‘not in l' ’ must be ‘not in $l \cup l'$ ’. In Haskell we represent sets using lists and set union with concatenation.

7.1.2 Mapping from variables to ValuePs

Formally, a substitution is a partial function from variable names to values (with a finite domain). But because the `ValueP` data type can represent a cofinite set of possible values, we will make a `SubstP` type which maps variables to `ValuePs`. Each `SubstP` stands for a set of substitutions, for example, the `SubstP` mapping x to `(NotIn [3, 4])` stands for a cofinite set of substitutions, all ρ such that $\rho(x) \neq 3$ and $\rho(x) \neq 4$.

In code, it is easiest to represent a partial function as a list of $(var, value)$ pairs and provide a lookup function. The empty `SubstP` is the empty list, and adding to the function means adding a new pair to the list, and so on.

```
type SubstP  $\alpha$  = [(Var, ValueP  $\alpha$ )]
```

The function `from_substp` looks up a variable and returns its associated `ValueP`, plus the remainder of the `SubstP` after removing that pair. So it combines the operations of looking up a variable, and removing that variable from the substitution. There are two possible results from looking up a variable: either it was found, or it wasn't. If it was found, then `from_substp` should return the associated value and the rest of the substitution. The type `Found` defines the two possible results.

```
data Found  $\alpha$  = Found (ValueP  $\alpha$ , SubstP  $\alpha$ ) | NotFound
from_substp :: Var  $\rightarrow$  SubstP  $\alpha$   $\rightarrow$  Found  $\alpha$ 
from_substp v s = case (lookup v s) of
  Nothing  $\rightarrow$  NotFound
  Just val  $\rightarrow$  Found (val, filter (( $\neq$  v) . fst) s)
```

The result of `from_substp v s` is `NotFound` if v is not defined in s .

7.1.3 Unification

To see why unification on **SubstPs** is important, look again at the definition of part of the logic, for example:

$$T \models^{\rho} \alpha \mapsto \xi[\varphi] \Leftrightarrow T \equiv \rho(\alpha) \mapsto \rho(\xi)[T'] \wedge T' \models^{\rho} \varphi$$

Let us express this differently, giving the semantics of (T, φ) as the set of substitutions ρ such that $T \models^{\rho} \varphi$.

$$\llbracket T, \alpha \mapsto \xi[\varphi] \rrbracket = \{\rho : T \equiv \rho(\alpha) \mapsto \rho(\xi)[T'] \wedge T' \models^{\rho} \varphi\}$$

Assuming that T has the shape $a \mapsto x[T']$,

$$\begin{aligned} \llbracket T, \alpha \mapsto \xi[\varphi] \rrbracket &= \{\rho : \rho(\alpha) = a \wedge \rho(\xi) = x \wedge T' \models^{\rho} \varphi\} \\ &= \{\rho : \rho(\alpha) = a\} \cap \{\rho : \rho(\xi) = x\} \cap \llbracket T', \varphi \rrbracket \end{aligned}$$

So the result of matching a formula against a tree can be expressed as the intersection of some sets of substitutions.

The first set $\{\rho : \rho(\alpha) = a\}$ can be represented with the **SubstP** mapping α to $(\text{One } a)$. Similarly the second set is represented by $[(\xi, \text{One } x)]$. The third set of substitutions can be defined by induction, if we assume that all results of formulæ are representable with a **SubstP** object.

Many other formulæ for matching tree structure can also be expressed in this way: they can be reduced to either simple sets, or the intersection of sets produced by other formulæ. We hope to show by induction that this holds for all formulæ—but that is not true, as we'll soon see. For the time being, we'll continue with the current approach.

If the **SubstP** sp represents the (possibly infinite) set of substitutions S , and sp' represents S' , then we want `unify_substps` sp sp' to represent $S \cap S'$. We define `unify_substps` in terms of a simpler operation `extend_substp`, which attempts to add a new variable mapping to a **SubstP**. It first finds the existing **ValueP** for that variable, if any. If the variable was not mentioned before, the new mapping can simply be added to the list; if the variable was known before then the values are unified. The unification might fail, which is why the return type is `Maybe (SubstP α)`.

`extend_substp :: Eq α \Rightarrow (Var, ValueP α) \rightarrow SubstP α \rightarrow Maybe (SubstP α)`

```

extend_substp (var, val) s = case (from_substp var s) of
  NotFound      -> Just ((var, val) : s)
  Found (val', rest) -> do
    val'' <- agree val val'
    return ((var, val'') : rest)

```

The `do`-notation is here a shortcut for working with `Maybe` values. The expression `val'' <- agree val val'` can be best understood as ‘*attempt to give val'' the result of agree val val'*’. If that call to `agree` fails, or in other words returns `Nothing`, then this whole function returns `Nothing`. But if `agree val val'` returns `Just x`, `val''` is bound to x and the function continues.

You can think of `extend_substp` as trying to add some extra information to a `SubstP`, and returning `Nothing` if the extra information is inconsistent with what was already there. One more auxiliary function is the standard `foldr` lifted to `Maybe` types:

```
foldr_maybe :: (α → β → Maybe β) → β → [α] → Maybe β
foldr_maybe f z [] = Just z
foldr_maybe f z (x : xs) = do
    a ← foldr_maybe f z xs
    f x a
```

This is just a list catamorphism that ‘could go wrong’. It tries to reduce the list using the function `f`; but if the call of `f` fails, or the recursive call to `foldr_maybe` fails, then the whole thing fails and the result is `Nothing`. Because a `SubstP` is really a list, we can use this function to work on `SubstPs` by picking off one (variable, value) pair at a time. Now define `unify_substps` as:

```
unify_substps :: Eq α ⇒ SubstP α → SubstP α → Maybe (SubstP α)
unify_substps = foldr_maybe extend_substp
```

In other words picking variable assignments one at a time from one `SubstP`, and trying to merge each one into the other `SubstP`. If all of them succeed we have a `SubstP` combining the information from both, otherwise `Nothing`.

7.1.4 Inadequacy of the first attempt

Then can we work using the `SubstP` objects instead of with sets of substitutions, call `unify_substps` to implement \cap , and use singleton substitutions like $[(\alpha, a)]$ for the base cases? Unfortunately it is not quite that easy.

While the result of most of the structural matching formulæ can be defined using \cap on sets of substitutions, the definition of $|$ is different. Returning to the semantics defined in terms of sets of ρ , we have

$$\llbracket (T, \varphi | \psi) \rrbracket = \bigcup \{ \llbracket (T_1, \varphi) \rrbracket \cap \llbracket (T_2, \psi) \rrbracket : T \equiv T_1 | T_2 \}$$

So at least we need to implement \cup as well as \cap . And we haven’t even mentioned how to handle \neg so far. But considering for the moment how to represent \cup with our `SubstP` objects, you can see that a single `SubstP` is not sufficient. How would you represent $\{\{x \mapsto 3\}, \{x \mapsto 4\}\}$, which is the union of two singleton sets? This set of substitutions means ‘ x is either 3 or 4’.

7.2 Second attempt: a disjunction of `SubstPs`

What we need is a set of `SubstPs`, whose meaning is the disjunction of the individual `SubstPs`. As usual in Haskell we store a set as a list. The above example could be represented as $[[("x", \text{One } 3)], [("x", \text{One } 4)]]$. Now we need to implement \cap and \cup in terms of these disjunctions.

To extend `unify_substps` to sets of `SubstPs`, we take all possible pairs of one element from the first set and one element from the second set, and try to unify the two elements. If the unification is not `Nothing`, it is part of the result set.

```

pairs :: [α] → [β] → [(α, β)]
pairs [] _ = []
pairs (x : xs) ys = [ (x, y) | y ← ys ] ++ pairs xs ys
unify_substp_sets :: Eq α ⇒ [SubstP α] → [SubstP α] → [SubstP α]
unify_substp_sets s s' = catMaybes (map (uncurry unify_substps) (pairs s s'))

```

This works as follows. First, `pairs` takes two lists and returns a list of all (x, y) pairs where x is some element from the first list, and y some element from the second. This gives us all pairs of one `SubstP` from each set. Next, we apply `(uncurry unify_substps)` to each pair, that is, we try to unify the two. The result of unifying might be a new `SubstP`, or it might be `Nothing`. Finally, `catMaybes` is the standard library function to remove `Nothing` elements from a list. So the result is a set of all `SubstPs` which were successfully created by unifying one element from each input set.

We can use `unify_substp_sets` to implement \cap on sets of substitutions; the size of the output list grows exponentially in the size of the input lists, but it is still finite. We still have a finite data structure to represent an infinite set of substitutions.

What about \cup ? In fact we will not define \cup separately but go straight on to negation. Once negation is implemented, \cup can be defined in terms of \cap and \neg .

7.2.1 Negation

Remember that a `SubstP` is a mapping from variable names to either `One x` or `NotIn xs`. It gives a set of possible values for each variable. We now define how to negate a `SubstP`, to give something which represents the complement of that set.

An ordinary `SubstP` can be thought of as a conjunction of variable assignments. For example `[("x", One 3), ("y", One 4)]` specifies `x` has value 3, ‘and’ `y` has value 4. The negation of a `SubstP` can be represented as a disjunction of variable assignments; the negation of the above `SubstP` is `[("x", NotIn [3]), ("y", NotIn [4])]`, saying `x` is not 3, ‘or’ `y` is not 4. Clearly the set of substitutions represented by this disjunction is the complement of those represented by the original `SubstP`. We give it the data type

```
type Disj α = [(Var, ValueP α)]
```

which is the same physical type as a `SubstP`, but the meaning is different.

To define the negation function which turns a `SubstP` into a `Disj`, we first define complement on `ValuePs`. We would like to turn a single `ValueP` into something representing its complement; a `ValueP` represents a set of values, and we want something representing everything not in that set. Looking at the

definition of the `ValueP` type, it's clearly not possible, in general, to make a single `ValueP` which is the complement of a given `SubstP`. But we can do it if we make the result be a set of `SubstPs`, and interpret them as a disjunction.

```
negate_valuep :: ValueP α → [ValueP α]
negate_valuep (One x)    = [ NotIn [ x ] ]
negate_valuep (NotIn ls) = map One ls
```

For example the complement of `NotIn [red, blue]` is the list `[One red, One blue]`. Similarly, we can try to get the 'complement' of a `(Var, ValueP)` pair as a list of such pairs:

```
negate_pair :: (Var, ValueP a) → [ (Var, ValueP a) ]
negate_pair (var, val) = map (pair (const var, id)) (negate_valuep val)
```

Using `negate_pair` we can define

```
substp_to_disj :: Eq α ⇒ SubstP α → Disj α
substp_to_disj = concat · map negate_pair
```

This takes each variable assignment in the input and turns it into a disjunctive list of `(var, val)` mappings for the complement of its original value. Then the results are concatenated, so that the result, interpreted as a disjunction, represents the negation of one `SubstP`.

But as mentioned above, we are not working with individual `SubstPs` but with lists of them, the meaning of each list being the disjunction of its `SubstP` elements. How can we negate such a list?

Well the negation of a disjunction of `SubstPs` can be written as:

$$\begin{aligned} & \neg(\sigma_0 \vee \sigma_1 \vee \dots) \\ &= \neg\sigma_0 \wedge \neg\sigma_1 \wedge \dots \end{aligned}$$

Now as shown above, the negation of each σ_i can be given as a disjunction of variable mappings $x_{ij} \mapsto a_{ij}$, so

$$= (x_{00} \mapsto a_{00} \vee x_{01} \mapsto a_{01} \vee \dots) \wedge (x_{10} \mapsto a_{10} \vee x_{11} \mapsto a_{11} \vee \dots) \wedge \dots$$

This is then the dual of our normal data structure: instead of a disjunction of `SubstPs`, we have a conjunction of `Disjs`.

Every substitution contained in this expression is contained in every individual `Disj`. To find all substitutions, we pick all combinations of one variable assignment from each `Disj`. Each such combination is potentially a `SubstP`, depending on whether it is consistent. If the different variable assignments chosen disagree on the value of some variable, then no substitution exists for that combination. Otherwise, it is converted to a `SubstP` and added to the result. The result is the disjunction of all `SubstPs` found.

As an example, take the list of `SubstPs`: `[[("x", One "red"), ("y", NotIn ["eyes"])], [("x", NotIn ["blue", "green"]), ("y", NotIn ["shoes"])]]`, and find its negation. Write out the allowed values as a table:

	x		y
0.	{red}	<i>and</i>	$S - \{\text{eyes}\}$
<i>or</i> 1.	$S - \{\text{blue, green}\}$	<i>and</i>	$S - \{\text{shoes}\}$

where S is the set of all strings. The meaning of this is the disjunction of **SubstPs** **0** and **1**; expressed in English, **x** can be ‘red’ and **y** anything except ‘eyes’, or **x** can be anything except ‘blue’ and ‘green’ with **y** anything but ‘shoes’. There are some assignments which are allowed by both rows, so there’s some redundancy, but that doesn’t matter.

To negate this table, we negate the individual **SubstPs**:

	x		y
0’.	$S - \{\text{red}\}$	<i>or</i>	{eyes}
<i>and</i> 1’.	{blue <i>or</i> green}	<i>or</i>	{shoes}

The meaning is now the conjunction of **0’** and **1’**, but each row is now a disjunction of variable assignments. Take an example assignment permitted by the first table—for example $\{\mathbf{x} \mapsto \text{red}, \mathbf{y} \mapsto \text{tie}\}$, which is included in **0**. See that it is not permitted by the negation: it is not included in **0’** because **0’** requires **x** to not be ‘red’ or **y** to be ‘eyes’, and the negation is the conjunction of **0’** and **1’**.

Conversely, pick an example assignment allowed by the negation, such as $\{\mathbf{x} \mapsto \text{purple}, \mathbf{y} \mapsto \text{shoes}\}$. This is included in **0’** by matching **x**, and in **1’** by matching **y**. But it is not included in the original disjunction of **SubstPs**, because it fails to match **0** on **x**, and fails to match **1** on **y**.

Now we have to convert this dual data structure back to a disjunction of **SubstPs**. To do that, we pick one possible variable assignment from each row—since each row is a disjunction—and ‘and’ them together into a **SubstP**. By taking all such combinations we cover all assignments allowed by the negation. The six possible combinations of taking one assignment from each row are called **A** to **F**:

	x		y
A.	$(S - \{\text{red}\})$ <i>and</i> {blue}		<i>anything</i>
B.	$(S - \{\text{red}\})$ <i>and</i> {green}		<i>anything</i>
C.	$S - \{\text{red}\}$		{shoes}
D.	{blue}		{eyes}
E.	{green}		{eyes}
F.	<i>anything</i>		{eyes} <i>and</i> {shoes}

This is close to the ordinary **SubstP**, but some of the rows give two or more **ValuePs** for the same variable, or *anything* for some variables. If *anything* is given for a variable, then we need not mention that variable in the output **SubstP**, we can just leave it unconstrained. And if several **ValuePs** are given for the same variable, we use `unify_substps` defined earlier. If they fail to unify, then that row is ‘impossible’ and is removed from the final **SubstP**. Applying this to the table above gives:

	x	y
A.	{blue}	
B.	{green}	
C.	$S - \{\text{red}\}$	{shoes}
D.	{blue}	{eyes}
E.	{green}	{eyes}

Note that row **F** was impossible, because it specified both {eyes} and {shoes} as the value for **y**.

Expressed in Haskell syntax, this is the `SubstP`: `[[("x", One "blue")], [("x", One "green")], [("x", NotIn ["red"]), ("y", One "shoes")], [("x", One "blue"), ("y", One "eyes")], [("x", One "green"), ("y", One "eyes")]]`.

The output has some redundancy—the rows not restricting **y** already contain all the assignments given by the rows mapping **y** to {eyes}. And to convert back to a `SubstP` required an exponential number of combinations to be gathered and each one tested. [21] gives a way of reducing the time complexity of negation, but it wasn't implemented in this project. Stripping out redundant rows could be done by calling `unify_substps` on pairs of rows, so that if two rows can be unified to a single row they can both be removed and replaced with their unification. But this too would take exponential time for a simple implementation (test all possible pairs of rows in the `SubstP` to see if their unification exists). Also, removing seemingly useless extra rows from the table will affect the query language, because it will affect the number of different results returned—see 10.8.

Having explained how to do negation, we now give its implementation in Haskell. Define

```

combos :: [[α]] → [[α]]
combos []      = [ [] ]
combos (l : rest) = concat (map (λ e → (map (e :) (combos rest))) l)

```

to take some lists, and return all combinations of one element from each list. Now we need a function to do the normalizing process above, to turn a simple list of `(Var, ValueP)` pairs into a `SubstP` which has exactly one `ValueP` for each variable mentioned. Well for that we can use `extend_substp` defined earlier:

```

list_to_substp :: Eq α ⇒ [(Var, ValueP α)] → Maybe (SubstP α)
list_to_substp = foldr_maybe extend_substp []

```

In other words, to take a list of pairs and attempt to combine them into a single `SubstP`, we can start with the empty list and try to add the pairs one at a time. The result might be `Nothing`, if some of the pairs conflict.

Then our function to negate a set of `SubstPs` is:

```

negate_substps :: Eq α ⇒ [SubstP α] → [SubstP α]
negate_substps = catMaybes · map list_to_substp · combos · map substp_to_disj

```

This turns each `SubstP` into a `Disj` representing its complement, then picks all combinations of one variable assignment from each `Disj`, and tries to turn each one into a new `SubstP`. The standard library function `catMaybes` takes a list of `Maybe α` to a list of `α` by weeding out all `Nothing` elements.

7.2.2 Existential quantification

The meaning of $\exists \mathbf{x}.\varphi$, expressed in terms of the set of ρ needed to make a tree satisfy the formula, is

$$\llbracket T, \exists \mathbf{x}.\varphi \rrbracket = \llbracket T, \varphi \rrbracket - \{\mathbf{x} \mapsto ?\}$$

in other words the same set as φ , but with the variable \mathbf{x} erased from the results. To implement this we need a function to remove a variable from a set of `SubstPs`:

```
delete_from_substp :: Var → SubstP α → SubstP α
delete_from_substp v = filter ((≠ v) · fst)
delete_from_substp_set v = map (delete_from_substp v)
```

7.2.3 Summary

We have a data structure capable of representing possibly infinite sets of ρ in finite space, and functions for \neg and \wedge . If we take the semantics of logical formulæ—normally defined by giving a set of trees T for a formula φ and substitution ρ —and reexpress them as the set of ρ for a formula φ and tree T , then most components of the logic can be defined as basic one-element sets of substitutions, or compositionally using \neg and \wedge . (The structural formulæ look at the shape of the tree and give either the empty set, or a set built up with \wedge from one-element sets, and existential quantification can be implemented by removing a variable from a set.) Therefore, this set of functions is sufficient to implement most of the logic. However, we have ignored one kind of formulæ: equality tests on variables!

7.3 Final version: store (in)equality info too

An equality test $\alpha_1 = \alpha_2$ asserts that the two α_i are equal, and the α_i could each be either a constant or a variable. An assertion that two constants are equal is easy to deal with, it is either true or false. If we have $\mathbf{a} = \text{value}$, where \mathbf{a} is a variable and ‘value’ is a constant, then the set of substitutions that causes this to match is simply $\{\mathbf{a} \mapsto \text{value}\}$. An assertion that two variables are equal is where it gets interesting.

So far we have mapped variables to either ‘one value’, or a set of ‘not these values’. But in both cases the `SubstP` object just associates variables with values. It does not associate variables with other variables. What should be the set of `SubstPs` returned by $\mathbf{a} = \mathbf{b}$?

The implementation of [21] has the restriction that such variable equality tests are not allowed, although the straightforward case of equality between one variable and a value is handled. But our implementation does handle general equality tests, and in this section we explain how. Please skip to the next chapter if you feel that the apparatus for dealing with ordinary lists of `SubstPs` is complicated enough, and you don't mind sidestepping the issue of equality tests.

We could extend the idea of a `ValueP` so that as well as denoting a set of values, it could denote a set of variables. So perhaps `NotIn [3, 4, y]` could mean 'any value except 3, 4, and whatever `y` happens to be'. This sounds general enough for any combination of \neg and \wedge applied to variable equality. But it doesn't fit well into the structure we've outlined so far, because `agree` can't be defined as neatly as before. Consider the agreement of values `x` (which means 'whatever `x` holds') and `5`. There is no single `ValueP` or even set of `ValuePs` which holds the information contained in both these.

Instead, we store variable-equality information side by side with the mapping from variables to `ValuePs`. We define a new structure called a `SubstPV` which combines a `SubstP` with a list of equal and not-equal relationships between variables. Many of the concepts and steps are the same as before, so this section is presented a bit faster.

7.3.1 Absorbing (in)equality info into sets of values

First revisit the `agree` function. Currently it takes two `ValuePs` and returns a new `ValueP` which combines the information from both (or fails with `Nothing`). But a `ValueP` represents a set of values, and in fact the set returned is not enough to guarantee that two values picked from it will be equal. Take the cofinite sets `NotIn [4]` and `NotIn [5]`. The agreement of these will be `NotIn [4, 5]`, which is fine as far as it goes. But you could pick two values from that set which would not be equal. On the other hand, calling `agree` to implement an equality constraint does not always lose information. If you have the sets `One 5` and `NotIn [6]`, then the agreement of these is `One 5`, and any two elements drawn from that set must be equal.

We define `agree'` to act like `agree`, except that it returns an additional Boolean parameter saying whether the equality information has been absorbed into the new `ValueP` returned.

```

agree' :: ValueP α → ValueP α → Maybe (ValueP α, Bool)
agree' (One x) (One y)   | x == y = Just (One x, True)
                        | otherwise = Nothing
agree' (One x) (NotIn l) | x ∉ l = Just (One x, True)
                        | otherwise = Nothing
agree' (NotIn l) (One x) = agree' (One x) (NotIn l)
agree' (NotIn l) (NotIn l') = Just (NotIn (union l l'), False)

```

It's clear that `agree'` is related to equality, in some way. We also need an equivalent function, `disagree`, related to inequality. Later we'll see how these two are used.

Define `disagree` as a function which takes two `ValuePs`, and if possible returns two more which give a subset of the original values, such that if you pick one value from each subset the two values picked will differ. Except that sometimes this is not possible, so there is a Boolean flag returned to say whether the ‘not equal’ information has been absorbed into the new pair of `ValuePs`.

```
disagree :: Eq α ⇒ ValueP α → ValueP α → Maybe (ValueP α, ValueP α, Bool)
disagree (One x) (One y) | x ≠ y = Just (One x, One y, True)
                        | otherwise = Nothing
disagree (One x) (NotIn l) | x ∈ l = Just (One x, NotIn l, True)
                        | otherwise = Just (One x, NotIn (x : l), True)
disagree a@(NotIn _) b@(One _) = do
    (b', a', absorbed) ← disagree b a
    return (a', b', absorbed)
disagree (NotIn l) (NotIn l') = Just (NotIn l, NotIn l', False)
```

For example, `disagree (One 5) (NotIn [6, 7]) = (One 5, NotIn [5, 6, 7], True)`. If you pick a value that is 5, and pick another value that is not 5, 6, or 7, then the two values must differ. Because the two sets are disjoint the Boolean value returned is `True`. But `disagree (NotIn [0]) (NotIn [1, 2])` gives back the same sets of values. It is not possible to find a pair of `ValuePs` which give all pairs of different values from the original pair of `ValuePs`, and only those pairs of values. So the Boolean value returned is `False`; the information that the two values must be different has not been fully absorbed into the new sets returned. As you would expect, if the two sets cannot be made to disagree—if they both specify the same single value—then the result is `Nothing`.

It seems that `disagree` is in some way dual or opposite to `agree`, but there wasn’t time to investigate it formally. One difference between them is that for agreement, you get a single `ValueP` returned which tries to combine the information from both arguments, but for disagreement two different `ValuePs` are returned.

7.3.2 Storing a list of variable relationships

The possible relationships between variables which we want to keep track of are sets of just two assertions: ‘these two variables are equal’, and ‘these two variables are not equal’. We define `data Rel = Equal | NotEq`—just a renamed Boolean type—to distinguish these two cases.

Then `data VarRel = VarRel Rel Var Var` is a piece of information giving a relationship between two variables. For example,

```
VarRel Equal "x" "y"  means  the value of x equals the value of y;  
VarRel NotEq "a" "b"  means  the value of a differs from the value of b.
```

If we have such a `VarRel`, and two `ValuePs` for the variables mentioned, then `apply_rel_to_valueps` attempts to give two new `ValuePs` taking account of the information. For example suppose we have `VarRel Equal "x" "y"` as above, and we also happen to know that `x` has values in `One "pig"` while `y`’s possible

values are the set `NotIn ["dog"]`. Given the information that `x` and `y` are equal, can we narrow down the sets of values assigned to them? This is what `apply_rel_to_valueps` tries to do.

```

apply_rel_to_valueps :: Eq α =>
  Rel → ValueP α → ValueP α → Maybe (ValueP α, ValueP α, Bool)
  apply_rel_to_valueps Equal v0 v1 = do
    (v', complete) ← agree' v0 v1
    return (v', v', complete)
  apply_rel_to_valueps NotEq v0 v1 = disagree v0 v1

```

The return value is `Just (v0', v1', True)` if the relation was absorbed into the allowable values, `Just (v0', v1', False)` if the relation was not fully absorbed, and `Nothing` if the relation cannot be satisfied with the sets passed in. With the above example, `apply_rel_to_valueps Equal (One "pig") (NotIn ["dog"])` returns `Just (One "pig", One "pig", True)`. That means the equality information was fully absorbed (`True`), and the new values for `x` and `y` are `One "pig"` and `One "pig"` respectively. Since the information said that the two variables were equal, it's no surprise that the two new sets of values are the same.

This function is clearly just a wrapper for `agree'` and `disagree`, calling whichever is appropriate. Note that the ordering of `v0` and `v1` does not matter, if you swap them then they are just swapped in the result. This is to be expected since equality and inequality are symmetric.

A list of `VarRel`s will store the complete set of relationships between variables. Now there are various properties of `=` and `≠`, and it's a design decision how many of these properties to store explicitly in the list of relationships, and how many to infer. For example, if we wish to store the information `x=y` and `y=z`, should the list also store the implied constraints like `y=x`, `x=z` and so on? For this implementation we will use a maximal set of constraints, that is, closed under symmetry, under transitivity of `=`, and under substitution of equals for equals.

The function `consequences` takes a set of variable relationships and a new relationship to add, and returns a list of some extra relationships which would need to be added to keep the set maximal. In other words, the extra things which you need to check if you want to add a new constraint.

```

consequences :: VarRel → [VarRel] → Maybe [VarRel]
consequences vr@(VarRel Equal v0 v1) vrs
  | (VarRel NotEq v0 v1) ∈ vrs = Nothing
  | vr ∈ vrs                    = Just []
  | otherwise                    = Just (union
    [mkvarrel r v0 v'' | VarRel r v' v'' ← vrs, v' = v1 ∧ v'' ≠ v0]
    [mkvarrel r v1 v'' | VarRel r v' v'' ← vrs, v' = v0 ∧ v'' ≠ v1])
consequences vr@(VarRel NotEq v0 v1) vrs
  | (VarRel Equal v0 v1) ∈ vrs = Nothing
  | vr ∈ vrs                    = Just []
  | otherwise                    = Just (union
    [mkvarrel NotEq v0 v'' | VarRel Equal v' v'' ← vrs, v' = v1]

```

```
[mkvarrel NotEq v1 v' | VarRel Equal v' v'' ← vrs, v' = v0])
```

All this function does is compute one level of transitivity and equals-for-equals substitution. It does not do full closure, for that you would need to call it recursively with each extra item returned. Also the function does not give every extra relationship to add, because it doesn't consider symmetry of = and \neq . It assumes that the existing set is closed under symmetry, but the extra relationships returned are given only one way round. We'll see below how the results from consequences are used.

7.3.3 Representing a set of substitutions

Previously, to represent a set of substitutions we took a disjunction of `SubstPs`, where each `SubstP` itself represented a possibly-infinite set of ρ . We continue with this technique, but now we use a disjunction of `SubstPVs`, which are defined as:

```
data SubstPV a = SubstPV (SubstP a) [VarRel]
```

That is, the new structure consists of an old-style `SubstP` associating variables with `ValuePs`, and additionally a list of `VarRels` associating variables with other variables. This gives us a data structure which represents the sets of ρ we want to handle.

For most parts of the logic the semantics is the same as before, with an empty list of variable relationships on the end. For example the semantics of $\mathbf{a} \mapsto [\text{nil}]$ is `[SubstPV [("a", a)] []]` if the tree has the shape $\mathbf{a} \mapsto [\text{nil}]$, and `[]` otherwise. The equality tests are what's new, and in particular equality between two variables. We define the semantics of $\mathbf{x} = \mathbf{y}$ as the set of substitutions represented by `[SubstPV [] [VarRel Equal "x" "y", VarRel Equal "y" "x"]]`. Note how the list of relationships has to be maximal.

Now we have to define \cap and \neg on sets of `SubstPVs`, as we did before with sets of `SubstVs`. The method is much the same: define how to break down a `SubstPV` into individual pieces of information, and how to merge a new piece of information into an existing `SubstPV` (giving `Nothing` if the result is inconsistent). Then unification on individual `SubstPVs` works by breaking down one `SubstPV` into pieces and merging them into the other, and unification on sets of `SubstPVs` can be done by picking all pairs consisting of one element from each set, trying to unify each pair and if the result is not `Nothing` including it in the output set. Negation of sets is done by defining a dual representation which holds a conjunction of disjunctions of pieces of information, and picking all combinations taking one piece from each disjunction, and for those combinations which are consistent including them in the result.

7.3.4 Unification

The 'piece of information' we work with is called a `VarInfo`. It holds one assertion about a variable—either that it has a certain set of possible values, or that it is equal or unequal to another variable.

```
data VarInfo  $\alpha$  = Val (Var, ValueP  $\alpha$ ) | VR VarRel
```

Breaking a `SubstPV` into a conjunction of these is simple:

```
substpv_to_varinfos :: SubstPV  $\alpha$  → [VarInfo  $\alpha$ ]
substpv_to_varinfos (SubstPV s vrs) = (map Val s) ++ (map VR vrs)
```

Then there are two cases for merging in this extra information: when the information is a `(Var, ValueP)` pair and when it's a `VarRel` relationship between two variables. First we define how to merge a `VarRel` into a `SubstPV`.

```
merge_varrel_into_substpv :: Eq  $\alpha$  ⇒ VarRel → SubstPV  $\alpha$  → Maybe (SubstPV  $\alpha$ )
merge_varrel_into_substpv vr@(VarRel rel var0 var1) (SubstPV s vrs) = do
  conseq ← consequences vr vrs
  (val0, s') ← return (from_substp' var0 s)
  (val1, s'') ← return (from_substp' var1 s')
  (val0', val1', absorbed) ← apply_rel_to_valueps rel val0 val1
  s''' ← return ((var0, val0') : (var1, val1') : s'')
  if absorbed
    then return (SubstPV s''' vrs)
    else (foldr_maybe merge_varrel_into_substpv
      (SubstPV s''' (nub (vr : (flip_varrel vr) : vrs)))
      conseq)
```

The function looks scary but it is quite simple. The `←` lines are like let-assignments, but assignments that might 'fail'. If one of the expressions on the rhs of a `←` returns `Nothing`, then this whole function returns `Nothing`. You can think of the calls to `consequences` and `apply_rel_to_valueps` as calls which might go wrong—the new variable relationship might be obviously inconsistent with the existing relationships, or it might be inconsistent with the possible values given to variables. The calls prefixed with `return` are 'pure' and cannot fail. The function works as follows:

First, find all the 'consequences'—all the extra variable relationships which would come about if this relationship were added. That means taking account of properties of `=` and `≠` as discussed above. Next, find the current `ValuePs` associated with `var0` and `var1`. Attempt to apply the relationship to these two, and set `s'''` to the new `SubstP` with the new pair of `ValuePs`. If the equality or inequality information was fully absorbed into the new sets of values, the function need not bother to add it to the list of `VarRel` constraints, and can just return `s'''` and the original list of constraints.

On the other hand, if the information could not be fully absorbed then the constraints have to be updated. We know that at least `vr` itself is consistent with the variable assignments we have, so add it and its symmetric twin to the list. Now we have the consequences to check; the `foldr_maybe` is one recursive call for each consequence, trying to merge *it* into the `SubstPV` object. If any of these recursive calls fails then, by the way `foldr_maybe` works, the whole function fails.

(You may object that even if the new variable relationship (or constraint) `vr` is fully absorbed into the sets of allowable values, surely we still need to check

its ‘consequences’. But the new possible values for the two affected variables are a *subset* of the previous possible values, so all the old constraints remain satisfiable. And we know that the new constraint *vr* always holds, because the new set of possible values guarantees this. Therefore, we know that for any value-assignment satisfying the old constraints, the new constraint will also be satisfied, and thus so will the consequences of the new and old constraints. We don’t need to check them separately!)

Two helper functions are needed, `from_substp'` which looks up a variable in a `SubstP` but returns the unconstrained value (`NotIn []`) if the variable was not found, and `flip_varrel` to give the symmetric twin of a variable relationship (`y=x` from `x=y`, etc.).

```
from_substp' :: Var → SubstP α → (ValueP α, SubstP α)
from_substp' var s = case from_substp var s of
  Found (vp, s') → (vp, s')
  NotFound → (NotIn [], s)
flip_varrel (VarRel r v0 v1) = VarRel r v1 v0
```

That covers how to merge a new variable relationship into a `SubstPV`. The other case is merging a variable assignment. We already have code to merge a new variable assignment into a `SubstP`, so it can be extended as

```
merge_val_into_substp :: Eq α ⇒
  (Var, ValueP α) → SubstPV α → Maybe (SubstPV α)
merge_val_into_substp (var, vp) (SubstPV s vrs) = do
  s' ← extend_substp (var, vp) s
  foldr_maybe merge_varrel_into_substp (SubstPV s' []) vrs
```

This takes the brute force approach: first merge the new assignment into the `SubstP`, and then go through every variable relationship and re-add it to check it is still consistent.

Having functions to handle both cases, it’s possible to define `extend_substp` to merge an arbitrary `VarInfo` piece of information into a `SubstPV`, and thus to convert a list of `VarInfo` into a single `SubstPV` (or `Nothing`) by merging them one by one, and to unify two `SubstPV`s by a similar technique. Then unifying two sets of `SubstPV`s consists of taking all possible consistent pairs of one `SubstPV` from each set, as we did for the sets of `SubstP`s earlier.

```
extend_substp :: Eq α ⇒ VarInfo α → SubstPV α → Maybe (SubstPV α)
extend_substp (VR vr) spv      = merge_varrel_into_substp vr spv
extend_substp (Val (var, vp)) spv = merge_val_into_substp (var, vp) spv
list_to_substp :: Eq α ⇒ [VarInfo α] → Maybe (SubstPV α)
list_to_substp = foldr_maybe extend_substp (SubstPV [] [])
unify_substps :: Eq α ⇒ SubstPV α → SubstPV α → Maybe (SubstPV α)
unify_substps spv = (foldr_maybe extend_substp spv) · substpv_to_varinfos
unify_substp_sets :: Eq α ⇒ [SubstPV α] → [SubstPV α] → [SubstPV α]
unify_substp_sets s s' = catMaybes (map (uncurry unify_substps) (pairs s s'))
```

7.3.5 Negation

If we can negate a `VarInfo` piece of information, then we can convert each `SubstPV` to a conjunction of `VarInfos`, negate each one, and take the result as a disjunction which represents the negation of the original `SubstPV`. Then negation of sets of `SubstPVs` uses the picking-all-combinations technique explained earlier.

Negating an equal or not-equal relationship between variables is easy! And we already know how to negate `(Var, ValueP)` pairs. So we can negate `VarInfo` objects and everything else is familiar.

```
negate_varinfo :: VarInfo α → [VarInfo α]
negate_varinfo (Val p)           = map Val (negate_pair p)
negate_varinfo (VR (VarRel Equal v0 v1)) = [ VR (VarRel NotEq v0 v1) ]
negate_varinfo (VR (VarRel NotEq v0 v1)) = [ VR (VarRel Equal v0 v1) ]

type DisjV α = [VarInfo α]
substpv_to_disjv :: Eq α ⇒ SubstPV α → DisjV α
substpv_to_disjv = concat · map negate_varinfo · substpv_to_varinfos

negate_substpvs :: Eq α ⇒ [SubstPV α] → [SubstPV α]
negate_substpvs =
    catMaybes · map list_to_substp · combos · map substpv_to_disjv
```

Existential quantification

As before, the result of $\exists \mathbf{x}.\varphi$ is obtained by removing all reference to \mathbf{x} from the result of φ .

```
delete_from_varrels :: Var → [VarRel] → [VarRel]
delete_from_varrels v = filter (λ (VarRel _ v0 v1) → (v0 ≠ v ∧ v1 ≠ v))

delete_from_substp :: Var → SubstPV α → SubstPV α
delete_from_substp v (SubstPV s vrs) =
    SubstPV (delete_from_substp v s) (delete_from_varrels v vrs)
```

7.3.6 Summary

Although a list of `SubstP` objects is sufficient to represent most of the sets of substitutions we want to handle, it can only give associations between variables and values, and cannot relate one variable to another. To handle assertions that two variables are equal or unequal, we define a `SubstPV` which holds a `SubstP` and additionally holds a list of relationships between variables. When adding a new relationship between two variables to the list, we must check that it is consistent with the existing relationships; that it is consistent with the set of values allowed for those two variables; and finally, that any additional relationships following from the new one and the existing list can also be added successfully.

We define functions to see whether a piece of equal or not-equal information is consistent with two sets of allowable values. The result is either failure, or

two new sets which are subsets of the original two and take account of the extra information. Sometimes it is possible for the extra information to be fully ‘absorbed’ into the new sets, so that it is no longer necessary to store it separately. At other times the two new sets, being both infinite, cannot guarantee the property that a pair of elements picked from them will satisfy the equal or not-equal constraint, so that information must be retained.

Since we already know how to merge a plain variable assignment into a `SubstP`, we can continue doing that with `SubstPVs`. The difference is we must additionally check that the new variable-to-possible-values list is still consistent with the equal and not-equal constraints on variables.

Unification and negation are very similar to before: we can break a `SubstPV` into a conjunction of pieces of information, and merge them into another `SubstPV` to unify, with unification on sets of `SubstPV` picking all pairs and trying to unify each pair. Negation of a set works by splitting each `SubstPV` into pieces of information, negating each piece, and then picking all combinations of one piece from each `SubstPV` to include in the result.

7.4 Technical notes on the implementation

We’ve given a code walkthrough of the machinery which manipulates sets of substitutions. This is defined in the libraries `valuep.hs` and `subst.hs`. But its application to the trees requires some more coding.

To start with, a tree formula supports three kinds of variables, for labels, addresses and leaf data. These can be of different types—usually we take labels and leaf data to be strings, and addresses to be integers. If we want to make our tree-querying code fully general, we must parameterize it by three types, and have separate `SubstPV` structures for the three kinds of variable. Also there are tree variables to consider, so that’s four kinds. Instead of keeping results as a list of `SubstPV`, we use a list of tuples, each tuple being four separate `SubstPVs`, one for each kind of variable. Then all the operations must be ‘extended’ to work with these tuples. For example, instead of turning a single `SubstPV` into its complement `DisjV`, instead we must apply the same operation to each part of the 4-tuple. As you can imagine, this code is uninteresting ‘boilerplate’ and not worth covering in this report.

(This illustrates a weakness in the Haskell language. When tree variables were added, all the code needed to be changed from working on 3-tuples to working on 4-tuples. With a templating system like that in C++ it is possible to generate at compile time code to work on tuples of a specified size and shape.)

The code to match against trees is not completely uninteresting, but it does follow automatically from the structure of trees and the query semantics defined earlier. The match function has, essentially, the type

$$\text{match} :: \text{RVs } \alpha \ \xi \ \delta \rightarrow \text{Logic } \alpha \ \xi \ \delta \rightarrow \text{Tree } \alpha \ \xi \ \delta \rightarrow [\text{ST } \alpha \ \xi \ \delta]$$

where `RVs` is a set of recursion variables (a map from variable names to formulae), and α , ξ , and δ are the types chosen for labels, addresses and leaf

data in the tree, and ST is a 4-tuple of $SubstPVs$ for label, address, data and tree variables. To give a flavour of how it works, here is the case for matching leaf data:

```

match rvs (LLeafData d) (LeafData d') = case (match_vl d d') of
  Nothing → []
  Just sd → [ (empty_substpv, empty_substpv, sd, empty_substpv) ]
match rvs (LLeafData _) _ = []

```

Here the first equation is for matching the $LLeafData$ formula against a tree which has the shape $(LeafData d')$. $empty_substpv$ is defined as $(SubstPV [] [])$, the unspecified $SubstPV$ which allows all possible substitutions. Perhaps ‘empty’ is exactly the wrong word to use! But it is empty in that it gives no information. The function $match_vl$ matches a ‘variable-or-constant’ against a constant, returning either $Nothing$ (if two constants do not match), or the $SubstPV$ which assigns the constant to the variable. Here it is called to match a data variable-or-constant (from the logical formula) against a piece of leaf data (from the tree). The result becomes the third part of the 4-tuple, the set of substitutions for data variables.

Matching the $LLeafData$ formula against anything else fails, that is, it returns the empty list.

The other structural cases of $match$ do similar matching against the tree, calling $match_vl$ to check variables-or-constants in the formula against the data found in the tree. Parallel composition tries all ways to split a parallel composition in two (which is simply finding sublists of a list), and for each splitting calls $match$ recursively on each half of the tree and tries to unify the two results. Then all the lists of STs generated are concatenated together, so the result is the union of results from all possible splittings. (Duplicates are not removed when concatenating, so if there is more than one splitting which returns a given substitution, that substitution will appear more than once in the result. See 10.8.)

The ‘and’ and ‘not’ cases use unification and negation on lists of ST 4-tuples, as explained above. Existential quantification removes the given variable from the list of 4-tuples, and equality tests return a singleton 4-tuple which contains only the information that two variables (of the same kind) are equal.

Recursion is in fact rather simple to implement. Each call to $match$ passes around an RVs object which maps variable names to formulae; then $\mu\mathbf{R}.\varphi$ just adds $\{\mathbf{R} \mapsto \varphi\}$ to the list and evaluates φ . A recursive call is just a lookup of the variable name.

If you want more detail than this brief summary, please look at the code in `tree_query.hs`. It is not very interesting because most of the hard work is done by the `subst.hs` library, which was explained earlier in this chapter.

Chapter 8

Query language

We have a way of matching a formula against a tree and finding the set of substitutions which make it match. Until now it has been implicit that these substitutions constitute the ‘answer’ to the question asked. But a useful query language should return the answer in a more useful form than just a set. It should be possible to get the result in a form that’s just as flexible as the input data—that is, a tree.

We can do this by creating an output template, which is the same shape as a tree but contains variables as placeholders. Instantiating the output template by replacing variables with their value in a particular substitution will give us our result. But very often, there is more than one set of substitutions produced. How can we handle all of them?

To start with, we can restrict consideration to finite sets of substitutions. Whatever tricks we use to keep track of infinite sets within the query engine, it’s not useful for the end result to be an infinitely large tree or an infinite set of trees. Since the data model itself (see 5.1) doesn’t have any provision for representing infinite trees, we must stick with finite trees if we want to produce output in the same format as the input data. This sounds very restrictive, as if all the work on implementing fully-general negation were wasted. But you can still work with infinite sets *inside* the logical formula, it’s just the output stage that is restricted. Later on we discuss whether the restriction to finite output could be lifted, but for the time being we build a query language taking finite trees to finite trees.

But even if the number of substitutions is finite, it could still be more than one. Following [1] we build one result tree for each substitution found, and use parallel composition $|$ to join them together into a single result.

But our data model is not quite the same as in [1]. There, any two trees could be joined together with $|$; whereas in our model, a tree can be either a set of branches *or* some leaf data, so leaf data cannot sit in parallel with branches. This means that in general you cannot always join two trees together with $|$, because one of them might be leaf data. The answer we adopt is to check at run time that none of the results has that shape; later in Evaluation we discuss

how a different data model might avoid the problem.

Before describing the query language that was implemented, we first present a simple, cut-down variant to introduce some of the ideas. Then we demonstrate why this simple language is inadequate, and introduce the full version. Also, we at first ignore the question of unique addresses and graphical links, adding them later.

8.1 A simple query language

Define a ‘tree template’ as:

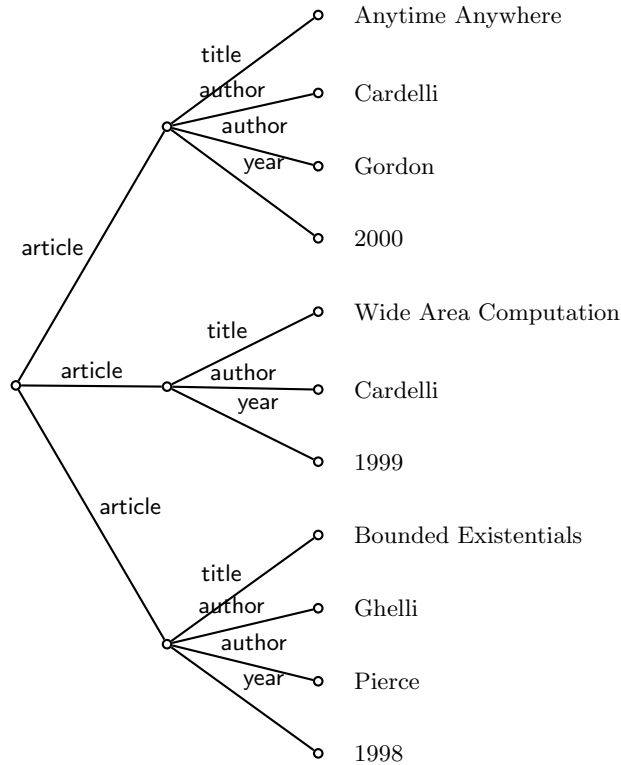
$M^T ::=$	δ	data expression
	B^T	branches to subtrees
$B^T ::=$	nil	empty
	$\alpha \mapsto \xi[M^T]$	branch, label expression α , address expression ξ
	$\alpha @ \xi$	graphical link, as above
	$B^T B^T$	parallel composition

where a data expression δ is either a data variable or some constant leaf data; similarly for label expressions α and address expressions ξ . It’s clear how, given a particular substitution ρ , we can turn a tree template M^T into a finished tree $\rho(M^T)$ by ‘filling in’ all the variables with their assigned values.

To build a basic query language, define

$$\text{from } M \models \varphi \text{ select } B^T \stackrel{def}{=} \mathbf{Parr}\{\rho(B^T) : M \models \rho \varphi\}$$

where $\mathbf{Parr} s$ is simply the parallel composition of every branch in s . This allows us to capture all the information in the substitutions found, and return it in a tree shape. For example, if we adapt an example from [1]:



then the following query builds a result tree associating authors with years when they published:

```

from   T ⊨ (article ↦ [author ↦ [a] | year ↦ [y] | true] | true
select pair ↦ [person ↦ [a] | published ↦ [y]]

```

The result of the query is the output template instantiated once for each substitution making the formula match; that is,

```

pair ↦ [person ↦ [Cardelli] | year ↦ [2000]]
| pair ↦ [person ↦ [Cardelli] | year ↦ [1999]]
| pair ↦ [person ↦ [Gordon] | year ↦ [2000]]
| pair ↦ [person ↦ [Ghelli] | year ↦ [1998]]
| pair ↦ [person ↦ [Pierce] | year ↦ [1998]]

```

Well, it certainly returned all the information, but perhaps not in the format you'd most prefer. Note for example the two separate subtrees for 'Cardelli' giving two different years. What if we wanted to have an author and all of his or her publication dates in a single subtree?

Inadequacy of the simple query language

It's easy to show that this basic query language cannot return a result with all an author's dates in a single subtree. Remember that it works by finding all substitutions ρ such that $T \models^\rho \varphi$, and then for each substitution *separately*

instantiating the output template. Thus, if there are two different ways for \mathbf{y} and \mathbf{a} to bind, there must be two separate subtrees in the output, even if it happens to be the same value for \mathbf{a} both times.

8.2 General *from/select* querying

The extra feature we need is the ability to make *nested* queries or ‘subqueries’. The new query language defines a query as

$Q ::=$	$from\ Q \models \varphi\ select\ Q'$	subquery
	\mathbf{t}	tree variable
	nil	empty
	$Q \mid Q'$	composition
	$\alpha \mapsto [Q]$	branch
	δ	data

Apart from the *from/select* and the tree variables \mathbf{t} , the structure of queries is similar to that of trees. One difference is that while the grammar for trees is defined with two productions M and B to ensure that ‘data’ appears only at the leaves of the tree, the grammar for Q doesn’t have this restriction. It’s not needed because we are doing the check at run time. In the Evaluation section we discuss this further.

The semantics of a query Q is defined relative to a substitution ρ . The query evaluator $\llbracket Q \rrbracket_\rho$ is a function which takes a query and a substitution and returns a tree. It is defined as follows:

$\llbracket from\ Q \models \varphi\ select\ Q' \rrbracket_\rho$	$=$	$\mathbf{Parr}\ \{\llbracket Q' \rrbracket_{\rho'} : \llbracket Q \rrbracket_\rho \models \rho' \varphi \wedge \rho \sqsubseteq \rho'\}$
$\llbracket \mathbf{t} \rrbracket_\rho$	$=$	$\rho(\mathbf{t})$
$\llbracket nil \rrbracket_\rho$	$=$	nil
$\llbracket Q \mid Q' \rrbracket_\rho$	$=$	$\llbracket Q \rrbracket_\rho \mid \llbracket Q' \rrbracket_\rho$
$\llbracket \alpha \mapsto [Q] \rrbracket_\rho$	$=$	$\rho(\alpha) \mapsto \llbracket [Q] \rrbracket_\rho$
$\llbracket \delta \rrbracket_\rho$	$=$	$\rho(\delta)$

(We assume that ρ is the identity on constants.) All of the cases are simple, except that for *from/select* which works as follows. The ‘output’ query Q' is evaluated for each substitution ρ' which extends the input substitution ρ and which allows $\llbracket Q \rrbracket_\rho$ to satisfy φ . To say that ρ' extends ρ , written as $\rho \sqsubseteq \rho'$, means that the two substitutions agree on the values of variables mentioned in ρ , although ρ' may give its own valuation for variables not mentioned in ρ . This is the semantics defined in [1].

Intuitively, we want to first evaluate the subquery Q to find the tree it represents, and then find all substitutions letting this tree satisfy φ and build a result from these. Because Q' may have new variables not mentioned in Q or φ —and hence not included in the set of substitutions letting those two match—we need to allow new variables to be added to ρ' , but we don’t let it change the value of variables that have already been bound.

To make clear the restriction that the right hand side of a *from/select* cannot

be leaf data, we write

$$\begin{aligned} \mathbf{Parr} \emptyset &= \text{nil} \\ \mathbf{Parr} S \cup \{t\} &= t \mid (\mathbf{Parr} S), \text{ if } t \neq \delta, \\ &\text{undef, otherwise.} \end{aligned}$$

Leaf data or data variables may appear inside a query; just not directly on the rhs of a *from/select*.

With this query language we can build more flexible result trees as the result of a query. Returning to the problem of presenting an author with all his or her publication dates in a single subtree, the query

```
from T ⊢ article ↦ [author ↦ [a] | true] | true
select person ↦ [ name ↦ [a] |
  (from T ⊢ article ↦ [author ↦ [a] | year ↦ [y] | true] | true
   select year ↦ [y]) ]
```

first binds **a** to the name of an author and then, in a ‘person’ subtree constructed for that author, runs a subquery to find all the author’s dates of publication. The result is

```
person ↦ [name ↦ [Pierce] | year ↦ [1998]]
| person ↦ [name ↦ [Ghelli] | year ↦ [1998]]
| person ↦ [name ↦ [Gordon] | year ↦ [2000]]
| person ↦ [name ↦ [Cardelli] | year ↦ [1999] | year ↦ [2000]]
```

which gives a single ‘Cardelli’ subtree, as wanted.

8.3 Adding addresses and links

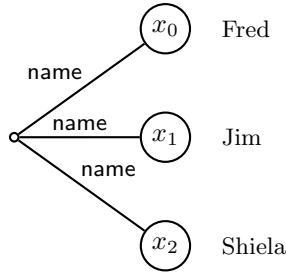
Remember that in our data model, each subtree must have a unique address. Only the root of the whole tree does not need an address. We normally omit writing the addresses when they are not important, and similarly in the logic section we defined $\alpha \mapsto [\varphi]$ as a shorthand for $\exists \mathbf{x}. \alpha \mapsto \mathbf{x}[\varphi]$ for formulæ where the address of the subtree is not important. But when it comes to building results in the query language we do have to worry about addresses and making sure each is unique.

To make the query language follow the data model, so that result trees can be constructed containing addresses and graphical links, we could remove the existing $\alpha \mapsto [Q]$ case and replace it with two new cases:

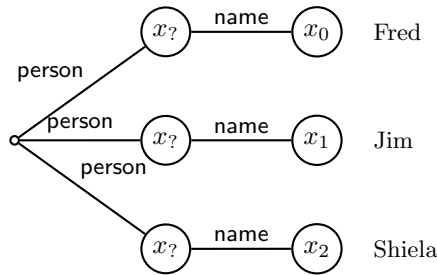
$$\begin{aligned} \llbracket \alpha \mapsto \xi[Q] \rrbracket_\rho &= \rho(\alpha) \mapsto \rho(\xi) \llbracket [Q] \rrbracket_\rho \\ \llbracket \alpha @ \xi \rrbracket_\rho &= \rho(\alpha) @ \rho(\xi) \end{aligned}$$

This would let us select addresses from the input tree, and include those same addresses in the output—in just the way that we can currently select labels or leaf data and use those in the output. But because addresses have a restriction that they must be globally unique, just selecting existing addresses may not be enough.

Consider a query to ‘restructure’ some data. Suppose we have the input tree



and we’d like to rearrange it to follow the same format used elsewhere in this report: where each person is a subtree reached by a ‘person’ arc from the root, and things like ‘name’, ‘age’ and ‘shoe size’ are branches within each person’s subtree. The tree we want to end up with should look something like:



If the result tree is to be valid, each address must be unique. So we need to find six unique addresses somehow. You could get x_0 , x_1 and x_2 from the input tree, but then how to generate addresses for the nodes marked $x_?$? In fact this is impossible, because in the query language so far the only way to select addresses is by matching them from the input. The input tree contains only three addresses, so there’s no way to generate the six we require.

It would be possible of course to hardcode some new addresses in the query, but that would be limited to a fixed number and in any case you’d have no guarantee that the addresses you hardcoded did not clash with ones that were selected from the input tree.

In order that the query language can be used to construct results which are bigger than the input—in terms of having more nodes and hence more unique addresses—it is necessary to add a new case:

$$\llbracket \alpha \mapsto [Q] \rrbracket_\rho = \rho(\alpha) \mapsto x[\llbracket Q \rrbracket_\rho], \text{ where } x \text{ is new}$$

This mirrors the addition to the logic of the plain $\alpha \mapsto [\varphi]$ query which ignores address information. You can choose to ignore addresses when matching a tree; and you can choose not to specify addresses when building up the result. In this case a fresh address will be chosen for you by the system.

8.3.1 Copying subtrees containing links

So far we have added some special treatment for addresses, but still the general way of querying is to match addresses as any other data item and to include them in the result in the same way that labels and leaf data are included. Similarly, a graphical link $\alpha@x$ is added to the result just by finding $\rho(\alpha)$ and $\rho(x)$. But if we consider the *meaning* of graphical links, arguably addresses and links to them should be treated differently from other kinds of data.

The first question is, should the query system enforce that graphical links in the result always point somewhere. Should the system make a check at runtime and warn if an $a@x$ link has been included, but the subtree with address x is nowhere in the result? The answer is definitely no. One of the important features of our data model is that it allows dangling pointers which point to addresses ‘outside’ the current tree.

Also, what should happen when a tree variable is included in the output template (that is, the right hand side of a *from/select* expression) and the value of that tree variable is a subtree containing graphical links? Should the system ‘follow’ the links to make sure that the destination of the links is also included in the output? For example, in the query

$$\begin{aligned} & \text{from } (\text{person} \mapsto x_0[\text{name} \mapsto [\text{Shem}] \mid \text{begat}@x_1] \\ & \quad \mid \text{person} \mapsto x_1[\text{name} \mapsto [\text{Arphaxad}] \mid \text{begat}@x_2]) \\ & \models \text{person} \mapsto [\mathbf{t} \wedge (\text{name} \mapsto [\text{Shem}] \mid \text{true})] \mid \text{true} \\ & \text{select found} \mapsto [\mathbf{t}] \end{aligned}$$

we are asking whether the destination of the $\text{begat}@x_1$ graphical link should be somehow included in the query result, just because it is pointed to by the tree bound to \mathbf{t} . Again, the answer must be no. Doing this kind of copying would be necessary, if the data model did not allow dangling pointers. Since the data model does allow graphical links which point to addresses ‘elsewhere’ (indeed, there was an example of such a dangling link in the input tree itself), we should assume that this was the intention of whoever wrote the query. It would be possible to write a different query which included the subtree pointed to, so we should allow the user the flexibility of choosing whether or not to ‘chase’ graphical links in this manner.

Now, what is the behaviour of the query language as currently defined? Well apart from the special $\alpha \mapsto [Q]$ case which generates a fresh address, the addresses x are just matched with variables; and then when building a result, a graphical link $\alpha@x$ is simply instantiated with values from ρ . Nothing special happens that depends on where $\rho(x)$ points. So the simple handling of graphical links has the behaviour we want—that the destination of a link is not included in the result unless it is fetched separately. Using the \diamond operator defined in the logic, the destination of a graphical link x can be found with $(\diamond(\exists \mathbf{a}.\mathbf{a} \mapsto x[\mathbf{t}])),$ which binds the tree variable \mathbf{t} to the subtree with address x . So there is no need for the query language to provide any special functionality here, except perhaps as a convenient shortcut.

Another thing to consider is including a particular subtree more than once in the output. Suppose that we wanted a query which would rewrite the

‘articles’ database discussed earlier to make each article appear twice:

```
from T ⊨ article ↦ [t] | true
select article ↦ [t] | publication ↦ [t]
```

This sounds like a useless thing to do, but perhaps it would be necessary if we have one program expecting to look under `article` branches, and another which looks under `publication`, and we want to make a single database usable by both. In any case, the query serves to illustrate copying a subtree to two places. We would like to support that because it is certainly possible in [1]’s query language.

Because of the requirement for globally unique addresses, the query written above will fail with a runtime error. The subtree `t` cannot simply be included in two different places, because the addresses it contains would also be duplicated. This is a limitation of the query system at present, and it shows how requiring unique addresses can cause problems.

You could argue that it’s not such a problem after all, because now that graphical links are available it should no longer be necessary to make copies of subtrees. The `publication` branches could instead be written as graphical links *pointing* to the article subtrees. That certainly seems more sensible; but it doesn’t change the fact that we do have a serious restriction on the query language which was not present in the query language defined in [1]. Copying subtrees could still be necessary—for example if you wanted to make a copy of an article and then change some information about the copy without affecting the original. In Evaluation we discuss possible extensions to the query language to overcome this problem.

8.4 Implementation of the query language

We define a data type for queries following the grammar for Q given above:

```
data Query α ξ δ = FromSelect (Query α ξ δ) (Logic α ξ δ) (Query α ξ δ)
                  | QTreeVar Var
                  |QParr [Query α ξ δ]
                  |QBranch (VC α) (VC ξ) (Query α ξ δ)
                  |QLink (VC α) (VC ξ)
                  |QLeafData (VC δ)
```

where `VC α` is either a variable name, or a constant of type α .

The function `make_parr` composes together a list of trees into a single tree. It is this function which does the runtime check that you are not trying to compose together leaf data. The function `lookup_treevar` looks up a tree variable in a substitution; it does a runtime check that the tree variable is not bound to an infinite set of trees. This corresponds to the restriction (shared by [21]) that the semantics of a query or subquery must be finite.

Now the query function can be defined:

```

query :: (Ord  $\alpha$ , Ord  $\xi$ , Ord  $\delta$ )  $\Rightarrow$  ST  $\alpha$   $\xi$   $\delta$   $\rightarrow$  Query  $\alpha$   $\xi$   $\delta$   $\rightarrow$  Tree  $\alpha$   $\xi$   $\delta$ 
query st (FromSelect q phi q') = let
  qr = query st q
  sts = match phi qr
  sts' = unify_st_sets sts [ st ]
  sts'' = nub sts' in
  make_parr (map ( $\lambda$  st'  $\rightarrow$  query st' q') sts'')
query st (QParr qs) = make_parr (map (query st) qs)
Other cases not shown.

```

The cases not shown are structural matching involving looking up a variable; or the case to look up a tree variable with `lookup_treevar`. The interesting case is that for `FromSelect`; it follows the semantics defined earlier. In particular, the call to `unify_st_sets sts [st]` is the way of enforcing that $\{st\} \sqsubseteq sts'$ and $sts \sqsubseteq sts'$.

The line `sts'' = nub sts'` removes duplicates from the set. This is important because the matching algorithm, as it is written, can return multiple copies of the same substitution when a formula can match in ‘more than one way’. This could lead to multiple copies of the same data being included in the result of a query. In the Evaluation we discuss whether this feature could actually be useful, and whether it has a well-defined semantics. The actual implementation differs from the code here in that a Boolean parameter chooses whether to eliminate duplicates or not. For the ‘vanilla’ form of the logic and query language, the duplicates must be removed.

(Another way to think about removing duplicates is to suppose that the parallel composition `|` in trees has a setlike behaviour, so that if there are two identical trees composed in parallel they are structurally congruent to just one. That would automatically remove the problem of selecting two duplicate trees next to each other, but it would not help when the same address is selected twice in completely different parts of the result. In our implementation, `|` constructs a multiset, not a set; the congruence $T | T \equiv T$ does *not* hold, and so the query language must be careful not to select two parallel copies of the same tree if it wishes to avoid duplicating addresses.)

Finally, for each substitution `st'` found we run the subquery `q'`, and then use `make_parr` to combine the results.

The parallel composition case is simply structural recursion, but again the runtime check of `make_parr` is needed to make sure you’re not trying to parallel compose things which are leaf data.

After running `query` it is necessary to check dynamically that the result has unique addresses—although for testing and experimentation this check might be skipped. To handle the special template $\alpha \mapsto [Q]$ which inserts a fresh address, we adopt the convention that an address of `-1` is intended to be ‘fresh’, and a function `fix_addrs` goes through the result tree replacing all the `-1`s.

Chapter 9

Summary of the implementation work

9.0.1 Haskell version

There are two implementations. The main one is written in Haskell, a strongly-typed lazy functional language. This implementation fully handles the logic and query language. It uses the techniques explained in this chapter, and the code samples given in this report are very close to what's used in the implementation. (Sometimes there are changes to facilitate testing, but these differences are minor.)

The code consists of a library `subst.hs` for handling the sets-of-substitutions explained in this chapter, and another `tree_query.hs` which handles the structural details of querying trees, by mapping a formula and a tree to the data structure provided by `subst.hs`. It also provides the query language, which is fairly simple to implement once you're able to find substitutions for a tree and a formula.

9.0.2 Small Perl version

The second implementation is in Perl, an imperative, dynamically typed scripting language. It does not handle the full logic because it doesn't cope with negation or general equality tests, the two 'difficult' parts explained in the previous chapter. The purpose of this implementation was to show how the idea of graphical links can be naturally represented using pointers or references in an imperative program. The logic implemented in Perl is slightly different to the Haskell version, because it has a different (arguably more natural) treatment of graphical links. We'll cover this in the Evaluation section.

A tree, in the Perl implementation, is represented as either a plain string (for leaf data) or as a hash table mapping labels to subtrees. Looking up a string in a hash table is a constant-time operation, so following a branch to a subtree or a graphical link can be done much more quickly than in the Haskell implementation. However, as we shall see, the standard logic defined on these

trees needs tweaking to get the benefit of using hash tables. Perl's standard hash tables associate only one value with each key; it was necessary to define a new `MultisetHash` type which stores a list of values for each key, and as with Haskell, pretend that the list is really a set.

In fact Perl does not allow embedding one data structure inside another; all nested data structures are built using references. So it turns out that inside the computer there is no real difference between a branch pointing to a subtree and a graphical link pointing to that subtree—both are represented using a reference. The difference between branches and graphical links comes when you consider how to copy part of a tree, which will be discussed later.

The Perl version has another interesting feature: a tree is stored as a hash table. This means that finding a branch with a given name could be implemented as a constant-time lookup. This would work well together with the $|_1$ optimization discussed in the Evaluation.

Chapter 10

Evaluation and future work

10.1 Efficiency of $|$ operator

The definition of $T \models \varphi | \psi$ is that *some* possible splitting of the tree T allows the two halves half to satisfy φ and ψ respectively. This means that to get the complete set of substitutions letting the formula match, the query engine must try *all* splittings of T and take the union of the substitutions found for each one.

How many possible splittings are there? Because trees are unordered, the number isn't as high as it could be—but still a tree with w branches in parallel has 2^w possible splittings. Thus a $\varphi | \psi$ query will take exponential time in the width of the tree.

It is not possible to optimize away this problem, because the query engine must try all the splittings; it can't know in advance which are going to fail, because that is undecidable in general. (No proof, but note that the logic contains recursion.) However, many of the common uses of $|$ could be dealt with better. For example,

$$T \models \text{name} \mapsto [\mathbf{n}] | \text{true}$$

We can see from looking at the formula that $(\text{name} \mapsto [\mathbf{n}])$ matches only on trees of width exactly one. The query engine only really needs to try all splittings $T \equiv (T_0, T_1)$ such that T_0 has width 1. If T has width w , then this is w splittings—linear, a clear improvement.

We could allow the user to give hints to the query engine by introducing a new $|_1$ operator defined as follows:

$$\begin{aligned} T \models^p \varphi |_1 \psi &\Leftrightarrow \exists T_1, T_2 \in \mathcal{T}. \quad (T \equiv T_1 | T_2 \\ &\quad \wedge (T_2 \equiv (\alpha \mapsto \xi[T_2']) \vee T_2 \equiv (\alpha \mapsto \xi)) \\ &\quad \wedge T_1 \models^p \varphi \wedge T_2 \models^p \psi) \end{aligned}$$

In other words just the same as $|$ except that the right hand formula only gets a chance to match trees of width 1. Note that $|_1$ is not commutative. Now, with

this addition to the logic, you could write

$$T \models \text{true} \mid_1 \text{ name} \mapsto [\mathbf{n}]$$

which would execute more efficiently if T is wide.

That's an improvement but it would be better to make the optimization automatic. Define $W^1 :: \varphi \rightarrow \mathbb{B}$ to work out whether a formula matches only on trees consisting of a single branch. This is undecidable in general, but we can make a conservative approximation.

$$\begin{aligned} W^1(\alpha \mapsto \xi[\varphi]) & \text{ always} \\ W^1(\alpha @ \xi) & \text{ always} \\ W^1(\varphi \wedge \psi) & \Leftrightarrow W^1(\varphi) \vee W^1(\psi) \\ \neg W^1(\varphi), & \text{ all other cases} \end{aligned}$$

Note that $W^1(\neg\varphi) \Leftrightarrow \neg(W^1(\varphi))$ does not hold. With this function, we can define a translation \rightsquigarrow on formulæ:

$$\begin{aligned} \varphi \mid \psi & \rightsquigarrow \varphi \mid_1 \psi, & \text{if } W^1(\varphi) \\ \varphi \mid \psi & \rightsquigarrow \psi \mid_1 \varphi, & \text{if } W^1(\psi) \\ \dots\varphi\dots & \rightsquigarrow \dots\varphi'\dots, & \text{if } \varphi \rightsquigarrow \varphi' \end{aligned}$$

The last case here is to indicate contextual closure. The intention is to apply the \rightsquigarrow step as many times as possible until it can no longer match. (In fact to do that there needs to be a new case for W^1 : $W^1(\varphi \mid_1 \psi) = \text{ff.}$) Note that there are two possible rewrites for $\varphi \mid \psi$ when both φ and ψ are known to require a tree of width 1; either of these throws away the information that $\varphi \mid \psi$ requires width 2. This suggests a more general approach.

You can generalize the W^1 function to statically detecting the tree width required by any formula. Let $W(n, \varphi)$ mean ‘ φ requires a tree of width n ’, defined as

$$\begin{aligned} W(0, \text{nil}) & \text{ always} \\ W(1, \alpha \mapsto \xi[\varphi]) & \text{ always} \\ W(1, \alpha @ \xi) & \text{ always} \\ W(n, \varphi \mid \psi) & \Leftrightarrow \exists n_0, n_1. (W(n_0, \varphi) \wedge W(n_1, \psi)) \\ W(n, \varphi \wedge \psi) & \Leftrightarrow W(n, \varphi) \wedge W(n, \psi) \\ \neg W(n, \varphi), & \text{ all other cases} \end{aligned}$$

Then we could define many variants of \mid , of the form $\mid_m \mid_n$, hinting to the query engine that it need only try splittings of the tree with width m in one half and width n in the other. For cases where the width of one subquery is known but not the width of another, define \mid_n to try all splittings with width n on the right hand side (and any width on the left). The optimization stage would look like

$$\begin{aligned} \varphi \mid \psi & \rightsquigarrow \varphi \mid_m \mid_n \psi, & \text{if } W(m, \varphi) \wedge W(n, \psi) \\ \varphi \mid \psi & \rightsquigarrow \varphi \mid_n \psi, & \text{if } W(n, \psi) \\ \varphi \mid \psi & \rightsquigarrow \psi \mid_n \varphi, & \text{if } W(n, \varphi) \\ \dots\varphi\dots & \rightsquigarrow \dots\varphi'\dots, & \text{if } \varphi \rightsquigarrow \varphi' \end{aligned}$$

Again the idea is to try as many \rightsquigarrow rewrite steps as possible—with a rule being applied only when all earlier rules fail to match.

Now the query engine, when matching $\varphi|_n\psi$ against a tree of width w , would need to try only wC_n splittings rather than 2^w . Similarly when evaluating ${}_m|_n$ it would need to try only wC_m splittings, where $w = m + n$ is the total width of the current tree. And if $w \neq m + n$, it need not try any splittings at all!

A further optimization would be to notice that two queries requiring different widths cannot possibly both match:

$$\varphi \wedge \psi \rightsquigarrow \neg \text{true}, \text{ if } \exists n, m \text{ with } n \neq m, W(n, \varphi), W(m, \psi).$$

In fact the optimizer presented so far, while sound, is much less good than it could be because it fails to handle negation. To deal with \neg you would probably need to define a dual to W and keep track of what widths of trees a formula *can* match, in addition to the width (if any) that it *must* match.

10.2 Comparison with TQL

TQL, Tree Query Language, is the query tool defined in [1]. It allows you to write *from/select* style queries (see 8) to build new information trees (see 4.2.4) from existing ones, by matching against formulæ of the ambient logic containing free variables. The implementation of this language is described in [21] and the aim of this project was to implement something similar for the new data model, with graphical links, addresses and leaf data. How do the two implementations compare?

The essential principles are the same. The logic is very similar, with only the structural matching cases differing. In both cases the implementations use a special technique to handle the problems caused by negation; the presentation in this report is different to that in [21] (and slightly easier to understand, it is hoped) but the essential idea is the same. The definition of *from/select* queries for this project again follows the TQL version very closely.

10.2.1 Path expressions

One feature implemented for the ambient query language which is not included in this tool is *path expressions*. This is a more convenient syntax for querying, encapsulating the ‘| true’ idiom and other useful things such as transitive closure of a path (Kleene star). For example, the path expression

$$\text{.person} \mapsto [(\text{.child})^*[\text{.name} \mapsto [\text{Bob}]] \wedge \mathbf{t}]$$

binds \mathbf{t} to all subtrees reachable via some **person** branch, which either contain a **name** \mapsto [Bob] branch, or can reach a subtree containing such a branch by following zero or more **child** branches. (The syntax of a path expression is be slightly different in TQL, due to the different data model.) Path expressions are translated into standard formulæ, so they don’t add any power, but they do help to build a more concise query language.

10.2.2 Data model differences

There are some differences between the two query tools which are caused by the different data models. In TQL there are no addresses and no graphical links—so no need to worry about uniqueness of addresses in the result of a query. But with our tool it is not allowed to select two copies of the same subtree in building a query result, for reasons explained in chapter 8.

Also, while the ambient information trees have labelled branches but only nil as leaves, our data model stores information both at the branches and at the leaves. This gives a more natural expression of many data structures but means a separation between ‘leaf data values’ and ‘label values’. In the logic as currently defined, there is no way to test equality of a data variable and a label variable. This means that if something appears as leaf data in the input tree, it must appear as leaf data (if at all) in the output; similarly branch labels must be output as branch labels. Again, with the TQL query tool this issue simply does not arise. As discussed in section 6.4, it would be possible to add comparisons between labels and leaf data to the logic, if they are restricted to be the same type.

10.2.3 Fully general equality tests

The implementation of TQL handles general negation, and the same approach is adapted here. But general equality tests of the form $\mathbf{x} = \mathbf{y}$, that is, between two variables, are not dealt with by the TQL tool.

Therefore, we can claim to have implemented the entire logic, while TQL misses some of it. (As mentioned before, the two logics are very similar, with only the structural cases differing significantly). Whether this extra feature adds any power to the logic remains to be decided; we still have to either prove that every formula containing general equality tests can be reduced to one which does not, or else find a counterexample.

The machinery to handle equality between variables is quite complex, and takes up around half of chapter 7. It means extending the kinds of infinite sets supported: you have to invent a data structure which says things like ‘the value of \mathbf{x} is in this set, the value of \mathbf{y} is in that set, *and* $\mathbf{x} \neq \mathbf{y}$ ’. Keeping this extra equality and inequality information is a lot of housekeeping, and if equality tests turn out not to be essential it would be better to handle them by translation to simpler formulæ.

(I originally implemented the mechanism for equality tests because I thought that the TQL query tool supported them. Then later I heard from its author that while comparisons of a variable against a single value were supported, general variable-to-variable comparisons were not. Apparently they could have been implemented, but it was not thought worth the effort. After doing an implementation myself, it is hard to disagree with that judgement.)

10.2.4 Front end

The TQL tool is written in Java and provides a semi-graphical interface to load and save data files and enter queries. This implementation does not have a front end, and relies on the interactive mode of the Hugs environment. There are some advantages to this—not least being able to use the existing Haskell parser and type checker to read data and queries—but it does make the project further away from being a standalone tool. Designing a usable file format for storing semistructured data following our data model would be a useful extension, and an *efficient* file format for on-disk querying would be a whole project in itself.

As a consolation for the lack of polish, at least the small Perl implementation (which handles only a subset of the logic) can generate pretty-looking trees in L^AT_EX format for inclusion in this report.

10.2.5 Report

One of the aims of this project is to present the ideas at a more accessible level; in particular to explain the techniques for handling negation. This report is far from perfect, and the sections giving lots of Haskell code are not as crystal clear as they could be. But it is hoped that the report does a better job than [21] of explaining the issues to the non-specialist.

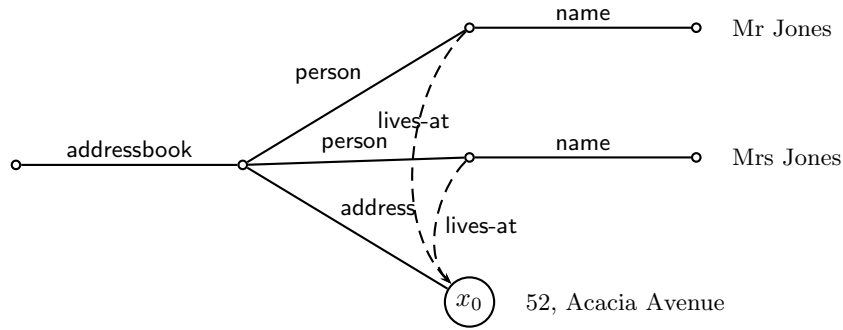
10.3 Selecting subtrees in a query

Because all addresses in a tree must be unique, there are restrictions on how a tree can be built up on the right hand side of a *from/select* query, as we saw earlier. In particular, it is not possible to select two copies of the same subtree, and this means that you cannot in general write queries to duplicate trees. How can this be remedied?

We could add a new case to the definition of queries: $\text{copy}(\mathbf{t})$, where \mathbf{t} is a tree variable. The semantics of this query under a substitution ρ would be the same as $\llbracket \mathbf{t} \rrbracket_{\rho}$, but with every address x renamed to a fresh address.

What about graphical links in the subtree being copied—should their destinations be renamed too? Clearly links which point outside \mathbf{t} are not affected by the renaming, so they should be left alone. But for links which point from somewhere in \mathbf{t} to somewhere else in \mathbf{t} , should the destination be altered so that it remains within the copied subtree?

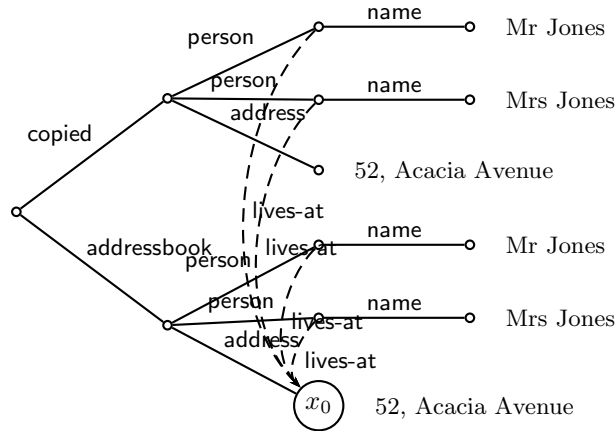
Taking the input tree T , an address book mapping people to addresses, as follows:



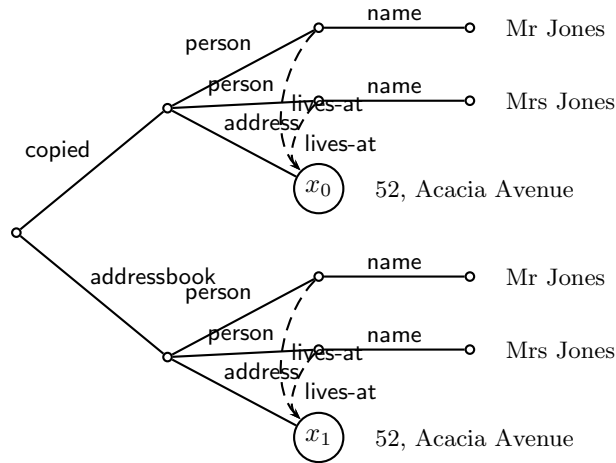
and the query

from $T \models (\text{addressbook} \mapsto [\mathbf{t}])$
select $\text{addressbook} \mapsto [\mathbf{t}] \mid \text{copied} \mapsto [\text{copy}(\mathbf{t})],$

the result without renaming the destination of links is



That seems silly—we went to all the trouble of copying the ‘addressbook’ tree, including the ‘Acacia Avenue’ subtree, but then didn’t use the newly created subtree. The ‘copied’ subtree taken by itself would have dangling pointers, which doesn’t seem right when the original tree on its own had none. If the query engine arranges to rename the targets of links, the result becomes



which seems much more satisfactory. So perhaps `copy(t)` should always rename the targets of graphical links pointing within the copied tree, so that they continue to point within the tree after its addresses have been replaced. On the other hand, perhaps there are some graphical links which are not intended to be moved—should we allow the data model to specify this somehow?

This is another opportunity to make an analogy between our data model and Unix directory structures. As mentioned in the Background, the equivalent of graphical links in that model are called symbolic links. A symlink refers to a file by name and is permitted to ‘dangle’ by pointing to a filename that does not exist. When copying a directory tree, the symlinks are copied but their destinations remain the same. We assume some familiarity with Unix-style pathnames.

However, there are really two kinds of symlink: relative and absolute. An absolute symlink names a file starting from the root of the file system, but a relative symlink starts from the directory containing it, and moves up or down the tree. It is thus possible to refer to the same file in two different ways. For example, consider the directory `/d0/`, containing a plain file `f` and two symbolic links:

```

relative  →  f
absolute  →  /d0/f

```

Then both these symlinks point to the same file. If the directory `/d0/` is recursively copied (using the `cp(1)` command, or by being backed up and then restored somewhere else), the copied directory will have the same two symbolic links, with the same destination filenames—but the relative link will point to the copy of `f` in the new directory, while the absolute link will point to the original `/d0/f`. And if `/d0/f` no longer exists, the absolute link becomes dangling.

The difference between relative and absolute links exists almost by accident in Unix, because of the way symbolic links are specified by giving a filesystem path. But our data model works by giving addresses directly as the target of graphical links, not path expressions. So while there’s clearly some precedent for distinguishing two kinds of link which behave differently on copying, it’s not obvious how to adapt the data model to do this. Also it is not clear that

the equivalent of ‘absolute’ links would be useful; the address book example seemed to demonstrate that the destination of graphical links should always be moved. So we don’t yet know whether two different kinds of graphical link should be added, although the idea of using path expressions as the target of a link deserves further investigation.

10.4 Comparison with XML

10.4.1 Ordering

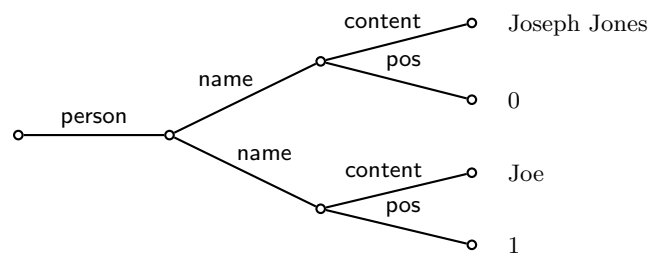
As mentioned in the Background section, the biggest difference between XML and our data model is that our model is unordered whereas XML works with ordered trees.

There are two possible approaches to convert from XML to our data model. One is to convert subelements to subtrees, and throw away the ordering information. This is appropriate for applications where the ordering is not important, or can be recalculated from other data in the file. This of course depends on the meaning of the XML document and cannot be decided syntactically. Perhaps a future schema language for XML should allow you to specify explicitly which ordering information is important, and which is just ‘accidental’.

The other approach is to preserve the ordering information by turning it into some other data in the unordered trees. For example, the XML document

```
<person>
  <name>Joseph Jones</name>
  <name>Joe</name>
</person>
```

might have a meaning that the more formal names are listed earlier, with nicknames coming later. It might be translated to:



But you can imagine that other parts of the same document might not need to preserve order. Perhaps the ‘person’ elements at the top level do not have any particular ordering, but are presented in whatever order the user chooses (alphabetical, newest first, or whatever). Associated with the description of the file format there could be information about which elements require ordering information, and which do not.

10.4.2 Pointers

XML has graphical links, of a sort, using `ID` and `IDREF` attributes. However for the document to be valid these links cannot ‘dangle’, they must point within the same document. This means you cannot in general split an XML document tree into several subtrees: the subtrees might have pointers elsewhere in the document and so be invalid when taken by themselves.

Some XML applications (such as XHTML[22]) have explicit ‘links’ to other documents referred to by URL. These links are to the top level of another document, not to elements within it, although proposals such as XPath[23] would allow a richer set of links between documents. The idea of referring to a URL ties in with the idea mentioned earlier that graphical links might work using path expressions instead of addresses.

10.5 Alternate meaning of graphical links

In this section we talk about the Perl implementation, its handling of graphical links, and whether that makes sense for a query language.

In the logic as defined, matching a graphical link is done with the formula $\alpha @ \xi$. This allows you to bind an address variable to the destination of the link, but then it is up to you to find the subtree pointed to ‘by hand’. It is possible to write a formula to do that using \diamond or recursion, but it doesn’t always seem natural. Often we would like to just ‘follow’ a graphical link in the logic and make assertions about whatever tree it points to.

While the Haskell implementation works by a straight translation of the data model into Haskell—so that a graphical link in the tree is represented by a pair of label and address—the Perl version takes a different approach and represents a graphical link with a pointer to the subtree it points to. This makes following a link as fast as a single dereference operation, there is no longer any need to search down from the top of the tree for a particular address.

In fact, Perl stores all subtrees by reference, so inside the machine there is no real difference between a contained subtree and a graphical link. It is just a convention we adopt to say that the branch to a subtree is the ‘main’ connection to it and the graphical links are just ‘aliases’. To the computer they are all just pointers. To distinguish the two, the Perl implementation uses names beginning `gl-` for graphical links.

Then writing a formula that says ‘the tree is a graphical link with label α , pointing to a tree satisfying φ ’ is just the same as the conventional $\alpha \mapsto [\varphi]$ query, except that if you want to follow a graphical link you specify a name beginning `gl-`.

The addresses, in fact, are not visible to the logic. It is not possible to write a formula which asserts that two graphical links point to the same place; all you can do is make assertions about the trees pointed to. Since our data model definitely does distinguish between two copies of the same tree (for example, between the ‘addressbook’ and ‘copied’ subtrees given earlier in this chapter) this

is not good enough. Users need to be able to make assertions about addresses themselves, at least to the extent of equal and not-equal comparisons.

So we have a Haskell implementation where the only way to ‘follow’ a graphical link is by writing a formula with \diamond to search for that address, and a Perl implementation which makes it easy and fast to follow graphical links, but does not allow direct querying of addresses. Is it possible to combine the advantages of both?

We suggest adding a new formula to the logic, in addition to the existing $\alpha@x$, as follows:

$$T \models^{\rho} \alpha \dashrightarrow [\varphi] \Leftrightarrow \exists \xi. (T \equiv \rho(\alpha)@x \wedge T' \models^{\rho} \varphi,$$

where T' is the tree with address ξ).

If the link is dangling then the tree fails to satisfy this formula; thus $\alpha \dashrightarrow [\text{true}]$ is a check to see whether a graphical link points anywhere.

Then it would be possible to easily write queries which follow graphical links, making them more useful and ‘transparent’. You could create a derived operator which follows either a branch or a graphical link with a given label; in fact this might be the most common way of matching subtrees, so that if later on it was decided to represent some information as graphical links rather than subtrees the old queries would still work.

An implementation which stored graphical links as pointers would be able to test $\alpha \dashrightarrow [\varphi]$ as efficiently as $\alpha \mapsto [\varphi]$. An implementation which didn’t use pointers (like our Haskell version) could still provide somewhat faster access than at present, since following the link would be done in code rather than by evaluating a complex formula.

10.6 Different data models

In every query language, the first part of the logic definition follows the structure of the data. So if a tree has empty trees, branches and parallel composition, the logic will have atomic formulæ to match the empty tree, a tree with one branch and two trees in parallel. To avoid repetition we could overkill the problem and define the logic automatically from the data structure.

An algebraic datatype is the fixed point of a functor. The datatypes available in Haskell and other functional programming languages can in principle be built from a finite number of elements (`True`, `False`, `Zero` and so on) plus the functors \times and $+$. For example a list can be defined as

$$\mathbf{L} \ a = \mathbf{Nil} + (a \times \mathbf{L} \ a)$$

and the datatype `List` would be the fixed point of this equation. I think. What logic can we derive from the definition of \mathbf{L} ?

The top level of \mathbf{L} is $+$. It can be either of the form `Left ...` or `Right ...`

So we need two formulæ to match an L:

$$l \models \text{Left } \varphi \Leftrightarrow l = \text{Left } x \wedge x \models \varphi$$

$$l \models \text{Right } \varphi \Leftrightarrow l = \text{Right } x \wedge x \models \varphi$$

The formal definition is as follows.

- If a, b are types and x has type a , then $\text{Left } x$ has type $a + b$.
- If a, b are types and y has type b , then $\text{Right } y$ has type $a + b$.
- If a, b are types and φ is a formula for matching a value of type a , then $\text{Left } \varphi$ is a formula for matching a type of value $a + b$. See above for the definition.
- Similarly $\text{Right } \varphi$.

- If a, b are types, x has type a , y has type b , then $\text{Pair } x y$ has type $a \times b$.
- If φ matches an a then $\text{First } \varphi$ matches an $a \times b$, if the first part of the pair matches.
- Similarly $\text{Second } \varphi$.
- A useful derived formula $\text{Both } \varphi \psi$ is defined as $(\text{First } \varphi) \wedge (\text{Second } \psi)$.

This lets us define a logic for any data structure built from $+$ and \times . For example we could define a logic over lists of a , provided we were given formulæ to match Nil (trivial) and elements of type a . But for semistructured data this is not adequate, because we want to ignore ordering.

Essentially the two functors above and their associated logic have no way to represent a set or multiset where the order is unimportant. You can get a list, but two lists are different (distinguishable by the logic) if they contain the same elements in a different order. What we need is a new functor Multiset which takes a type a to a type $\text{Multiset } a$. We assume there is some way to construct these multisets, perhaps from an ordinary list. There are three formulæ on a multiset:

- If a is a type, then Nil is a formula matching the type $\text{Multiset } a$. (For any a .)
- If a is a type and φ is a formula matching an a , then $\text{One } \varphi$ is a formula matching a $\text{Multiset } a$.
- If φ and ψ are both formulæ matching a $\text{Multiset } a$, then $\text{Parr } \varphi \psi$ is also a formula matching a $\text{Multiset } a$.

Satisfaction for these formulæ is defined as:

$$m \models \text{Nil} \Leftrightarrow m \text{ is the empty multiset}$$

$$m \models \text{One } \varphi \Leftrightarrow (m \text{ has exactly one element } x) \wedge x \models \varphi$$

$$m \models \text{Parr } \varphi \psi \Leftrightarrow \exists m_0, m_1. (m = m_0 \cup m_1 \wedge m_0 \models \varphi \wedge m_1 \models \psi)$$

where \cup is some kind of union on multisets. Standard multiset union seems a bit odd at first, it's the *least* relation such that $x \subseteq x \cup y$ and $y \subseteq x \cup y$. With multisets this means take the maximum of the number of each different element in x and y , not the sum. For querying semistructured data we'd probably prefer a different way of splitting a multiset into m_0 and m_1 , such that the total number of elements is preserved; in other words additive union \uplus .

It is believed that that these three formulæ are orthogonal, minimal and complete (in the sense that all the structure of a multiset is exposed, and two different multisets can be distinguished). Actually Nil is not strictly needed, we could define it as a derived predicate as follows:

$$\begin{aligned} m \models \text{Some} &\Leftrightarrow m \models \text{One True} \vee m \models \text{Parr Some True} \\ m \models \text{Nil} &\Leftrightarrow m \not\models \text{Some} \end{aligned}$$

assuming a formula True which always matches. But Nil is nice to have because it reflects our intuitions about sets. And besides the above definition is recursive, so without recursion Nil is not redundant.

There is a less powerful alternative for querying multisets: if α is a formula matching an a and φ is a formula matching a Multiset a , then Parr1 $\alpha \varphi$ is a formula matching a Multiset a . Satisfaction:

$$m \models \text{Parr1 } \alpha \varphi \Leftrightarrow \exists a, m'. (m = a \cup m' \wedge a \models \alpha \wedge m' \models \varphi)$$

noting as before that standard multiset union may not be quite what we want for \cup . It is possible to define Nil and One (but not Parr) in terms of Parr1. This all-in-one formula may be powerful enough for some logics, such as matching a graph by removing one edge at a time. It corresponds to the $|_1$ alternative to parallel composition discussed earlier.

10.6.1 Converting the tree and graph logics into this form

Converting even fairly simple data structures like the trees with graphical links into datatypes built from $+$, \times and Multiset can get syntactically awkward, but the same power is there. First show how the trees with graphical links are defined in this framework:

$$\text{Tree } a \ x \ d = (\text{Multiset } (((a \times x) \times \text{Tree } a \ x \ d) + (a \times x)) + d)$$

So a tree is either a multiset or some data d . The elements of the multiset are either labelled branches (with a , x and a subtree) or graphical links (with a and x). Now we illustrate how the logic is derived from the datatype, giving everything provided by the original logic. The interesting cases are:

$$\begin{aligned} \varphi | \psi &\Rightarrow \text{Left (Parr } \varphi \ \psi) \\ \text{data } \delta &\Rightarrow \text{Right } \delta \\ \alpha \mapsto \xi[\varphi] &\Rightarrow \text{Left (One (Left (Both (Both } \alpha \ \xi) \ \varphi)))} \end{aligned}$$

A simple labelled directed graph data structure is simpler:

$$\text{Graph } a \ x = \text{Multiset } (a \times (x \times x))$$

Then taking one edge off the graph can use `Parr1`, or splitting the graph into two disjoint parts uses `Parr`. This is just included to illustrate that within the general framework we can model different data structures, not just unordered trees, and derive a logic automatically for each one.

If we wanted an *undirected* graph, where the edge $a(x, y)$ is not distinguishable from $a(y, x)$, then we'd need a new functor, an 'unordered pair'. We could define this as: if a is a type, then $\angle a$ is a type, meaning two elements of a in any order. If φ and ψ are formulæ matching an a , then `Two φ ψ` is a formula matching an $\angle a$. It matches if the two elements of the $\angle a$ match φ and ψ either way round. Note that both elements must be of the same type, you cannot make an unordered pair of two different types. Partly for this reason it is not possible to build `Multiset` out of `+`, `×` and `∠`.

10.6.2 Variable matching

Having built a logic using the connectives above, we still need to define formulæ to match the atomic data types. For example with a multiset of strings, we need some way of matching strings.

To do this we follow the same approach used in the main part of the report: introduce variables and then the result of matching a formula against a tree is the set of substitutions letting the tree satisfy the formula. For example, the result of the query

$$\text{Multiset } \{a, b, c\} \models \text{Parr } (\text{One } x) \text{ (True)}$$

is a set of three possible substitutions: $\{\{x \mapsto a\}, \{x \mapsto b\}, \{x \mapsto c\}\}$.

The techniques explained in this report to handle infinite sets of substitutions (caused by unrestricted use of negation and variable equality) can be applied with this general data model too.

10.6.3 Implementation

A short Haskell program uses the standard data type `Either`, and defines its own types `Pair` and `Multiset` to let you build new algebraic data types which will automatically be queryable.

The type `Pred α β` is a 'predicate' matching a value of type α and returning a value of type β . If we did not have variables, then β could be \mathbb{B} , but since we are trying to gather substitutions, the type used for β is a set of substitutions. Then each data type `T` has its associated 'lifting' functions to convert a value of type `Pred α β` to a value of type `Pred (T α) β` .

As a proof of concept there are functions to translate the tree data model, and a simple graph model, into the general framework; and similarly to translate a formula of the tree logic into a `Pred` value.

For example, here is the Haskell definition of the `Pair` functor:

```
data Pair  $\tau$   $\tau'$  = Pair  $\tau$   $\tau'$ 
```

```

frst :: Pred  $\tau$   $\beta$   $\rightarrow$  Pred (Pair  $\tau$   $\tau'$ )  $\beta$ 
scnd :: Pred  $\tau'$   $\beta$   $\rightarrow$  Pred (Pair  $\tau$   $\tau'$ )  $\beta$ 
both :: Pred  $\tau$   $\beta$   $\rightarrow$  Pred  $\tau'$   $\beta$   $\rightarrow$  Pred (Pair  $\tau$   $\tau'$ )  $\beta$ 

frst p (Pair x _) = p x
scnd p (Pair _ y) = p y
both p0 p1 pair = (frst p0 pair)  $\cap$  (scnd p1 pair)

```

The type β is the result type of a query, such as booleans (for the case when we don't handle variables) or sets of substitutions. The \cap operator is the equivalent of 'and' on values of type β , for example with the framework used in this project it would be the function `unify_substpv_sets`.

Given a value of type `Pair τ τ'` , there are two possible queries on it. Either you can provide a predicate to match a value of type τ , and ask whether the first element of the pair satisfies that predicate—or you can provide a predicate to match τ' and ask about the second element of that pair. These two cases correspond to `frst` and `scnd`. The function `both` is a useful derived operator to match both parts.

This idea is analogous to the standard `map` and ordinary functors: but instead of a mapping on types and a mapping on functions, we have a mapping on types and a mapping on predicates.

The Haskell implementation of the general querying was abandoned halfway through the project, to concentrate on the particular case of unordered trees. Although building a data structure out of `Either`, `Pair` and `Multiset` is just as powerful as defining one directly in Haskell, it is syntactically a lot more awkward, and extending the logic to arbitrary data structures is not the primary aim of this project. So this implementation was just a testbed for interesting new features. One such feature it did implement was an easy way to switch between different kinds of β ; that is, different representations of the results of a query. This allowed experimentation with another way to handle the problem of negation: negation by failure, explained in the next section.

10.7 Alternate meanings of negation

The techniques used for handling infinite sets of substitutions work very well for the logic as it is currently defined. But the trouble is that they are inflexible—it is difficult to add new kinds of formulæ stating things about variable assignments. Recall how much extra code was required just to add equality tests between variables.

The trouble is that when you have an infinite set of values, it's difficult to add any restriction to that set. Suppose for example that a new formula were added to the logic, `x` like `pattern`. The semantics is easy to define: the formula holds iff the value of `x` matches `pattern`, which could be some kind of string match—suppose that `x` like `'B%'` means the first letter of `x` is `'B'`, for example.

When it comes to evaluating this formula, how can we reconcile it with infinite sets? If the value of `x` is currently `(One "Blue")` and we wish to unify

this with the information that the first letter of x is ‘B’, then there’s no difficulty in building a new set of substitutions which contains that information. But what if the value of x were `NotIn ["Red"]`? Then we’d need to represent the set of substitutions with the following property:

- The value of x is not ‘Red’,
- but whatever value it is, it starts with ‘B’.

How can our data structures store that? We’d need to add another extension to the data structure, another list of constraints, a new set of rules for testing those constraints and checking they are consistent with each other. And then when the next new feature is added to the logic we have the same trouble again. We saw that even something as basic as equality tests leads to a lot of bookkeeping.

Compare this with the situation when you always know the substitutions will be finite. In that case, adding the information that the first letter of x is ‘B’ is as simple as enumerating all values for x and throwing away those which don’t meet the criterion.

But as we’ve seen, negation in general can have infinite semantics, so we can’t get rid of the infinite sets without also getting rid of negation. Or can we? Negation by failure provides a possible answer to this problem, although in the end it may prove unsatisfactory.

Under NBF, the meaning of $\neg\varphi$ is that it succeeds iff φ with the ‘current substitution’ fails. This definition doesn’t really fit with the implementation we have chosen so far, where the result of a match is a set of substitutions which are returned all at once. To implement negation by failure, define a ‘negated set’ of substitutions, which has the meaning ‘any variable assignment not covered by one of the elements in this set’. If a formula contains no negations, then negated sets will never be used, only positive sets of substitutions. \neg is defined as turning a positive set into a negated set, and vice versa. It is possible to unify (\wedge) two positive sets or unify two negative sets without problems.

That is, the possible substitutions are either a finite set of variable assignments mapping each variable to just one value—or the negation of such a finite set taken as a whole. If a set of substitutions is a finite set of variable assignments, then it is easy to filter on any predicate such as `like` just by throwing away the values which do not match.

On the other hand, if the current set of substitutions is the negation of such a finite set, then filtering on x like *pattern* involves throwing away all the values which *do* match. In both cases there is no need to consider infinite sets of values.

This means that with NBF there are no limits on what predicates and tests can be added to the logic. There is no need to do lots of work keeping track of constraints to be checked later—a constraint such as `like` or equality can be checked immediately, by looking at all possible values. Since a practical query language is normally expected to support a wide range of comparisons and operations on data, this is a strong advantage.

10.7.1 Incompleteness of negation by failure

But unifying a positive set and a negative set may flounder if the negative set contains variables which are not mentioned in the positive set. For example, take the two sets:

$$\{\{\mathbf{x} \mapsto 1\}, \{\mathbf{x} \mapsto 2\}\}$$
$$\text{Not}\{\{\mathbf{y} \mapsto 3\}\}$$

The first set, which is a positive set, means that \mathbf{x} has the value 1 or 2. The second set, a negated set, means all substitutions *except* those giving \mathbf{y} the value 3. Now we would like to unify these two to get a new set, which should also be finite. This is not possible: if the new set were a positive set then it could not store the information that \mathbf{y} is not 3. But if it were a negated set, it could not store the information that \mathbf{x} has the value 1 or 2, because a negated set gives a finite set of values and then says ‘any assignment except these’.

In general, you cannot introduce new variables inside a negation. When that happens, it is called ‘floundering’ and is normally a runtime error (although some systems will ignore the error, throw away some information, and later given an incorrect result). This problem with negation by failure is well documented in the field of logic programming.

10.7.2 Implementation

An implementation of negation by failure was written early on in the project, for comparison with the general negation approach. But the final code does not support NBF.

10.8 Multiplicity

The tree logic as originally specified collected a set of substitutions which let the formula φ match the graph T . In other words, all ρ such that $G \models^\rho \varphi$. We use R to refer to this set of substitutions. In the implementation there may be various cunning schemes to keep track of an infinite number of different substitutions using finite space, but the semantics is clear: we have a set of substitutions assigning values to variables, and \wedge in formulæ has the effect of \cap on the sets of substitutions, \neg is set complement, and so on.

Then to construct the result using `from/select` we take each $\rho \in R$ apply it to the output template. The result is constructed by joining together each such application with `|`. Now with the data model as it currently stands it is not allowed to select two copies of the same subtree in parallel, because addresses would no longer be unique. But suppose for a moment that this restriction were lifted. Would it ever be useful to select n copies of a particular subtree, where n depends in some way on the set of substitutions found? To do that, we’d want to allow R to contain the same substitution ‘more than once’.

It turned out that the implementation of the logic, quite by accident, would return a list containing two copies of the same substitution ρ when there was more than one way for $T \models^\rho \varphi$ to be satisfied. ‘More than one way’ usually means that there is a parallel composition in the formula such that two or more different splittings would let the formula be satisfied. If R had been strictly limited to set semantics, then of course these duplicates would not have arisen. The implementation, however, has a multiset-like semantics for R . In this section we ask whether multiple results in this way make sense, by reference to what users expect and what other database systems do. If we can decide what would be most useful in a query language, then we can try to formalize it mathematically.

Taking R as a set gives us a query language that doesn’t allow duplicate results or any easy way to find out counts (such as ‘how many authors’). This doesn’t match what SQL-based relational databases do; they return multiple results by default although you can do `select distinct`. Since our data structures do normally allow ‘several’ of something (for example, several tree edges labelled `author` in parallel with each other), the query language ought to work in the same spirit.

We need to pick a definition of \cap to use in the semantics of \wedge , a definition of \cup for \vee , and some kind of complementation for \neg . We use the semantics $\llbracket T, \varphi \rrbracket$ defined as the set of substitutions letting T satisfy φ .

Also we should decide the meaning of parallel composition, $\llbracket T, \varphi \mid \psi \rrbracket$. Assume for the moment that its semantics is the *union* of all possible splittings, in other words $\bigcup \{ \llbracket T_0, \varphi \rrbracket \cap \llbracket T_1, \psi \rrbracket : T \equiv T_0 \mid T_1 \}$. So the choice of what kind of union will affect the meaning of \mid .

One choice, as mentioned above, is to treat R as a set. Then the meanings of \wedge , \vee and \neg are obvious. With a multiset-based data structure (like trees with multiset \mid) such a logic would be incomplete because it would be unable to distinguish the trees `author` \mapsto `[Cardelli]` and `(author` \mapsto `[Cardelli]` \mid `author` \mapsto `[Cardelli])`. This does not arise in our data model because every subtree must have an address, and it must be unique. So in our data model the two ‘author’ subtrees would be distinguishable, if only by their different addresses.

However, having to match addresses in order to count the number of occurrences is awkward. It might be better if you could write simpler queries using the $\alpha \mapsto [\varphi]$ derived operator, which hides the details of addresses from the user, but still be able to distinguish between T and $T \mid T'$ when T and T' differ only by addresses, not by data.

If we don’t treat R as a set, the next step up is a multiset. This can be thought of as mapping each substitution ρ to a natural number, whereas an ordinary set maps them to Booleans. As long as it is possible, as in the current implementation, to explicitly flatten the results to a set (the equivalent of SQL’s `select distinct`), we can still keep all the power of plain set-based results, while having the opportunity to work with multiple results when wanted. If R is a multiset, then what are the definitions of the set operators?

From the subset inclusion ordering on multisets we get \cup defined as the join of two multisets and \cap as their meet. But while these operators have nice

properties mathematically, it's not clear that they are appropriate for a query language. In fact, if we take $|$ as using the standard union operation to combine its results, then we can prove that multiset union will be identical to set union! The proof is by induction on the structure of formulæ. The result of any $T \models \varphi$ query will not contain any duplicate substitutions, (for any T). Base cases such as nil are easy. Then clearly multiset union and multiset intersection will not introduce any duplicate substitutions, if ρ occurs once in R and once in R' , it will still occur just once in $R \cup R'$. Since the other query operators including $|$ are defined in terms of \cup and \cap (or \vee and \wedge if you prefer), this property always holds.

So it makes no sense to use multiset union everywhere. If we decided to implement logical operators in terms of multiset operators, we'd have to do something different for $|$ or we might as well just go back to sets. Suppose that for $|$ we used additive union \uplus to combine the different splittings. So if a tree T has two splittings $A | A'$ and $B | B'$, and ρ is in the set of substitutions generated for both those splittings, then ρ will appear twice in the set of substitutions for matching T .

Does it do any good to use multiset union and intersection, but with additive combining of results for parallel composition? Well certainly it does more than using plain sets (or plain multisets, as explained above), because you can now get multiple results. But does it give the semantics users expect? Suppose we have the tree $(\text{pub} \mapsto [\text{author} \mapsto [\text{Cardelli}]] | \text{pub} \mapsto [\text{author} \mapsto [\text{Cardelli}]] | \text{pub} \mapsto [\text{reviewer} \mapsto [\text{Cardelli}]])$, and the query $(\text{pub} \mapsto [\text{author} \mapsto [\mathbf{x}]] | \text{true}) \vee (\text{pub} \mapsto [\text{reviewer} \mapsto [\mathbf{x}]] | \text{true})$. Then the substitutions returned will be the union of the left half and the right half. The left half matches with the substitution multiset $\{\mathbf{x} \mapsto \text{Cardelli}, \mathbf{x} \mapsto \text{Cardelli}\}$, and the right half matches with $\{\mathbf{x} \mapsto \text{Cardelli}\}$. The multiset union of these two gives $\mathbf{x} \mapsto \text{Cardelli}$ twice, the maximum of the two counts. But what does that mean in terms of the original data? I would have expected to get 'Cardelli' three times. Also note that if you rearrange the query to move $\text{pub} \mapsto [\dots]$ to the outside, the semantics changes.

Also, what should the definition of \neg be in a multiset world? It seems that whatever definition you pick, you would lose the property that $\neg\neg\varphi \equiv \varphi$. Either that or lose the property $T \models \neg\varphi \Leftrightarrow \neg(T \models \varphi)$. We can think of \neg as a function $\mathbb{N} \rightarrow \mathbb{N}$, the most sensible definition seems to be $0 \mapsto 1, n > 0 \mapsto 0$.

Suppose we used additive union everywhere. That would make the previous example return 'Cardelli' three times (since he is a reviewer once and an author twice). But what should be the definition of intersection? There are some properties we might like to keep, such as distributivity of \vee over \wedge . The property of union being the *least* upper bound of two sets obviously no longer holds, so we shouldn't feel guilty that intersection is no longer the greatest lower bound. But we would like to keep some properties related to negation (like duality) if possible.

To recap, the aim is to find definitions of \cup and \cap that provide useful query functionality, returning multiple results in a way that is intuitive. Doing this, however, will mean sacrificing some mathematical equivalences, and we must decide which equivalences are worth keeping and which we can compromise on.

Perhaps we should make sure that when the multisets happen to look like

ordinary sets—with each substitution occurring either zero or one times—then all the familiar properties from sets still apply. In other words, if it looks like a set, then it still behaves like a set. Multiset union and intersection, plus the negation defined above, are like this. However, once elements start appearing more than once in the multiset, some of the set properties are lost.

We suggest that a good operator to go along with additive union is multiplicative intersection. In other words the count of ρ in $A \cap B$ is (count of ρ in A) \times (count of ρ in B). This choice clearly has lots of the properties we would like, and it *almost* has duality between \cup and \cap if you restrict consideration to ordinary sets.

It so happens that the current implementation handles \wedge by doing a Cartesian product of two sets of substitutions, and then removing those pairs from the result which do not unify. So it uses additive union and multiplicative intersection. Now you could say that we are just trying to retrospectively justify implementation decisions that have already been taken, by appealing to vague notions of what might be most ‘intuitive’. But there is precedent. Look at what SQL does:

```
=> create table a (name varchar(1000));
=> create table b (bname varchar(1000));
=> insert into a values ('fred');
=> insert into a values ('fred');
=> select * from a;
  name
-----
  fred
  fred
(2 rows)
=> insert into b values ('fred');
=> insert into b values ('fred');
=> insert into b values ('fred');
=> select * from b;
  bname
-----
  fred
  fred
  fred
(3 rows)
=> select count(*) from a;
      2
=> select count(*) from a,b;
      6
=> select count(*) from a,b where name = bname;
      6
```

The result of joining tables a and b gives $3 \times 2 = 6$ rows.

However, using \times to implement \cap is not perfect. While additive union seems like a good match for users’ expectations when querying, this definition

of intersection is less obvious. For example, consider the tree

```

pub ↦ [author ↦ [Cardelli]]
| pub ↦ [author ↦ [Cardelli]]
| pub ↦ [reviewer ↦ [Cardelli]]
| pub ↦ [reviewer ↦ [Cardelli]]
| pub ↦ [reviewer ↦ [Cardelli]]

```

That is, ‘Cardelli’ appears twice under `author` and three times under `reviewer`. The query $(\text{pub} \mapsto [\text{author} \mapsto [\mathbf{x}]] \mid \text{true}) \wedge (\text{pub} \mapsto [\text{reviewer} \mapsto [\mathbf{x}]] \mid \text{true})$ will give a result with $x \mapsto \text{Cardelli}$ six times. It’s not necessarily obvious how this comes from the data. The explanation is that there are two different ways for `author` \mapsto `[x]` to match ‘Cardelli’, and for each of those ways there are three possibilities for the reviewer match, so $2 \times 3 = 6$ results in total. Well yes—but is this what the user wants?

Probably in this case six is probably a better number of results to return than two, which you would get with multiset intersection. But a user would probably be most interested in just one result, a yes/no answer. While with \vee you may want to build up several results, with \wedge this behaviour is rather counterintuitive and it is easy to get huge result sets from small queries. So maybe we should have an asymmetry between these two operators, implementing \wedge as the dual of \vee with respect to the \neg operator defined earlier. This would mean that the result of the above query would be just one $x \mapsto \text{Cardelli}$.

On the other hand, we could leave \wedge with a multiplication-based semantics and make the user to explicitly eliminate duplicates if that is wanted. It’s not yet clear what the right approach is.

10.8.1 Conclusions

The logic as currently defined is complete, and can always distinguish two trees which are not structurally congruent. However, sometimes doing so is awkward because it involves explicitly comparing addresses, and there is no clear way to get a *count* of how many times a particular pattern occurs. This happens because according to the formal definition, substitutions are treated as a set, so the same ρ cannot appear more than once.

However, if we look at other database systems (in this case SQL), we see that returning multiple results can be useful even when the theoretical framework specifies sets. (The relational model is definitely set-based, yet SQL happily returns multiple copies of the unless the query specifically forbids it.) This should lead us to ask whether our query tool should do the same, and keep track of how many ‘ways’ there are for a given substitution to be returned. This question was suggested by the fact that the current implementation already does.

The query language as it stands has problems with selecting many copies of a subtree, because the result tree needs to have unique addresses. But if we assume that this restriction can be overcome—and the extension suggested in section 10.3 could do just that—then we can consider whether selecting more

than one copy would be useful, and what the semantics should be.

Moving from sets to multisets can be thought of as a compromise between familiar mathematical properties—since the behaviour with ordinary sets is well understood—and presenting a richer language to the user. However, if we decide to change the semantics to accommodate ‘what users want’, we have to decide what that is. We give some reasons for the choice of \uplus as the semantics of \cup ; and either multiplicative intersection or the dual of \uplus as the semantics of \cap . But this is a subjective judgement. Looking at SQL’s behaviour gives some harder evidence, but still this is not conclusive.

The obvious next step is to conduct user research, perhaps even psychological research, to find out under what circumstances multiple results would be expected and how many should be returned. Then there would be some objective criteria to compare the possible different semantics, in addition to considering their mathematical properties.

10.9 Alternate semantics for recursion

The current definition of $\mu\mathbf{R}.\varphi$ is defined as a least fixed point with respect to set inclusion. Recursion is restricted to cases where \mathbf{R} appears positively, and with this we have a monotonicity result for set inclusion which makes sure the least fixed point exists.

However, the implementation of recursion for this project is different. It works very simply: keep a mapping from recursion variables to formulæ, and then when \mathbf{R} is encountered look it up to find a formula φ , and continue processing with φ . On the one hand, there is no restriction that \mathbf{R} appear only positively, the mechanism works correctly whether it is positive or negated. But on the other hand, many recursive formulæ will fail to terminate (for example $\mu\mathbf{R}.\mathbf{R}$). In this section we suggest an alternative semantics for recursion which doesn’t have any syntactic restrictions, but takes account of the possibility of nontermination (so the semantics remains computable). The work is not finished yet, so this section is just a sketch.

The idea is that instead of taking the semantics of a formula to be a set of trees, we make it a partially-defined set of trees or ‘pst’. A pst gives *some* information about which trees are in the set, but not necessarily complete information.

Take an element of the domain to be $E = (A, B)$ with the meaning that A is the elements definitely in E , B ’s elements are definitely not in E , and of course $A \cap B = \emptyset$. Then

$$\begin{aligned} \perp &= (\emptyset, \emptyset) \\ (A, B) \sqsubseteq (A', B') &\Leftrightarrow A \subseteq A' \wedge B \subseteq B' \\ \neg(A, B) &= (B, A) \\ (A, B) \cap (A', B') &= (A \cap A', B \cup B') \end{aligned}$$

An alternative way of expressing the same thing is to take the function space $T \rightarrow \mathbb{B}_\perp$.

To show that this semantics is computable, we need to show that the partial order is a ccpo, and that \neg and \cap are continuous. To show that every chain has a least upper bound, consider that \sqsubseteq is defined pointwise in terms of \subseteq on the components of the pair, and a chain in \subseteq always has a lub. Proving that \neg and \cap are continuous remains to be done.

A formula can be identified with a pst, so $\mu\mathbf{R}.\varphi$ will be the least fixed point of a function $\text{pst} \rightarrow \text{pst}$. If all the pieces used to construct formulæ (namely \neg and \cap) are continuous then this function is continuous and so the least fixed point exists.

For example, the semantics of $\mu\mathbf{R}.\mathbf{R}$ should be the least fixed point of the identity function. That is clearly \perp —which matches the behaviour of our implementation, which fails to terminate when given this formula. However, it's not so clear what it means for the semantics of a formula to be a partly but not fully defined set of trees: that is, (A, B) where A and B are nonempty, but some trees are in neither A nor B . Does that mean that evaluation of the formula will terminate on some trees but not others? Or that the problem of deciding whether a tree is in this (infinite) set is semi-decidable?

We also need to check the property that when the set of trees given by this semantics is fully defined, it corresponds to the result given by the standard semantics.

Another question is how to reexpress this semantics in terms of taking a formula and a tree, and finding the possible ρ . It seems as though there should be an equivalent formulation with partially-defined sets of substitutions, continuity of \neg and \wedge , a bottom element that carries no information and means nontermination, and so on—but the details haven't been worked out yet.

So this is just a possible idea to formalize the behaviour of the current implementation, we don't yet know for sure that it will work and we haven't proved that it does correspond to what the program does (only that for a few cases such as $\mu\mathbf{R}.\mathbf{R}$ it seems to give the right result). It's an area for further investigation.

10.10 Static checking of queries

Some queries will fail with our data model because they attempt to parallel compose leaf data, or to duplicate addresses. It should be possible to warn about these problems before the query is evaluated—not in every case, but at least in some of them. As the data model grows more complex, it becomes more useful to have some kind of static checking that the results of queries do conform to the model. [24] defines a type system for the ambient calculus, but it is not clear whether this has any relation to our data model and query language.

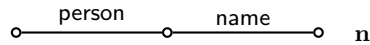
Could make sure that the semantics of a query is finite? That is necessary in the current implementation, even when the internal workings of the logic may handle infinite sets. Is there some way of deciding, without waiting for the query to run, whether it will produce an infinite result? This is undecidable for the relational tuple calculus[25], and it is suspected (but not known) that it is

undecidable in our query language too. However, it would be possible to create a conservative ‘safety checker’ for queries. Such a checker would reject as unsafe a number of queries which were in fact safe; the question is whether it can be right often enough to be useful.

Alternatively, perhaps the data model itself could be changed so that two trees can always be composed in parallel without problems. If we took the current B production as the grammar for trees, this would be the case.

10.11 Allowing query results to be infinite

Implementing support for fully general equality tests between two variables was quite hard, but it has an interesting side-effect. If you can handle a formula $\mathbf{x} = \mathbf{y}$, then you can put variables in the *input tree* and match against those just as easily as matching against constants. For example, for the tree

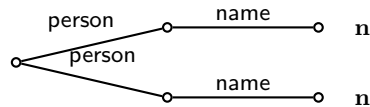


where \mathbf{n} is a variable, and for the formula $\text{person} \mapsto [\text{name} \mapsto [\text{Mickey}]]$, the set of substitutions is

$$\{\{\mathbf{x} \mapsto \text{Mickey}\}\}.$$

Or you could match a variable \mathbf{x} in the tree against a variable \mathbf{y} in the formula, and end up with all substitutions where $\mathbf{x} = \mathbf{y}$.

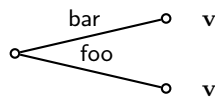
This feature sounds contrived and useless. What could it possibly mean to have variables in the tree? One answer could be that it represents uncertainty or many possible trees. For example, to say ‘there are two people with the exact same name, but I really can’t remember what it is’, you could write the tree



Still that does not sound particularly useful. But consider what happens when a *from/select* query has an infinite set of substitutions to deal with. This set is not ‘infinite’ within the program, it is represented in finite space with mappings from variables to sets of values, and relationships between variables. Suppose that we have the query

$$\text{from } T \models \varphi \text{ select } \text{foo} \mapsto [\mathbf{x}] \mid \text{bar} \mapsto [\mathbf{y}]$$

and it happens that $T \models \varphi$ returns an infinite set of results. In particular, suppose it returns all substitutions which give \mathbf{x} and \mathbf{y} the same value. Currently, there would be no way to instantiate the result from this infinite set. But if we allow variables in trees, the result of this query can be written as



where \mathbf{v} is a variable specially created for the result tree.

Now, if the above query were in fact a subquery in a larger *from/select* expression, it would be possible to pass this tree—an infinite set of trees—to the next query and have processing continue. Matching a formula against the tree containing variables will work as expected.

Now the top-level result must be finite, since usually the user won't want an output containing variables. (Although a returned tree like the one above might be a useful diagnostic, indicating to the user the place where he or she needs to constrain the query a bit more.) However, there is the possibility to make subqueries with infinite semantics, even when the top-level query has a finite result. This is a step up from the current situation, where infinite results are allowed only inside the *logic* and cannot be used in a *from/select*, even when the query as a whole has finite semantics.

However, it's possible that if a query has finite semantics, then it can always be written using subqueries with finite semantics. If that were true then this extension would not add any power to the query language. Whether it is true or not isn't currently known.

This is just a sketch of how adding variables to trees would help extend the query language to infinite results. Another issue which needs to be addressed is tree variables: if a tree variable were bound to a tree which itself contained variables, and we wished to unify this tree variable with another tree variable, then it is no longer enough just to compare the two trees and see if they are equal. For example, a simple equality test would say that the trees $\text{size} \mapsto [\mathbf{d}]$ and $\text{size} \mapsto [\mathbf{e}]$ are 'different'. But in fact they can unify, as long as $\mathbf{d} = \mathbf{e}$. Therefore it would be necessary to write a small unification engine for trees which returns the necessary variable equalities, or failure.

Also, not every infinite result is necessarily representable by the simple measure of putting variables in the tree. The set of all substitutions with \mathbf{x} *not* equal to \mathbf{y} , for example, cannot be represented in this way. So we might define the output of a query to be a tree plus a set of 'side conditions' which constrain the values in this tree. These side conditions would be added to the substitutions used to match against the tree in the parent query.

Towards the end of the project the trees-with-variables querying was implemented, since it comes for free once equality tests are done, but the need for new code to handle tree variables meant there was not enough time to implement the query mechanism with possibly-infinite results. So for the time being, the code shares the limitation of [21] that the results of a query must always be finite. There is no such restriction in the formal definition of the query language in [1].

Chapter 11

Conclusions

The aim of the project was to implement a query language based on a data model of unordered trees with graphical links, and a logic defined on that data model. The logic is based on the ambient logic, in particular sharing its \parallel parallel composition operator, which is a different approach to first-order logic. There was a previous implementation of a very similar query language on a simpler data model, and we hoped to do the same job for the new data model.

The main difference of our data model is the graphical link, which lets you build a wider range of data structures than you could with just plain trees—as the examples in the data model chapter demonstrate.

We succeeded in implementing the full logic, in terms of finding all sets of variable assignments (substitutions) letting a tree satisfy a given formula. Negation is difficult because it leads to infinite sets of substitutions, but we could adapt the technique used in the previous implementation to represent these in finite space. Equality tests also have infinite semantics and were not dealt with before. We developed a technique to handle these, as a set of constraints between variables, so every part of the logic is now implemented.

The semantics of the query language was already defined; it involves finding a set of substitutions and instantiating a result template to give an output tree. Because there could be more than one result, it is necessary to join them together into a single tree, for which we use parallel composition. Our data model doesn't allow unrestricted parallel composition so there must be some runtime checks on the query result. Also, we have a restriction that addresses be globally unique which is also checked at runtime. We showed the need for nested subqueries to get the most flexible language.

As well as the implementation, this report is intended to serve as an introduction to the data model, logic and query language. We've tried to give motivating examples for each feature, or at least those whose purpose was not always clear from the earlier literature.

The main implementation language is Haskell, used because it maps well onto the formal specification of the project, and is concise enough to explain by giving code samples in the report. A small secondary implementation in

Perl explores an alternative way of handling graphical links, since they are like pointers in computer memory.

It turned out that the implementation differed from the specification in some aspects, such as the behaviour of recursion and whether multiple copies of the same substitution should be included in a 'result'. These happened by accident, but we've tried to evaluate whether the new behaviour could be useful when building a query language, and to give well-defined semantics for it.

There remains a lot of work to do, in particular with trying to optimize implementation of the logic so that the query language becomes fast enough to use on large data sets. The ultimate aim is to integrate standard database techniques with the new data model and logic. Other areas to work on are returning multiple copies of the same result in a query, and defining a computable semantics for recursion that handles negated recursive calls.

Appendix A

Summary of the logic

This just restates what was introduced earlier, so it's easier to refer to. Logical formulæ are defined as:

label expression	$\alpha ::= a$	label
	\mathbf{a}	label variable
address expression	$\xi ::= x$	address
	\mathbf{x}	address variable
data expression	$\delta ::= d$	leaf data
	\mathbf{d}	data variable
formulæ	$\varphi, \psi ::=$	
tree structure	\mathbf{nil}	match empty tree
	$\alpha \mapsto \xi[\varphi]$	match tree branch (address ξ)
	$\alpha @ \xi$	match graphical link to address ξ
	δ	data
	$\varphi \psi$	composition
classical logic	\mathbf{true}	true
	$\varphi \wedge \psi$	conjunction
	$\neg \varphi$	negation
equality tests	$\alpha_1 = \alpha_2$	label equality
	$\xi_1 = \xi_2$	address equality
	$\delta_1 = \delta_2$	data equality
recursion	$\mu \mathbf{R}. \varphi$	least fixed point
	\mathbf{R}	recursive call
tree variable	\mathbf{t}	
there-exists	$\exists \mathbf{a}$	exists label
	$\exists \mathbf{x}$	exists address
	$\exists \mathbf{t}$	exists tree

Satisfaction is defined in terms of a substitution ρ which maps label variables

a to labels, address variables **x** to addresses, data variables **d** to leaf data, and tree variables **t** to trees:

$$\begin{array}{ll}
T \vDash^\rho \text{nil} & \Leftrightarrow T \equiv \text{nil} \\
T \vDash^\rho \alpha \mapsto \xi[\varphi] & \Leftrightarrow T \equiv \rho(\alpha) \mapsto \rho(\xi)[T'] \wedge T' \vDash^\rho \varphi \\
T \vDash^\rho \alpha @ \xi & \Leftrightarrow T \equiv \rho(\alpha) @ \rho(\xi) \\
T \vDash^\rho \delta & \Leftrightarrow T \equiv \rho(\delta) \\
T \vDash^\rho \varphi | \psi & \Leftrightarrow \exists T_1, T_2 \in \mathcal{T}. (T \equiv T_1 | T_2 \wedge T_1 \vDash^\rho \varphi \wedge T_2 \vDash^\rho \psi) \\
T \vDash^\rho \text{true} & \text{always} \\
T \vDash^\rho \varphi \wedge \psi & \Leftrightarrow T \vDash^\rho \varphi \wedge T \vDash^\rho \psi \\
T \vDash^\rho \neg \varphi & \Leftrightarrow \neg(T \vDash^\rho \varphi) \\
T \vDash^\rho \alpha_1 = \alpha_2 & \Leftrightarrow \rho(\alpha_1) = \rho(\alpha_2) \\
T \vDash^\rho \xi_1 = \xi_2 & \Leftrightarrow \rho(\xi_1) = \rho(\xi_2) \\
T \vDash^\rho \delta_1 = \delta_2 & \Leftrightarrow \rho(\delta_1) = \rho(\delta_2) \\
T \vDash^\rho \mu \mathbf{R}. \varphi & \Leftrightarrow T \vDash^\rho \varphi[\mu \mathbf{R}. \varphi / \mathbf{R}] \\
T \vDash^\rho \mathbf{t} & \Leftrightarrow \rho(\mathbf{t}) \equiv T \\
T \vDash^\rho \exists \mathbf{a}. \varphi & \Leftrightarrow \exists \mathbf{a}. T \vDash^{\rho[\mathbf{a} \mapsto \mathbf{a}]} \varphi \\
T \vDash^\rho \exists \mathbf{x}. \varphi & \Leftrightarrow \exists \mathbf{x}. T \vDash^{\rho[\mathbf{x} \mapsto \mathbf{x}]} \varphi \\
T \vDash^\rho \exists \mathbf{d}. \varphi & \Leftrightarrow \exists \mathbf{d}. T \vDash^{\rho[\mathbf{d} \mapsto \mathbf{d}]} \varphi
\end{array}$$

The definition for recursion is the least fixed point of its equation given above, in that any other fixed point would match more trees for some ρ and at least as many for all other ρ .

Appendix B

Testing

The Haskell implementation has a suite of 75 test cases which exercise different parts of the code. In fact, querying a tree with a complex formula covers all of the code, but smaller test cases for individual functions are useful to track down errors. Where possible, the test cases were written along with the code they test, to give some reassurance that the low-level functions worked correctly before moving on to implementing more complex routines using them. In Haskell, a bug in a small function somewhere can cause completely mystifying errors on the large scale! But most of the tests are ‘top-level’ tests for running a query, or at least for matching a formula against a tree and checking the substitutions returned.

In particular, there are test cases for all the example formulæ and queries mentioned in this report (apart from those which are flagged as incorrect, or mentioned as future improvements). Also every example mentioned in [1] as an example use of the TQL query tool has been added to the test suite, except for one which was not possible due to differences in the data model.

The small Perl implementation also has a test suite, but it is less comprehensive. The purpose of the Perl version is to experiment with graphical links and to generate diagrams, so it doesn’t need as much testing.

The Haskell test suite is in the file `test.hs` and works simply by having a list of functions, input values, and expected outputs. Each function is run on its input values and the output is compared with what’s expected. In Haskell a ‘set’ value (such as a set of variable assignments) is represented as a list, but the `==` function on lists takes account of ordering. It was necessary to define new data types to wrap the lists and define new `==` and `!=` functions which disregard ordering. These new data types are the main reason why the running code differs from the presentation in this report.

There is not a complete code listing here, but just to give the general idea here is one particular test case of the 75:

- 'Using recursion, we look for email at the current level or,
- recursively, at any inner nesting level'.

```

qttest6 :: QueryTest
qttest6 = ("6",
  (FromSelectT
    articles_email
    (Fun "R" (LOr
      (LOr
        (LCompose (LBranch (C "email") (V "ea") (LTreeVar "X")) LTrue)
        (LCompose (LBranch (C "e-mail") (V "ea") (LTreeVar "X")) LTrue))
      (LCompose
        (ExistsLabel "l" (LBranch_IgnoreAddr (V "l") (Recurse "R"))
          LTrue)))
    (QBranch (C "email") (V "ea") (QTreeVar "X"))),
  Parr [ Branch "email" 4 (LeafData "luca@microsoft.com"),
        Branch "email" 9 (LeafData "ghelli@upisa.it") ])

```

This is one of the examples taken from [1], and it illustrates just how much syntax is needed to express the logic and queries in Haskell. This particular test will be run using the `query` function and it says, *from* the `articles_email` database (not shown), *satisfying* the specified formula (look recursively for ‘email’ or ‘e-mail’ subtrees and bind `ea` to their address and `X` to their contents), *select* the template email $\mapsto \mathbf{ea[X]}$. The expected result is a tree with two ‘email’ branches. Those email addresses, by the way, are completely fictitious—the test suite does not check that yet!

The test suite can be run from Hugs by loading `test.hs` and typing `run_tests`.

Appendix C

Getting the code

The working copy of the code is in <http://membled.com/work/projects/my/>. Ask me if you want access to the CVS repository (which would be a much more convenient way to grab the code).

Bibliography

- [1] Luca Cardelli and Giorgio Ghelli. A query language based on the ambient logic. In David Sands, ed., *Lecture Notes in Computer Science 2028*. Springer, 2001
- [2] Serge Abiteboul, Dan Suciu, and Peter Buneman. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999
- [3] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pp. 365–377. 2001
- [4] PNG (Portable Network Graphics) Specification, Version 1.2
URL <http://www.libpng.org/pub/png/spec/png-1.2-pdg.html>
- [5] Extensible Markup Language (XML) 1.0 (Second Edition). World Wide Web Consortium
URL <http://www.w3.org/TR/2000/REC-xml-20001006>
- [6] ISO 8879: Standard Generalized Markup Language. International Organization for Standardization (ISO), 1986
- [7] Kristoffer Rose. Generating Fast Validating XML Processors (extended abstract)
URL <http://flexml.sourceforge.net/>
- [8] SAX, the Simple API for XML
URL <http://www.saxproject.org/>
- [9] An SGML System Conforming to International Standard ISO 8879
URL <http://www.jclark.com/sp/nsgmls.htm>
- [10] XML Schema 1.0. World Wide Web Consortium
URL <http://www.w3.org/XML/Schema>
- [11] N. Klarlund, A. Moller, and M. I. Schwartzbach. DSD: A schema language for XML. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*. Portland, OR, 2000
- [12] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. In *International Workshop on the Web and Databases (WebDB)*. Dallas, TX, 2000

- [13] Anders Møller and Michael I. Schwartzbach. The XML Revolution: Technologies for the future Web. December 2001
- [14] X.500: The Directory. International Telecommunications Union (ITU), 1993
- [15] W. Yeong and T. Howes. RFC 1777: Lightweight Directory Access Protocol. 1995
- [16] D W Chadwick. Understanding X.500—The Directory. Chapman and Hall. ISBN 0-412-43020-7
- [17] Luca Cardelli. Describing semistructured data. *SIGMOD Record*, December 2001
- [18] Philippa Gardner Luca Cardelli and Giorgio Ghelli. A spatial logic for querying graphs. In International colloquium on automata, languages and programming. To appear
- [19] Luca Cardelli, Philippa Gardner, and Giorgio Ghelli. Reasoning about trees with dangling pointers, 2002. In preparation
- [20] Silvano Dal Zilio. Fixed points in the ambient logic. In Lecture Notes in Computer Science. Springer, 2001
- [21] Giovanni Conforti and Orlando Ferrara. TQL algebra and its implementation, October 2001
- [22] XHTML 1.0: the Extensible Hypertext Markup Language. World Wide Web Consortium
URL <http://www.w3.org/TR/xhtml1/>
- [23] XPath 1.0. World Wide Web Consortium
URL <http://www.w3.org/TR/xpath>
- [24] Giorgio Ghelli Luca Cardelli and Andrew D. Gordon. Types for the ambient calculus. To appear in I&C special issue on TCS'2000
- [25] S. Abiteboul, R. Hull, and V. Vianu. Foundations of databases. Addison-Wesley, 1995