

TYPE-SAFE COMPUTATION WITH HETEROGENEOUS
DATA

by

FREEMAN YUFEI HUANG

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada

August 2007

Copyright © Freeman Yufei Huang, 2007

Abstract

Computation with large-scale heterogeneous data typically requires universal traversal to search for *all* occurrences of a substructure that matches a possibly *complex* search pattern, whose context may be *different* in different places within the data. Both aspects cause difficulty for existing general-purpose programming languages, because these languages are designed for homogeneous data and have problems typing the different substructures in heterogeneous data, and the complex patterns to match with the substructures. Programmers either have to hard-code the structures and search patterns, preventing programs from being reusable and scalable, or have to use low-level untyped programming or programming with special-purpose query languages, opening the door to type mismatches that cause a high risk of program correctness and security problems.

This thesis invents the concept of pattern structures, and proposes a general solution to the above problems - a programming technique using pattern structures. In this solution, well-typed pattern structures are defined to represent complex search patterns, and pattern searching over heterogeneous data is programmed with pattern parameters, in a statically-typed language that supports first-class typing of structures and patterns. The resulting programs are statically-typed, highly reusable for different data structures and different patterns, and highly scalable in terms of the

complexity of data structures and patterns. Adding new kinds of patterns for an application no longer requires changing the language in use or creating new ones, but is only a programming task. The thesis demonstrates the application of this approach to, and its advantages in, two important examples of computation with heterogeneous data, i.e., XML data processing and Java bytecode analysis.

Keywords: pattern structure, Pattern Calculus, generic programming, heterogeneous data, type safety, XML processing, Java bytecode checking.

To My Father

I dedicate this thesis to my dear father, the late Professor Shutang Huang, who gave me all the most important personalities, including persistence, discipline, logicalness, and intelligence, for the success of this thesis. He was not only my first and all-the-time teacher, but also my life-time role model. He was a man of so few needs or wants but giving so much to the family. He was a man of few words, not saying “I love you” often, but exhausting his best to pour care and love on us, till his last minute. I regret that I did so little in return. I regret that I had not taken better care of his health, not paid more attention to his feelings, not spent more time with him and listening to him. However, I have no chance to compensate. I even have no chance to say “I am sorry, Dad”. My only hope is that this thesis can bring him some happiness, as he was always proud of every bit of my progress and achievement.

We miss you so much, Ba-ba.

About my father: The late Shutang Huang, 1930.07.13–2005.05.03, then Professor Emeritus in Mathematics of Zhongshan University in Guangzhou, China, devoted his whole career life to university education and research in Differential Geometry, with the latest focus on Riemannian Geometry, and devoted all his time beside career to caring of family till the last minute. He was much respected and loved by his colleagues, students, and the family.

Co-authorship

Published Paper “Programming with Heterogeneous Structures: Manipulating XML data Using `bondi`” [57] co-authored by myself, C. Barry Jay, and David B. Skillicorn is based on materials in early versions of Section 3.2 and Chapter 4.

Published Paper “Adaptiveness in Well-Typed Java Bytecode Verification” [56] co-authored by myself, C. Barry Jay, and David B. Skillicorn is based on material in an early version of Chapter 5.

Acknowledgments

I would like to thank my supervisor, Prof. David Skillicorn for his guidance, encouragement, support, and patience. He has been of great assistance throughout the course of the thesis research. I have been obtaining from him, not only strong academic guidance and financial support, but also enthusiasm and skills for efficient research work.

I would also like to thank Prof. Juergen Dingel and Prof. Pat Martin, who are members of my supervisory committee, for their ongoing guidance and review of my research and their invaluable and insightful comments.

Special thanks to Prof. Barry Jay at University of Technologies, Sydney, Australia, for his novel theory, the Pattern Calculus, and his collaboration, which made this thesis possible. I also appreciate Barry's patience in answering my questions, providing insightful opinions and references when I was finalizing the thesis.

Contents

1	Introduction	1
1.1	Problem to Solve	1
1.2	Motivating Scenario	7
1.3	Thesis Contribution	12
1.4	Organization	13
2	Literature Review	14
2.1	Computation with Heterogeneous Data	14
2.2	Enforcing Program Security	20
2.2.1	Security Policies and Enforcement	20
2.2.2	Security of Mobile Programs	23
2.2.3	Security with Java	25
2.3	XML Data Processing	26
2.3.1	XML Query Languages	27
2.3.2	Native XML Processing	28
2.4	Pattern Calculus and <code>bondi</code>	33
2.4.1	Static Typing and Expressiveness	33
2.4.2	Structure Polymorphism in Pattern Calculus	35

2.4.3	Path Polymorphism in <code>bondi</code>	44
2.4.4	Pattern Polymorphism in <code>bondi</code>	46
2.4.5	Summary	47
3	Expressing Computations on Heterogeneous Data	49
3.1	The Challenge of Expressing Pattern Searching	50
3.1.1	Pattern Searching in OCaml	51
3.1.2	Pattern Searching in Java	54
3.1.3	Pattern Searching using SYB in Haskell	58
3.2	Pattern Searching with Pattern Structures	60
3.3	Understanding and Creating Pattern Structures	73
3.4	Summary	78
4	XML Data Processing	80
4.1	Representing and Transforming XML Data	81
4.2	Vertical XPath Patterns	86
4.3	Horizontal Regular Expressions	94
4.4	Summary	96
5	Java Bytecode Analysis	98
5.1	Representing Java Bytecode	99
5.2	Semantic Checking of Bytecode	104
5.3	Java Bytecode Instrumentation	107
5.4	Summary	109
6	Conclusion and Future Work	110

Bibliography

116

Chapter 1

Introduction

1.1 Problem to Solve

Data-intensive applications manipulate large-scale data, which are organized in computable structures. Tabular structures have been widely used, especially in relational databases where data is in the form of relational table, a homogeneous list of tuples (records) of primitive items. Recently, data are becoming not only larger in size, but also more complex in structure. In many situations they are better represented as heterogeneous structures for processing.

A data structure is said to be (structurally) heterogeneous if it has multiple sub-structures that are not all the same. For example, suppose a piece of geographical data contains information about names, areas and populations of countries and their provinces and cities. It is more natural to represent such data as one heterogeneous tree with sub-trees for countries, each of which has different sub-trees respectively for country name, area, population and provinces, and each province sub-tree has different sub-trees respectively for province name, area, population and cities, and

so on. Practical examples of heterogeneous data include semi-structured data, such as XML data, and unstructured data, such as plain textual documents and static program codes.

A basic computation in typical manipulation of large data is a search for target items that match a *pattern*. General speaking, a pattern, or a search pattern in the sense of data processing, is a template of value or structure that is exhibited by every member of a group of things or parts of things in common. In programming for data processing, a pattern may be as *simple* as a primitive value, or a data structure with element values and sub-structures; it may also be as *complex* as a combination of several simple patterns; the values and structures in a pattern may be represented by variables. Pattern matching is the act of checking for the presence of the constituents of a given pattern in a piece of structured data, and the matching values or structures are called targets. Typical manipulation of large data consists of looking for targets that match a pattern, then extracting information from, or updating, the content of the matching targets.

In the case of heterogeneous data, such pattern-searching computation requires finding *all* matching substructures whose context may be *different* in different places within the whole data. For example, in a geographical data repository, a task to increase population of a country by 1% requires updating all population elements (the targets) under different substructures within the country, such as cities, provinces, and the country itself. Also, the complexity of the search pattern may be *different* for different tasks. A province-wide population increase requires searching for populations within the specific province (a complex pattern) to update, while leaving the other provinces unchanged. All these aspects are the results of data heterogeneity. A

practical program for handling heterogeneous data should be able to adapt to all of them.

However, handling data heterogeneity is a challenge for most existing general-purpose programming languages. When developing a program for a pattern searching task, it is a problem to arrange types for the pattern and the substructures, because the pattern will be compared with substructures of different types at different places while traversing the whole data. The typing problem is even worse when the program is to be reusable for different pattern search tasks, usually with different data structures, different patterns and/or different complexity of patterns. In most languages, changes in data structures and search patterns require re-design and re-compiling of the programs, and sometimes even require changes to the programming languages. Reusability is very limited, if possible at all.

Of course, these problems can be avoided by programming in a low-level untyped way, using the same primitive data type such as character string, or objects inheriting the same super class, to represent all different substructures and the patterns to compare with them. But then programmers need to manually insert checking code in the programs in order to verify the types of values at runtime, to ensure type safety. This is tedious and error-prone, and the extra checking code incurs runtime overhead. Even doing so does not guarantee correctness and security of the resulting programs. Intensive analysis and testing are still required, or, alternatively, sophisticated execution monitoring must be used. This is especially a problem in situations where programs are developed by untrusted parties, so that proper programming, analysis and testing are not guaranteed. For example, mobile programs, such as mobile codes

and mobile agents, are likely to be used in data-intensive applications. Dynamically-arriving mobile programs may come from untrusted or even unknown sources, giving no confidence that the programs are correct and safe to run. Execution hosts need to verify them for the hosts' security. Such verification should be done as quickly as possible for reasonable response time. Full analysis and testing are not possible in this situation, while execution monitoring requires a much more complex runtime system, so that it is, in its own way, riskier for security, and incurs runtime overhead.

The difficulties with general-purpose languages led to the design of special-purpose query languages. However, solutions using query languages are unsatisfactory for general processing of heterogeneous data. The problem with using a query language is that it creates an interface between data extraction and data use. For example, in a typical web environment, the data itself is in a back-end system that supports a query language, in which the patterns to search for are typed. But the results of data queries must be passed to a front-end program developed in a general-purpose language for further processing. The existence of a boundary between two programming environments requires a common format, usually quite a low-level one such as character strings, by which the back-end and front-end communicate. This has been called the *impedance mismatch problem* [6, 104]. It creates a type-safety problem similar to untyped programming, requiring extra programming effort and runtime overhead for the front-end to translate and verify the untyped messages from the back-end, and requiring intensive analysis/testing or sophisticated execution monitoring, for correctness and security purposes.

Overall, existing programming approaches have difficulty realizing the following desirable characteristics when implementing large-scale heterogeneous data processing

programs. Each approach may be good at one or other of these characteristics, but they have not been able to achieve these characteristics together:

- *Type Safety.* Well-typed programs, in a statically-typed language, are desirable. Typed programs can be verified by static type checking for correctness and security, before execution. Static type checking is efficient, and a type checking program is much smaller than a sophisticated analysis or monitoring system, and so easier to develop, verify and maintain. This is especially beneficial for systems involving mobile programs.
- *Reusability.* Programs should be able to process different pieces of data, data of different structures, and different search patterns, in order to reduce the programming effort for these different cases.
- *Scalability.* Reusability also means that programs are able to adapt to increasing heterogeneity of data, and to increasing complexity of search patterns.
- *Uniform language environment.* Implementing a whole application in one single language or at least in one single type system is desirable, to avoid type mismatches among cooperating programs. Such a mismatch would cause extra workload in programming and verification/testing for correctness and security, and cause a runtime performance penalty.

This thesis propose a general solution to achieve all the above characteristics, even in the presence of arbitrarily heterogeneous data. The thesis proposes a general kind of well-typed constructs called *pattern structures*, based on a newly-designed general-purpose programming language that supports first-class typing of structures and patterns, and proposes a programming technique that uses pattern structures.

An instance of a pattern structure can encapsulate all of the information needed to guide a pattern searching program to traverse a piece of heterogeneous data and to reach all targets. Using pattern structures in programming, in addition to first-class typing of structures and patterns, provides sufficient expressiveness to develop parametric programs that can take both well-typed data and well-typed patterns as arguments, even if the structures of the data are arbitrarily heterogeneous and patterns are arbitrarily complex. Most importantly, this can be done in one single language framework.

Such a programming technique is advantageous in implementing computation with heterogeneous data, especially when security of the resulting programs is critical and so preferably to be enforced by static type checking. Two good examples of such computations are XML data processing, and Java bytecode analysis, which have recently gained extensive use in practice. The thesis demonstrates the applications of the proposed approach to implementing XML data manipulation, and Java bytecode security checking/instrumentation, as illustrations of the potential use of the approach in practice. It shows that, using the proposed approach, the computation with XML data and Java bytecode data can be straightforwardly expressed as well-typed highly-reusable programs in a single programming environment, while in other existing programming approaches, achieving all these at the same time is difficult, if possible at all. Actually, the proposed approach fits well with not only Java bytecode analysis, but also any program code manipulation in general. For example, although not demonstrated in this thesis, the approach is expected to be advantageous in implementing aspect-oriented programming frameworks [39], which involve weaving of crosscutting aspect codes into programs [101].

1.2 Motivating Scenario

This section shows the many levels of need for expressiveness in implementing computation with heterogeneous data. To do so, a programming scenario of such computation tasks is examined, which motivated this thesis.

Suppose we have a data repository containing geographical information and we want to carry out the following operation: *Add 1% to the population of all the Canadian cities*. How could we express such an operation?

The first way is what might be called assembly language programming: a specific program that traverses the specific structure of the repository, finds all of the places where Canadian cities are present, and then finds their population elements and adds 1% to them. The problem is that if we decide to change the task in any way we have to rewrite and recompile the whole program.

The situation can be improved by making a program with a parameter, passing to it the amount by which the populations are to be incremented as argument. So the computation might be expressed as something like:

incrementPopsOfCanadianCities(1%)

This small change increases the generality of the program. With this program, we can make increments of different percentages to population without rewriting or recompiling the program. Simply speaking, the program is reusable.

It is also desirable to make the operation that is to be done to the populations of Canadian cities into a parameter as well. So the computation might be written as:

updatePopsOfCanadianCities(incrementby, 1%)

Now it is trivial to decrement the populations instead, by passing a decrement function to the first parameter, without any change to the program. We obtain another level of program reusability. Most existing programming languages should have no problem with programs up to this point.

We may need to apply the operation to several data repositories from different sources. So we might also like to have a program to take a data repository as argument too:

updatePopsOfCanadianCities(incrementby, 1%, data)

Taking atomic data values of the same type as arguments is easy for most existing languages, such as taking the percentage value as argument. But it is a different story here because *data* is a compound value, an instance of some data structure. The program needs to traverse the structure of *data*, find all population values and update them.

For most programming languages, it is not a big problem to make a program take any instance of one particular data structure as argument, for example, a program accepting any list, or a program accepting any binary tree, because the program can hard-code the way of traversing the expected structure and recognizing each type of substructure met en route. But then each and every possible structure would need a different hard-coding program to handle it. What about a program that can accept any kind of data structure? Most languages have problems when parameter *data* needs to accept argument of different kinds of data structures, for example, the argument passed to *data* at runtime may be a list, a binary tree, or something else. It is even worse when the data structures are heterogeneous, for example a tree with different types of sub-trees and different types of leaf values. Hard-coding for

all possible structures and all possible substructures in one program is clumsy and error-prone, and the program would need to be rewritten and recompiled whenever there is a need to include a new data structure or a change in an existing structure.

However, it is desirable to have one such single program that can accept as argument any instances of arbitrary structures, so that the program is reusable for different data repositories of different structures. Then when the size and heterogeneity of the data structure increase, there is no need to change the program or build more complicated programs. Such a program will not only be reusable, but also scalable in terms of the size and heterogeneity of data structures.

The next level of parameterization is to make the target where the operation is applied into a parameter as well. So we might write:

$$\text{updateCanadianCities}(\text{Pops}, \text{incrementby}, 1\%, \text{data})$$

Pops here is actually the pattern with which the target substructures must match. So this program is reusable for different patterns. Now it is trivial to increment (or decrement) cities' *areas* instead of their populations using the same program. Most existing programming languages have trouble typing this program, because potential target substructures to match may appear in different contexts. It is hard to arrange an appropriate type for the pattern *Pops*, which will be compared with different types of substructures along the paths to different instances of population elements.

A further extension is to make the particular units within Canada that are being considered into a parameter. So we might write:

$$\text{updateInCanada}(\text{City}, \text{Pops}, \text{incrementby}, 1\%, \text{data})$$

Now the program is generic in the pattern that describes *where* the increment is to be applied (populations under cities). It can update all city populations regardless of whether cities are immediately below countries, e.g., the capital Ottawa, or accessed via intermediate layers such as provinces. It can also be used for a task to update provinces' areas instead of cities' populations.

Now let us parameterize on the country too:

```
update("Canada", City, Pops, incrementby, 1%, data)
```

This code involves a side-condition to check on a related structure: the name of the country must be "Canada".

Note that the multiple parameters for "Canada", *City* and *Pops* are all related and it is the *connections* between them that define the real parameter of interest. So we could rewrite the computation as:

```
update(Canadian_City_Pop)(incrementby, 1%, data)
```

The program used here has a parameter to accept a *complex* pattern argument. If we want to search for more complicated patterns within the geographical database, we don't have to keep building more complicated functions; rather, the complexity can be expressed in the choice of a complex pattern for the pattern parameter of the standard *update* function. We obtain another level of program scalability in terms of the complexity of patterns.

This scenario shows the many levels of need for parameterization in processing arbitrary, and typically heterogeneous, data structures. The situation is common in real applications where input data are organized as heterogeneous trees, or even

graphs. It is desirable to make parameterized programs and capture as much behavior of a computation as possible as parameters, so that the programs are reusable for different values of the parameters, and are scalable and easy to maintain. Although theoretically a computation task can always be programmed without using any parameter by hard-coding all the operand values and structures needed by the computation, such programming would be very tedious and error-prone, and the resulting programs would be hard to understand and maintain. As the complexity of data structures and patterns increase, such programming becomes more and more complicated.

However, it is a challenge to develop highly-parameterized but statically-typed programs. It will be shown in this thesis that, while it is easy to have programs parameterized on increment operation and percentage value, most programming languages have typing problem for the rest of the parameterization needs, especially for parameterizing patterns.

One possible solution is to implement the parameterized programs at a low-level, breaking the static-typing requirement. For example, substructures, and patterns to match them, can all be represented as strings, or all as objects, which can easily be parameterized, and matching becomes string comparison or object type identification. Another possible solution is to use special-purpose query languages that support well-typing of patterns. However, as described in the previous subsection and in more detail in the following chapters, both solutions are unsatisfactory, incurring extra workload for programming, verification, testing, and extra runtime overhead, and are especially disadvantageous in security-critical situations such as those involving mobile programs.

The programming technique proposed by this thesis is a general solution that can achieve all the above-mentioned levels of parameterization in a statically-typed way, in a single language framework.

1.3 Thesis Contribution

The contributions of this thesis are:

- It argues that existing general-purpose programming languages in practical use are not equipped to handle heterogeneous data, and shows why the languages have difficulty to program with heterogeneous data.
- It invents the concept of *Pattern Structures*, and proposes a general programming technique that constructs well-typed pattern structures to represent complex patterns in a language that support first-class typing of structures and patterns, and expresses pattern searching computation in the same language with the help of pattern structures. This solution enables programming for heterogeneous data processing in a well-typed, highly reusable, and highly scalable manner.
- It establishes proof-of-concept application scenarios of XML data processing and Java bytecode analysis, two important examples of computation with large heterogeneous data that have been extensively used in practice. Applying the proposed solution to these scenarios demonstrates the potential of the solution in practical use, and shows that the solution provides a better programming framework for programs that process large-scale distributed data, and for programs that verify security-critical programs.

1.4 Organization

The rest of the thesis is organized as follows. Chapter 2 reviews related research and results in computation with heterogeneous data, mobile computation, security enforcement, XML data processing, and the Pattern Calculus. Chapter 3 shows why existing standard programming languages have difficulty in programming with heterogeneous data, then introduces the concept of pattern structure and the programming technique using pattern structures to solve the difficulty. Chapter 4 demonstrates how the use of pattern structures can lead to better programming practice for applications with XML data processing. Chapter 5 demonstrates how the use of pattern structures can lead to better programming practice for implementing Java bytecode verification and instrumentation for security enforcement. Chapter 6 draws the conclusion and discusses possible future work.

Chapter 2

Literature Review

2.1 Computation with Heterogeneous Data

As processing power and storage capability of computers are increasing rapidly, application data are also becoming extremely large in size and complex in structure. Structurally homogeneous tabular data structures, such as in relational databases, used to be predominant in organizing large data. However, heterogeneous data structures, such as general trees and graphs, have recently been getting extensive use in practice, in settings where data are structurally complex and unsuitable to be represented as relational tables. For example, XML [18] is a common format to represent online information for exchangeability. Most leading search engines and e-commerce service providers, including Google, Yahoo, EBay and Amazon, are providing XML views of their databases [24, 35]. They are also turning their online service interfaces into Web Services [14, 50, 68, 81], a set of XML-based standards, so that the search, negotiation and use of online services are all in XML style. XML's hierarchical data model makes it natural to represent semi-structured XML files by heterogeneous tree

structures during processing. Other examples of heterogeneous data include unstructured textual documents and program codes that are usually transformed into syntax parse trees for analysis; molecular structure information in biopharmaceutics and biochemistry, which is usually represented as graphs for structural analysis [2, 15, 71, 94]; standard RDF [69] files that describe online knowledge as resource description graphs for computer programs to comprehend; and so on.

Computations with these large heterogeneous data tend to be globally distributed, due to the global distribution of either the data, the data users, or both. To adapt to the increasing size, heterogeneity and distribution of data, the architecture of the computations has been evolving more and more towards moving computation to data:

- Client/server.

Traditional client/server architecture splits a data processing task into two parts: the data server processes its database and generates intermediate results, and user(client) programs on remote computers further process the results. For the server and its clients to communicate, a query language or a remote procedure call interface such as CORBA [84] and Java RMI [61] is used. With the rise of heterogeneous data, a quick (but not necessarily satisfactory) solution is to use the mature client/server architecture with some adaptation. This has been the case for most XML applications, with the emergence of a large number of XML query languages (see discussion regarding query languages below). It has also been the case in many online molecular data servers [2, 10, 15, 71, 94]. Each of these servers usually supports both queries in a query language and remote invocations of a set of standard analysis program procedures. A disadvantage of using client/server architecture is that the ways to manipulate the data are

restricted by the, typically limited, expressiveness of the query language and/or the limited set of procedures available on the server for remote invocation. This may not be a problem for relational data, whose structure is homogeneous and simple, but can be a problem for heterogeneous data, whose structure is complex. There may be too many possible ways of data manipulation to be completely covered by the query language and pre-stored procedures. This has been one of the motivations to the solution of moving data to computation.

- Moving data to computation.

The approach of replicating raw data to user-designated compute servers for processing was first used in particle physics for simulation data processing [9], where the computation with the large data requires intensive processor power that cannot be satisfied by a single regular data server but needs a pool of powerful compute servers. Later on, many molecular data servers (e.g., [10, 15]) allowed users to download raw data files to their local computers, automatically or manually, for user-customized analyses. They did so because of the abundance of methods for analyzing molecular data, some of which might have been developed after the servers were implemented and some might have been users' business secrets. These might not all have been expressible using query languages, or covered by the standard analysis procedures on the servers. Storing data on users' computers gives users direct access to the data. Users can implement any computation they want, using any language with sufficient expressiveness. However, this approach requires that every user has full access to some powerful computer. Another problem is that moving large data on global network is difficult because of technical and economical limitations to

network bandwidth. Even if solutions like peer-to-peer file sharing [30] and data compression can be used, when the data are extremely large, at the scale of a petabyte or even more, moving the data to users' local computers or user-designated compute servers becomes impractical.

- Moving computation to data.

The immovability of extremely large data has led to the idea of coupling super processing power with the data servers, with users sending their custom programs to the data servers for execution [8, 99]. Large data are fully processed on the data servers without any moving. Mobile-program techniques such as mobile code [19] and mobile agents [47] may be used to move user programs towards data. However, opening data servers to mobile programs raises security concerns for the servers. Protecting mobile program hosts against erroneous and malicious mobile programs is an active research area. A review on security enforcement techniques, especially for mobile programs, can be found in Section 2.2.

Besides architecture, another issue regarding computation with heterogeneous data is the programming approach for the computation. As described in Section 1.1, one basic computation in manipulating heterogeneous data is to search for a pattern in the data and apply some operation on the target elements found. In practice, there are two styles of implementations for such computation: indirect data manipulation through a data management system with a special-purpose query language (remote procedure call interfaces can be deemed as a simplified form of query language), and direct data manipulation fully in a general-purpose programming language. Neither of them has been satisfactory so far.

The great success of using SQL [48] in handling relational databases has led to attempts to extend the approach to heterogeneous data manipulation. A large number of special-purpose query languages for XML data processing [1, 13, 22, 26, 28, 33, 91] have been proposed, emerging from both the database community and the structured-text community. Graph query languages also arose in the bioinformatics community for structural search and analysis of molecular data [31, 44], and in the Semantic Web [98] community for online knowledge processing [85, 97].

However, special-purpose query languages usually have limited expressive power designed for specific applications. This creates problems, which become prominent in the presence of heterogeneous data:

- Poor extensibility. The limited expressiveness ties programmers' hands when confronting diverse and sophisticated computations with heterogeneous data. The need for a new kind of (complex) patterns usually means redesign of the query language, or invention of a new one, resulting in big changes to the implementation of the data management system (e.g., [37]).
- Impedance Mismatch problem [6, 104]. Due to the limited expressiveness of query languages, query results must be passed into user programs developed in a general-purpose language for further processing, incurring type mismatches between the two programming environments. This creates the chance of security holes; causes tedious extra workload for the programmer and the program verifier to ensure type correctness of the user program; and causes runtime overhead for translation between the two environments.

A review of these problems can also be found in Section 2.3 in the context of XML query languages.

Manipulating heterogeneous data directly in general-purpose programming languages has not been well-addressed, until recently. All standard programming languages currently in practical use (e.g., Java [45], C++ [100], OCaml [76], Haskell [89], etc.) have built-in mechanisms to express computation with homogeneous data structures, such as homogeneous lists (arrays) and binary trees with only one type of node values, but not computation with heterogeneous data structures. For example, when expressing pattern searching over heterogeneous trees using these languages, one has to either (1) assume a specific tree structure and hard-code all the types of nodes and the ways of branching at each type of nodes (the program is not reusable for other tree structures); or (2) program at a low untyped level, (say, express nodes by strings and searches by string comparisons, or by subclasses of a top class so that searching requires lots of class casting), use runtime type checking and allow runtime errors, incurring significant performance drawbacks and security risks.

Demonstration of the programming problem with heterogeneous data can be found in Subsection 2.3.2 in the context of XML data, and in Section 3.1 for arbitrary heterogeneous data.

Significant efforts to improve the situation have only been seen recently, including the Pattern Calculus [62, 66], and a recent proposed extension to Haskell based on an approach called “scrap your boilerplate” (SYB) [73, 74]. The Pattern Calculus allows first-class terms of structures and patterns with static types, and allows a generalized pattern matching, achieving parametric polymorphism on both structures, data access paths, and search patterns. It provides the basis for this thesis. Discussion of the Pattern Calculus is in Section 2.4. The SYB extension to Haskell is built on Haskell’s type class system and type-safe casting, achieving type-safe traversal of arbitrary

heterogeneous data by function overloading. However this approach is not able to support programs with parameters to accept search patterns, so that the change of search pattern requires re-programming, and programming for complex patterns is difficult [72]. Discussion of the SYB approach can be found in Section 3.1.3.

A related research topic, the visitor design pattern [42, 49, 87], also concerns traversal of a heterogeneous object structure. However, it is more of a conceptual programming scheme than a program template or library. It is still a problem to implement a highly reusable visitor that can adapt to arbitrary object structures using existing programming languages in practice [43, 79].

2.2 Enforcing Program Security

2.2.1 Security Policies and Enforcement

A security policy defines what programs, or what actions of programs, are or are not acceptable for the purpose of maintaining security of computing systems that execute the programs. Automata are common abstractions of computing systems, so that it is also common to formalize security policies using set- and automata-theoretic approaches [96]. A security policy can be represented as a predicate $\mathbb{P}(T)$ on the set T of all possible state-transition traces of an automaton, where a state-transition trace $t \in T$ is a sequence of state-action pairs representing program execution steps, in the form $(q_0, a_0), (q_1, a_1), (q_2, a_2), \dots$, where q_0, q_1, q_2, \dots are system states, a_0, a_1, a_2, \dots are possible execution actions, and $(q_i, a_i) \rightarrow q_{i+1}$ conforms to the transition relation of that automaton. Theoretically, a program g is violating a security policy if its set of all possible execution traces T_g does not satisfy $\mathbb{P}(T)$, i.e., $\mathbb{P}(T_g) = false$.

Security policies can be enforced by verifying static codes of target programs, or by monitoring the execution of target programs at runtime.

Traditional verification of static program code uses program-analysis techniques. It explores an entire target program g to collect information and fully construct T_g , and then verifies $\mathbb{P}(T_g)$, as in [67]. In practice, $\mathbb{P}(T)$ is sometimes in the form of a set $\{P_{a_0}(q), P_{a_1}(q), \dots\}$ of predicates on program states, where $P_{a_i}(q)$ specifies the possible starting states upon which an action a_i can be executed. When walking through a program g , all actions are symbolically executed and, at each step, the corresponding predicate to satisfy is picked up. All predicates are combined in the order of the walk-through to generate a single, monolithic predicate called the verification condition \mathbb{C} . The task of verifying $\mathbb{P}(T_g)$ becomes to verify \mathbb{C} [3, 4, 32].

Although such verification has the advantage of zero runtime overhead, complexity is a problem. To verify the predicate $\mathbb{P}(T_g)$ or \mathbb{C} , one has to either prove the predicate in a sound logic, as in [3, 32], which cannot be fully automated and may be undecidable; or use model-checking [59, Chap.3] to verify that the predicate holds for all possible values of the states in program execution [40, 67], which may suffer from state explosion and incur exponential complexity. This disadvantage is unacceptable especially for mobile program security verification, as discussed in Subsection 2.2.2.

To avoid the complexity problem with traditional static verification, more recent practice uses static type checking to verify security policies [16, 54, 75, 90, 93, 107]. This approach uses a statically-typed language for target programs. In a statically typed program, types of all syntactic terms are annotated or can be inferred. There are semantic rules specifying what types of terms can be used in expressions and operations, and how the types of expressions and operations are derived from types of

their component terms. Most importantly, these typing rules provably imply all the desired security policies. Verification of the policies reduces to checking the typing rules by scanning through target program code and searching for some (counter) semantic patterns of syntactic terms. Such pattern-searching is decidable and efficient.

Another static approach to work around the complexity problem, specifically designed for security of mobile programs (see Subsection 2.2.2), is to use Proof-Carrying Code [4, 41, 95, 106]. It requires the creator of a target program to attach, to the program, a formal proof of security, so that the execution hosts only need to check the proof, which is efficient and fast. However, this approach pushes back the difficult work of theorem proving to programmers, makes it difficult for the programmers to accept the approach.

On the other hand, runtime security monitors have low complexity and can enforce policies related to runtime values, but incur significant runtime overhead. A traditional monitor is typically a program running concurrently with target programs, observing the trace of current execution of each target, and verifying at each step that (q_i, a_i) is allowed, according to the policies enforced, before the target can execute action a_i [96]. The observation is usually done through some event-triggering mechanism. System control is transferred back and forth between the target and the monitor, incurring many expensive context switches. The Java security manager [61] is an example.

To improve the runtime overhead of runtime security monitoring, a common practice is to use code instrumentation, an approach that merges monitors with target programs by inserting checking code before security-relevant actions in target program

code [29, 36, 105]. With this approach, a target program and its monitoring code become a single program and are executed in the same context, eliminating the need for context switches. An implementation of code instrumentation usually includes scanning through a target program, locating the monitoring points by searching for some semantic patterns, and inserting appropriate monitoring code at these points.

Note that both static verification and code instrumentation involve manipulation of program codes, which are better represented as heterogeneous parse trees. So the proposed approach of this thesis for manipulating heterogeneous data is applicable to implementation of these security enforcement mechanisms themselves. More details are in Subsection 2.2.3 and Chapter 5.

2.2.2 Security of Mobile Programs

Mobile programs are programs that move to remote computers dynamically at runtime through networks, for execution on the remote computers. In practice there are two forms, mobile code [19] that is static program code to be sent to execution hosts (one hop only) and be executed to termination, and live mobile agents [47, 58] that can autonomously move themselves from computer to computer (multiple hops) and resume execution upon arrival at each computer en route.

Using mobile programs is the major approach to move computation towards data, one of the three ways to implement computation with remote large heterogeneous data (Section 2.1). It gives data users the freedom to implement whatever data manipulation they want, unrestricted by the algorithms pre-established on data servers; it also saves the bandwidth of moving large data around. When remote heterogeneous data are extremely large and immovable, mobile programs become the only choice.

However, accepting mobile programs poses a greater security risk for execution hosts, and makes security enforcement more difficult. In a general and open distributed system, mobile programs may come from unknown or untrusted users. The execution hosts have no control over the development of the mobile programs, but they are expected to launch the programs for execution upon receipt and without significant delay, to achieve a reasonable response time. Full program analysis or intensive testing for security would cause unacceptable delay; proof-carrying program code is easy to verify, but programmers need to do theorem proving, which cannot be fully automated and may be undecidable; pure execution monitoring causes no delay on launching a mobile program upon receipt, but incurs significant runtime overhead and performance penalty for the mobile program. Among security enforcement techniques (see Subsection 2.2.1), static type checking [16, 54, 75, 90, 107] and execution monitoring with code instrumentation [36, 52, 92] are the most applicable to mobile programs. Both type checking and code instrumenting are efficient and fast, so that both approaches do not cause unacceptable delay for launching a mobile program, but the execution of the instrumented code is a runtime overhead. Static type checking does not create any extra work to do at runtime, so it is the only technique that can both be run efficiently and have no runtime overhead.

Since static type checking is a preferred security enforcement technique, especially suited for mobile programs, and there is strong need to move computation with large heterogeneous data to data servers (Section 2.1), it is desirable to implement computation with large heterogeneous data in a statically-typed way for efficient security checking. Programming with query languages and programming at a low untyped level are both inferior for security reasons.

2.2.3 Security with Java

The popular Java programming language [45] is highly portable because Java source programs are compiled into bytecode, which is executable on standard Java virtual machines [77]. This makes Java especially suitable for developing mobile programs, which can adapt to different kinds of computer hardware at which they arrive.

A standard Java virtual machine imposes security on Java bytecode programs by two means: static type checking [75] to enforce basic memory safety, and a pure runtime monitoring mechanism called the security manager [61] to enforce authorization on invoking library methods. There are many finer-grained application-specific or user-specific security policies not enforced by these standard mechanisms. Intensive effort has been put into enforcement of these policies, by either improving the type system of the language then using static type checking [16, 20, 25], or using execution monitoring with code instrumentation [23, 52, 88, 92].

Researchers have paid much attention to security enforcement mechanisms, but little to the security and reusability of the mechanism implementations themselves, assuming that they can be realized by existing programming approaches without problems. Unfortunately, this is not true:

- Security problem. The security of a security enforcement program itself is critical to the system protected by the program, hence it needs to be fully guaranteed. It is desirable to implement a security enforcement program in a fully typed way so that it is possible to ensure full security by efficient static type checking. However, bytecode programs are syntactically heterogeneous trees.

Both static type checking and code instrumentation to enforce security on bytecode programs involve scanning through programs for patterns indicating violation, or potential of violation, of security policies, and further processing at the matching locations. Although it is always possible to implement such computation at a low, untyped level with different kinds of tree nodes (syntax terms) of a bytecode program all represented as text strings, it is a challenge for existing programming languages to represent bytecode programs as typed heterogeneous data trees and express the manipulation of such trees in a well-typed way.

- Generality and reusability problem.

Different researchers focus on different categories of security policies, are concerned with different language abstractions, arrange different actions upon finding patterns of insecurity, and usually end up with separate implementations. For example, Pandey and Hassii [88] chose to enforce customized access control policies by searching calls to related methods and inserting checking codes before the calls; Chander and Mitchell [23] came up with a similar solution, but rather than inserting, they replaced method names being called with new ones; Sabelfeld and Myers [93] enforced information-flow policies by checking types of lower-level variables and instructions. Their expressions of security policies and their implementations of the enforcement are quite different from each other.

2.3 XML Data Processing

XML [18] is a common format to represent online data for exchangeability and machine-comprehensibility. XML data elements are declared hierarchically, hence

XML data are better represented as heterogeneous trees (or graphs if there is recursive or mutual recursive declaration). A basic operation in processing XML data is to search over a data tree for items that match a pattern, and then apply some computation to the matching items.

There are two styles of complex search patterns in XML data processing. The first is *vertical* patterns, as in the search for the population of cities of Canada, given a piece of geographical data in XML format. Such a pattern describes the relationship of XML elements (populations, cities, countries) from different levels of a hierarchy. Location paths, expressed in the popular XPath [27] language, fall in this category. The second is *horizontal* patterns, as in the search for cities having child elements for name, population, either timezone or continent, and zero or more rivers, i.e., the pattern `(cityName, population, timezone|continent, river*)`. Such a pattern describes relationship of XML elements under the same parent element on the same level of the hierarchy. The regular-expression patterns handled in XQuery [53] and CQuery [102] are examples. Vertical and horizontal patterns can be combined into even more complex search patterns.

There is a rich literature on algorithms to process XML data, assuming the use of query languages. However, problems in using XML query languages have recently been noticed, leading to attempts to directly express XML data processing in general-purpose programming languages.

2.3.1 XML Query Languages

In the early years of XML, a large number of special-purpose XML query languages such as Quilt [22], Lorel [1], YATL [28], XML-QL [33], XQL [91] and XSLT [26]

were invented to handle query and transformation of XML data, emerging from the database community and the structured-text community. They are typically untyped, handling both tag names and element content as strings.

These query languages have very limited programming power, and are unable to express sophisticated computations on XML data. XSLT uses XPath [27] expressions as patterns to search for in XML data. Although XPath is powerful in expressing a wide range of vertical patterns, XSLT is limited in programming power, hence incapable of expressing sophisticated computation for further processing of search results. The other query languages are even more restrictive, both in expressing search patterns and in programming. None of them is able to express horizontal patterns systematically, although they can hardcode individual ones (e.g., sibling axes in XPath can represent simple horizontal patterns).

Due to the limited expressive power of query languages, in many settings the queries and their results must be passed, at runtime, to other application programs for further processing, incurring the Impedance Mismatch problem [6, 104] as described in Section 2.1. It creates greater security risks, since the type safety of XML manipulation programs relies on runtime type-checking by explicit checking codes, which are inserted by programmers at development time. The correctness and completeness of the checking code are not guaranteed.

2.3.2 Native XML Processing

In recent years, attempts have been made to merge XML processing into general-purpose programming languages. Typical approaches introduce special types to represent XML data elements and special expressions to represent search patterns, in

addition to standard programming language features. The most recent efforts include XJ [51], XQuery [13] and $C\omega$ [12, 78] focusing on vertical patterns, and XDuce [53, 103] and CDuce[102] focusing on horizontal patterns. In terms of programming style, XJ and $C\omega$ are object-oriented, while XQuery, XDuce and CDuce are functional.

XJ and XQuery enforce static typing against XML schemas rather than making XML data types native to the programming languages. They express search patterns by embedded strings. Therefore, type mismatches still exist to some extent. $C\omega$, XDuce and CDuce express XML data and search patterns fully in native mode with static typing, so that XML processing can be handled within a single language.

However, the inability to parameterize structures and patterns, and poor extensibility, are two common shortcomings of all of these languages. First, these languages can only parameterize XML data items, not structures of these items, nor the patterns to match the structures, because the latter are not first-class entities. Although XML data can be parsed into well-typed trees, traversal of the heterogeneous XML trees for pattern searching has to rely on runtime type-casts, and patterns have to be hard-coded in programs. Second, these languages only allow specific kinds of patterns and cannot be extended to other kinds easily in a type-safe way. An extension to a new kind of patterns requires new features to be added to the language; the type system has to be modified; and so does the compiler.

XJ extends Java with XML data types and XPath expressions, capable of handling vertical patterns conforming to XPath 1.0 [27]. It expresses XML element types as Java classes, and uses special embedded strings containing XPath expressions as search patterns. Static typing of these XML types and embedded pattern strings

against XML schemas is enforced by a special type checker. Because the type checking is against XML schema types, not native Java types, XML data and pattern expressions are not fully type-safe in Java. Since search patterns are just strings, there is the potential to include patterns other than XPath expressions, but only in a untyped way (or at best typed against XML schemas, not Java). Also, the special type checking requires that schemas for XML data are always available and trustworthy, which is unrealistic in many situations.

XQuery is designed to be a query language, but is equipped with some basic functional-programming features. It is intended to be a language for XML processing analogous to SQL for relational data processing. It aggregates many features from older XML query languages and SQL, and its data model and type system fully conform to XML [18] and XML Schema [38] specifications. It is a superset of XPath 2.0 [11], with XPath expressions being native, so it is fully capable of handling vertical patterns of the XPath form. On the other hand, XQuery has only very limited functional programming features. Except in user-interactive settings, its expressions are supposed to be embedded in host programs in other languages for processing of query results. So the impedance mismatch problem still exists, just as in XJ, since XQuery is only typed in terms of XML schemas. The mismatch between XML schema types and host-language types weakens the safety of XML processing programs. The only advantage over XJ is that, in the absence of XML schemas, XQuery expressions can still be type-checked to some extent based on the type information in the query expressions themselves.

$C\omega$ is intended to extend $C\#$, another general-purpose programming language, with native types that support both object-oriented, relational and semi-structured

data models, in order to unify the processing of all three kinds of data. It introduces three new kinds of types: stream, anonymous struct and choice, roughly equivalent to list, heterogeneous tuple and sum types in functional languages. It uses the notion of content class for expression of XML schemas. For example, suppose an XML schema for geographical data has a country element type, with name, population and zero or more provinces as child elements. It then can be encoded as a content class `Country` as:

```
class Country {
  struct{ string name; float population; Province* provs; };
  ... // appropriate constructor
  void increasePopulation(float percentage){...}
  ...
}
```

which contains an anonymous struct holding `name`, `population` and a stream of `Province`. In turn, `Province` is another content class (declaration not shown here) for province element type, which may have name, population and a stream of `City` as children, and so on. Suppose `canada` is an instance of `Country`. The pattern to search for the population of Canada can then be expressed as `canada.population`. To accommodate XPath-style vertical patterns, $C\omega$ also introduces filter expressions such as `Country[name=="Canada"]`, and transitive query expressions such as `Country...population` for population data appearing at arbitrary depth below countries. For example, the following method returns a stream of populations of cities in a given country:

```
virtual float* getPopulation(Country c1) {
  foreach (p in c1...City.population) yield return p;
}
```

The expressiveness of $C\omega$ for patterns in XML processing is roughly equivalent to XPath 1.0 [27] without backward axes. In contrast to XJ and XQuery, XML data and pattern expressions in $C\omega$ are fully native, expressed by identifiers all having $C\omega$ native

types. There is no impedance mismatch problem. However, the pattern to search, such as `c1...City.population` in the above method, has to be hard-coded in the program and cannot be passed as an argument to a method in a typed manner. Hence it is not possible to have a method reusable for different patterns, i.e., not possible to make a program like `get(somePatternType pattern, Country c1)`. Moreover, adding other kinds of patterns, for example XPath backward axes, self-nested structures, or horizontal patterns, would require large changes to the language.

XDuce and CDuce are functional programming languages with regular-expression types added to general-purpose functional language features. These two languages use regular expressions to denote XML element types, and to define horizontal patterns to match XML elements. For example, the country element type above in the `C ω` example can be declared in CDuce as:

```
type Country = <country>[Name Population (Province)*]
type Province = <prov>[Name Population (City)*]
type City = <city>[Name Population ...]
type Name = <name>[String]
type Population = <pop>[Int]
```

Ordinary pattern matching is supported by CDuce, and can be used to locate all population items in a piece of XML data and apply some update to them:

```
let updatePop (x:<_>[_*]) :<_>[_*] =
  let [ y ] =
    xtransform [ x ] with
      <pop>[(z & Int)] -> [ <pop>[(z*101/100)] ]
  in y
```

This CDuce function uses regular-expression type `<_>[_*]` to constrain both the parameter and result, meaning an element with any tag name and any content; and uses regular-expression type `<pop>[Int]` to match any element with tag name “pop” and an integer as content. It traverses the whole structure of a given piece of XML

data x using the macro iterative operator `xtransform`, finds all population elements, and increases every of them by 1%.

In XDuce and CDuce programs, patterns are well-typed and are handled natively. CDuce can even encode XPath-like vertical patterns with child axes (but not descendant axes). However, search patterns, such as `<pop>[Int]` in the above CDuce program, are not first-class terms and cannot be referenced and passed as well-typed arguments. Also, the two languages are not able to handle vertical patterns in general. Inclusion of new kinds of patterns such as vertical patterns would require significant extensions to the languages, if it is possible at all.

2.4 Pattern Calculus and bondi

2.4.1 Static Typing and Expressiveness

Enforcing a type system is one of the mainstream techniques to verify program correctness, and has become a common part of modern language design [21]. A type system consists of grouping language terms (variables and values) into categories called types, with a set of syntactic rules called typing rules which define what operations can be applied on what terms. For example, in Java bytecode (which can be deemed as the assembly language for Java Virtual Machine [77]), integral operators, such as *iadd*, *isub*, *iload*, and *istore*, can only be applied to values of integer type, not of floating-point type nor reference type. Checking programs against the typing rules of a type system can prevent errors and inconsistencies caused by programmer mistakes and, more recently, has been used to prevent security breaches in malicious programs. Specifically, it is possible to design a type system whose typing

rules imply all desirable security policies [17, 54, 70, 75, 90, 107], so that type checking on programs enforces the security policies.

Type checking is usually done on program code before a program is executed, unless a typing rule is defined in terms of runtime values. This requires developing programs in a statically typed way, fully conforming to a static type system, that is, a type system that does not rely on any runtime value.

However, expressing a computation in a statically typed way is not always possible. A static type system is conservative, at the price of expressiveness. It guarantees that the programs accepted are safe to run, but not necessarily that programs rejected are problematic. Given a language with a static type system, there is always some legal computation too complex to be expressible, and it is an endless process to search for more expressive type systems to express more sophisticated computation [80].

The Pattern Calculus [62, 63, 66, 83] aims at expressing reusable programs that can handle arbitrary data types. It provides a type system more expressive than those of the mainstream general-purpose programming languages in practical use. Rather than using special types for specific purposes, such as in [12, 53, 102], Pattern Calculus introduces general type abstractions for structures and for patterns to search in structures. It:

- treats structures and patterns as first-class values, just like data and functions, allowing them to be referenced by variables, passed as arguments and pattern-matched, all with first-class native typing;
- allows a generalized form of pattern matching, without requiring the pattern cases to be the same type, so that control flows can be (recursively) determined by structure, not just data.

These features provide higher expressiveness, enable new forms of polymorphism including structure polymorphism, path polymorphism and pattern polymorphism, which will be demonstrated in the following subsections. Rather than a full coverage of Pattern Calculus, the demonstrations will concentrate on features that directly enable the proposed approach of this thesis. A recently-designed experimental general-purpose language, called `bondi` [65] (formerly FISH2), implements the Pattern Calculus, and will be used to demonstrate the Pattern Calculus features in the following subsections and to demonstrate my proposed approach in the rest of this thesis. `bondi` syntax is similar to OCaml with some minor differences, hence this thesis will not give full description but only explain when necessary.

2.4.2 Structure Polymorphism in Pattern Calculus

Pattern Calculus enables programs reusable for different data structures. The reusability is achieved by (1) extending the use of polymorphic types from representing data to representing structures, and (2) generalizing standard pattern-matching to match with polymorphic structure types.

Polymorphic types for data values are supported in many general-purpose programming languages, such as OCaml [76], Haskell [89], and Java [45], to enable reusability. For example, in OCaml [76], a list of datum elements of arbitrary type can be declared:

```
type 'a list =  
  Nil  
  | Cons of 'a * ('a list);;
```

whose elements are of polymorphic type `'a` (in OCaml, the single quote before a variable name indicates a type variable, to distinguish it from ordinary variables).

The `bondi` language implementation removes the need of single quote). Operations on lists can be programmed by using standard pattern matching. For example, a program to calculate the size of a list can be like this:

```
let rec listSize x = (* "rec" indicates recursive function in OCaml*)
  match x with
  | Nil -> 0
  | Cons (y, z) -> 1 + (listSize z);;
```

Function `listSize` is reusable for different types of list elements. It can be applied to a list of integers, a list of strings, or a list of lists, to name a few. It uses pattern matching to distinguish and handle different list constructors. The first case tries to match the argument against pattern `Nil`, and the second against pattern `Cons (y,z)`. Such a standard pattern matching construct is a common feature in mainstream functional languages such as OCaml [76] and Haskell [89]. The type systems of these languages require that every match case in a pattern matching construct must have the same type. Since the overall type of the function `listSize` is `('a list)->int`, every pattern must have type `('a list)`, and the specialized action for every case must have type `int`, so that the type of every case is consistent with the function type.

When there is a need to use a different data structure other than list, say, a binary tree like:

```
type 'a btree =
  Leaf of 'a
  | Node of 'a * ('a btree) * ('a btree);;
```

a different program of type `('a btree)->int` will be needed for the size calculation:

```
let rec btreeSize x =
  match x with
  | Leaf y -> 1
  | Node (y, z, w) -> 1 + (btreeSize z) + (btreeSize w);;
```

For programming efficiency and maintainability, it is desirable to have a single program able to compute the size of an arbitrary data structure. For example, it would be nice to have a function `size` of type `'b->int`, in which the parameter `x` (of type `'b`) would be polymorphic not only on the type of data elements, but also on the type of the data structure, so that it could handle both lists and binary trees:

```
let rec size x =
  match x with
  | Nil -> 0
  | Cons (y, z) -> 1 + (size z)
  | Leaf y -> 1
  | Node (y, z, w) -> 1 + (size z) + (size w);;
```

However, such a function would violate the type systems of the standard functional languages, because the pattern matching cases have different types.

Pattern Calculus achieves reusability for different data structures by generalizing the typing rule for pattern matching. Rather than requiring each match case being the same type as the function type, it only requires that the type of each case is a specialization of the default function type, and the specialization is determined by the type of the pattern. For example, function `size` above (after a slight syntax change) would be well-typed in `bondi`, because the default function type `'b->int` can be specialized to `('a list)->int` in the first two match cases, and to `('a btree)->int` in the last two cases.

The type specialization in pattern-matching constructs allowed by Pattern Calculus is based on unification [5], an important concept in term rewriting and logic programming. An unification process tries to make two terms (syntactically) equal by substituting into free variables.

Let $\{t_1/x_1, t_2/x_2, \dots, t_n/x_n\}t$ denote a term obtained from term t by substituting into free variables x_1, x_2, \dots, x_n in t with terms t_1, t_2, \dots, t_n respectively, i.e., for every

x_i , all its free occurrences in t are replaced by t_i (all bound occurrences of x_i are replaced by a fresh variable, say, y_i , beforehand). $u = \{t_1/x_1, t_2/x_2, \dots, t_n/x_n\}$ is called a substitution. The composition of two substitutions u_1 and u_2 , denoted u_1u_2 , is defined by $(u_1u_2)t = u_1(u_2t)$.

If there exists a substitution u for two terms t_1 and t_2 such that $ut_1 = ut_2$, u is called a unifier of the two terms, and the two terms are said to be unifiable. For example, terms $f(x, 4)$ and $f(3, y)$ are unifiable, and substitution $\{3/x, 4/y\}$ is a unifier of them, because

$$\{3/x, 4/y\}f(x, 4) = f(3, 4) = \{3/x, 4/y\}f(3, y)$$

A unifier v of terms t_1 and t_2 is called the most general unifier if, for every other unifier u of t_1 and t_2 , there exists a substitution w such that $u = wv$. It has been proven that if two terms are unifiable, there exists a unique most general unifier (subject to variable renaming) [5]. An unification process, denoted as $\mathcal{U}(t_1, t_2)$, is expected to take two terms as input, decide whether the two terms are unifiable, and if they are, return the most general unifier v (denoted by $v = \mathcal{U}(t_1, t_2)$), otherwise terminate without returning anything (denoted by $\mathcal{U}(t_1, t_2) \uparrow$).

Obviously, the termination of an unification process relies on the decidability of the computation. The simplest case, first-order unification, where higher-order variables (for functions) are not allowed, is decidable. Higher-order unification in general is undecidable, but second-order unification with linear occurrences¹ of second-order variables is decidable and there exists an efficient algorithm [34]. This enables efficient type inference for the generalized pattern matching allowed by the Pattern Calculus.

¹Linear occurrence of a variable in a term means that the variable appears exactly once in the term.

The type specialization in pattern matching allowed by Pattern Calculus is captured by a new kind of terms called *extension*, of the form `(at p use s else t)`, which means an extension to the default function t at the pattern p by the specialization s (not confused with the type specialization mentioned above). When such a term is applied to a term t_1 , if t_1 matches pattern p , s is applied, otherwise t is applied. In Pattern Calculus, patterns are either term variables, (constant) constructors, or application of one pattern to another. Although simple, these are sufficient to express (simple and complex) patterns defined in Section 1.1. The type of an extension term obeys:

$$\frac{t : T \rightarrow S \quad p : T_1 \quad s : vS}{\text{at } p \text{ use } s \text{ else } t : T \rightarrow S} v = \mathcal{U}(T_1, T) \quad (2.1)$$

$$\frac{t : T \rightarrow S \quad p : T_1}{\text{at } p \text{ use } s \text{ else } t : T \rightarrow S} \mathcal{U}(T_1, T) \uparrow \quad (2.2)$$

These typing rules for an extension term do not require that the type T_1 of the pattern must be the same as the parameter type T of the whole term (and of the default function), unlike other languages that support pattern matching. Rule (2.1) for successful matching only requires that type T_1 and T have a most general unifier v and the specialization s has type vS where S is the return type of the whole term. For example, if the default function type is `'b->int`, it suffices that p has type `('a list)` and s has type `int`, because `'b` and `('a list)` have a most general unifier $v = \{(a' \text{ list})/'b\}$, and constant type `int` $= v \text{ int}$. Rule (2.2) says that, even if T_1 and T cannot be unified, the term is still type-safe, but the specialization s can never be used (in practice this usually means a programmer mistake). A formal presentation of the complete type system of Pattern Calculus can be found in [66].

The typing rules for extension terms enable pattern-matching more general than the standard one in other languages, while maintaining type safety. Consider every pattern matching case `| p -> s` to be a fragment of an extension term:

(`at p use s else ...`). A generalized pattern matching construct, such as that in function `size` mentioned above, can be implemented as nested extension terms and can be statically typed. For example, the function `size` in `bondi` syntax looks like this:

```
let (size: b->int) x =
  match x with
  | Nil -> 0
  | Cons y z -> 1 + (size z)
  | Leaf y -> 1
  | Node y z w -> 1 + (size z) + (size w);;
```

Its overall function type is declared as `b->int` polymorphically, but each case does not have to have type `b->int`. Actually the cases are specialized to either `(list a)->int` or `(btree a)->int` (types of data structures in `bondi` are written in the way consistent with type applications, such as `(list a)`, not like `('a list)` in OCaml).

This function can be interpreted as:

```
at Nil use 0
else (at Cons y z use 1 + (size z)
      else (at Leaf y use 1
            else (at Node y z w use 1 + (size z) + (size w)
                  else error)))
```

and obviously the specialized type of every case satisfies rule (2.1). The ultimate default (if no pattern matches) is an error term `error` of type `b->int`, internally defined in the `bondi` implementation using an exception constructor of polymorphic type. It is implicit in the `bondi` surface syntax, and thus does not appear in the definition of function `size`.

The application of a `bondi` pattern-matching function on an argument is executed (evaluated) as a sequence of reductions of extension terms. Below are the basic reduction rules for extension terms (“>” represents the reduction relation of terms):

$$(\text{at } x \text{ use } s \text{ else } t) t_1 > \{t_1/x\}s \quad (2.3)$$

$$(\text{at } c \text{ use } s \text{ else } t) c > s \quad (2.4)$$

$$(\text{at } c \text{ use } s \text{ else } t) t_1 > t t_1 \quad \text{if } t_1 \text{ cannot become } c \quad (2.5)$$

$$(\text{at } p_1 p_2 \text{ use } s \text{ else } t) (t_1 t_2) > (\text{at } p_1 \text{ use } (\text{at } p_2 \text{ use } s$$

$$\text{else } \lambda y.(t (p_1 y)))$$

$$\text{else } \lambda x, y.(t (x y)))$$

$$) t_1 t_2$$

$$\text{if } t_1 \text{ is a constructed term}$$

$$(\text{at } p_1 p_2 \text{ use } s \text{ else } t) t_1 > t t_1 \quad (2.7)$$

$$\text{if } t_1 \text{ cannot become applicative}$$

The reduction here does not require explicit type information to guide the specialization of the polymorphic function. When the pattern is a variable x , the specialization s always applies with the free occurrences of x in s being substituted by the argument t_1 (rule 2.3). When the pattern is a constructor, the specialization applies only if the argument is also that same constructor (rule 2.4), otherwise the default is applied (rule 2.5). With an applicative pattern, the reduction tries to match components of the pattern with the components of an applicative argument respectively (rule 2.6). If the argument is not applicative, the default is applied (rule 2.7). Note that even if the pattern and the argument to compare with are of totally different types, the reduction can still proceed as in rules (2.5) and (2.7), although the pattern matching will definitely fail and the default function will be used.

Note that, similar to languages that support standard pattern matching, Pattern Calculus requires patterns to be linear, i.e., each variable appearing in a pattern must appear exactly once, no more and no less, to ensure correct binding of variables in specializations so that reductions will not be ill-behaved. Firstly, a variable in a

pattern must appear at least once, so that applicative patterns are not allowed to eliminate binding variables during reduction. For example, pattern $(y \rightarrow 0) x$ is not allowed, otherwise, $(\text{at } (y \rightarrow 0) x \text{ use } x \text{ else } t) 0$ would reduce to x , so that binding variable x would become free after reduction. Secondly, a variable in a pattern must appear no more than once. For example, pattern $(c x x)$ is not allowed, otherwise, application $(\text{at } c x x \text{ use } x \text{ else } t) (c p q)$ would reduce to q , with binding variable p being eliminated:

$$\begin{aligned}
& (\text{at } c x x \text{ use } x \text{ else } t) (c p q) \\
> & (\text{at } c x \text{ use } (\text{at } x \text{ use } x \text{ else } \lambda z.(t (c x z))) \\
& \quad \text{else } \lambda y, z.(t (y z))) (c p) q && \text{(rule 2.6)} \\
> & (\text{at } c \text{ use } (\text{at } x \text{ use } (\text{at } x \text{ use } x \text{ else } \lambda z.(t (c x z))) \\
& \quad \quad \quad \text{else } \lambda w.(\lambda y, z.(t (y z)) (c w))) \\
& \quad \quad \quad \text{else } \lambda w, u.(\lambda y, z.(t (y z)) (w u))) c p q && \text{(rule 2.6)} \\
> & (\text{at } x \text{ use } (\text{at } x \text{ use } x \text{ else } \lambda z.(t (c x z))) \\
& \quad \quad \quad \text{else } \lambda w.(\lambda y, z.(t (y z)) (c w))) p q && \text{(rule 2.4)} \\
> & (\text{at } x \text{ use } x \text{ else } \lambda z.(t (c x z))) q && \text{(rule 2.3)} \\
> & q && \text{(rule 2.3)}
\end{aligned}$$

Actually, in the `bondi` implementation, these ill-behaved reductions will not happen, since pattern linearity requirement is among the typing rules of the type system and is checked at the type inference stage. Programs with non-linear patterns will be rejected. Note that linear occurrence of variables in a pattern also implies that each function appearing in the pattern is a linear function, i.e., it uses each of its arguments exactly once.

Function `size` achieves both data polymorphism and structure polymorphism together, because its parameter of type `b` is polymorphic in both the type of the data structure and the type of the data elements. Actually, these two forms of polymorphism can also be expressed separately in `bondi`. The type system of Pattern calculus

treats every component of an applicative term as a first-class term which has a first-class type and can be polymorphic if typed by a type variable. Hence, the structures and the elements of constructed data can be typed and handled separately, and possibly polymorphically. For example, when declaring the type of a data parameter in `bondi`, one can use a polymorphic type for the structure only, separating it from the type for elements. Taking the mapping operation ² as an example, one can not only make a mapping program on lists, like:

```
map: (a->b) -> List a -> List b
```

but can also make a mapping program that is parameterized on arbitrary homogeneous structures:

```
map1: (a->b) -> c a -> c b
```

Function `map1` has a parameter of type `c a` and return type `c b`, where type variable `c` represents the polymorphic type for an arbitrary homogeneous structure, achieving structure polymorphism. For example, a list of integers, a binary tree of text strings, and a ternary tree of PDF files can all be passed to `map1` for processing. Type variables `a` and `b` are the types for data elements in the structure, achieving data polymorphism. The actual definition of `map1` is more complex than its type suggests as it relies on the theory of data structures developed in Pattern Calculus. Interested readers may refer to the Pattern Calculus paper [66].

Note that the standard Haskell [89] with type class system also achieves the structure polymorphism equivalent to that presented in the function `size` and `map1` above. Rather than allowing polymorphic parameters for different structures and generalizing the typing requirement of pattern matching, Haskell uses a different approach,

²Mapping is an operation which, given a data structure and a function, applies the function to every element of the data structure.

declaring all structure types of concern as instances of a type class, and defining specialized functions to overload for every type of structure.

Pattern Calculus supports not only first-class structures in constructed data, but also first-class patterns and first-class components in applicative patterns to match structures. This enables another way to express structure polymorphism independently: to use polymorphic variables in patterns to match structures. Doing so becomes necessary in the presence of heterogeneous data structures. This achieves not only structure polymorphism, but also another new form of polymorphism, *path polymorphism*, as described in the next subsection.

2.4.3 Path Polymorphism in `bondi`

The Pattern Calculus and `bondi` support parameterization of not only homogeneous structures, but also heterogeneous structures.

Recall the heterogeneous geographical data repository in Section 1.2, which may contain different types of substructures such as countries, provinces, cities, region areas, populations, etc. Suppose population elements in such a repository are represented by data type:

```
datatype population = Pop of float;;
```

`bondi` supports a program that is polymorphic on the repository structure, and that can apply an update operation to all population elements regardless of what structure the repository has and how heterogeneous it is:

```
let (updatePops:(float->float)->d->d) f x =  
  match x with  
  | Pop z -> Pop (f z)  
  | y z -> (updatePops f y) (updatePops f z)  
  | z -> z;;
```

Function `updatePops` takes two arguments, an update function `f` of type `float->float` and a piece of data `x` of polymorphic type `d`, and applies the update function to the content of all population elements wherever they occur in the data, and whatever the structure of the data is. It is implemented by the generalized pattern matching allowed by the type system of Pattern Calculus, and can accept a data repository of arbitrary, most probably heterogeneous, structure. The patterns of the three matching cases here are of different types. The first case is to match the target – a population element. The second and third cases cause the action to be propagated to all parts of the data structure. The pattern `y z` in the second case matches against any compound data (e.g., `Country m n q` where `Country` is a constructor and `m`, `n` and `q` are its arguments which may themselves be atomic or compound), and causes both parts of the compound (e.g., `Country m n` and `q`) to be updated recursively. The pattern `z` in the final case is used to terminate at atomic data. The three cases may be of different types, which would not be allowed in other languages that support pattern matching.

Note that in function `updatePops` polymorphism on data structures is achieved by using first-class term variable `y` in the applicative pattern of the second case to directly match the head component (the structure) of an arbitrary applicative term (constructed data). This variable may match terms of different types at different rounds of the recursive function calls, especially in a heterogeneous structure. With this mechanism, `updatePops` is able to reach all target population values wherever they are in a data repository. These targets may reside in substructures of different types in the given repository (e.g., some are provincial populations and some are city populations), and hence they have different access paths from the root of the whole structure.

Therefore, `bondi`'s support of using first-class polymorphic variables as the head components in applicative patterns achieves not only structure polymorphism, but also access path polymorphism which is especially useful in the presence of heterogeneous data structures.

The scrap-your-boilerplate (SYB) approach [73], developed in parallel with the Pattern Calculus and implemented as an extension to Haskell based on type class and type-safe casting, also achieves path polymorphism roughly equivalent to that in `bondi` function `updatePops` above. However the SYB approach does not support another new form of polymorphism achieved by the Pattern Calculus – pattern polymorphism.

2.4.4 Pattern Polymorphism in `bondi`

The third form of polymorphism supported by the Pattern Calculus and `bondi` is pattern polymorphism, due to the use of higher-order pattern terms [62], allowing programs to be parametric in the choice of patterns to search for, such as in function

```
let (update: lin(a->b)->(a->a)->d->d) \P f x =
  match x with
  | P z -> P (f z)
  | y z -> (update P f y) (update P f z)
  | z -> z;;
```

This is similar to function `updatePops` in form, but the pattern of target elements to update is created from polymorphic parameter `\P`. The function `update` arises naturally from function `updatePops`, but has a number of unusual technical features. First some conventions: capitalized variables such as `P` are always free, unless explicitly bound as in `\P` (to be thought of as λP). Thus, the pattern `P z` contains a free variable `P` and a bound variable `z`. Evaluation of `update Pop` will substitute constructor `Pop` for `P` so that the pattern of the first matching case becomes `Pop z`.

Some care is required when substituting into patterns, so such variables are required to be *linear* as indicated by the linear type `lin(a->b)`, meaning that the function of type `a->b` uses its argument exactly once. This is due to the linearity requirement for patterns, as explained in Subsection 2.4.2, that each variable in a pattern must appear exactly once, and so each function that appears in a pattern must be linear. Linear terms, i.e., terms of linear types, are explained in detail in [62]. They represent first-class values in `bondi`, similar to terms of function types, and can be referenced and passed as arguments to programs. In this thesis, all linear terms are constructors, although there are important exceptions when `bondi` is used in other situations. So, for now, `lin(a->b)` is the type of a constructor having one parameter of type `a` for a data structure of type `b`. For example, `Pop` has type `lin(float->population)`. Since a constructor value is mostly used in creating pattern cases, a linear term itself (representing a constructor) is also called a pattern within this thesis when it does not cause confusion.

2.4.5 Summary

The Pattern Calculus and `bondi` provide new expressive power and create the opportunity to represent sophisticated computations with large-scale data structures by statically-typed programs. Such greater expressiveness has been proven powerful and beneficial for object-oriented processing of relational data [82]. It also provides a basis for the programming technique proposed by this thesis, to express computation with arbitrarily heterogeneous data, and specifically to express XML data processing [57] and Java bytecode security enforcement [56], in a statically-typed, highly-parametric and highly-reusable way.

This thesis uses bondi for demonstration program examples.

Chapter 3

Expressing Computations on Heterogeneous Data

Data-intensive applications manipulate large amounts of data organized in data structures. Such manipulation typically consists of searching in a data structure for target data items that match a pattern, then extracting information from, or updating, the content of the matching targets. A pattern may be simple, or complex, as described in Section 1.1. There is always an implicit universal quantification with such pattern-searching computation, requiring traversal of the whole data to find *all* matching targets. For example, given a relational table `Countries` for country information, with data fields for country name, area and population respectively, a task to update the Canadian population to be 33,000,000 may be expressed by a SQL query like:

```
UPDATE Countries
  SET Population = 33,000,000
  WHERE Countryname = 'Canada'
```

It traverses the whole (homogeneous) list of records in the table and finds all records

whose country name is Canada to apply the update. The implicit universal quantification is common to all tasks, although different tasks take different action at each record. However, in the case of heterogeneous data, expressing universal quantification is a challenge, and is not as easy as the case for a homogeneous list in tabular structure.

Section 3.1 will show that making typed and parameterized programs for pattern searching in heterogeneous data is a problem for existing general-purpose programming languages in practical use. Section 3.2 describes my solution to this problem, i.e., adopting a general-purpose programming language that supports first-class typing of structures and patterns, and introducing a general kind of well-typed constructs called pattern structures, and its use in programming. It shows that the approach can express the universal searching with parameters for search patterns of any complexity in a statically typed way. The solution achieves higher level of program reusability and enables generic programming for arbitrary heterogeneity of data and arbitrary complexity of search patterns.

3.1 The Challenge of Expressing Pattern Searching

To demonstrate the challenge in implementing pattern searching computation, the following subsections attempt to develop programs for the scenario of Section 1.2 in general-purpose programming languages. Three languages, i.e., OCaml [76], Java [45], and the SYB extension [73] to Haskell [89], are chosen as representatives. OCaml enforces overstrict typing, and is not able to develop a program that can adapt to

different substructures in heterogeneous data. Java allows limited adaptation to heterogeneous substructures through subclassing. The SYB extension to Haskell achieves type-safe traversal of heterogeneous data by function overloading. All three are not able to pass search patterns as well-typed arguments to programs, and have difficulty handling complex patterns.

3.1.1 Pattern Searching in OCaml

Functional languages like ML strictly enforce static typing. No runtime checking is needed for ML programs. However, the type systems of most functional languages are over-restrictive and hence very inflexible in programming with heterogeneous data structures. OCaml [76] is a variant of ML and is used here as a representative for demonstration.

In OCaml [76], a heterogeneous data structure can be declared as a hierarchy of various data types with their constructors. A structure of geographical data repository may look like this:

```
type population = Pop of float;;
type cityname = CityName of string;;
type area = Area of float;;
type city = City of cityname * population * area;;

type provname = ProvName of string;;
type province = Prov of provname * population * area * city list;;

type countryname = CountryName of string;;
type country = Country of countryname * population * area * province list;;

type geodata = GeoData of country list;;
```

A geographical data repository will be an instance of type `geodata`. To explore such an instance and update all population values of Canadian cities by increasing each of them by 1%, one has to design a series of update functions:

```
let updatePop (Pop x) = Pop x*1.01;;

let updateCity (City (ctn, pop, ar)) = City (ctn, updatePop pop, ar);;

let rec updateCityList ctl = (* "rec" indicates recursive function in OCaml *)
  match ctl with
  [] -> [] (* [] is a short-hand form for Nil, and x::y for Cons (x,y) *)
  | c::tail -> (updateCity c)::(updateCityList tail);;

let updateProv (Prov (pn, pop, ar, ctl)) = Prov (pn, pop, updateCityList ctl);;

let rec updateProvList pl =
  match pl with
  [] -> [] (* the case for empty list *)
  | p::tail -> (updateProv p)::(updateProvList tail);;

let updateCountry (Country (cn, pop, ar, pl)) =
  if (cn == CountryName "Canada")
  then Country (cn, pop, ar, updateProvList pl)
  else (Country (cn, pop, ar, pl));;

let rec updateCountryList cl =
  match cl with
  [] -> [] (* the case for empty list *)
  | c::tail -> (updateCountry c)::(updateCountryList tail);;

let updateGeodata (GeoData cl) = GeoData (updateCountryList cl);;
```

This solution is statically well-typed, but obviously tedious. In order to traverse the heterogeneous tree of different data types, each and every different data type of nodes on all paths to target population values needs a different update function! And there is no way to make the pattern of targets a parameter. Consider some changes to the task:

- there is a change in the structure, say, there is a `capitalCity` item directly under `country`;
- the areas of Canadian cities rather than populations are to be updated;
- the population of Canadian provinces and of Canada also need to be updated;

Then many of the functions need to be redesigned. When the size of the data structure becomes bigger, and Canadian cities also appear in many other substructures, the programming will be more tedious. What makes it even worse is that when knowledge about the structure of the data parameter is not available or not complete at programming time, which is usually the case in modular software development practice, there is no way to design these functions.

In OCaml, pattern searching can be done in one function only when the data structure is homogeneous. For example, it is easy to update population values which are all in a list:

```
let rec updatePopList poplist =
  match poplist with
  | [] -> []
  | head::tail -> (updatePop head)::(updatePopList tail);;
```

So alternatively, the programming for heterogeneous structure can be simplified when the constructors of all concerned nodes, i.e., populations, cities, provinces and countries, are declared to be the same type to remove the heterogeneity:

```
type geodata = (* any component with population value(s) under it *)
  Pop of float;;
  | City of cityname * geodata * area;;
  | Prov of provname * geodata * area * geodata list;;
  | Country of countriname * geodata * area * geodata list;;
  | GeoData of geodata list;; (* for the list of countries *)

let rec updatePops data =
  match data with
  | Pop x -> Pop x*1.01
  | City (cn, pop, ar) -> City (cn, updatePops pop, ar)
  | Prov (pn, pop, ar, ctl) -> Prov (pn, pop, ar, updateList ctl)
  | Country (CountryName "Canada", pop, ar, pl) ->
    Country (CountryName "Canada", pop, area, updateList pl);;
  | Country (cn, pop, ar, pl) -> Country (cn, pop, ar, pl);;
  | GeoData cl -> GeoData (updateList cl);;

let rec updateList gl =
  match gl with
  | [] -> []
  | head::tail -> (updatePops head)::(updateList tail);;
```

Although simpler in form, this solution is not a substantial improvement. Again, the pattern of targets cannot be parameterized; changes of the structure or the pattern require changes to the function `updatePops`; and precise knowledge of the whole data structure is required when designing the function.

These OCaml examples indicate that the type system of the OCaml language is not expressive enough for computations with arbitrary heterogeneous data structures. One over-restriction is the typing requirement that all cases of a pattern-matching syntax “`match ... with`” must be the same type. This makes it impossible to match different types of substructures in different rounds of recursive calls to one function. It is also a problem that, although OCaml supports pattern matching, *patterns* are not first-class values, so there cannot be explicit terms of patterns in expressions, nor pattern parameters.

3.1.2 Pattern Searching in Java

Java (source) programming language [45] is usually deemed to be a language combining strong typing and programming flexibility. This subsection will show that using Java can make a little progress, but does not solve the difficulty of programming in the presence of heterogeneous data structures.

Suppose a data structure of geographical information for the scenario of Section 1.2 is represented as a hierarchy of field references of various Java classes, as follows.

```
// singly-linked list
public class List<E> { E node; List<E> next; ... }

public class Pop { float p; ... }
public class CityName { String n; ... }
public class Area { float a; ... }
public class City { CityName ctn; Pop pop; Area ar; ... }
```

```
public class ProvName { String n; ... }
public class Province { ProvName pn; Pop pop; Area ar; List<City> ct1; ... }

public class CountryName { String n; ... }
public class Country { CountryName cn; Pop pop; Area ar; List<Province> pl; ... }

public class GeoData { List<Country> cl; ... }
```

To save space, the class declarations are listed in a condensed form, and appropriate constructors and get/set methods are not listed here. It is worth mentioning that, in the declaration of the class `List`, a type parameter `E` is used to represent any possible type of list elements. It must be specialized to a concrete type when using the class `List`, such as `List<City>` and `List<Province>` in the above declarations. This feature is called generic types, and was introduced in the third edition of the Java Language Specification [45, Section 4.4]. It enables declarations of data structures and methods that can be used for different types of elements. However, it only works for homogeneous data structures, and does not help at all for heterogeneous structures.

If we use only the basic Java language features to implement the exploration of a heterogeneous object tree linked by field references, the program would have many class casts and need many runtime checks using `instanceof` instructions, manually inserted before every cast by the programmer. Class casts are not type-safe and ruin the static typing. Programs may crash at casting errors.

A better solution is to take advantages of the visitor programming scheme [49, 87]. Palsberg and Jay [87] implement a universal visitor as class `Walkabout` using Java's reflection API (`java.lang.reflect`) [60]. In `Walkabout`, a general method `visit(Object o)` takes an arbitrary object as argument, and checks if a specialized `visit` method exists for the actual class of the object. If such specialized method exists, it is invoked

on the object, otherwise, all accessible fields of the object are recursively visited by the general `visit` method (assuming no reference loop):

```
class Walkabout {
    void visit(Object v) {
        if ( v != null ) {
            Object [] os = v;
            Class vClass = v.getClass();
            Class [] cs = vClass;
            try { this.getClass().getMethod("visit",cs).invoke(this,os); }
            catch (java.lang.NoSuchMethodException e) {
                if (!(v instanceof Number)|(v instanceof Byte)|(v instanceof Short)|
                    (v instanceof Character)|(v instanceof Boolean)) {
                    java.lang.reflect.Field [] vFields = vClass.getFields();
                    for (int i=0; i<vFields.length; i++) {
                        try { this.visit(vFields[i].get(v)); }
                        catch (java.lang.IllegalAccessException e1) {;} // Cannot happen
                    }
                }
            }
            catch (java.lang.Exception e) {;} // Cannot happen
        }
    }
}
```

Given the class `Walkabout`, searching/updating objects of a specific class is easy, by making a simple subclass of `Walkabout` with a specialized `visit` method. For example, the following class can be used to increment population values in all objects of class `Pop` in an instance of class `GeoData`:

```
class UpdatePop extends Walkabout {
    public void visit(Pop pop) {
        pop.p = pop.p * 1.01
    }
}
...
GeoData data = ... ;
UpdatePop upd = new UpdatePop();
upd.visit(data);
...
```

In the new class we only need one specialized `visit` method for target `Pop` objects. The general method `visit(Object o)` in `Walkabout` takes care of all other objects

of various classes in the reference tree, without the need of many methods for the classes. This can be done because all the classes, although different from each other, are subclasses of class `Object`. Both `visit` methods are statically typed. This is an improvement compared with the OCaml solution in the previous subsection, in the sense that the program can be used for different data repositories of possibly different structures, and one single method can handle all different substructures in the universal traversal, even if these substructures are not known in advance.

However, this Java solution does not improve the situation as much as it seems at first.

- The general `visit` method uses runtime type checking and runtime exception handling heavily. Since this method will be invoked for every object in the whole reference hierarchy, the runtime overhead is significant;
- Although it is possible to use objects of class `Class` to represent patterns (not shown above) and pass them as parameters, every new pattern for objects of a different class (e.g., to update `Area` instead of `Pop`) would require a new `visit` method, so that a new subclass of `Walkabout` would be needed and the whole program would need to be recompiled;
- More importantly, when the search pattern for targets is complex, e.g., searching for populations of cities of Canada rather than all population elements, programming in this style starts getting complicated. When the pattern becomes arbitrarily complex in a hierarchical way, the programming will be very clumsy, programs hard to understand, and eventually becomes impossible in practice.

3.1.3 Pattern Searching using SYB in Haskell

The proposed extension to Haskell based on the “scrap your boilerplate” (SYB) approach [73] enables traversal of heterogeneous data by overloading, and processing of different data types by type-safe casting. In standard Haskell [89], programming to update population values in a data repository would be as clumsy as in OCaml (Subsection 3.1.1). However, with the SYB extension, updating all population values can be as simple a program as:

```
updatePops :: (population->population)->d->d
updatePops f = everywhere (mkT f)
```

Here `f` is the update function of type `population->population` to apply on a target population data type. `mkT` is a type extension function to extend `f` so that it can be applied to any data type, not just population. It is defined using a type-safe cast operator. The resulting extended function, `(mkT f)`, behaves like `f` when applied to a population, and like the identity function when applied to any other types.

`everywhere` is a recursive traversal function to apply function `(mkT f)` to every substructure in the given data. The type of `everywhere` is:

```
everywhere :: Term a => (forall b. Term b => b->b)->d->d
```

where `(Term a)` is a context constraint, meaning that `everywhere` can only traverse types that are instances of the type class `(Term a)`. Type class `(Term a)` hosts a generic one-layer map function. Every type that will be encountered in the traversal must be declared as instance of this class, and must have its own version of the map function defined. Function `everywhere` is then recursively defined using the map function, and the traversal is performed by overloading the map function for different substructures.

Function `everywhere` takes two arguments, a polymorphic function of type `(forall b. Term b => b->b)` and an arbitrary piece of data of type `d`, and applies the function argument to every node in the structure of the given data. Note that the type of the function argument is a quantification over all types of class `Term a`, because the function will be applied on different types of data during the traversal. Such rank-2 types are not supported by standard functional languages including standard Haskell, where a polymorphic function must be bound to one certain type before the whole computation starts.

Obviously, function `everywhere` is reusable for different structures of heterogeneous data, and can find targets wherever they appear in a heterogeneous structure. This is an improvement comparing with the situation in OCaml (and standard Haskell), and is roughly equivalent to the expressiveness of the visitor solution in Java.

At a glance, function `everywhere` seems also reusable for different types of targets, because it is polymorphic on the function `f` to be applied to targets. However, this reusability is downgraded by the need to supply a different function for `f` in the case of a different type of targets. For example, to update all area elements rather than population ones, a update function of type `area->area` instead of `population->population` is needed. Re-programming is always necessary for a different type of targets. Furthermore, since the polymorphism on target type relies on the type of the function `f`, a complex pattern to search for would incur clumsy programming to provide such as function. For example, there is no way for the Haskell version of function `updatePops` above to handle the update of population of all Canadian cities, except that the search for the cities and their populations within Canada would be handled by the function argument for `f`. An attempt to relate this approach to XPath processing [72] also

indicates that it will be very difficult, if possible at all, to extend the approach to achieve reusability for complex patterns systematically.

On the contrary, in the proposed approach of this thesis, reusability for patterns relies on explicit pattern parameters of first-class types, so that re-programming for different patterns is minimized, and in many cases can be eliminated at all, no matter how simple or complex the patterns are. This is described in the next section.

3.2 Pattern Searching with Pattern Structures

It can be seen from the previous section that the difficulty in expressing computation with heterogeneous data comes from the universal quantification under heterogeneity, that is,

- there is a need to find *all* occurrences of a pattern;
- the contexts of the occurrences may be *different* in different places;
- the situation is even worse when the quantification is for a *complex* pattern, which is a combination of several different simple patterns.

This section presents a general solution to address these problems, by showing the paces of developing more and more sophisticated programs for computation tasks of increasing difficulties. It introduces the idea of pattern structures and programming using pattern structures, in a general-purpose programming language that implements Pattern Calculus [62, 66], taking advantages of the first-class typing of structures and patterns and the generalized form of pattern matching supported by the language.

At present, the only language that implements Pattern Calculus is `bondi` [65]¹. It is used in this section and the following two chapters to write example programs for demonstration. `bondi` syntax is similar to OCaml and some important distinctions have been explained in Section 2.4. So its syntax will not be explained here unless necessary.

As a convention, `a`, `b`, `c`, `d`, ... are used as variables for types and ..., `w`, `x`, `y`, `z` are variables for values.

To demonstrate the idea, let us go back to the geographical data updating task described in Section 1.2: *Add 1% to the population of all the Canadian cities*, and see how to express programs with various kinds of parameters using `bondi`.

First of all, a datatype of populations can be defined as:

```
datatype popul = Pop of float;;
```

This declaration introduces both a new type `popul` and a new term, its constructor `Pop`, of type `float->popul`. Here is a function for updating populations by pattern-matching:

```
let (atPopIncrementBy1Percent:popul->popul) x =
  match x with
  | Pop z -> Pop (z * 1.01);;
```

When applied to a term of the form `Pop z`, it returns `Pop (z*1.01)`. This function can be parameterized with respect to the action to be taken on populations, by defining

```
let (atPopApply:(float->float)->popul->popul) f x =
  match x with
  | Pop z -> Pop (f z);;
let incrementBy1Percent x = x*1.01;;
let atPopIncrementBy1Percent = atPopApply incrementBy1Percent;;
```

¹At the time of writing this document, there is another language under development called `bronte`, which implements an improved version of the Pattern Calculus [64]

Evaluation of the new version of `atPopIncrementBy1Percent` reduces to the old one by substituting for the variable `f`.

More generally, when population values are stored in larger data structures, e.g., a list, they can all be updated using the following function:

```
datatype list a =
  | Nil
  | Cons of a and (list a);;

let (atListApply: (a->b) -> list a -> list b) f x =
  match x with
  | Nil -> Nil
  | Cons y z -> Cons (f y)(atListApply f z);;
```

`list` is a basic data structure, which is recursively declared, as in other functional languages. Conventionally, `[]` is a shorthand for an empty list `Nil`, and `x::y` for `Cons x y`.

The Function `atListApply` is defined by pattern-matching over the two list constructors. It takes a function `f` as its first argument and applies it to every element (of type `a`) of a list. For example,

```
let updateFloatInList = atListApply incrementBy1Percent;;
```

acts on lists of floats and

```
let updatePopInList = atListApply atPopIncrementBy1Percent;;
```

acts on lists of populations. This illustrates how `atListApply` is reusable for different choices of types `a` and `b` for list entries. Similarly, many other functions can be defined for other homogeneous structures. For example, we can imagine a function `atBinTreeApply` for a binary tree of populations.

There is no surprise up to this point. Standard functional languages such as OCaml and Haskell can also express these programs in the same way, parameterizing

instances of a particular data structure, and the operation to be applied on the elements in the structure.

However, in OCaml, every new structure of populations (say, a ternary tree) would require a new function, while in `bondi` (and also in Haskell) only one program is needed, with a polymorphic parameter for data of arbitrary structure. For example, lists, binary trees and ternary trees can all be handled by function `map1` as described in Subsection 2.4.2:

```
map1: (a->b) -> c a -> c b
```

Even `map1`, however, is not flexible enough for practical use, since typical large-scale data are not as homogeneous as indicated by type `c a`, which means that a structure of type `c` contains only one type of elements. For example, a repository of geographical data will probably have elements of populations as well as city names and areas, and these elements of different types may appear in different substructures. Simply put, a structure of geographical data is most probably heterogeneous.

Fortunately, a function `updatePops` that acts on all population values wherever they occur in a heterogeneous data repository is possible in `bondi`, as described in Subsection 2.4.3:

```
let (updatePops:(float->float)->d->d) f x =  
  match x with  
  | Pop z -> Pop (f z)  
  | y z -> (updatePops f y) (updatePops f z)  
  | z -> z;;
```

The generalized pattern matching here, supported by `bondi`, enables matching with different type of substructure in each recursive call, and is critical to express universal quantification over a heterogeneous data structure. For example,

```
updatePops incrementBy1Percent [Pop x1, Pop x2];;
```

evaluates to `[Pop x1*1.01,Pop x2*1.01]`, but

```
updatePops incrementBy1Percent ([Pop x1],Pop x2);;
```

evaluates to `([Pop x1*1.01],Pop x2*1.01)` even though the populations appear on different levels of the data structure. This adaptiveness to heterogeneity of the data structure remains when the data structure gets more complex. Suppose a concrete structure of geographical data is declared as:

```
datatype popul = Pop of float;;
datatype cityname = CityName of string;;
datatype area = Area of float;;
datatype city = City of cityname * popul * area;;

datatype provname = ProvName of string;;
datatype province = Prov of provname * popul * area * list city;;

datatype countryname = CountryName of string;;
datatype country = Country of countryname * popul * area * list province;;

datatype geodata = GeoData of list country;;
```

A geographical data repository is an instance of type `geodata`. Suppose:

```
let somedata = GeoData ... ;;
```

Then applying program `updatePops` as

```
updatePops incrementBy1Percent somedata;;
```

can update all populations within `somedata`, no matter where they appear, under cities, provinces, or countries.

Hence, the program `updatePops` is not only reusable for data repositories of different structures, but also scalable in the sense that it works for heterogeneous data structures of any complexity. The visitor solution in Java in Subsection 3.1.2 and the SYB solution in Haskell in Subsection 3.1.3 are roughly as expressive as this program.

Examining the program above, it is clear that the constructor `Pop` is playing a completely passive role, and so is ripe for parameterization. As described in Subsection 2.4.4, constructors are first-class linear terms supported by the Pattern Calculus and `bondi`, so is able to act as arguments:

```
let (update:lin(a->b)->(a->a)->d->d) \P f x =
  match x with
  | P z -> P (f z)
  | y z -> (update P f y) (update P f z)
  | z -> z;;
```

Function `updatePops` is actually a specialization of the generic function `update` with constructor `Pop` as argument, and the population update task can be carried out by:

```
update Pop incrementBy1Percent somedata;;
```

A different constructor (say, `Area`) can be passed to the first parameter of `update` for a different type of target elements to be updated in a different task (say, updating area values not populations), without the need to change the program. Recall from Section 2.4.4 that, in this thesis, all linear terms are constructors, and a constructor value is mostly used in creating pattern cases, so a linear term is also called a pattern within this thesis when it does not cause confusion. Further, because a pattern created from a linear term (such as `(P z)` or `(P y z)`) is a simple pattern (to match only one (sub)structure), sometimes a linear term is also called *singular* pattern to distinguish it from complex patterns (represented by instances of pattern structures as described later).

So the `bondi` program `update` is reusable for different singular patterns of target structures. This reusability from explicit first-class pattern parameters stands out above other existing programming languages.

Rather than updating, some processing tasks with large-scale data just collect information. For example, one may want to check if all population values are positive.

Similar to `update`, a function `check`, that simply checks that some property holds for the content of every target structure, can be defined as:

```
let (check:lin(a->b)->(a->bool)->d->bool) \P f x =
  match x with
  | P z -> f z
  | y z -> (check P f y) && (check P f z)
  | z -> True;;
```

where `True`, `False` are two constant constructors of type `bool` as usual and `||` is logical-or. Then validation of population values can be done as:

```
let isPositive x = (x > 0.0);;
check Pop isPositive somedata;;
```

Again, the program `check` achieves parameterization of the data structure, the pattern to match targets, and the operation to apply on targets (the boolean function to check the content of targets).

Although the expressiveness achieved by `bondi` so far stands out above other existing programming languages, it is not yet sufficient for the real situations of processing large-scale data. For example, function `update` updates all population values, without distinguishing the contexts they are in. What if the task becomes: *Add 1% to all population values of cities* (but not of provinces and countries)? In such a case, the computation needs to search for all occurrences of pattern `City` first. Under every matching target, all occurrences of pattern `Pop` must be searched for. It is actually a nesting of two universal quantifications: (*for all cities (for all populations, ...)*). One possible solution for such a nesting of universal searching is nested calls, like:

```
update City (update Pop incrementBy1Percent) someData;;
```

where different patterns are passed into different level of the nested calls. Similarly, the task *verify that all population values of cities are positive* can be done by:

```
check City (check Pop isPositive) somedata;;
```

However, this solution is unsatisfactory, since more complicated patterns will then require nesting of more updates and checks, so that the complexity of programming can increase unboundedly, and automatic processing of patterns would be difficult if possible at all. And there is still the challenge of checking side-conditions, e.g., that the cities must be in Canada.

Obviously, a systematic way to represent complex patterns is desired. Given that singular patterns are first-class values in `bondi`, it is a natural idea to construct abstract structures containing these values to represent complex patterns, analogous to data structures containing first-class datum values.

When designing a data structure, a programmer needs to define constructors that construct instances of the structure from primitive data and smaller structured data. For example, structure `list` is defined as having optional constructors `Nil` and `Cons`; structure `country` is defined as having one unique constructor `Country`. The choice of constructors is made by the programmer based on the ways of combining smaller data into bigger ones.

This is also the case when designing structures for complex patterns. The choice of constructors for a structure is up to its programmer, who should make decision based on the ways of combining simpler patterns into more complex ones under consideration, according to the needs of the application at hand.

As an example, consider this structure definition:

```
datatype boolpat
  at a b =
    | Bsingle of lin(a->b) and (a->bool)
  at (a1,a2)(b1,b2) =
    | Bconcat of lin(a1->b1) and boolpat a2 b2
    | Bdisj of boolpat a1 b1 and boolpat a2 b2
    | Bconj of boolpat a1 b1 and boolpat a2 b2;;
```

Here “**at**” indicates matching with different forms of the type arguments. `boolpat` takes two type arguments, each of which can be either atomic, or pairs of types. Constructor `Bsingle` represent the base case, a singular pattern for boolean verification. Its first parameter is a singular pattern, and the second a boolean function which will be used to verify the content of the target structures matching the singular pattern. Three other constructors are declared recursively, representing three ways of combining smaller patterns into more complex ones: `Bconcat` represents concatenation of a singular pattern with a pattern structure; `Bdisj` represents inclusive disjunction of (the verification results for) two `boolpat` structures; `Bconj` represents conjunction of (the verification results for) two `boolpat` structures.

Although the declaration of `boolpat` is similar to those of data structures, and it uses the same syntax keyword `datatype`, `boolpat` is not a data structure, because its concrete instances will contain values of singular patterns, which are linear function types rather than primitive data types. Such a structure is called a *pattern structure* in this thesis. A pattern structure declaration may include primitive data types, if the patterns of concern will match some primitive values, or the specializations of some pattern cases need data arguments. An example can be found in the next section when defining another version of `boolpat`, called `boolpat2`. Conceptually, ordinary functions should not be parts of a pattern for data processing, because it is not possible to search for a function in a data structure. However, a pattern structure declaration can include ordinary function types for programming convenience, as in `boolpat`. The ordinary function in an instance of a pattern structure should not be used in creating pattern cases to match with data, but can be used to define the specializations for cases. In the pattern cases to match with patterns, as in the

function `checkps` below, it is possible to have the ordinary functions appearing in pattern cases, but the functions should never appear as the first component of any applicative term in the patterns, due to the linearity requirement for patterns.

Pattern instances of a pattern structure can be declared, similar to declaring compound data instances of a data structure. The complex search pattern in the task *verify that all population values of Canadian cities are positive* can be represented by a pattern instance of `boolpat`, as:

```
let is x y = (x == y);; // a general comparison function
let isCanada = (is "Canada");;
let ps1 = Bconcat Country (Bconj (Bsingle CountryName isCanada)
                                (Bconcat City (Bsingle Pop isPositive)));;
```

Handling of a pattern structure is similar to that of a data structure. Since structures are first-class in `bondi`, an instance of a pattern structure, such as `ps1`, is a first-class value, can be referenced, be passed to a program as argument, be traversed and decomposed by pattern matching. Given all these, it is now possible to design programs having complex patterns as parameters. For example, a more general version of function `check`, which has a pattern parameter of type `boolpat`, can be declared as:

```
let (checkps: (boolpat a b) -> d -> bool) ps x =
  match p with
  | Bsingle \P f -> check P f x
  | Bconcat \P p -> check P (checkps p) x
  | Bdisj p1 p2 -> (checkps p1 x) && (checkps p2 x)
  | Bconj p1 p2 -> (checkps p1 x) || (checkps p2 x);;
```

Function `checkps` uses pattern matching to traverse recursively the complex pattern passed in through the first parameter `ps`. This is similar to the traversal of a homogeneous data structure, like function `atListApply` defined earlier in this section. The difference is that `atListApply` finds a data substructure at each step of

the traversal, while `checkps` finds a pattern substructure. Once `checkps` finds pattern substructure representing a singular pattern, it invokes function `check` to perform pattern searching on the second argument, the actual data of some heterogeneous data structure, based on the singular pattern it finds, and applies the boolean function to the end targets. For example, if `checkps` finds that the given complex pattern matches `Bsingle \P f`, it knows that it contains a singular pattern `P` for end targets and a boolean function `f` to apply to the end targets. It then invokes `check P f` on the second argument, the actual data. If the given complex pattern is a `Bconcat \P ps1`, it is a singular pattern `P` concatenated with another complex pattern `p`. Hence `checkps` should be invoked recursively on `p` and a data substructure matching `P`. If the given complex pattern is a `Bdisj p1 p2`, it is an inclusive disjunction of the checking results respectively for complex pattern `p1` and `p2`, and so on.

Function `checkps` is designed specifically for pattern structure `boolpat`. It plays two roles:

- it is a semantic interpreter of complex patterns of structure `boolpat`. What it does with `Bsingle`, `Bconcat`, `Bdisj` and `Bconj`, respectively reflects what these constructors mean, i.e., the ways of combining smaller patterns into more complex ones.
- it is a program for universal traversal of an arbitrary data structure. It uses the information it gets from the complex pattern argument to decide what to do at each traversal step.

Now it is trivial to express the task *verify that all population values of Canadian cities are positive*:

```
let result1 = checkps ps1 somedata;;
```

What a simple form for the expression. The complexity of the pattern for a universal searching, i.e., the different ways of relating all singular patterns involved in the task, is totally captured by the (many options of) internal organization of a pattern structure, whose instance is passed to the pattern searching program as a whole. Function `checkps` achieves all the levels of parameterization described in Section 1.2. It is reusable for different boolean functions to verify target structures, different structures of data repositories, different singular pattern components of complex patterns, and different organizations of these singular patterns. It is also reusable for arbitrary increases in the complexity of the data structures and the complexity of patterns (within the given pattern structure declaration), achieving high scalability on these complexities. In other programming languages, programming with complex patterns is clumsy, and introducing a new kind of complex pattern would require significant re-programming and even re-design of the type systems, if it is possible at all.

Now let us go back to the updating task: *Add 1% to the population of all the Canadian cities*. In the same way as `boolpat`, a pattern structure to capture complex patterns for updating can be declared as:

```
datatype updpat
  at a b =
    | Usingle of lin(a->b) and (a->a)
  at (a1,a2) (b1,b2) =
    | Uconcat of lin(a1->b1) and updpat a2 b2
    | Uboth of updpat a1 b1 and updpat a2 b2
    | Ucond of boolpat a1 b1 and updpat a2 b2;;
```

The base case `Usingle` is the same as that in `boolpat`, except that the second parameter is a updating function rather than a boolean function to apply to the content of target structures.

There are also three constructors here for pattern combination. `Uconcat` has the same meaning as in `boolpat`. `Uboth` indicates combination of two updating tasks, both of which need to be done on a given data repository. Note that the conditional constructor `Ucond` takes two parameters of different types, one `boolpat`, the other `updpat`. The `boolpat` serves as a side condition. Only when the checking of this condition is satisfactory will the update proceed.

The complex pattern *the population of all the Canadian cities* can be expressed as:

```
let ps2 = Uconcat Country (Ucond (Bsingle CountryName isCanada)
                                (Uconcat City (Usingle Pop incrementBy1Percent)));;
```

Here is a general `updateps` function taking a pattern of type `updpat` as argument:

```
let (updateps: (updpat a b) -> c -> c) p x =
  match p with
  | Usingle \P f -> update P f x
  | Uconcat \P p -> update P (updateps p) x
  | Uboth p1 p2 -> updateps p2 (updateps p1 x)
  | Ucond bp1 p2 -> if (checkps bp1 x)
                    then updateps p2 x
                    else x ;;
```

Similar to function `checkps`, function `updateps` acts as an interpreter of the constructors of `updpat`, and performs universal traversal of the given data structure of type `c` based on the information it gets from the given complex pattern of type `updpat`. It invokes function `update` to do pattern searching for singular patterns and do the final update on end targets. It also invokes `checkps` to verify side conditions.

Now the task *Add 1% to the population of all the Canadian cities* can be done by:

```
let result2 = updateps ps2 somedata;;
```

Besides checking and updating, other operations for manipulating heterogeneous data, for example, folding(extracting), can be implemented in a similar way.

3.3 Understanding and Creating Pattern Structures

What is a pattern structure, really? A pattern structure is actually a formal definition of the syntax to express a group of possible complex patterns being considered in a particular application. It defines the way to express the handling of final targets, and to express different ways of combining simpler patterns into more complex ones. The recursive cases in its declaration are actually combining operators to combine patterns. What kinds of combination to consider is up to the programmer based on the application at hand. In the `boolpat` case, the basic task is to verify every final target using a boolean function, and the possible combinations are concatenation, inclusive disjunction and conjunction.

A pattern structure only defines a syntax to express complex patterns. The function(s) to use it serve as its semantic interpreter. For example, function `checkps` actually defines what `Bsingle`, `Bconcat`, `Bdisj` and `Bconj` mean, i.e., what to do when instances of them are given. For a pattern structure to make sense, at least one function to use the structure needs to be defined, such as `checkps` for `boolpat` and `updateps` for `updpat`. The pattern structure and the function(s) are expected to be designed together by their programmer(s).

By default, universal traversal of a data structure visits every substructure recursively and exhaustively without doing anything else but branching to other substructures. A pattern searching task is a specialization of such standard operation, probably doing different things at different substructures. A complex pattern, represented by an instance of a pattern structure, actually captures all of the specialization

information about what to do at each traversal step and how to get to the final targets. It serves as a description of a pattern searching task, although this description is partial, that is, not covering the substructures at which only the standard traversal operation applies. For example, instance `ps1` of pattern structure `boolpat`:

```
let ps1 = Bconcat Country (Bconj (Bsingle CountryName isCanada)
                                (Bconcat City (Bsingle Pop isPositive))));;
```

represents a complex pattern, which defines the tasks:

- at each node matching pattern `Country`, check all its child nodes against complex pattern `(Bsingle CountryName isCanada)`, and if it matches (a conjunction), check all its child nodes against pattern `(Bconcat City (Bsingle Pop isPositive))`;
- at each node matching `CountryName`, verify its content using the boolean function `isCanada`;
- at each node matching `City`, check all its child nodes against pattern `(Bsingle Pop isPositive)`;
- at each node matching `Pop`, verify its content using the boolean function `isPositive`;

Functions to use a pattern structure interpret the information carried by the instances of the pattern structure, and implement the specialized traversal. A function having a pattern structure parameter can adapt to different complex patterns encoded as instances of the structure. For example, a task to check Australian city areas instead of Canadian city populations needs a different instance of `boolpat` than `ps1`:

```
let ps3 = Bconcat Country (Bconj (Bsingle CountryName (is "Australia"))
                                (Bconcat City (Bsingle Area isPositive))));;
```

but the task can still be done by function `checkps`.

The roles of a pattern structure and its accompanying functions are analogous to the case of query languages. A pattern structure plays the role of a query language. An instance of the pattern structure, representing a complex pattern, is like a query, which defines what target elements to look for. The data-processing function having a pattern structure parameter is like the query language engine in a database management system, which can accept and interpret any query and traverse a database for target elements accordingly. However, a significant difference is that the definition of a pattern structure is a programming task, while creating a query language is a language design task. Another difference is that the definition of pattern structures, the declaration of pattern instances, the interpretation of pattern instances, and the search for patterns, are all done in one language environment and are expected to be programmed at the same time by the same programmers, while query languages, their interpreter engines, and the use of queries in computations, involve multiple language environments and are usually developed separately.

Therefore, using pattern structures in `bondi` has both the advantages of using query languages and the advantages of computing in a single programming environment. As demonstrated in Section 3.1, the existing approaches to do pattern searching in a single language environment without using query languages would require re-programming and re-compilation for new search patterns. With the use of pattern structures in `bondi`, new search patterns can be instances of existing pattern structures, and so can be handled by existing programs having parameters of those pattern structures. At the time of writing this thesis, there is only an interpreter available for `bondi`, so a new instance of pattern structure must be interpreted before it can

be passed to a program. It is expected that in a compiler-based environment the instantiation of pattern instances can be automated at runtime, so that the change of task does not need re-compilation at all.

A pattern structure may need to be modified, or a new one may need to be created, which means re-programming, if:

- the operation on final targets changes. For example, if the operation on targets is to update them rather than to verify them logically, a new pattern structure `updpat` must be created.
- the way of combining simpler patterns into more complex ones changes, or a new kind of combinations emerges. For example, if `boolpat` needs to express the Kleene star of a singular pattern, a new constructor may need to be added to the declaration:

```
datatype boolpat
  at a b =
    | Bsingle of lin(a->b) and (a->bool)
    | Bkstar of lin(a->b) and (a->bool)
  at (a1,a2)(b1,b2) =
    | Bconcat of lin(a1->b1) and boolpat a2 b2
    | Bdisj of boolpat a1 b1 and boolpat a2 b2
    | Bconj of boolpat a1 b1 and boolpat a2 b2;;
```

where an instance of the form `Bkstar P f` represents a basic task of verifying all final targets matching P^* . Alternatively, a programmer may choose to declare a new pattern structure rather than changing `boolpat`.

There are many ways to combine simpler search patterns into more complex ones. When new ways of combination are needed, programmers are totally free to change old pattern structures or declare new ones as necessary according to computations at

hand. But they also need to change the existing programs or create new ones that use those pattern structures. Fortunately, this is just a programming task, while in other existing languages it would be a task of language re-design, if it were possible at all. By programming using pattern structures, extending a data processing system for arbitrary new kinds of complex patterns becomes fairly easy. Readers can find new pattern structures for more complex patterns in Chapter 4.

One might argue that the need of programming effort for new kinds of search patterns weakens the claim of reusability. It does. Fortunately, it is easy to declare a pattern structure covering a wide range of possible patterns and a wide range of pattern complexity, with only a few constructors, like `boolpat`. What is more striking is that pattern-searching programs using pattern structures adapt to the complexity of patterns as simple patterns are piled up into more complex ones, without reprogramming. In other programming languages, when search patterns become more complex, not only is reprogramming required, but also the difficulty of programming increases, and such increase is unbounded when pattern complexity increases arbitrarily.

Despite the name “pattern structure”, any kinds of first-class terms, not just singular patterns, can appear in a structure. There is usually a need to declare a pattern structure with data and ordinary functions in it, for programming convenience. These data and functions may conceptually not be parts of the patterns to search for in target data, but are used in creating the specialized actions for various pattern cases. It has been seen that `boolpat` contains ordinary boolean functions, for example. One may also declare a slightly different version of `boolpat` to include the value against which the boolean function verifies the targets:

```
datatype boolpat2
  at a b =
    | Bsingle of lin(a->b) and (a->a->bool) and a
  at (a1,a2)(b1,b2) =
    | Bconcat of lin(a1->b1) and boolpat a2 b2
    | Bdisj of boolpat a1 b1 and boolpat a2 b2
    | Bconj of boolpat a1 b1 and boolpat a2 b2;;
```

so that we do not have to define a boolean function in advance every time we need one. For example, pattern instance `ps1` can be defined without prior declaration of boolean functions `isCanada` and `isPositive`:

```
let ps1 = Bconcat Country (Bconj (Bsingle CountryName (==) "Canada")
                                   (Bconcat City (Bsingle Pop (>) 0))));;
```

Here `(==)` is the function for equality test, a prefix form of the `==` operator. This is a convention in functional languages, applicable to all built-in binary operators. A binary operator in parentheses is used as the function name for the prefix form of the operator.

3.4 Summary

One basic computation in manipulating heterogeneous data is universal pattern searching. Existing general-purpose programming languages in practice are problematic for expressing such computations, because their type systems are designed for homogeneous data structures, and have problems typing the pattern matching in heterogeneous data. Specifically, it is a problem to arrange types for the different substructures in a heterogeneous data structure and the pattern to match with these substructures; it is an even bigger problem to arrange a type for the pattern when the pattern is complex, and may have different complexity in different tasks.

These are solved by using well-typed pattern structures and expressing the pattern searching computation in a language supporting first-class types of structures and patterns. Using this approach, a pattern searching program can be fully statically typed, and can adapt to different structures/substructures of the data to process, different patterns to search for, and different complexities of the patterns. Type safety, high reusability and high scalability can be achieved altogether.

Chapter 4

XML Data Processing

XML is becoming the standard format for data published and exchanged on the Internet. A wide variety of Internet applications involve XML data processing. However, existing solutions for processing XML data, either using query languages or processing natively in general-purpose programming languages, are far from satisfactory (Section 2.3). The dissatisfaction comes from the heterogeneous, semi-structured nature of XML data.

Given that XML data is naturally suitable to be represented as heterogeneous-structured data, the proposed approach of this thesis is immediately applicable to XML data processing. Applying the approach to expressing XML data processing is a perfect illustration of the advantages of the approach. By using pattern structures, it is easy to make natively-well-typed XML processing programs that are reusable for data from different XML schemas and for different search patterns; and it is easy to include new kinds of patterns by programming, without any change to the programming language.

Section 4.1 briefly describes how XML data are represented as heterogeneous data

in `bondi` and are processed easily by the programs introduced in Section 3.2; Section 4.2 shows it is only a matter of programming to fully support XPath patterns natively in `bondi` for XML data processing; Section 4.3 shows the handling of another kind of patterns, horizontal patterns in regular-expression style, again natively in `bondi`, for XML data processing. These complex patterns have been considered individually in other existing approaches but have never appeared fully together in one language.

Note that the demonstration in this chapter is based on hand-made artificial XML data examples, not production XML data. The `bondi` encoding of the examples has gone through the `bondi` interpreter though, to ensure the correctness.

4.1 Representing and Transforming XML Data

Due to the hierarchical data model of XML, it is straightforward to represent XML data as heterogeneous trees in `bondi`. Actually, the geographical data example used in Section 3.2 can be a representation of a piece of XML data. Consider the following XML schema:

```
<xs:element name="cityname" type="xs:string"/>
<xs:element name="popul" type="xs:decimal"/>
<xs:element name="area" type="xs:decimal"/>

<xs:element name="city">
  <xs:complexType><xs:sequence>
    <xs:element ref="cityname"/>
    <xs:element ref="popul"/>
    <xs:element ref="area"/>
  </xs:sequence></xs:complexType>
</xs:element>

<xs:element name="provname" type="xs:string"/>
<xs:element name="province">
  <xs:complexType><xs:sequence>
    <xs:element ref="provname"/>
    <xs:element ref="popul"/>
  </xs:sequence></xs:complexType>
</xs:element>
```

```

        <xs:element ref="area"/>
        <xs:element ref="city" minOccurs="0"
            maxOccurs="unbounded"/>
    </xs:sequence></xs:complexType>
</xs:element>

<xs:element name="countryname" type="xs:string"/>
<xs:element name="country">
    <xs:complexType><xs:sequence>
        <xs:element ref="countryname"/>
        <xs:element ref="popul"/>
        <xs:element ref="area"/>
        <xs:element ref="province" minOccurs="0"
            maxOccurs="unbounded"/>
    </xs:sequence></xs:complexType>
</xs:element>
<xs:element name="geodata">
    <xs:complexType><xs:sequence>
        <xs:element ref="country" minOccurs="0"
            maxOccurs="unbounded"/>
    </xs:sequence></xs:complexType>
</xs:element>

```

Of course, a validating XML parser is needed to transform textual XML files into bondi data format for processing. Assuming such parsing, the above schema can be represented as bondi data structures (mostly a repeat from Section 3.2):

```

datatype cityname = Cityname of string;;
datatype popul = Popul of float;;
datatype area = Area of float;;
datatype city = City of cityname * popul * area;;

datatype provname = Provname of string;;
datatype province = Province of provname * popul * area * list city;;

datatype countryname = Countryname of string;;
datatype country = Country of countryname * popul * area * list province;;

datatype geodata = Geodata of list country;;

```

For demonstration clarity, this chapter does not consider recursive and mutual recursive definitions in XML schemas. Such definitions would bring higher heterogeneity, making XML data heterogeneous graphs rather than trees.

The pattern structures `boolpat` and `updpat`, and their corresponding functions `checkps` and `updateps`, introduced in Section 3.2 are then immediately applicable to XML data processing. XML processing computation such as verifying or updating populations of Canadian cities can be expressed with these facilities, as described in Section 3.2.

Besides checking and updating, another common important operation on XML data is transformation, for example, transforming an XML file into an HTML file. Extracting information from an XML file, such as getting a list of city names from a geographical data file, is also a transformation operation. XML data transformation can be implemented in `bondi` as a folding operation, using the pattern-structure approach.

A function `foldleftp` can be defined for folding in heterogeneous data based on a singular pattern, in a similar way to `check` and `update`:

```
let (foldleftp:lin(a->b)->(e->a->e)->e->d->e) \P f w x =
  match x with
  | P z -> f w z
  | y z -> foldleftp P f (foldleftp P f w y) z
  | z -> w;;
```

This function takes a piece of data `x`, of arbitrary structure (type) `d`, searches for target elements matching singular pattern `P`, and puts their content into a data container `w` of another structure `e`. Parameter `f` is a function responsible for combining a single value of type `a` into the result container.

More sophisticated folding operations, based on complex patterns, need to use pattern structures. Consider this structure:

```

datatype foldpat
  at a b c =
    | Fsingle of lin(c->b)
  at (a1,a2) (b1,b2) c =
    | Fconcat of lin(a1->b1) and foldpat a2 b2 c
    | Fcond of boolpat a1 b1 and foldpat a2 b2 c;;

```

The purpose of the constructors of `foldpat` is similar to the corresponding ones in `boolpat` and `updat`. A difference is that `foldpat` has a type parameter `c` to expose the type of the content of target elements. Folding elements satisfying a complex pattern of type `foldpat` can then be done by:

```

let (foldleftps:((foldpat a b c)->(e->c->e)->e->d->e) p f w x =
  match p with
  | Fsingle \P -> foldleftp P f w x
  | Fconcat \P p1 -> foldleftp P (foldlefts p1 f) w x
  | Fcond p1 p2 -> if (checkps p1 x)
                    then foldleftps p2 f w x
                    else w;;

```

Many essential XML transformation operations can be expressed using `foldleftp` and `foldleftps`. For example, extracting information from XML data based on a search pattern and a filter is the most common kind of XML query. It can be easily implemented by `foldleftps`.

Suppose we want a list of names of Canadian cities. This query consists of three components: the pattern to search for: `...country...city...cityname`; the data filter: `country name is Canada`; and the way to construct the result. The search pattern can be represented as an instance of a `foldpat` containing a `boolpat` pattern as a side condition, and the filter is a boolean function carried by that `boolpat` pattern:

```

let ps4 = Fconcat Country (Fcond (Bsingle CountryName isCanada)
                                (Fconcat City (Fsingle CityName)));;

```

An accumulating function needs to be given to the folding function as a parameter, to accumulate matching elements. Different accumulating functions can be designed

to construct the final result in different ways. If the result is to be a list of strings for city names, i.e., of type `list string`, the accumulating function can be as simple as:

```
let (listInsert: list a -> a -> list a) x y =  
  match x with  
  | Nil -> Cons y Nil  
  | Cons z r -> Cons y (Cons z r);;
```

Of course, more sophisticated accumulating functions can be designed, for example to check for duplicates, or to build results in a structure other than a flattened list of string.

Given the pattern and accumulating function, the task is straightforward:

```
let result3 = foldleftps ps4 listInsert [ ] somedata;;
```

Many other XML transformation operations, such as extraction while preserving or restructuring original structures, indexing and sorting, are basically folding operations as well and can be implemented in a similar way using the folding functions. More examples are available in an earlier technical report [55].

As reviewed in Section 2.3, there are two styles of search patterns commonly seen in XML data processing: vertical patterns and horizontal patterns. Pattern structures `boolpat`, `updpat` and `foldpat` have been able to represent a wide range of vertical patterns. More patterns can be covered by adding new constructors to these three pattern structures, or defining new pattern structures as needed. This is just a programming task, in contrast to other XML processing approaches, where new kinds of patterns need new language features at best, and are impossible at worst.

The following two sections demonstrate the handling of complex patterns in vertical XPath style and horizontal regular-expression style.

4.2 Vertical XPath Patterns

The specification of a location path in XPath 1.0 [27] expresses a wide range of vertical patterns. A location path expression consists of a sequence of location steps delimited by “/”. Each step may contain three kinds of components:

```
Step ::= AxisName::NodeTest [PredicateExpression]*
```

Each kind of component can be represented in `bondi` by a pattern structure, and so can the location steps and location paths. This section shows how to declare pattern structures for them, and how to program with these structures. For clarity, only XML elements are considered, leaving out attributes, namespaces, comments, processing instructions etc., which will need to be handled in a full practical implementation of the approach.

Axes can be represented by constant constructors of type `axis`, as follows:

```
datatype axis = | Child
                | Descendant
                | DescendantOrSelf
                | Following
                | FollowingSibling
                | Parent
                | Ancestor
                | AncestorOrSelf
                | Preceding
                | PrecedingSibling
                | Self;;
```

A node test can be a test for a node name or a node type. Since only XML elements are considered, a node type test is not necessary. A node name test can match with the wildcard “*” or a specific (element) name:

```
datatype nametest a b = | NodeName of lin(a->b)
                       | Any;;
```

Recall that `lin(a->b)` here denotes “linear function type `a->b`”, the form of types for singular patterns.

The specification of predicate expressions in XPath 1.0 [27] is very general. However, for clarity, the demonstration here restricts each expression to contain at most one location path and no library function calls. So expressions such as `[(sales/price)*(inventory/quantity) > 350.0]` are excluded. Arithmetic operations are also excluded. A typical predicate expression is of the form `[sales/price < 9.0]`. This is conceptually similar to `boolpat` in Section 3.2 and `boolpat2` in Section 3.3, except that each stage of the path here is a (more expressive) `step` rather than a singular pattern, and there is no need for conjunction and disjunction. Formally, a predicate can be defined as:

```
datatype predicate (a1,a2) (b1,b2) =
  | PredGoal of step (a1,a2) (b1,b2) and (a1->a1->bool) and a1
  | PredStage of step a1 b1 and predicate a2 b2;;
```

where `step` is another pattern structure to be defined later. To represent a list of (zero or more) predicates in a well-typed way, a pattern structure similar to `list` is needed:

```
datatype predlist (a1,a2) (b1,b2) =
  | NilPred (* for empty list *)
  | ConsPred of predicate a1 b1 and predlist a2 b2;;
```

Now the pattern structure for location steps is:

```
datatype step (a1,a2) (b1,b2) =
  Step of axis and nametest a1 b1 and predlist a2 b2;;
```

and that for location paths is:

```
datatype locpath
  at a b c =
  | PathGoal of step (c,a) b
  at (a1,a2) (b1,b2) c =
  | PathStage of step a1 b1 and locpath a2 b2 c;;
```

Note that `locpath` is similar to `foldpat` defined in the previous section, in that it uses an extra type parameter `c` to expose the content type of the element of the final step in a path for programming convenience, giving the flexibility for programmers to use the pattern structure for different tasks (say, either boolean checking or updating), and apply whatever functions they like to the final matching targets. Again, `locpath` uses `step` rather than singular pattern for each stage of the path.

The declarations of `predicate`, `predlist` and `step` are mutually recursive. In the XPath specification, matching the corresponding definition in the XPath specifications. Each step of a location path can contain a list of predicates, and a predicate in turn can contain a location path which is a sequence of steps, just as in the following XPath (abbreviated) location path for national population:

```
country[//city/cityname="Kingston"]/population
```

where the predicate `[//city/cityname="Kingston"]` in the location path contains another location path `//city/cityname`. Note that `step` is used, rather than `locpath` as in the XPath specification, in the definition of `predicate`, because otherwise a predicate would also need an extra type parameter `c` for its own path, so conflicting with the extra type parameter of its context location path.

So far, only a small number (six) of pattern structures have been defined, i.e., `axis`, `nametest`, `predicate`, `predlist`, `step`, and `locpath`, but they have been able to represent a small but most commonly used subset of the XPath location path specification. This subset can be described as:

$$\begin{aligned}
\textit{LocationPath} & ::= \textit{Step} \mid \textit{Step} \textit{'/' LocationPath} \\
\textit{Step} & ::= \textit{AxisName} \textit{'::' NameTest Predicate}^* \\
\textit{AxisName} & ::= \textit{'child'} \mid \textit{'descendant'} \mid \textit{'descendant-or-self'} \mid \textit{'following'} \\
& \quad \mid \textit{'following-sibling'} \mid \textit{'parent'} \mid \textit{'ancestor'} \mid \textit{'ancestor-or-self'} \\
& \quad \mid \textit{'preceding'} \mid \textit{'preceding-or-self'} \mid \textit{'self'} \\
\textit{NameTest} & ::= \textit{'*'} \mid \textit{aLocalElementName} \\
\textit{Predicate} & ::= \textit{'[' PredExpr ']'} \\
\textit{PredExpr} & ::= \textit{LocationPath RelationalOp Constant} \\
\textit{RelationalOp} & ::= \textit{'='} \mid \textit{'!='} \mid \textit{'>'} \mid \textit{'<'} \mid \textit{'>='} \mid \textit{'<='} \\
\textit{Constant} & ::= \textit{StringConstant} \mid \textit{NumberConstant}
\end{aligned}$$

A location path expression (given as a pattern to search for) conforming to this specification can then be transformed into a pattern instance in `bondi` for pattern searching computation. Assume that a validating parser is available to parse a pure-text XPath expression into tokens conforming to the above specification (a validating XPath parser is a basic component in any existing XPath engine). The parser will also transform abbreviated XPath expressions into full ones, and validate the typing for the relational operations in predicates. Also assume that all element names appearing in XPath expressions are declared as `bondi` data types and corresponding constructors in the way that each data type name is the same as the element name. Let mapping \mathbb{C} return the constructor name for a given element name (in this thesis, a constructor name is also the same spelling as the corresponding XML element name but capitalized at the first character), for example, $\mathbb{C}(\textit{city}) = \textit{City}$. Also, let mapping \mathbb{F} gives a predefined `bondi` function name for a relational operator. For example, $\mathbb{F}(=) = (=)$, $\mathbb{F}(>) = (>)$, and so on, recalling that a binary operator in parentheses stands for a function in prefix form of the same operation. The transformation \mathbb{T} of a location path into an `bondi` pattern instance can then be (somewhat informally)

described as:

```

T($LocationPath) =
  | $Step → PathGoal T($Step)
  | $Step / $LocationPath → PathStage T($Step) T($LocationPath)

T($Step) =
  | $AxisName :: $NameTest $Predicate* → Step T($AxisName) T($NameTest) T($Predicate*)

T($AxisName) =
  | child → Child
  | descendant → Descendant
  | descendant-or-self → DescendantOrSelf
  | following → Following
  | following-sibling → FollowingSibling
  | parent → Parent
  | ancestor → Ancestor
  | ancestor-or-self → AncestorOrSelf
  | preceding → Preceding
  | preceding-or-self → PrecedingOrSelf
  | self → Self

T($NameTest) =
  | * → Any
  | $aLocalElementName → NodeName C($aLocalElementName)

T($Predicate*) =
  | [] → NilPred
  | $Predicate $Predicate* → ConsPred T($Predicate) T($Predicate*)

T($Predicate) =
  | [$PredExpr] → T($PredExpr)

T($PredExpr) =
  | $Step $RelationalOp $Constant → PredGoal T($Step) F($RelationalOp) $Constant
  | $Step / $PredExpr → PredStage T($Step) T($PredExpr)

```

Let us see how to transform the location path for national population mentioned above: `country[//city/cityname="Kingston"]/population`. It is in abbreviated form. Its full form is:

```
child::country[descendant::city/child::cityname="Kingston"]/child::population
```

The transformation of this path expression will be shown from bottom up and split-
ted into separate steps for readability, although the transformation rules above are
described in a top-down manner. Start from the predicate in the path:

```
let step4city = Step Descendant (NodeName City) NilPred;; (* for "descendant::city" *)
let step4cname = Step Child (NodeName Cityname) NilPred;; (* for "child::cityname" *)
let pred4cntry = PredStage step4city (PredGoal step4cname (==) "Kingston");;
```

and the whole location path for national population can be encoded as:

```
let step4cntry = Step Child (NodeName Country) (ConsPred pred4cntry NilPred);;
                                                    (* for "child::country[...]" *)
let step4pop = Step Child (NodeName Pop) NilPred;; (* for "child::population" *)
let poppath = PathStage step4cntry (PathGoal step4pop);; (* the whole path pattern *)
```

As a demonstration, a polymorphic function `updatepath` is designed below to accept an XPath pattern as argument, update the content of every XML element that matches the pattern, for example updating the population of every country with a city named “Kingston”.

First of all, some helper functions are needed for clarity. Function `checkstep` has three parameters `s`, `f` and `x`. It verifies, in a piece of XML data `x`, whether there exists an element matching the pattern (a single location step) `s`, with the content of this element satisfying a boolean function `f`. Let us start with the simple cases, considering only descendant and child for the axis of a pattern step, and ignoring the possibility of a wildcard in the name test. Recall from Section 3.2 that multiple children of one element (structure) are represented as a tuple of elements, which are nested pairs of product type in `bondi`.

```
let (checkstep:(step (a1,a2) b)-> (a1->bool)->c->bool ) s f x =
  match s with
  |Step Descendant (NodeName N) p1 ->
    match x with (
      |N z ->(checkpredlist p1 z) && (f z)
      |y z ->checkstep s f y || checkstep s f z )
  |Step Child (NodeName N) p1 ->
    match x with (
      |N z -> (checkpredlist p1 z) && (f z)
      |Pair y z -> checkstep s f y || checkstep s f z )
  |t -> false;;
```

Note the simplicity in the code for traversing the piece of XML data x (the inner pattern matching). Only two cases are needed for the generalized pattern matching. Also note that the pattern t in the final case is like a wildcard that can match anything, catching objects that do not match any preceding cases.

In function `checkstep`, another function `checkpredlist` is invoked, which is defined as follows:

```
let (checkpred:(predicate a b)->c->bool) p x =
  match p with
  |PredGoal s f y -> checkstep s (f y) x
  |PredStage s sp->checkstep s (checkpred sp) x;;

let (chkpredlist:(predlist a b)->c->bool) pl x =
  match pl with
  | NilPred -> true
  | ConsPred p spl -> (checkpred p x) && (chkpredlist spl x);;
```

These two functions evaluate predicate p and a list of predicate pl respectively against a given piece of XML data x . Note that the three functions above are mutually recursive. By using them, update functions with XPath patterns can be easily defined, as follows.

Function `updatestep` updates elements that match the singular pattern of a single location step. Again, for now, only “descendant” and “child” are considered for an axis in a step:

```
let (updatestep: (step (a1,a2) b)->(a1->a1)->c->c) s f x =
  match s with
  |Step Descendant (NodeName N) pl ->
    match x with (
    |N z -> if (chkpredlist pl z) then N (f z)
    |y z ->(updatestep s f y)(updatestep s f z))
  |Step Child (NodeName N) pl ->
    match x with (
    |N z -> if (chkpredlist pl z) then N (f z)
    |Pair y z -> Pair (updatestep s f y)(updatestep s f z))
  |t -> x;;
```

Function `updatepath` updates elements that match a complex XPath pattern, a `locpath`:

```
let (updatepath: (locpath a b c)->(c->c)->d->d) lp f x =
  match lp with
  | PathGoal s -> updatestep s f x
  | PathStage s slp -> updatestep s (updatepath slp f) x;;
```

Similarly, other general functions that compute with `locpath` can be defined. For example, function `checkpath` verifies that there exists at least one element matching a given XPath pattern, and that the content of that element satisfies a given boolean function:

```
let (checkpath: (locpath a b c)-> (c->bool)->d->bool ) lp f x =
  match lp with
  | PathGoal s -> checkstep s f x
  | PathStage s slp -> checkstep s (checkpath slp f) x;;
```

Adding more cases of patterns only incurs more programming work. For example, adding the other axis cases can be done in a modular way, by adding cases to the pattern matching in the program code of `checkstep` and `updatestep` respectively; no changes to the other functions or the language are needed.

The program examples here are intended for proof-of-concept demonstration, not a full implementation for the XPath specification. XPath features not covered here can be handled in a similar way, but programming with them may require novel algorithms that are beyond the scope of this thesis. There is a rich literature on XPath evaluation algorithms, for example [7, 46, 86], and it is expected that most algorithms can be adapted to our approach.

4.3 Horizontal Regular Expressions

In contrast to vertical patterns that describe the relationship of XML elements from different levels of an XML data hierarchy, a horizontal pattern matches the occurrences of sibling elements on the same hierarchical level under the same parent element. For example, `province[provname, population, area, city*]` is a horizontal pattern. New languages have been created to specifically handle XML processing with horizontal patterns in regular-expression style [53, 102]. Regular-expression types are introduced in those approaches, incurring a significant amount of work for designing and verifying type systems and compilers. These languages are not able to handle vertical patterns, and there is no easy way to extend them to do so.

This section shows that horizontal patterns can be handled by using pattern structures in `bondi`, again without the need to change the language or create a new one. For demonstration, only four cases of horizontal regular expression are considered: singular patterns, Kleene star of singular patterns, pattern concatenation, and pattern alternation.

```
datatype regexp
  at a b =
  | Single of lin(a->b)
  | Kstar of lin(a->b)
  at (a1,a2)(b1,b2)
  | Concat of regexp a1 b1 and regexp a2 b2
  | Altern of regexp a1 b1 and regexp a2 b2;;
```

The transformation rules to transform a regular expression of XML element names into an instance of pattern structure `regExp` is quite straightforward, so the formal definition of the transformation is omitted here. Below are a few basic examples:

pattern “city” is	Single City
pattern “(city*)” is	Kstar City
pattern “(provname, city*)” is	Concat (Single Provname) (Kstar City)
pattern “(province state)” is	Altern (Single Province) (Single State)

Treating instances of this pattern structure as horizontal patterns, a function `localmatch` can be designed (as a helper function) to check the existence of such patterns in a given tuple of XML elements and stop immediately at the first mismatch. Recall from Section 3.2 that multiple children of one element are represented as a tuple of elements, which are nested pairs of product type in `bondi`.

```

let (localmatch:(regexp a b)->c->bool) r x =
  match r with
  | Single P -> (
    match x with
    | Pair (P y) z -> true
    | P z -> true
    | z -> false )
  | Kstar P -> (
    match x with
    | Pair (P y) z -> localmatch (Kstar P) z
    | z -> true )
  | Concat (Single P) r2 -> (
    match x with
    | Pair (P y) z -> localmatch r2 z
    | z -> false )
  | Concat (Kstar P) r2 -> (
    match x with
    | Pair (P y) z -> localmatch r z
    | z -> match r2 z )
  | Concat (Concat r3 r4) r2 -> localmatch (Concat r3 (Concat r4 r2))
  | Concat (Altern r3 r4) r2 -> localmatch (Concat r3 r2) || localmatch (Concat r4 r2)
  | Altern r1 r2 -> (localmatch r1 x) || (localmatch r2 x);;

```

Table 4.1: Function `localmatch`

As shown in Table 4.1, in function `localmatch`, generalized pattern matching

against data structures is simple, requiring no more than three cases most of the time. Pattern matching against the pattern structure constructors, on the other hand, should have cases covering all possible constructors.

Once the basic machinery is in place, complicated computations that sound difficult can be implemented easily. For example, a more useful function `globalsearch` checks the existence of a horizontal regular-expression pattern in an entire piece of arbitrary XML data. It can be defined by using `localmatch`, as in Table 4.2.

```
let (globalsearch:(regexp a b)->c->bool) r x =  
  match r with  
  | Pair y z -> if (localmatch r (Pair y z)) then true  
                 else (globalsearch r y) || (globalsearch r z)  
  | y z -> if (localmatch r (Pair y z)) then true  
            else (globalsearch r y) || (globalsearch r z)  
  | z -> false;;
```

Table 4.2: Function `globalsearch`

4.4 Summary

Application of the pattern-structure approach to XML data processing demonstrates the advantages of the approach in manipulating heterogeneous data. By using pattern structures, XML data processing can be done natively in a single general-purpose programming language. The XML processing programs can adapt to data with different XML schemas, different patterns, and different complexity of patterns. The inclusion of XPath patterns and horizontal regular-expression patterns (and any other kinds of patterns when necessary) only requires declarations of new pattern structures and

new programs parametric on these structures, rather than changes to the programming language itself. These XML processing programs are statically typed, allowing efficient correctness/security verification by type checking. This is especially beneficial when mobile programs are sent to process extremely large XML data repositories that are immovable.

Chapter 5

Java Bytecode Analysis

Program code manipulation is another good application scenario for the approach of using pattern structures to handle heterogeneous data. This chapter specifically examines manipulation of Java bytecode.

Security enforcement for Java bytecode programs in practice tends to use either static type checking, or runtime monitoring with code instrumentation (see Section 2.2.3). Both security mechanisms manipulate static bytecode programs. Static type checking scans through a bytecode program and searches for syntactic patterns indicating violation of certain (semantic) typing rules. Code instrumentation also searches for syntactic patterns in a bytecode program, looks for locations that have the potential to violate desired security policies, and inserts runtime checking code to prevent the violations. The common computation in both security mechanisms is pattern searching all over the whole bytecode program of concern, which is actually a heterogeneous tree of different bytecode syntactic terms. The proposed approach for general heterogeneous data processing is immediately applicable.

Using the pattern-structure approach to implement bytecode security enforcement

is advantageous in terms of security and reusability. To enforce security on the security enforcement programs themselves, static type checking is desirable for efficient verification and full security guarantee. This requires security enforcement programs to be developed in a statically-typed language. On the other hand, security enforcement programs must readily adapt to changes of security policies or target code syntax to avoid high costs for program development and maintenance. Such changes may be frequent, reflecting a changing environment, or user choices. It has been shown in Section 2.1 and Section 3.1 that the existing general-purpose programming languages have problems achieving both static typing and reusability at the same time in processing heterogeneous data, especially in the presence of complex search patterns. This chapter shows that the approach proposed in this thesis solves the problem.

The rest of this chapter is structured as follows. Section 5.1 describes how a Java bytecode program is represented as heterogeneous data in **bondi**. It also serves as an informal review of the bytecode syntax specification. Section 5.2 shows that static type checking, and more generally, semantic checking, can be expressed in a well-typed and highly reusable way by using pattern structures in **bondi**. Section 5.3 shows that the same programming framework can be used for bytecode instrumentation.

5.1 Representing Java Bytecode

Any program executable in a Java virtual machine is in Java class file format [77, Chapter 4], also called Java bytecode. The format is well-typed and tree-structured, so that it is straightforward to define the tree structure of Java bytecode in a typed programming language, parse any given class file into a syntax tree of the structure, and analyze the file by manipulating the tree. Although well-typed, the tree is highly

heterogeneous.

This section declares, in `bondi`, the data structures for Java bytecode for use in the rest of the chapter. Although not shown here, similar declarations can be made in other languages, either functional or object-oriented, without essential differences. The difference between languages is not in the structure declaration, but in the capability to implement manipulation of the data.

The declaration here preserves most of the bytecode format specification, except that the constant pool is eliminated and references to constant-pool entries are replaced with the actual entry values. The decision to eliminate the constant pool is not essential, but aids presentation clarity. Also note that the presentation here of the declaration is in top-down order for easy reading. However, the declaration should be in a reverse order when being compiled.

The top structure for a Java class file is the datatype `clazz`:

```
datatype clazz = Clazz of (list accessFlag) * thisClassName * superClassName
                * (list interfaceName) * (list field) * (list methd) * (list attribute);;
```

which is a straight translation of the specification (omitting the magic number and versions, which are not of our major concern), with the references to constant pool for this class, super class, super interfaces, fields and methods all replaced by datatypes representing the corresponding actual data entries.

There are, in total, 13 access flags defined in the current bytecode specification [77] for either classes, fields, methods, or both. Each of them can be represented by a constant constructor:

```
datatype accessFlag = | Public
                    | Private
                    | Static
                    | ... ...
                    | InterfaceFlag
                    | ... ... ;;
```

References to this class, super class and super interfaces are represented by their fully qualified names wrapped by corresponding constructors, while fully qualified names are constructed from simple strings such as “java/lang/Object”:

```
datatype fqname = FQname of string;;
datatype thisClassName = ThisClassName of fqname;;
datatype superClassName = SuperClassName of fqname;;
datatype interfaceName = InterfaceName of fqname;;
```

Now it is possible to declare a dummy instance of type `clazz`:

```
let dummyclazz = Clazz ([ ], ThisClassName (FQname "Dummy"),
                       SuperClassName (FQname "Object"), [ ], [ ], [ ], [ ]);;
```

The datatype for fields is declared from a list of access flags, the field name, the Java source field type, and a list of attributes:

```
datatype fieldName = FieldName of string;;
datatype fieldType = | Byte
                    | Character
                    | Integer
                    | ... ..
                    | ObjRef of fqname
                    | Array of fieldType;;

datatype field = Field of (list accessFlag) * fieldName * fieldType
                  * (list attribute);;
```

Note that, for `fieldType`, the primitive Java types are represented by constant constructors while the Java reference types (object type and array type) are unary constructors. The declaration for arrays is recursive.

The datatype for methods is similar to that of fields but has a slightly more complex datatype hierarchy, because a Java method type consists of Java source types for parameters and return value. A concrete method will also contain a `Code` attribute with a list of bytecode instructions representing the method implementation.

```

datatype methodName = MethodName of string;;
datatype paramType = ParamType of fieldType;;
datatype returnType = | Void
                    | RetType of fieldType;;
datatype methodType = MethodType of list paramType * returnType;;
datatype methd = Methd of (list accessFlag) * methodName * methodType
                    * (list attribute);;

```

Now consider a datatype for attributes. Besides the special `Code` attribute mentioned above, there are other 8 attributes for classes, fields, methods, and even for the `Code` attribute itself.

```

datatype attribute =
  | ConstValue of baseValue
  | SourceFile of string
  | Exceptions of list ffname
  | Code of maxStack * maxLocals * (list (int * instruction))
        * (list exceptHandler) * list attribute
  | ... .. ;;

datatype maxStack = MaxStack of int;;
datatype maxLocals = MaxLocals of int;;

```

The declaration for datatypes `baseValue` and `exceptHandler` are not listed here. Datatype `baseValue` represents the base values acceptable in Java bytecode: integer, long, float, double, and string. Note that, although Java bytecode supports all of the primitive types of Java source language, it does not support all values of these types. For example, values of byte, character, short integer and boolean types in Java source code are all translated into integer values in bytecode.

In the `Code` attribute declaration above, we choose to represent an instruction list using datatype `list`, and attach an instruction number to each instruction for convenient handling of branching instructions. Again, this variation from the bytecode specification is not essential. The instruction list can also be represented by any other appropriate structures.

The datatype for instructions is just an enumeration of the instruction set, with each instruction represented by an appropriately parameterized constructor. For example, the constructor for static-method-invocation instructions takes a method reference as a parameter; the constructor for integer-loading instructions takes an integer parameter representing the index of the local variable to be loaded, and so on:

```
datatype instruction =  
  | InvokeVirtual of methodRef  
  | InvokeInterface of methodRef  
  | InvokeSpecial of methodRef  
  | InvokeStatic of methodRef  
  | New of refClassName  
  | Return  
  | Ireturn  
  | Goto of int  
  | Ificmplt of int  
  | Dup  
  | Iload of int  
  | Istore of int  
  | Aload of int  
  | Astore of int  
  | Iconst of int  
  | Bipush of int  
  | Iinc of int * int  
  | ... .. ;;
```

The datatypes for method reference and class reference used above are declared as:

```
datatype refClassName = RefClassName of fqname;;  
datatype methodRef = MethodRef of refClassName * methodName * methodType;;
```

Up to this point, all the data structures declared are personal choices, and there are always alternatives. They would look very similar if declared in a standard functional language like OCaml [76]. One could also declare them as classes in an object-oriented language such as Java itself, so that bytecode data are objects. Although *bondi* is designed for experimental purpose, not for production, the expressive power of the pattern calculus has enabled basic imperative [83] and object-oriented [63] features to be incorporated into the language, in addition to functional features. Any alternatives

from other general-purpose programming languages can also be realized in `bondi`. The many ways to declare the data structure are not substantially different. The difference is in the ways to express the computation with these data structures.

5.2 Semantic Checking of Bytecode

Static checking of syntactic patterns on Java bytecode programs can enforce not only typing rules, but also any semantic rules not defined by the bytecode specification. Syntactic patterns implying (violation of) security policies can be represented as instances of appropriate pattern structures, and functions with these pattern structures as parameters can be used to verify whether the patterns appear in a target bytecode program.

The demonstration starts from the simplest case, checking for a singular pattern in a single Java class file. Consider a simple security policy: every Java method can request no more than 256 memory units for its operand stack. To verify this policy against a single Java class file, every `MaxStack` item specifying maximum operand stack size of the corresponding method must be checked. The pattern structure `boolpat` and function `checkps` from Section 3.2 are immediately applicable to expression of this computation.

Suppose the target class file is transformed as an instance `c1` of type `clazz`. The singular pattern `MaxStack` can be represented by the `Bsingle` constructor of `boolpat`. Expression of the computation is then trivial:

```
let c1 = Clazz (... ..);;
let noMoreThan256 x = x<=256;;
let ps1 = Bsingle MaxStack noMoreThan256;;
let result1 = checkps ps1 c1;;
```

The function `checkps` scans through the class file `c1` to find all `MaxStack` items and verify their values. It returns true to `result1` if the value of each and every `MaxStack` item in the Java class is less than 256, and false otherwise. It is also possible to use function `check` with a singular pattern rather than a pattern structure as parameter. But to keep consistent with the demonstration for complex patterns, the more general `checkps` is used.

The function `checkps` is reusable for different syntactic patterns, different boolean functions to verify the matching items, and different declarations of the bytecode data structure. For example, if `MaxLocals` values rather than `MaxStack` ones are to be checked, the function `checkps` can be used without any change:

```
let ps2 = Bsingle MaxLocals noMoreThan256;;
let result2 = check MaxLocals noMoreThan256 c1;;
```

That is also the case even if the syntactic pattern is complex. Consider an access control policy that disallows class `c1` from creating new instances of `java/io/File` class, but allows it to use instances created by other classes through reference. The syntactic pattern is complex: first locate `New` items representing object instantiation instructions, then search for `FQname` items that are under `New` items, leaving all other `FQname` items unchecked.

```
let notfile x = x!="java/io/File";;
let ps3 = Bconcat New (Bsingle FQname notfile);;
let result3 = checkps ps3 c1;;
```

Although `boolpat` only provides three constructor options to combine smaller patterns into more complex ones, it is able to express a wide variety of security policies. Many basic typing policies of the bytecode specification can be encoded by `boolpat`. Type checking for these policies can be done simply by function `checkps`.

Of course, there are more ways than the three constructors to combine simple search patterns. It is not hard to declare new pattern structures as needed for more sophisticated policies, and new parametric functions for these structures, to express semantic checking with arbitrarily complex syntactic patterns.

Semantic rules representing security policies can then be totally captured by appropriate pattern structures, no matter how complex they are, and be passed to checking functions as arguments. It is possible to construct pattern structures on the fly, allowing users to configure their customized security policies at runtime, without any need to change the checking programs.

Function `checkps` is highly reusable for different complex patterns and different data structures, but can only verify single Java class files. Although this is sufficient for the standard Java bytecode verification, it cannot enforce many non-trivial customized security policies, which require searching for syntactic patterns across multiple class files.

For example, a policy to disallow creation of `java/io/File` objects may also want to make sure the objects to be created do not belong to subclasses of `java/io/File`. This involves searching not only in a target class file, but also upwards along the inheritance path, starting from the class name of an object to be created.

With the need to search across multiple class files, search patterns for the security policies become complicated, and the amount of programming workload for verification is much more significant. However, no matter how complicated a search pattern is, it can always be captured by declaring an appropriate pattern structure. This is still subject to further research, but a possible strategy is that `fqname` items (fully-qualified names) under different substructures in a class file represent references to

other classes, and are the channels to reach other classes during pattern searching. Checking for a policy may require to go along some of them but not others, and this will vary from policy to policy. For example, when checking for the policy that disallows creation of new instance of class `java/io/File` and its subclasses, the verification needs to check the class references of all `New` instructions, and all ancestors of these classes, but does not need to check the class references in method calls and in exception lists. A highly parametric pattern structure, such as a (heterogeneous) list of patterns, may be used to determine, from the parameters of the structure, which of the channels should be followed, and such a structure can be passed as a parameter to the checking function.

5.3 Java Bytecode Instrumentation

Many security policies that are based on runtime values cannot be enforced by static-time verification. For example, the policy that restricts each program not to launch more than 2 threads cannot be checked statically. Target programs have to be monitored at runtime and the number of threads launching has to be counted. Except for Java's standard security manager, most runtime monitoring for security enforcement of Java bytecode is done by code instrumentation, an efficient approach that scans through target bytecode programs, locates the critical points that need monitoring based on some syntactic patterns, and inserts or replaces monitoring code into the target programs. Since monitoring code is part of the target programs being executed, the runtime overhead for program context switch is much lower than running a pure monitor separately.

Although code instrumentation and static type checking are two different categories of security mechanisms, the static part of the code instrumentation technique is similar to static type checking. They both search over target program codes for syntactic patterns, which can then be highly parameterized using pattern structures. The difference is that static type checking simply rejects target programs upon finding of the patterns of security violation, while the code instrumentation technique inserts monitoring code or replaces insecure code with secure code.

They are indeed not very different when expressed in `bondi`. Pattern searching of both techniques can be parameterized using pattern structures, and the insertion and replacement operations of the instrumentation technique can be captured by a parameter to a pattern structure, just like the function parameters in pattern structures `boolpat` and `updpat` (see Section 3.2). Actually, pattern structure `updpat` and function `updateps` are already sufficient for expressing many code instrumentation tasks.

Consider an instrumentation task to replace in class file `c1` all instantiation of class `java/io/File` with a safe version `security/safeFile`, which is the same as `java/io/File` but incorporates runtime monitoring code for certain security policies.

This can be expressed by using `updateps`:

```
let replacefile x = if x=="java/io/File"
                   then "security/safeFile"
                   else x;;

let ps6 = Uconcat New (Usingle FQname replaceFile);;
let newc1 = updateps ps4 c1;;
```

The pattern structure `updpat` captures not only the whole syntactic pattern to match the location for replacement, but also the replacement operation to apply at the matching points. This allows users to configure their security policies on the

fly, and define their preferences for insertion and replacement, without the need to change the instrumentation program. This approach not only can be used for bytecode instrumentation for security purpose, but also has the potential to be used in general program code patches and optimizations.

5.4 Summary

Implementing Java bytecode security enforcement using pattern structures is easy and simple, in contrast to the difficulty of doing so with other existing languages, especially in the presence of complex patterns to search for. By using pattern structures, programs for bytecode checking and instrumentation can be fully statically typed, allowing efficient correctness/security verification on the programs themselves by static type checking, giving full confidence to trust these security-critical programs. The programs can also be highly adaptive to different security policies and the heterogeneity of bytecode structures. Security policies can be totally captured by appropriate pattern structures, which are possible to be configured or reconfigured according to user input at runtime, and passed to enforcement programs as arguments.

Chapter 6

Conclusion and Future Work

Existing general-purpose programming languages in practice have difficulty expressing computations with heterogeneous data, because their type systems are designed for homogeneous data structures. They have problem typing the different substructures in a piece of heterogeneous data, and typing the pattern to search for within such data. The problem becomes worse expressing computation with complex search patterns, which may be arbitrary combinations of singular patterns. Given these problems, applications with heterogeneous data have been forced to be implemented by untyped programs, or by using special-purpose query languages, both of which introduce the risk of type mismatch, and incurs a heavy workload of program analysis, testing and runtime type checking in order to verify program correctness and security.

This thesis provides a general solution to these problems. It invents the concept of pattern structures, which is an abstract structure to represent complex patterns, and proposes a programming technique that uses pattern structures for universal pattern searching in heterogeneous data, in addition to using a general-purpose programming language that treats structures and patterns as first-class terms and supports a

generalized form of pattern matching.

In the proposed approach, a complex search pattern can be encoded as a well-typed instance of a pattern structure, which encapsulates all the information needed to guide the universal traversal in heterogeneous data to find all matching target elements. It is a first-class value, can be referenced by name, passed around as an argument, combined with other patterns into more complex patterns, and itself be traversed/decomposed by pattern matching.

By using well-typed pattern structures, heterogeneous data processing programs can be fully statically typed, and can be made parametric over data of arbitrary structures, and over arbitrary search patterns. These programs are type-safe, highly reusable for different structures and different patterns, and highly scalable with respect to complexity of data structures and complexity of patterns. In most other programming languages, different data structures and especially different search patterns require different programs, and programming becomes unboundedly difficult as complexity of the data structures and complexity of patterns increase.

Using pattern structures, defining a new kind of pattern is just a programming task: changing an existing pattern structure or creating a new one, and changing or creating the corresponding programs, in contrast to most other programming languages where redesign of the programming languages is necessary.

The applications of the approach to XML data processing and Java bytecode analysis demonstrate the advantages of the approach over other existing approaches. Programming using pattern structures can easily implement XML data processing with different kinds of patterns, such as XPath patterns and regular-expression patterns

which are handled by different languages in other approaches. The resulting programs are statically typed, allowing efficient static type checking for correctness and security, and are reusable for different XML schemas and different XML patterns. As well, programming using pattern structures can implement both type checking and instrumentation for Java bytecode, two security enforcement mechanisms that are usually implemented separately in different programming frameworks. The resulting programs are again statically typed, so it is easy to verify their own security, and are reusable for checking different security policies that might be configured on the fly by users.

The proposal and demonstration of the approach in this thesis are only the first steps towards a practical solution for computation with heterogeneous data. There is a long way to go to bring the approach into practical applications. There are issues remaining unsolved or open for further investigation. Also, the example programs in Chapter 4 and 5 are proof-of-concept demonstrations and have run only on hand-made example data. A lot of implementation effort is still required to make them practical.

- The usefulness of the proposed approach is limited by the difficulty of designing the functions with pattern parameters.
 - Representing complex patterns natively by pattern structures rather than by using query languages moves the workload from language implementation to the programming of the functions with pattern parameters. Given the functional style of the `bondi` programming, some patterns trivial to express in query languages may be difficult to handle in `bondi`. For example,

when designing the pattern structures for XPath in Section 4.2, if a predicate expression involved more than one location path as in the full XPath specification, the `bondi` functions to handle XPath patterns, i.e., `checkstep`, `checkpred` and `checkpath`, would be much more difficult to design. When such difficulty outweighs the reusability and type safety the functions can achieve, the approach may lose its advantages over implementing a query language.

- Although some basic imperative and object-oriented features have been incorporated into `bondi`, the claim from the Pattern Calculus papers [63, 83], that one can add imperative features and object-oriented features to `bondi` at will, is preliminary. Without standard imperative features, programming will be clumsy in many situations.
- The claim for reusability of the proposed approach relies on the possibility of creating interfaces that translate readable descriptions of complex patterns into instances of pattern structures automatically. It may be difficult for application programmers to understand pattern structures. Even if they do, real patterns in practice may still be tedious to encode. For example, a non-trivial XPath pattern encoded in `bondi` may be much harder to read and understand than its original XPath expression.
- The demonstration programs in the thesis assume that the target data are fully stored in internal memory. It is not yet clear, when the data are too large to be fit in memory, how it would affect the effectiveness of the proposed approach.
- The performance issue, although important, is not the goal of this thesis, and is

expected to be addressed on another level. Functional languages are still deemed inefficient at runtime. Given `bondi` programs are in functional style and have even higher level of abstraction and polymorphism than standard functional programs, their performance may seem an issue for some readers. However, the performance issue is not very relevant to the research here. Similar to other software engineering approaches such as the object-oriented programming technique, the proposed technique aims at improving type safety and reusability at high-level programming, relaxing human programmers' burden on error checking and repetitive programming during software development and maintenance. Performance issue is expected to be addressed at a lower level, for example through optimization at compilation level, by automated tools. Such a tool is not yet available for the proposed technique though. Currently, `bondi` is the only language can be used for the proposed technique, and only one experimental interpreter is available for it. `bondi` programs are executed by interpretation, and is indeed slow. A compiler-based language support for Pattern Calculus, with more imperative features and optimization mechanisms built-in, is desirable.

- The demonstration examples in Chapter 4 and 5 only consider subsets of the XML and Java bytecode specifications, and assume effective and efficient transformation of the original target data into `bondi` data structures. All these are subject to practical implementation.

Besides XML data processing and Java bytecode manipulation, more applications are desirable to demonstrate the usefulness of the approach. The Java bytecode manipulation example can be extended to manipulation of arbitrary programming code, which may lead to more suitable applications. For example, an aspect-oriented

programming framework [39] needs to analyze aspect-oriented programs and insert the crosscutting aspect code into programs at either compile time, deployment time, load time, or runtime [101]. This is called weaving. The approach of this thesis seems to be advantageous in implementing weaver programs.

Bibliography

- [1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.
- [2] F. H. Allen and K. J. Lipscomb. The Cambridge structural database. *Encyclopaedia of Supramolecular Chemistry*, pages 161–168, August 2004.
- [3] B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Trans. on Programming Languages and Systems*, 11(1):147–167, January 1989.
- [4] A. Appel, N. Michael, A. Stump, and R. Virga. A trustworthy proof checker. *J. of Automated Reasoning, special issue on Proof-Carrying Code*, 31(3-4), 2003.
- [5] F. Baader and W. Snyder. Unification theory. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, volume I*, pages 447–533. Elsevier Science Publishers, 2001.
- [6] F. Bancilhon and D. Maier. Multi-language object-oriented systems: New answers to old database problems. In K. Fuchi and L. Kott, editors, *Future Generation Computers II*, Amsterdam, North-Holland, 1988.

-
- [7] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, , and V. Josifovski. Streaming XPath with forward and backward axes. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 455–466, March 2003.
- [8] G. Bell, J. Gray, and A. Szalay. Petascale computational systems. *Computer*, 39(1):110–112, January 2006.
- [9] W.H. Bell, D.G. Cameron, R. Carvajal-Schiaffino, A.P. Millar, K. Stockinger, and F. Zini. Evaluation of an economy-based file replication strategy for a data grid. In *Proc. of the 3rd IEEE Int. Symposium on Cluster Computing and the Grid (CCGRID 2003)*, pages 661–668, Tokyo, Japan, May 12-15, 2003.
- [10] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. GenBank. *Nucleic Acids Research*, 34(Database Issue):D16–D20, 2006.
- [11] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML path language (XPath) 2.0 - W3C candidate recommendation, June 2006.
- [12] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in *Cw*. In A. P. Black, editor, *Proc. of 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, Glasgow, UK, July 25-29, 2005. Springer.
- [13] S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language - W3C candidate recommendation, June 8, 2006.
- [14] D. Booth and C. K. Liu. Web services description language (WSDL) version 2.0 part 0: Primer, W3C candidate recommendation, March 2006.

-
- [15] P. E. Bourne, K. J. Address, W. F. Bluhm, L. Chen, N. Deshpande, Z. Feng, W. Fleri, R. Green, J. C. Merino-Ott, W. Townsend-Merino, H. Weissig, J. Westbrook, and H. M. Berman. The distribution and query systems of the RCSB protein data bank. *Nucleic Acids Research*, 32(Database Issue):D223–D225, 2004.
- [16] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 2004.
- [17] C. Braghin, D. Gorla, and V. Sassone. A distributed calculus for role-based access control. In *Proc. of 17th IEEE Computer Security Foundations Workshop (CSFW 2004)*, pages 48–60, Pacific Grove, CA, USA, June 2004.
- [18] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (fourth edition) - W3C recommendation, August 2006.
- [19] R. R. Brooks. Mobile code paradigms and security issues. *IEEE Internet Computing*, 8(3):54–59, May/June 2004.
- [20] K. B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, Cambridge, Massachusetts, London, England, 2002.
- [21] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, FL, USA, 1997.

-
- [22] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. *Lecture Notes in Computer Science*, 1997:1–25, 2001.
- [23] A. Chander, J. C. Mitchell, and I. Shin. Mobile code security by java byte-code instrumentation. In *Proc. of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, Anaheim, CA, USA, June 2001.
- [24] N. Chase. XML and web services reference guide. *InformIT.com*, Last updated: July 2006. www.informit.com/guides/guide.asp?g=xml&rl=1.
- [25] S. Chong, A. C. Myers, K. Vikram, and L. Zheng. Jif reference manual, June 2006. www.cs.cornell.edu/jif/doc/jif-3.0.0/manual.html.
- [26] J. Clark. XSL transformation(XSLT): Version 1.0 - W3C recommendation, November 1999.
- [27] J. Clark and S. DeRose. XML path language (XPath): Version 1.0 - W3C recommendation, November 1999.
- [28] S. Cluet and J. Siméon. YATL: a functional and declarative language for XML. draft manuscript, March 2000. www-db.research.bell-labs.com/user/simeon/icfp.ps.
- [29] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. of 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66. ACM Press, 2000.

- [30] N. Daswani, H. Garcia-Molina, and B. Yang. Open problems in data-sharing peer-to-peer systems. In *Proc. of the 9th International Conference on Database Theory (ICDT 2003)*, pages 1–15, Siena, Italy, January 2003.
- [31] Daylight Chemical Information Systems, Inc. SMARTS - a language for describing molecular patterns, 2006. www.daylight.com/dayhtml/doc/theory/theory.smarts.html.
- [32] D. Detlefs. An overview of the extended static checking system. In *Proc. of the 1st Formal Methods in Software Practice Workshop*, 1996.
- [33] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. *Computer Networks*, 31(11–16):1155–1169, 1999.
- [34] G. Dowek. Higher-order unification and matching. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, volume I*, pages 1009–1062. Elsevier Science Publishers, 2001.
- [35] B. DuCharme. Amazon’s web services and XSLT. *XML.com*, August 04, 2004. O’Reilly Media, Inc.
- [36] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Ithaca, NY, USA, January 2004.
- [37] R. Fagin, P. G. Kolaitis, R. Kumar, J. Novak, D. Sivakumar, and A. Tomkins. Efficient implementation of large-scale multi-structural databases. In *Proc. of the 31st International Conference on Very Large Data Bases (VLDB 2005)*, pages 958–969, Trondheim, Norway, August 30 -September 2 2005. ACM.

- [38] D.C. Fallside and P. Walmsley. XML schema part 0: Primer, second edition - W3C recommendation, October 2004.
- [39] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley Professional, October 2004.
- [40] N. Francesco and G. Lettieri. Checking security properties by model checking. *Software Testing, Verification and Reliability*, 12(3):181–196, September 2003.
- [41] M. Franz, D. Chandra, A. Gal, V. Haldar, C. W. Probst, F. Reig, and N. Wang. A portable virtual machine target for proof-carrying code. *Science of Computer Programming*, 57(3):275–294, September 2005.
- [42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Visitor. In *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 331–344. Addison-Wesley Professional, January 15, 1995.
- [43] J. Gibbons. Design patterns as higher-order datatype-generic programs. In *ACM SIGPLAN Workshop on Generic Programming 2006*, Portland, Oregon, USA, September 2006. ACM Press.
- [44] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *Proc. of the 16th Int. Conf. in Pattern Recognition (ICPR 2002)*, Quebec, Canada, August 2002.
- [45] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 3rd Edition*. Addison-Wesley, 2005. java.sun.com/docs/books/jls/.

- [46] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, 2002.
- [47] R. S. Gray, G. Cybenko, D. Kotz, and D. Rus. Mobile agents: Motivations and state of the art. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2002.
- [48] J. R. Groff and P. N. Weinberg. *SQL: The Complete Reference, Second Edition*. Mcgraw-Hill / Osborne Media, Berkeley, CA, USA, August 2002.
- [49] C. Grothoff. Walkabout revisited: The Runabout. In *Proc. of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, pages 103–125. Springer-Verlag, July 21-25, 2003.
- [50] M. Gudgin, M. Hadley, and T. Rogers. Web services addressing 1.0 - core, W3C recommendation, May 2006.
- [51] M. Harren, M. Raghavachari, O. Shmueli, M. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: Facilitating XML processing in java. In *Proc. of the 14th International World Wide Web Conference*, pages 278–287, Chiba, Japan, May 2005. ACM Press.
- [52] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2), 2004.
- [53] H. Hosoya and B.C. Pierce. XDuce: A typed XML processing language. *ACM Trans. on Internet Technology*, 3(2):117–148, 2003.

- [54] K. Hristova, T. Rothamel, Y. A. Liu, and S. D. Stoller. Efficient type inference for secure information flow. In *Proc. of the 2006 workshop on Programming languages and analysis for security (PLAS'06)*, pages 85–94, New York, NY, USA, 2006. ACM Press.
- [55] F. Y. Huang, C. B. Jay, and D. B. Skillicorn. Programming with heterogeneous structure: Manipulating XML data using **bondi**. Technical Report 2005-494, School of Computing, Queen's University, March 2005. www.cs.queensu.ca/TechReports/Reports/2005-494.pdf.
- [56] F. Y. Huang, C. B. Jay, and D. B. Skillicorn. Adaptiveness in well-typed Java bytecode verification. In *Proc. of the 16th Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON 2006)*, pages 248–262, Markham, Ontario, Canada, October 2006. IBM Press.
- [57] F. Y. Huang, C. B. Jay, and D. B. Skillicorn. Programming with heterogeneous structures: Manipulating XML data using **bondi**. In V. Estivill-Castro and G. Dobbie, editors, *Proc. of 29th Australasian Computer Science Conf. (ACSC 2006)*, pages 287–295, Hobart, Tasmania, Australia, January 2006. ACS.
- [58] F. Y. Huang and D. B. Skillicorn. The spider model of agents. In S. Pierre and R. Glitho, editors, *Proc. of the 3rd Int. Workshop on Mobile Agents for Telecommunication Applications (MATA 2001)*, pages 209–218, Montreal, Canada, August 2001. Springer-Verlag.
- [59] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

-
- [60] Sun Microsystems Inc. Java 2 platform standard edition 5.0 API specification, 2004. java.sun.com/j2se/1.5.0/docs/api/index.html.
- [61] Sun Microsystems Inc. Java 2 platform standard edition 5.0 documentation, 2004. java.sun.com/j2se/1.5.0/docs/index.html.
- [62] C. B. Jay. Higher-order patterns. unpublished, 2004. www-staff.it.uts.edu.au/~cbj/Publications/higher_order_patterns.pdf.
- [63] C. B. Jay. Methods as pattern-matching functions. In *Informal Proc. 11th Int. Workshop on Foundations of Object-Oriented Languages*, Venice, Italy, January 17, 2004. doc.ic.ac.uk/scd/F00L11/patterns.pdf.
- [64] C. B. Jay and D. Kesner. Pure pattern calculus. In Peter Sestoft, editor, *Proc. of the 15th European Symposium on Programming (ESOP 2006)*, Vienna, Austria, March 27-28 2006. Springer.
- [65] C.B. Jay. bondi webpage, 2004. www-staff.it.uts.edu.au/~cbj/bondi.
- [66] C.B. Jay. The pattern calculus. *ACM Trans. Program. Lang. Syst.*, 26(6):911–937, 2004.
- [67] T. P. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, pages 89–103, 1999.
- [68] Web services choreography description language version 1.0 – W3C candidate recommendation, November 2005.

- [69] G. Klyne and J. Carroll. Resource description framework (RDF): Concepts and abstract syntax, W3C recommendation, February 2004.
- [70] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.
- [71] J. Kopp and T. Schwede. The SWISS-MODEL repository: New features and functionalities. *Nucleic Acids Research*, 34(Database Issue):D315–D318, January 2006.
- [72] R. Lämmel. Scrap your boilerplate with xpath-like combinators. In *Proc. of POPL'07*. ACM Press, January 2007.
- [73] R. Lämmel and S. Peyton-Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).
- [74] R. Lämmel and S. Peyton-Jones. Scrap your boilerplate with class: Extensible generic functions. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, September 2005.
- [75] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 2003.
- [76] X. Leroy. The Objective Caml system release 3.09 - documentation and user's manual, October 2005. caml.inria.fr/pub/docs/manual-ocaml/index.html.

- [77] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, 2nd Edition*. Addison-Wesley, 1999. java.sun.com/docs/books/vmspec/.
- [78] E. Meijer, W. Schulte, and G. Bierman. Unifying tables, objects and documents. In *Proc. of DP-COOL 2003*, Uppsala, Sweden, August 2003.
- [79] B. Meyer and K. Arnout. Componentization: The visitor example. *Computer*, 39(7):23–30, July 2006.
- [80] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 365–458. MIT Press, Cambridge, MA, USA, 1991.
- [81] N. Mitra. SOAP version 1.2 part 0: Primer, W3C recommendation, June 2003.
- [82] C. Murdaca and C. B. Jay. A relational account of objects. In V. Estivill-Castro and G. Dobbie, editors, *Proc. of 29th Australasian Computer Science Conference (ACSC 2006)*, pages 297–306, Hobart, Tasmania, Australia, January 2006. ACS.
- [83] Q. T. Nguyen, C. B. Jay, and H.Y. Lu. The polymorphic imperative: a generic approach to in-place update. In *Proc. 10th Computing: The Australasian Theory Symposium*, ENTCS, Dunedin, New Zealand, January 19, 2004. doc.ic.ac.uk/scd/F00L11/patterns.pdf.
- [84] Object Management Group, Inc. CORBA/IIOP specification, v3.0.3, March 2004.
- [85] C. Ogbuji. Versa: Path-based RDF query language. *XML.com*, July 20, 2005. O’Reilly Media, Inc.

- [86] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. of Workshop on XML-Based Data Management at EDBT 2002*, Prague, Czech Republic, March 2002.
- [87] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proc. of the 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC'98*, pages 9–15, Vienna, Austria, 1998.
- [88] R. Pandey and B. Hashii. Providing fine-grained access control for Java programs via binary editing. *Concurrency: Practice and Experience*, 12(14):1405–1430, 2000.
- [89] S. Peyton-Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, 2003.
- [90] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Trans. Program. Lang. Syst.*, 27(2):344–382, March 2005.
- [91] J. Robie, J. Lapp, and D. Schach. XML query language (XQL), December 1998.
- [92] A. Rudys and D. Wallach. Enforcing Java run-time properties using bytecode rewriting. In *Proc. of the International Symposium on Software Security*, Tokyo, Japan, November 2002.
- [93] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. on Selected Areas in Communications*, 21(3), January 2003.
- [94] L. Salwinski, C.S. Miller, A.J. Smith, F.K. Pettit, J.U. Bowie, and D. Eisenberg. The database of interacting proteins: 2004 update. *Nucleic Acids Research*, 32(Database Issue):D449–D451, January 2004.

- [95] R. R. Schneck and G. C. Necula. A gradual approach to a more trustworthy, yet scalable, Proof-Carrying Code. In *Proc. of the 18th Conference on Automated Deduction (CADE'02)*, pages 47–62, Copenhagen, Denmark, July 2002.
- [96] F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.
- [97] A. Seaborne. RDQL - a query language for RDF, W3C member submission, February 2004.
- [98] N. Shadbolt, T. Berners-Lee, and W. Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21(3):96–101, May/June 2006.
- [99] D. B. Skillicorn. The case for datacentric grids. In *Proc. of the 2nd Workshop on Massively Parallel Processing, IPDPS 2002*, Fort Lauderdale, FL, USA, April 15-19, 2002.
- [100] B. Stroustrup. *The C++ Programming Language (Special Edition)*. Addison-Wesley, MA, USA, 2000.
- [101] The AspectJ Team. The AspectJ development environment guide, 1998-2005. www.eclipse.org/aspectj/doc/released/devguide/index.html.
- [102] V. Benzaken and G. Castagna and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proc. of 2003 ACM SIGPLAN Int. Conf. on Functional Programming*, pages 51–63. ACM Press, 2003.
- [103] V. Gapeyev, M.Y. Levin, B.C. Pierce, and A. Schmitt. XML goes native: Runtime representations for Xtatic. Technical Report MS-CIS-04-23, University of Pennsylvania, October 2004.

-
- [104] P. Wadler. Links, January 2004. homepages.inf.ed.ac.uk/wadler/papers/links/links-blurb.pdf.
- [105] T. Watanabe, K. Yamada, and N. Nagatou. Towards a specification scheme for context-aware security policies for networked appliances. In *Proc. of the IEEE Workshop on Software Technologies for Future Embedded Systems (WSTFES '03)*, pages 65–68, Washington D.C., USA, May 2003. IEEE Computer Society.
- [106] N. Whitehead, M. Abadi, and G. Necula. By reason and authority: A system for authorization of Proof-Carrying Code. In *Proc. of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 236–250, Pacific Grove, CA, USA, June 2004.
- [107] L. Zheng and A. C. Myers. End-to-end availability policies and noninterference. In *Proc. of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 272–286, Aix-en-Provence, France, June 2005.