University of Strathclyde

Department of Computer and Information Sciences

Type Projections over Self-Describing Data

by

Fabio Simeoni

A thesis presented in fulfilment of the requirements for the degree of

Doctor of Philosophy

May 18, 2011

To Cathie and Sam, for the wait.

Abstract

We present a model for binding statically typed programs and self-describing data based on dynamic projections of type assertions, or *type projections*. The model exploits the potential for loose coupling which inheres in self-description in order to endorse a principle of *typeful programming* over shared data: the type of the data should vary in accordance with the processing requirements of individual programs.

Loose coupling is key between programs that execute in cooperating but autonomous runtimes. Accordingly, type projections may be employed at the boundary of typed languages to regulate the exchange of untyped data across the nodes of distributed systems. *External type projections* give rise to a class of *data binding tools* for mainstream languages and standard self-describing formats, such as XML.

Loose coupling is also desirable between programs that execute within a single runtime, where conventional typechecking regimes may unduly constrain the interpretations of shared data. Type projections may then be employed *within* typed languages to regulate the exchange of typed data across local application components. *Internal type projections* give rise to a class of *metamorphic languages*, where an unconventional ‘data-first’ approach to dynamic typechecking extends support for typeful programming beyond the scope of polymorphism and coercions.

Preface

This Dissertation is submitted 'by portfolio' which, according to Part 3 of the University of Strathclyde's Calendar 2009-2010, "*may consist in whole or in part of work previously published by the candidate, provided the thesis is so composed as to present a connected record of research in a field of study.*"

The Dissertation is structured accordingly in two Parts. Part I contains a presentation of type projections as a general-purpose model of type-safe bindings to self-describing data, and it is previously unpublished. It contains the most general and comprehensive account of the concepts, motivations, and applications that are associated with type projections. This includes an analysis on the role of types in programming languages which culminates with the identification of the principle of typeful programming, the motivation that unifies all the applications of type projections. It also includes a principled account of self-describing data and its implications for data processing which, to the best of my knowledge, is the first systematic attempt to analyse self-description within the formal literature.

Part II contains a selection of three publications on type projections. The selection summarises the main findings of a collaborative research investigation that spanned the years 2000-2004. It illustrates the evolution of the themes that are presented in a conclusive form in Part I, and it explores in more depth the design space defined by type projections. The selection includes:

- SIMEONI, F., MANGHI, P., LIEVENS, D., CONNOR, R. C. H., AND NEELY, S. An approach to high-level language bindings to xml. *Information & Software Technology 44*, 4 (2002), 217–228.
- SIMEONI, F., LIEVENS, D., CONNOR, R. C. H., AND MANGHI, P. Language bindings to xml. *IEEE Internet Computing 7*, 1 (2003), 19–27.
- MANGHI, P., SIMEONI, F., LIEVENS, D., AND CONNOR, R. C. H. Hybrid applications over xml: integrating the procedural and declarative approaches. In *Fourth ACM CIKM International Workshop on Web Information and Data Management (WIDM 2002)*, pp. 9–14.

The contributions of individual publications to the theme of the Dissertation are discussed in Section 1.4. For convenience, they are also repeated at the beginning of the corresponding Chapters.

Acknowledgements

The research presented in this Dissertation carries the contribution of many people. Working in a group, ideas are circulated and developed as a communal effort, and it is impossible to know from where they have originated. I would then like to thank Richard Connor, David Lievens, Paolo Manghi and Steve Neely for the part that each played in the development of the concepts presented here.

I own a particular debt of gratitude to Richard Connor, for his patient supervision and for his key role in my own development as a researcher. This work is deeply rooted in his early intuitions on typed approaches to semistructured data. I am also very grateful to David Lievens, who more than anyone else has shared with me views on type projections and self-describing data. The connection between internal type projections and coercions is his own. I would also like to thank all those who have read and provided useful comments and advice on this Dissertation.

Finally, my gratitude goes to my family for their understanding, patience, and love. Thank you for putting up with this.

Statement

I certify that I am wholly responsible for the work carried out in this Dissertation, that I am the main author of the publications which are contained therein, and that I have led the research investigation which is reported in those publications. More specifically, I am responsible for:

- the conceptualisation and articulation of the motivations for *extraction mechanisms* based on type projections, as well as the formalisation of their definition reported in Chapter 4;
- the conceptualisation and articulation of the motivations for *language bindings* to XML based on type projections, as well as the implementation of the prototype binding for Java reported in Chapter 5;
- the conceptualisation and articulation of the motivations for *metamorphic languages* based on type-projections, as well as the design of the language for hybrid applications over XML reported in Chapter 6;
- the evolution of the conceptualisation of type projections as a binding model for type-safe programming over self-describing data, from its original inception as a mechanism to extract regular subsets of semistructured data, to its broader role as an endorser of *typeful programming* over self-described data which is shared across or within language runtimes (cf. Chapter 1, Chapter 2, and Chapter 3).

Contents

List of Figures

Part I

Concepts, Motivations, and Applications

Chapter 1

Type Projections

We present a type-safe model for binding statically typed programs to data whose type is statically unknown. As such, the model extends the theory and practice of dynamic typechecking within statically typed languages [1], most noticeably those that deal with persistent or remotely defined data (e.g. [2, 3, 4]).

The novel assumption is that the data may be completely untyped at the point of binding. In fact, it may not be a language value at all. It may originate from outside the language and live in a file system, a database, or a network stream. If the data *is* a language value, its type is simply discarded for typechecking purposes.

What we require from the data is for it to be structured and sufficiently *self-describing* to enable runtime verification of the type that the program asserts for its input. Precisely, the data must be isomorphic to a rooted and directed graph in which edges are labelled with a description of target vertices. Collectively, we think of labels as *metadata* and say that the data contains its own metadata.

Dynamic typechecking then takes the form of a direct match between the asserted type and the metadata embedded in the data. Informally, the match is successful if the metadata indicates that (part of) the data can be interpreted as the self-describing representation of some value of the asserted type. By materialisation or through a view, the data can be manipulated thereafter as the value, i.e. with the algebra of the asserted type and under the full force of a static typechecking regime.

Overall, the model capitalises on self-description to identify interpretations of the data which remain latent within its structure, and to uphold them in the runtime of the language. Types convey the required interpretations but no single type must govern the entire lifetime of the data. Rather, different types may be bound to it on demand and under a notion of dynamic typechecking which is unconventionally *data-first*. We explain these bindings as *type projections* and describe the binding model as one based on *type projections over self-describing data*.

1.1 A First Example

Conceptually, the model is straightforward and may be immediately illustrated with an example. The pseudo code below defines a procedure `print` which displays contact data for a set of employees. For simplicity of context, we assume a block-structured and lexically scoped procedural language, where blocks are associated with levels of indentation in the code.

```
print(x:T)
 for e in x
  show(e.name)
  show(e.salary)
  for n in e.numbers
   show(n)
```

`print` uses the abstractions of the type system to assert that its input has no more and no less than the properties it needs to output. Assuming that the type system includes a small number of scalar types and purely structural abstractions for record and set values, `print` asserts the following type:

```
T:=set(record[name:txt,salary:num,numbers:set(txt)])
```

and annotates accordingly its formal parameter `x`. Essentially, `T` describes personnel data as a set of records with a string field for the name of employees, a numeric field for their salary, and a set-valued field for their phone numbers, all of which are strings.

At a later time, `print` is bound to some data d about employees. For the time being, it is immaterial whether d originates from outside the language or whether it is already represented in the runtime as one of its values. What does matter is that d may be mechanically processed as the labelled graph[1] in Figure 1.1.

At the point of binding, `T` is projected over d in an attempt to establish the former as a valid interpretation of the latter. Again, we abstract away the precise form and time in which projection occurs, as these depend on whether d is internal or external to the runtime, as well as on how projections interface with the language (e.g. integrated in its design or supported by a library).

The projection is successful if there is a subset d' of d which can be interpreted as the self-describing representation of a value v of `T`. Ultimately, this depends on the definition of precise projection rules for the target type system, and while we expect different rules for different languages, we do not exclude that multiple sets of rules may be available for a single language.

For this example we dispense with formal rules and rely on the intuition to identify d' as the 'subtree' of d whose edges are depicted in bold in Figure 1.1 (see Chapter 4 for the corresponding formal rules). The intuition is the following: the interpretation of record values is based on the names of their fields (e.g. `name`); the interpretation of collection values requires zero or more repeated labels (e.g. `employee`), possibly the field names of enclosing records (e.g. `number`); the interpretation of atomic values is based on the literals on some manifest syntax (e.g. `20000` for integers) or on standard

[1]For readability, we draw the graph in a quasi-radial style and mark its root node with a label **R**.

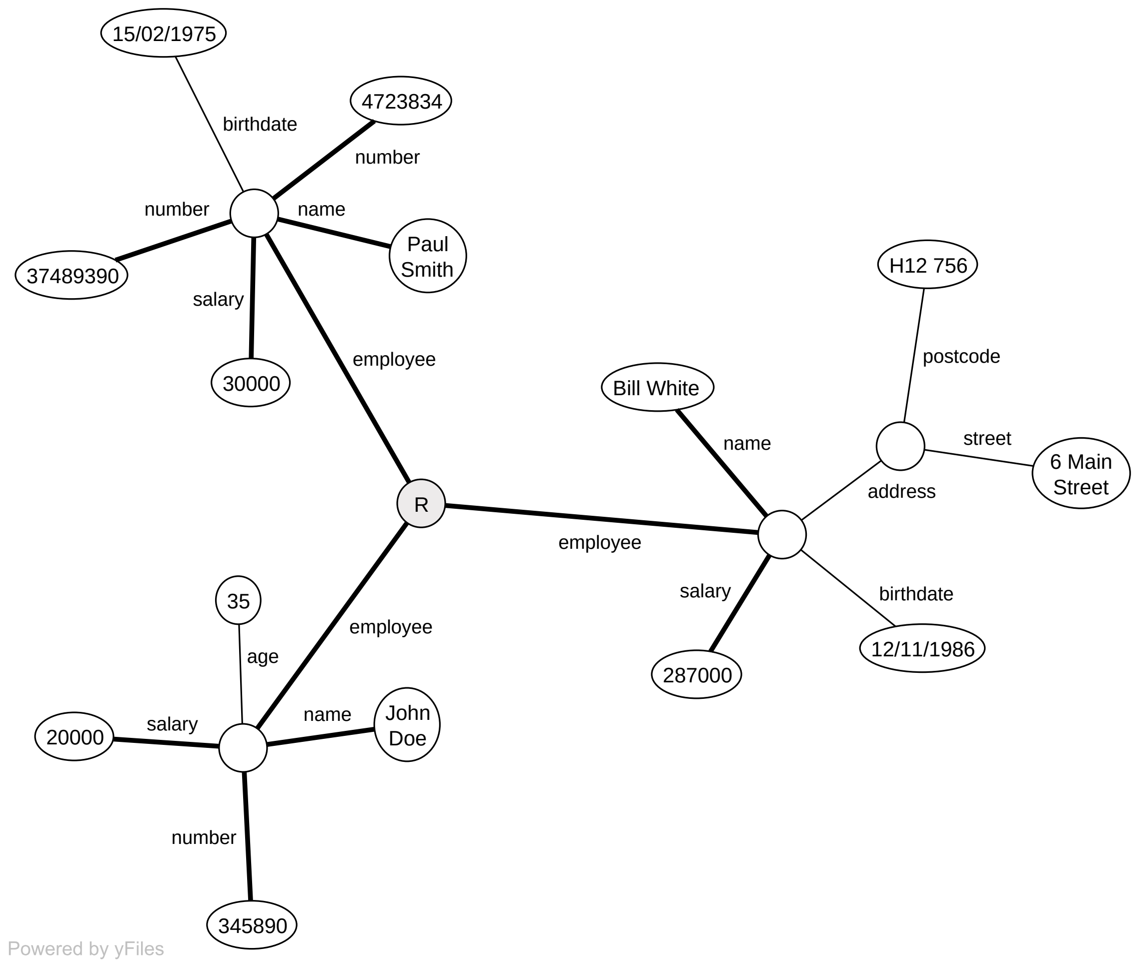

Figure 1.1: A simple example of type projection

coercions. Depending on the context of application, d' may be materialised into, or else viewed as, its typed counterpart v before being bound to `x` during the execution of `print`.

1.2 Motivations

Mechanics aside, the outcome of the projection above is no more and no less than the following: `print` can process d under a type that fully aligns with its requirements, and not another. It is because of this alignment that `print` can be so easily authored, understood, and maintained; that the typechecker can give some confidence in its correctness; that the runtime can optimally meet its memory requirements. As we discuss at length in Chapter 2, these are the implications of a fundamental programming principle, namely that the non-functional properties of a program are strongly related to the type that models its input. No type is intrinsically better than others, but some types may serve some programs better. This is the principle of *typeful programming*, and the ultimate motivation for type projections is to endorse it when conventional typechecking strategies fail to.

In a conventional strategy, the safety of bindings descends from type inferences: `print` can process d only if `T` can be inferred from the type that describes d at the point of binding. The inference raises no requirements for self-description and, in principle, may be carried out before `print` executes, i.e. statically. There are cases, however, when relying on type inference is either impossible or undesirable.

The obvious case is when d is created outside the jurisdiction of the type system, i.e. it is 'external' to the language. External data enters the language runtime as the output of parsers, i.e. under types that describe the generic structures of its representation format and independently from the requirements of consuming applications. If the programmer does not take responsibility for remodelling the data, the advantages of typeful programming are lost. In this sense, conventional typechecking strategies operate in 'closed' worlds and do not serve well application requirements that transcend the support of a single language or programming system. Extended with type projections, however, a typechecking strategy can capitalise on global standards for self-describing data and preserve typeful programming at the boundaries of individual systems. We discuss *external projections* and 'open' application domains in Section 1.2.1.

The less obvious case is when d *is* a language value but the programmer deems its type inadequate to judge the validity of `T`. We know of no conventional type system, for example, where `T` could be inferred as a type for d. A tree type or a finite union type may be plausible creation types for d, not least because they abstract away differences across employees. In doing so, however, they also hide their commonalities, including that they all have the properties predicated by `T`. Typeful programming may require multiple interpretations of the same data, yet not all them may be related by type inference. Like type casts and coercions, type projections can extend conventional strategies to preempt false negatives from the typechecker and adjust the evidence available for its next inferences. Unlike type casts, projections validate type assertions against the self-describing data rather than its creation type. Unlike coercions, projections are not limited to scalars and may preserve identity. The failure of a projection preserves safety when expectations prove incorrect. Its success marks a true 'metamorphosis' of the data in support of typeful programming. We discuss *internal projections* and metamorphoses in Section 1.2.2.

1.2.1 External Projections

Type projections befit application systems that require a controlled alternation of typed and untyped regimes over richly structured data. Somewhere and sometimes – such as when typechecking the procedure `print` above – the data may be associated with types which guide, optimise, and constrain its manipulation. In other places and points in time, type associations become less convenient and the same data is more easily managed in untyped forms.

These requirements are increasingly common. Typed regimes are enforced by mainstream languages and applications systems, sometimes orthogonally to the longevity and location of the data. Database systems specialise in the handling of long-lived data (e.g. [5]); programming languages host transient data and export typed values onto the file system and over the network through proprietary serialisation facilities (e.g. [6, 7]); some experimental languages go one step further and extend the jurisdiction of the type system over a persistent store (e.g. [4]); some middleware systems bridge the run-time of heterogeneous languages under a single type system and the transparencies of remote procedure calls [8, 9]. Upon crossing system boundaries, however, the data loses its type and is encoded in file or network formats. As an external resource, it may be manipulated with a variety of tools, undergo further transformations and relocations, and finally acquire the same or a new type upon entering a different system.

Data escapes the control of a type system because of some source of complexity within the solution domain. Complexity may lie in data that is too irregular and unstable to be conveniently typed. This is the domain of *semistructured* data [10, 11], an important theme within this thesis as well as one of its original motivations. We discuss semistructured data at length in Chapter 3.

Complexity may also arise from the architecture of application systems. In *loosely-coupled* distributed systems, for example, individual components retain a significant degree of local autonomy and differ substantially in tools, means, and purpose. Heterogeneity suggests a lightweight approach to interoperability which minimises synchronisation requirements. The implications are non-trivial and the impact potentially endemic within the system. Infrastructures need to be widely deployed (e.g. the Web); protocols and formats need to be standards with pairwise compatible and open source implementations (cf. Web Services); interfaces need to be uniform and generic (cf. SOA architectures and REST architectures). In contrast, exchanging typed data creates tight coupling within the system because it reveals stronger assumptions on *how* the data is consumed. Partly, this raises additional requirements on how the data should be mechanically accessed, i.e. with which technologies. More significantly, it curtails the ways in which the data may be interpreted: more or less abstractly; partially or in its entirety; in association with an algebra rather than another; in the form in which it becomes available or after a number of arbitrary transformations. Exchanging data is a necessary step for interoperability while exchanging types is not.

Moving beyond the constraints of typed regimes may appear liberating, but it risks losing the advantages of the technologies that enforce them, including familiar programming paradigms, complete computational models, rich platforms, and extensive tool support. Type projections can help avoiding these costs in an opportunistic manner. They can be employed at system boundaries, typically at the end of a pipeline of arbitrary data transformations, in order to interpret and process the data with the models and the technologies that are most convenient for local purposes. Projecting partial models of the data, for example, allows bindings to regular subsets of data which is otherwise semistructured; this effectively regains for parts of the data and for specific programs the advantages that could not be extended to the entire data and across all programs. Partial projections also insulate programs from changes that occur outside the bound subsets; this increases their resilience to the evolution of the data.

As to self-describing data, this is already commonplace in the domains for which we advocate type projections. Self-description is preferred for the representation of semistructured data because it preserves structure without constraining it into regular forms. It is also preferred for the loosely-coupled exchange of data because it identifies structure without preempting its interpretation. There is in fact a global standard to which requirements for self-description have rallied for the past ten years, the *eXtensible Markup Language* [12]. XML gives scope to the deployment of type projections as well as a point of reference for their definition and implementation.

1.2.2 Internal Projections

If type projections give local control over the interpretation of the data, then it is natural to question whether this control may only be exercised at the language boundary; whether a controlled alternation of typed and untyped regimes is only beneficial across the local components of distributed application systems; whether principles of loose-coupling and open-world assumptions may only be upheld for programs that execute in different runtimes.

These observations motivate a more radical application of our binding model, one in which type projections promote typeful programming *within* the context of a single language runtime. In such a language, data would retain its identity but change structure and behaviour as it percolates through the call stack of a local application.

To a degree, such changes are already supported in mainstream languages. Parametric and inclusion polymorphism do allow different interpretations of the same data under schemes of static type inference. The assumption of static typing, however, limits the possibilities: interpretations may only progressively abstract over structure and behaviour. More qualitative and dynamic changes are also common, such as those induced by coercions. These however apply mostly to scalars and thus induce copies of immutable values.

Type projections resort to dynamic typechecking to enable a wider range of interpretations, hence a wide range of structural and behavioural changes. Under a sufficiently eclectic type system, the data may be equally manipulated as, e.g., a graph, a collection of records, an object encapsulated behind a range of different interfaces, a heterogeneous array, or a dictionary. Idioms and viewpoints that are normally treated as mutually exclusive may be unified in the context of a single program. As a graph, a semistructured dataset may be subjected to declarative querying, rule-based transformations, or recursive traversal; as a dictionary or heterogeneous array, it may retain extensibility while supporting efficient lookup; as a sequence of records, a regular subset of it could be iterated over under a static typechecking regime prior to being ingested in a relational database.

We refer to such changes as *metamorphoses* and identify *metamorphism* as the generalisation of polymorphism and coercion which is obtained by lifting the assumption that the changes are statically typechecked, or that they are restricted to scalars, or that they do not preserve identity. We posit that metamorphoses may play a significant role in modern programming, where data is manipulated under an increasing range of viewpoints and yet within scopes which are increasingly smaller (e.g framework programming [13]).

1.3 Applications

As with all binding models, applications of type projections have a number of contextual dependencies. Resolving these dependencies means making choices that shape the design of an application and change how the design may be formalised for proofs of correctness and completeness. Thus type projections characterise in principle a number of concrete binding models, each of which reflects a coordinated set of choices about its dependencies.

The host language is the first dependency. What classes of interpretations can be described by its type system and instantiated in its value space? How do programmers denote type projections? How does the typechecker deal with them? How are their failures handled? The data-first approach to dynamic typechecking introduces its own dependencies. The format of the self-describing data becomes an element of application design rather than a detail of its implementation. How is self-describing data represented and, vice versa, what data can be represented in self-describing form? Most importantly, what rules define the self-describing representation of language values, i.e. determine the outcome of a type projection?

Arguably, the question of most consequence concerns the relationship between the host language and the self-describing data. In line with the divide between the deployment scenarios discussed in the previous Section, applications that focus solely on external projections raise different design issues from those that instead embrace internal projections.

1.3.1 Data Binding Tools

Applications in the first class serve as *data binding* tools for existing languages and self-describing formats. Today tools of this kind enjoy wide application, particularly in the arena of XML processing [14, 15]. They exist to dispense programmers from the tedious and error-prone task of modelling XML data in terms of language values and, vice versa, to convert the generated model back to its self-describing representation. Differently from parsers, they do not hard-code the model as an implication of the target format (e.g. the data is a tree or the data is a stream of parsing events [16, 17]). Rather, they can be instructed to generate models that align with application-specific requirements. Thus data binding tools are motivated by the principle of typeful programming, and the arguments we set forth in this Dissertation generalise and put on firmer ground those that populate the literature and other informal fora.

Yet most tools do not follow the principle to its logical conclusion and, ultimately, curtail the potential of self-describing data. They are engineered under the assumption that the same interpretation of the data may be prescribed to all its consumers. This interpretation is no longer dictated by its representation format, as with parsers, it is now dictated by its producers and serves as a shared type. Types still convey the intended interpretation of the data but do not originate within the host language. Following a well-known scheme in distributed computing [9], they are defined in language-neutral type systems (e.g. [18]) and are then translated into language types, e.g. as object types which encapsulate the binding logic as well as the bound data. The generative approach offers transparencies but suffers from problems of integration with local type systems and application design. More problematically, it is indicative of a closed-world approach to system design which takes control away from consumers and does not concede that their requirements may diverge. For example, consumers cannot bind only to the parts of the data which are relevant to their requirements. As a result, they must regenerate types even when the irrelevant parts change, and they cannot conveniently bind to regular subsets of semistructured data.

With type projections, data binding tools may follow a different approach. By projecting rather than generating types, a tool based on type projections does not suffer from mismatches across type systems and it does not prove intrusive with respect to application design. Most importantly, it gives control back to consumers and lets them project the type that best aligns with their requirements, including types that interpret the data partially. In Chapter 4 we define a framework for the formalisation of type projections which is appropriate for their use in binding tools. Based on the framework, we discuss in Chapter 5 the design of a prototype tool for binding Java and XML.

1.3.2 Metamorphic Languages

The second class of applications focuses on internal projections and includes *metamorphic languages*. Self-description enters the value space of the host language and type projections are expected to preserve identity, at least insofar as identity can be observed in the programming model. This changes their runtime semantics. The 'extraction' semantics naturally associated with binding tools and external data is less desirable, and projections can define *type views* over language values.

This raises non-trivial challenges for language design, most noticeably in relation to the soundness of typechecking in languages with mutable values. When multiple views are concurrently defined over the same data, updates made under one view may invalidate the soundness of other views. Adding and removing edges, for example, are legal changes under a mutable tree view, but they may void the static guarantees associated with a record view defined in a different scope over the same data.

The problems that arise when integrating static typechecking, type views, and mutability are well-known in languages with subtype polymorphism [19, 20, 21]. In a metamorphic language the problems generalise and so must solutions, whether these resort to dynamic typechecks, control mutability, limit projections, or combine a number of these approaches [22, 23]. An alternative approach is to introduce metamorphoses in a declarative language where, in the lack of update, concurrent interpretations of the data can never invalidate each other. Even renouncing the assumption that type projections preserve identity, i.e. induce true metamorphoses, can still deliver the benefits of typeful programming to a subset of programs (e.g. those that consume but do not update the data).

Overall, the design space for languages that accommodate internal projections is rich and virtually uncharted. In Chapter 6, we discuss a point in such space where metamorphoses coexist with updates under a simple confinement scheme. In particular, we outline the design of an imperative language with structural typing in which data changes upon crossing either one of two types of scopes. In *declarative scopes*, the data is a labelled tree and can be queried with an algebra of path expressions. In *procedural scopes*, the data is instead a recursive composition of mutable records and sets and can be manipulated with corresponding algebras. Since query evaluation semantics is standard, i.e. constructs new values, a type view that is defined in a procedural scope may only concur with a tree view defined in a declarative scope, and here the data is immutable. We then show that under this simple metamorphic scheme typeful programming supports *hybrid applications* over XML, integrating within a quasi-static language the different strengths of models and algebras that are traditionally associated with competing XML technologies.

1.4 Outline

This Dissertation is structured in two parts. The first part presents type projections as a general-purpose model of type-safe bindings. In this first Chapter we have introduced concepts, motivations and applications. In the next two Chapters we elaborate further on the key themes of typeful programming and self-describing data. In the process, we make the presentation self-contained and resolve pointers to related work. In particular:

- Chapter 2 overviews the rich body of notions that are associated with the use of types in programming languages, moving from the classic viewpoint of program correctness to a broader, model-oriented perspective that leads to the principle of typeful programming.

- Chapter 3 discusses the implications of labelled structures for data processing, focusing on the role of self-description as an enabler of typeful programming.

The second part of the Dissertation comprises a selection of three published works on type projections. The ideas set forth in this Chapter appear there still in an embryonic form but their implications are investigated in more depth in relation to concrete applications. In particular:

- Chapter 4 includes *An Approach to High-Level Language Bindings to XML* by Simeoni, Manghi, Lievens, Connor, and Neely, published in Elsevier's *Journal of Software and Technology* (44) in 2002. This first publication on type projections is largely motivated by the reconciliation of mainstream typed technologies and XML, particularly the type-driven 'extraction' of language values from regular subsets of potentially semistructured datasets. The focus is on the formalisation of the idea, and the main contribution is a language-independent framework for the rigorous definition of *extraction mechanisms* based on external type projections. The framework is instantiated to yield an extraction mechanism for an idealised language with a type system of purely structural abstractions, including record, collection, union, and recursive types. In applications to concrete languages, structural abstractions may be derived from abstract data types and coexist with types that characterise values that we do not expect to be represented in the data, such as functions. A high-level algorithm implements the mechanism and the implementation is shown to be correct and complete with respect to the formal definitions. The work presents also a simple metric that quantifies the accuracy with which a type projection matches the bound data. It then reports on a first application in distributed computing based on a correspondence between the type system of the idealised language and CORBA's IDL [8]. The application is investigated further in [24].

- Chapter 5 includes *Language Bindings for XML* by Simeoni, Lievens, Connor, and Manghi, published in IEEE's *Journal of Internet Computing* 7 (1) in 2003. Here the theme of typeful programming is illustrated in more depth with a systematic comparison of mainstream approaches to XML parsing [16, 17]. A new class of data binding tools enters the arena of XML processing and defines an ideal point of reference for motivating and applying type projections. A deeper understanding of typeful programming reveals the tight-coupling induced by models that are dictated by data producers. This forms the basis for discussing the advantages that type projections retain over mainstream approaches, such as [15]. The prototypical application of type projections evolves as a general-purpose data binding tool for Java, and a correspondence between the type system of the idealised language and Java's bridges the design of the tool to the formal results presented in previous work.

- Chapter 6 includes (an extended version of) *Hybrid Applications over XML* by Manghi, Simeoni, Lievens, and Connor, published in the Proceedings of the Fourth ACM CIKM International Workshop on *Web Information and Data Management (WIDM'02)* in 2002. This work embraces the full implications of typeful programming and marks a shift from external to internal type projections. The focus is thus on metamorphic languages, and type projections are employed to decouple the components of a single program in support of typeful programming. One such language is proposed which integrates procedural and declarative paradigms for hybrid applications over XML, as anticipated in Section 1.3.2.

Overall, this selection summarises the main findings of a collaborative research investigation that spanned the years 2000-2004 (see also [25, 26, 24, 27]). In doing so, it illustrates the evolution of the themes that we elaborate in their most general and comprehensive form in the first part the Dissertation. Evolution applies first to concepts: type projections originate as a mechanism to reassert conventional typechecking regimes over subsets of irregular and external data; they then evolve to inform general-purpose binding models inspired by the principle of typeful programming and the potential for loose-coupling inherent in self-description. The implications generalise accordingly. Applications are envisioned at first at the boundary of mainstream languages, as data binding tools. They then enter the design of novel languages in order to enable type-safe metamorphoses of language values.

We conclude in Chapter 7, where we take stock of the current state of the investigation, outline directions for further work, and overview less conventional applications of type projections. In this context, we also discuss relevant developments in the field which followed the research investigation but are nonetheless relevant to its themes.

1.5 Contribution and Thesis

In summary, the main contribution of the Dissertation is the identification a type-safe model for binding statically typed programs and self-describing data via type projections. The contribution is primarily analytical, in that we expound the concepts, motivations, and application domains that remain associated with the model. The analysis includes:

- an account of typechecking practices which culminates with the identification of the principle of typeful programming. This qualifies and generalises the role of types in programming languages beyond the conventional viewpoint of program correctness. Most importantly, it offers the motivation that unifies all the applications of type projections.
- an account of self-describing data and its implications for general-purpose data processing. To the best of our knowledge, this represents the first systematic attempt to analyse self-description in the formal literature. Type projections then emerge as a mechanism to exploit the potential for loose-coupling which inheres in self-describing data.
- the identification of a new class of tools based on type projections for binding statically typed languages and self-describing data, and their comparison with related tools in the mainstream.
- the identification of metamorphism, a novel form of type abstraction that generalises over polymorphism and coercions under a data-first approach to dynamic typechecking.

The analysis gives rise to a design space that stretches within and across language runtimes. We illustrate key points in this space with reference to published work, both in terms of formalisation and concrete applications. This includes:

- a formal framework for the definition of external type projections and its application to an idealised language with structural types. This serves a model of the design and implementation of binding tools for mainstream languages and standard self-describing formats.
- a report on the design and implementation of prototype binding tool for Java and XML.

- a proposal for the design of a metamorphic language that unifies procedural and declarative paradigms for computing over XML.

Collectively, these contributions form the evidence that supports our thesis, namely that binding models based on type projections can play a significant role in modern programming, both within and across the runtime systems of statically typed languages.

Chapter 2

Typeful Programming

In [28], Cardelli describes *typeful programming* in the following terms:

> *There exists an identifiable programming style which is based on the widespread use of type information, and which relies on mechanical and transparent typechecking techniques to handle such information. This typeful programming style is in a sense independent of the language it is embedded in; it adapts equally well to functional, imperative, object-oriented, and algebraic programming, and it is not incompatible with relational and concurrent programming.*

In this view, types are partial specifications of the intended behaviour of programs and typechecking is a tractable and reasonably predictable form of program verification. Typeful programming remains thus associated with *computational safety*, i.e with guarantees of correctness through error detection.

This *types-as-specifications* perspective is undoubtedly important and characterises a large body of research spanning more than three decades [29]. It is also relevant to the motivations of type projections, encompassing a rich body of notions that form essential background for their definition. In this Chapter, we give a selective overview of such notions, particularly those that revolve around typechecking strategies.

Equally, however, we posit that typed models of program inputs offer little guarantee of safety per se. Useful degrees of error detection may only be attained when the models are as adequate reflections of program requirements as can be afforded under the available type system. Most importantly, we argue that the benefits of programming against 'good' data models extends well beyond the enforceability of intentions. Independently of typechecking disciplines, adequate data models promote correctness by simplifying the way in which programs are written, understood, and maintained.

This is a broader and more qualitative perspective on types which becomes significant as soon as the interpretation of data may not be entirely under the control of its consumers. To emphasise this *types-as-models* perspective, we recast the style of typeful programming as a programming principle: *programming should be specified and mechanically verified against input models that are in optimal alignment with programming requirements*. It is precisely from this perspective that type projections are seen to promote typeful programming over self-describing data.

We make these arguments in the final part of the Chapter. This leads us to look beyond existing language support for typeful programming and to propose *metamorphism* as a generalisation of polymorphism and coercions which relies on type projections over language values.

2.1 Safety

Intuitively, a computation is safe if its errors do not go unnoticed [30]. In particular, a computation is safe if: (i) it is correct, that is it unfolds according to intentions, or (ii) its incorrect behaviour is detected by a vigilant runtime. Thus safety approximates correctness and promotes it through error detection.

The notion can be lifted to programs: a program is safe if it gives rise only to safe computations. Attached to a syntactic form, (some aspects of) safety become statically observable in principle. Through program analysis alone, a guarantee may be given that executing the program will never raise undetected errors. For reliability and convenience, the analysis may be automated and the guarantee offered as a system service. This raises a requirement for *safe languages*, i.e. languages in which programs are guaranteed to execute safely.

There are two broad strategies for the design of safe languages, depending on whether correctness is enforced during or before program execution. In dynamic strategies, program analysis generates runtime checks at points of potential failure. These points may be automatically identified. Alternatively, programmers may mark them with specifications of intended behaviour. In this case, the language may offer facilities for coding 'defensively', i.e. writing programs that explicitly check their own behaviour (e.g. *assertions* or *dynamic contracts* [31, 32, 33]). For convenience, we say that these strategies pursue *dynamic safety*.

In static strategies, the goal of program analysis is to prove correctness. In particular, (i) programs are rejected if they are provably incorrect, i.e. may generate *at least* one incorrect computation; and (ii) programs are accepted if they are provably correct, i.e. generate *only* correct computations. Program analysis becomes *program verification* and employs automated theorem proving techniques to validate formal specifications of intended behaviour. We say that these strategies pursue *static safety*.

In principle, static safety is more desirable than dynamic safety because it removes the burden of exhaustive testing. It also promotes efficiency, for provably correct programs do not require runtime checks. In practice, however, program verification is limited. Theorem proving techniques cannot compensate for incorrect, incomplete, undecidable, or simply intractable specifications [34]. This raises the issue of how static strategies should handle doubt: whether they should generate runtime checks to guard the execution of programs that may generate incorrect computations; or whether they should be conservative and refuse to execute programs that may still execute correctly. The first approach falls back to dynamic safety where necessary, reintroducing a dependency on testing. The second insists on static safety at the cost of false negatives. A static strategy may combine the approaches, showing more or less tolerance to different forms of potential failure.

2.2 Typechecking

Strategies for static safety continue to improve [35], but deployment is limited by the complexity of formal specifications and the opacity of automated proofs. In practice, formal specifications may be adopted but not verified, or they may be verified only in critical program sections. Alternatively, verification may concern program properties other than full-blown correctness [36].

As a case in point, static and dynamic strategies are equally pursued in the restricted context of *type safety*, the guarantee that operations are applied within their intended domain of definition. *Types* here model domains by generalising over some properties of their values while ignoring others, primarily identity [37].

Type safety is a trivial property of languages defined over a uniform value space. In the lambda calculus, values are computable mathematical functions defined over the domain of all such functions, so that any value can be safely applied to any other [38]. Type safety is threatened as soon as functions encode numbers and truth values, that is when values become partial functions [39].

Languages with heterogeneous value spaces pursue type safety by *typechecking* the application of operations, i.e comparing the types of operations and operands. Those that succeed are *strongly typed*. Java [6] and ML [40] are two examples of strongly typed languages in the mainstream. Languages that employ typechecking but still allow incorrect applications are *weakly typed*. C [41] and Pascal [42] are well-known examples of weakly typed languages. Languages that do not employ any form of typechecking are *untyped*, and they are rarely found among high-level languages. Machine and assembly languages are prototypical examples of untyped languages.

Typechecking strategies exemplify the approaches outlined in the previous Section. In *dynamic typechecking*, types annotate runtime representations of operations and values, successful type checks are a precondition for the application of operations, and their failure halts execution.

In *static typechecking*, types are associated with the denotations of operations and operands and successful type checks are a precondition to the execution of the entire program. Typechecking is carried out implicitly in the process of *typing* the entire program, i.e. inferring a type for it. The inference starts from its atomic expressions and proceeds recursively along its syntactic structure according to a set of well-defined *type rules*. Inference may be full or partial, depending on whether the type of variables remains implicit within the program or is explicitly denoted by its programmer [43]. In either case, types form a language of structured expressions, the *type language*. Type language and type rules form the backbone of the *type system* [30]. Typing is the responsibility of a dedicated language component, the *typechecker*.

2.3 Typed Languages

Static and dynamic typechecking can both lead to strong typing and usually coexist within a language (see Section 2.5). Traditionally, however, language design has shown bias towards either approach. Two strategies, in particular, characterise the majority of mainstream languages.

The first strategy is entirely based on dynamic typechecking but restricts its scope to predefined operations. This is the strategy of *dynamically typed languages*. The second strategy is dominated by static typecheking and concedes to dynamic type checking only when strictly necessary, e.g. to perform range checks and bind to external data (see Section 2.4.2). This is the strategy of *statically typed languages*.

Dynamically typed languages take a minimal approach to strong typing and catch errors as late as possible. Types annotate the runtime representation of values and type checks are built in the implementation of predefined operations. This can reduce the number of checks at runtime, putting less burden on the execution of type-correct programs. It also promotes heterogeneity and dynamism in the value space: generic lists, associative arrays, extensible objects are commonplace structures in dynamically typed languages. Unconstrained structures encourage exploratory and prototyping styles of programming, which befit small-scale development. They also serve well for middleware and framework programming, where data is manipulated under generic viewpoints. The strategy may prove too lazy when programs are in error, however, as incorrect invocations of user-defined operations are not detected upfront. Errors may then propagate through call chains, possibly outside the scope of the offending program.

Statically typed languages offer a different trade-off. Enforcing correctness as early as possible yields a better error detection tool and a more efficient runtime, as discussed in Section 2.1. It may also lift the notion of safety further into the application domain; for example, some languages compare type expressions by user-defined names (*nominal typechecking*) and channel data access through user-defined algebras (*data abstraction*). Issues of safety aside, statically typed languages employ types towards clearer structuring of programs.

The drawbacks are of course in the fallibility of program verification. For a typechecker, the risk of false negatives arise with programs that manipulate heterogeneous values, revealing only what is common to all of them. What differs across values is knowledge that the typechecker loses and for which it may no longer *deduce* a type. Such type may only be *induced*, often on the basis of explicit assertions (e.g. type casts). Inductive steps avoid false negatives but their validity must be checked at runtime, diluting the commitment to static typechecking.

The tension between statically typed languages and heterogeneity accounts for the regularity of their value spaces; the heterogeneous lists of dynamically typed languages are often replaced by homogeneous arrays and their associative arrays are constrained into record-like structures. Heterogeneity arises also with programs that require only partial knowledge of their inputs, however; lists may be homogeneous, but a program may wish to reverse any list regardless of the actual type of its elements. Barring such programs from execution prevents a fundamental form of code reuse and introduces well-known problems of redundancy and evolution.

Overall, statically typed languages risk a loss of algorithmic expression which does not surface in dynamically typed languages. To contain the loss, they must increase the sophistication of their typing inference, typically through *type abstraction*.

2.4 Type Abstractions

Intuitively, a type `T` abstracts over another type `T'` if `T` gives a looser description of the values of `T'`. As such, `T` may serve to type programs that are defined over the values of `T'` or *any* other of its specialisations.

Some type abstractions are 'transparent' to the typechecker, i.e. reveal the properties that are common to the values of all their specialisations. Transparent type abstractions serve well programs that exhibit the same behaviour across inputs of different types (*generic programs*). Statically typed languages that support such abstractions are *polymorphic* [44]. Polymorphic languages avoid false negatives with sophisticated typecheckers, often at the cost of added complexity.

Other type abstractions are instead 'opaque' to the typechecker, i.e. reveal no more of their values than they belong to some of their specialisations. Opaque type abstraction serve well programs that behave differently across inputs of different types (*type-*

driven programs). Statically typed languages that supports such abstractions make use of *union types*. Languages with union types avoid false negatives with liberal typecheckers, i.e. with concessions to dynamic typechecking.

2.4.1 Polymorphism

Polymorphism may take one of two common forms. In the first form, type abstraction relies on the universal quantification of type variables. Intuitively, a type `T` abstracts over a type `T'` if: $i)$ `T` includes (universally quantified) variables $\mathtt{X}_1$,$\mathtt{X}_2$,...,$\mathtt{X}_n$, and: $ii)$ the typechecker can infer *type parameters* $\mathtt{T}_1$,$\mathtt{T}_2$,...,$\mathtt{T}_n$ which reduce `T` to `T'` when they are bound to the corresponding variables. This is *parametric polymorphism* and an operation that reverses arbitrary lists is well served by it [45][1].

In the second form, type abstraction originates in record types [46] . Intuitively, a record type `T` (the *supertype*) abstracts over a record type `T'` (the *subtype*) if `T` describes a subset of the fields described by `T'`, recursively. This is *inclusion polymorphism* and is commonly found in object-oriented languages, often as an implication of inheritance [47].

Parametric and inclusion polymorphism have different implications and their formal treatments diverge accordingly. Most noticeably, supertypes are implicit abstractions and the knowledge that the typechecker loses upon inferring them may not be recovered [46, 48]. This is particularly problematic when values are mutable, as assignments to record fields which typecheck against supertypes may invalidate the constraints enforced by subtypes [19, 20, 23]. The convergence of parametric and inclusion polymoprhism is one way to address the problem: supertypes become explicit abstractions by including variables quantified over the range of their subtypes. This is

[1]Existential quantification of type variables yields another form of parametric polymorphism [37]. We note briefly that existentially quantified type abstractions may be used to hide the instantiations of their variables as the concrete implementations of abstract data types. This Dissertation, however, bears no direct relationship to them and we will not mention them further.

bounded parametric polymorphism [22].

2.4.2 Union Types

Union types are used to reintroduce heterogeneity on top of a regular value space, or to describe values that originate externally to the program. In either case, their values remain inaccessible until the typechecker can infer type specialisations for them. The inference may be automated, based on how the values are used in the program [49, 50]. Alternatively, it may require explicit type assertions, from simple type casts to sophisticated 'type-case' constructs based on pattern matching. The inference is inductive in all cases and may be disproved at runtime.

Union types may be finite or infinite, depending on whether the number of their specialisations is constrained. *Finite union types* are 'ad-hoc' type abstractions for case-based programming over data that may take one of a predefined number of forms. The united modes of Algol-68 [51] and derivatives are examples of finite union types within full-scale languages.

Infinite union types do not constrain their specialisations and thus serve as the highest type abstraction in the type system. They are commonly associated with persistent and distributed programming, as a trade-off between safety, generality, and resilience. The type `any` in CLU [52] and `REFANY` in Cedar/Mesa [53], Modula-2+ [54], and Modula-3 [55] are all examples of infinite union types in programming languages. The roots of type hierarchies in object-oriented language also qualify as infinite union types within mainstream languages (e.g. Java's `Object` type).

Union types often occur in forms that do not relate to type abstraction. A more mediated approach to typing unknown and possibly unrelated values is to describe their *disjoint* union. A *disjoint* or *discriminated union type* `T` describes pairs `(l,v)` where `l` 'discriminates' `v` as a value of one of the type disjuncts of `T`. In finite unions, discriminants may be plain labels that are statically associated with the disjuncts. In infinite unions, discriminants are representations of the type disjuncts. Discriminated union types occur in languages such as Pascal [42], Modula-2 [56], and Ada [57], but only within the context of record types and in association with enumerated (i.e. types with a finite number of values). They are also available in C [41] and its derivatives, where they receive dedicated constructors in the type language and value space. In an infinite form, discriminated union types are discussed in [1, 58] and implemented in experimental languages such as Amber [59] and Napier88 [4].

2.5 Hybrid Typechecking

With union types, languages concede to dynamic typechecking. An extensive use of union types, however, marks a 'dynamic' use of the language which may stretch beyond design intentions. Programs that are dominated by type-driven analyses risk to lose most of the advantages of static typechecking but little of its added complexity. As we note in Chapter 3, these programs may be better served by dynamically typed languages.

Some languages, however, pursue a balance between static and dynamic typechecking as a core design assumption. In *gradual typing*, for example, type annotations on program variables are optional. Programmers may introduce them partially and incrementally, progressing from dynamic to static safety across development phases. The typechecker makes standard use of the types it finds in the program, endorsing runtime optimisations when they prove compatible and reporting errors when they do not. Less conventionally, however, it deduces infinite union types when types are omitted and induces more specific types when types are reasserted, tentatively [60, 61]. As usual, dynamic type checks guard the inductions at runtime in order to preserve safety. Gradual typing is employed in full-scale languages such as Cecil [62], Dylan [63], and Python[64].

In *soft typing*, programs that can be proved correct execute unguarded, as usual. Those that cannot be proved correct generate warnings and execute with type checks at the points of potential failure. Finite union types are induced from the use of the data when deducing a more precise type proves impossible due to missing or contradictory information, i.e. when a conventional typechecker would report an error. [65, 66, 67].

2.6 Types as Models

In Section 2.1, we have introduced type safety as an approximation of computational safety. This served to put into perspective the guarantees that may be associated with its enforcement. Typechecking ensures that operations are applied within a given domain but protection from any other form of error rests upon defensive programming and extensive testing.

Even within this scope, however, guarantees need further qualification. The *intended* domain of an operation may only be partially modelled within the language. Typechecking may ensure that an operation is only applied to numbers, but this gives limited guarantees of correctness if the requirement is to apply it to, say, stock levels. Requirements of generality and tractability on language design contribute towards a semantic gap between the intended domains and those that can be typed within the language. Filling the gap rests again upon defensive programming: operations must explicitly check that their numeric operands do model stock levels.

Clearly, some types model intentions better than others, i.e. induce narrower gaps and decrease the need for defensive programming. Consider, for example, the following program fragments:

```
type T = set(record[
                   name:txt,
                   salary:num,
                   numbers:set(txt)])
```

```
rprint(x : T)
  for employee in x
    if (employee.salary>1000)
      show(employee.name)
      for n in employee.numbers show(n)
```

```
tprint(x : node)
  for employee in children(x)
    flag:bool = false
    name:txt = null
    for prop in children(employee)
      if (label(prop)=='name') name=value(prop)
      if (label(prop)=='salary'
          and (value(prop) as num)>1000)
        flag=true
      if (label(prop)=='number' and flag)
        show(name)
        for n in children(prop)
            show(value(n))
```

These fragments are meant to be functionally equivalent: both look for employees that earn above a given threshold and display their name and telephone numbers. They differ instead for their model of personnel: `rprint` chooses a set of records and navigates it with iterators and field dereference; `tprint` opts for a labelled tree of text-valued leaves and inspects the labels of internal nodes as it traverses parent-child relationships. In particular, `tprint` loops over employees and their properties, keeps track of names and matches on salaries, and finally reports upon encountering phone numbers.

Typechecking gives very different guarantees under the two models. Any attempt to access non-existing properties of employees is detected in `rprint` but it is not in `tprint`. For example, the typechecker cannot detect the incorrect use of `number` in place of `numbers` in `tprint` (line 10). Similarly, the typechecker cannot warrant numeric comparisons on salaries; these require explicit casts (`value(prop) as num` on line 8) and their validation must be deferred at runtime.

Similar differences would be observed when constructing employees. Under the first model, the typechecker can force the existence and uniqueness of names and salaries; programs need only to contribute with the proper initialisation of their properties. Under the second model, the typechecker can only impose constraints on the construction of trees; programs are responsible for enforcing the tree encoding of employees expected later by `tprint`. For example, the fact that phone numbers are grouped under `numbers` nodes rather than directly under employee nodes remains an unchecked convention. Similarly, it is up to programs to ensure that salary nodes contain numbers. Arguably, the implication of the tree model is that `tprint` forgoes much of the guarantees of type correctness of the language.

Looking beyond the scope of typechecking, it may not be immediately obvious that `tprint` is likely to contain a more insidious error. As it iterates over properties of employees, the program may encounter phone numbers before names or salaries. When this is the case, `tprint` does not display names or, worse, fails altogether to identify employees that satisfy the required condition. Effectively, the program makes an assumption on the order of the properties of employees, but it depends on the context whether this assumption is warranted or not.

There seems to be a clear relationship between such errors and the programming idioms used by `tprint`. While `rprint` can dereference the properties of interest, `tprint` needs to iterate across child nodes, inspect labels, cast values, and keep track of state it cannot immediately process. This results in a program that is exceedingly verbose and convoluted, hence more likely to contain logical errors.

2.7 Typeful Programming

The performance of the two models above reflects the different extent to which they embody program assumptions and requirements.

The record model captures explicitly the constraints associated with employees, including the existence of given properties, the cardinality with which they may occur, and the intended semantics of their values. This effectively exposes them to the system, which can then guarantee useful degrees of typechecking as well as space-time optimisations, including dedicated encodings for numeric values and convenient factorisations for employees. The programming algebra can also capitalise on constraints to simplify the construction and de-construction of employee values. Not only are simpler programs more likely to be correct, they are also easier to understand and thus to maintain and evolve over time. With the tree model, on the other hand, the constraints on employees remain implicit within programs, do not reflect on the programming algebra, and do not give scope to the system to offer useful services.

There is of course nothing inherently wrong with trees. The results of our analysis could have changed substantially if we had considered a different task or even assumed a different tree model. Indeed, we show in Chapter 3 that tree and graph models are preferred structures for semistructured data, and that they are also a common choice for generic programming over regular data. Similarly, we could have insisted on our sample task above but considered a different tree model, e.g. one that associates tree structures with higher-level algebras based on pattern matching or path expressions. We would have then found that neither linguistic convenience nor guarantees of safety are necessarily at odds with trees.

Our sample tree model, for example, may well be the ideal choice for tasks that require the 'deep' traversal of inputs and are less sensitive to their application semantics. Consider the performance of the two models when converting to upper case any text that occurs within slightly more structured personnel values (here `upper` is a pre-defined function):

```
type T = set(record[
                   name:record[
                         firstname:txt,
                         middlename:txt,
                         surname:txt]
                   salary:num,
                   numbers:set([record[
                               code:txt,
                               ext:txt,
                               num:txt)])
```

```
rcapitalise(x:T)
 for employee in x
   upper(employee.name.firstname)
   upper(employee.name.middlename)
   upper(employee.name.surname)
   for n in employee.numbers
    upper(code)
    upper(ext)
    upper(num)
```

```
tcapitalise(x:node)
  upper(value(x))
  for n in children(n)
   tcapitalise(n)
```

In this case, the algebras of set and record types offer little linguistic advantage, and in fact give weaker guarantees of safety. The generality of the trees matches the generality of the task and it is well served by a recursive solution. In addition, `rcapitalise` may only be applied to personnel and in fact must be modified as the model of personnel evolves over time.

These observations indicate that the non-functional properties of a program are highly sensitive to the model of its inputs, and that the performance of the model is highly sensitive to requirements, including the characteristics of the target programming environment. While no single model is likely to exhibit optimal performance all round, some models can significantly outperform others under a given set of programming requirements.

Overall, the evidence points to the identification of an informal principle, namely that *programming ought to occur under input models that are in optimal alignment* *with programming requirements*. Generalising over the definition given by Cardelli in [28] and quoted at the beginning of this Chapter, we refer to it as the principle of *typeful programming*. Not only do properties other than correctness depend on typeful programming, correctness itself rests more directly with typeful programming than with the sophistication of the typechecker.

Typeful programming is well known in the context of small-scale development, mainly in relation to the space-time efficiency of programs. The impact of input models upon storage consumption and performance requirements is at the foundation of most computer science curricula. Somewhat surprisingly, there seems to be much less awareness of its relevance to large-scale programming, where the implications extend to properties that are key in this context. As we discuss in Chapter 3, it is precisely in this context that the choice of input models is most easily constrained.

2.8 Metamorphoses

We have shown that the adequacy of typed models depends on the operation under scrutiny. From operation to operation, tree and record types may prove more or less convenient models of personnel. Typeful programming is then compromised when personnel is shared across operations, *unless* this sharing may occur under different models of the data. This raises a requirement against programming languages and programming systems, namely to decouple programs in their interpretation of shared data.

To clarify, this is not a requirement for interpreting trees as records, or viceversa. Their abstract semantics diverges and mixing them would be unsound. A record is more than the collection of named properties that it appears to have at any point in time. It also bears the guarantee that it will never lose any of those properties and that each property will always be unambiguously named. A tree value may satisfy these constraints at a given point in time, but there is no guarantee that it did in the past or that it will do in the future.

These guarantees, or the lack thereof, may not be found within the structure of individual records and trees, however. They are revealed by their generalisation into record and tree types, most noticeably by the associated algebras. While a record is not a tree and tree is not a record, structure alone may justify interpreting and using either one according to the type of the other. Any data that is modelled as a record could, at *any* point in time, be more conveniently manipulated with a tree algebra. Similarly, any data that is modelled as a tree may, at *some* point in time, be more conveniently manipulated with a record algebra. Typeful programming requires that values may be susceptible of different interpretations during their lifetime. We refer to such semantic changes as *metamorphoses*.

Conventional languages offer limited support for metamorphoses. The first type that describes a value, its *creation type*, may not be radically changed thereafter. In statically typed languages, the creation type is deduced or induced from the first denotation of the value within programs. In monomorphic languages it cannot change at all; in polymorphic languages it may only be abstracted over.

Polymorphism remains thus associated with a class of metamorphoses, those in which the interpretation of values is generalised to the effect that their structure is reduced (e.g. a list of integers becomes a list) and their algebra is narrowed (e.g. only some record fields may be dereferenced). The distinguishing feature of such changes is that they can be validated by deduction on the current type of the values, i.e. they can be statically typechecked.

Moving beyond these ‘quantitative’ metamorphoses requires dynamic validation. In addition, validation must be based on the structure of values rather than the type that captures their current semantics. Interestingly, metamorphoses of this kind occur routinely in most programming languages, as *coercions* between scalars. The intuition is indeed the same: records are morphed into trees like numbers are coerced into strings; some trees may be morphed into records like some strings may be coerced into numbers.

In conclusion, we can view metamorphism as the generalisation of polymorphism and coercions which is obtained by lifting their basic assumptions: that changes to values are statically validated; or that they are restricted to scalars; or that they do not preserve identity.

Chapter 3

Self-Describing Data

The notion of *self-describing data* surfaces almost ubiquitously in the practice of modern computing, most noticeably in the context of distributed systems and in relation to data storage and exchange standards. The obvious point of reference here is XML, a format that was originally intended for the structured representation of documents but it is now widely used as a general-purpose format for self-describing representations of structured data [12].

Its popularity notwithstanding, self-describing data is explained differently in different contexts, with interpretations that range from labelled data to typed representations of unlabelled data. The formal literature adopts the term in all its ambiguity but, to the best of our knowledge, contains no dedicated analysis of the notion of self-description and its implications for data processing. Self-description is framed within the pragmatics of system design, as a property of data representations used for storage and exchange. As such, it remains marginal to the academic discourse. The informal literature is richer in this respect, though analyses are here often narrow in scope. Interestingly, views on the role of self-description in modern computing prove invariably contentious.

In this Chapter, we address this gap and present a principled account of self-describing data. We start from a general characterisation of self-description and then consider the implications of a distinguished form of self-describing data, namely *labelled structures*. In the analysis, we draw relationships between self-description and *semistructured data*, *generic programming*, and *loose coupling*, arguably its strongest implication for software construction.

With loose coupling, we acknowledge the potential of self-description as an *enabler* of typeful programming and explain type projections as an attempt to exploit that potential in the context of a statically typed language.

3.1 Self-description

Under assumptions of systematic representation and processing, most data carries evidence of its own structure for the benefit of consuming processors. In natural language text, punctuation indicates sentence and paragraph structure to the human reader. In more rigidly formatted data, delimiters allow mechanical processing of structures with variable-length representations. The data contains some form of *metadata* and thus the trace of a semantics, if only an abstract one. Most data is then *self-describing*, to some degree and within some context of reference in which the metadatata is understood and the data is interpreted (Figure 3.1).

One way to define the context of reference is with a set of formatting rules. For example, consider the case of text structures organised into lines and then further divided within each line by some distinguished character. Within the context of this Comma-Separated-Values (CSV) format [68], newlines and delimiters are metadata and describe datasets as sequences of tuples with variable fields of variable length.

This variability within and across datasets is the primary justification for self-description. No metadata would be required if tuples were expected to have the same number of fields and each field the same number of characters. Instead of conflating

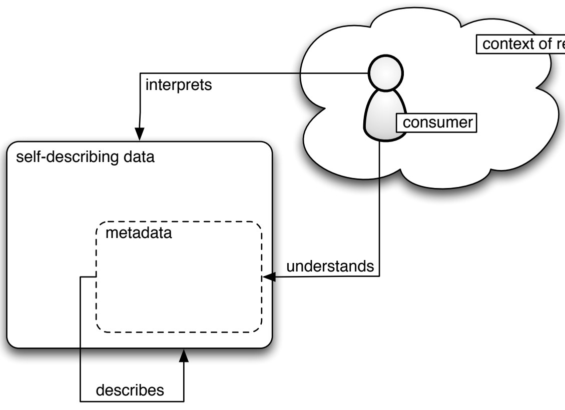

Figure 3.1: Self-Description

the size of the data, these numbers could be conveniently factored out in the context of reference (Figure 3.2).

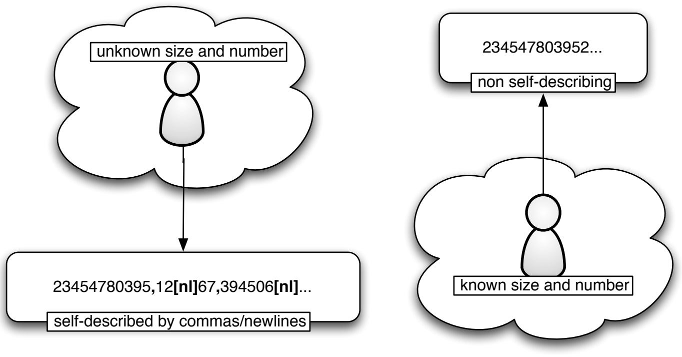

Figure 3.2: Self-description and Static Knowledge

Self-description captures within each dataset those aspects of its semantics that vary within and across datasets, and thus cannot be held constant within the context of reference. The requirement for self-description is in inverse proportion to our *static knowledge* of the data: lower degrees of static knowledge demand higher degrees of self-description, and vice versa.

The intuitions are relatively simple but become significant when self-description relates to application domains rather than abstract value spaces (e.g lists of tuples).

This passage from ‘abstract semantics’ to ‘application semantics’ is deceptively simple. It occurs when one relies on metadata not only to identify structure, but also to *name* its components. The first record of a CSV dataset, for example, is often intended as a ‘header’ for the rest of the data, in that its fields name those of the remaining tuples. In these cases, we speak interchangeably of self-describing data and *labelled structures*.

3.2 Generic Programming

Semantics that, informally, ‘moves’ from the context of reference into the data supports generic manipulations of the data.

Unaware of its application semantics, for example, we can exploit the metadata in a CSV dataset to reverse the order of its tuples, or to rearrange them based on the lexicographic order of their n-th field. We can more generally write a parser to model its abstract semantics within a language (e.g. as a bulk value or as a stream), ensuring its well-formedness and hiding encoding issues from other programs. An application may then more easily inject the parsed data into relational tables based on the application semantics of fields.

A first implication of self-describing data is thus for *generic programming*. The richer is the semantics represented within the data, and thus the metadata embedded in it, the larger is the class of programs which can be factored out of and reused within applications.

A typed variant of the CSV format, for example, may specify the semantics of tuple fields in terms of numbers, strings, or dates. We can then exploit the additional metadata for generic programming, e.g. towards more refined rendering or ordering of the data, but also for application programming, e.g. to validate the mappings of fields onto columns of relational tables.

Labelled structures broaden further the scope for generic programming. We can now write generic programs that are parametric with respect to labels, and thus with respect to application semantics. Using header tuples, for example, we can write a program that orders a CSV dataset based on predicate expressions on field names. More ambitiously, we can write a general-purpose query and transformation language for declarative specifications of arbitrary manipulations of the data. Even when application semantics *could* be held as constant, its exposure through a general-purpose format — ideally a strong standard — promotes the use of equally general tools in support of application programming. The wide deployment of similar tools for XML illustrates well the potential of self-describing data for generic programming [69, 70].

Notice that this does not contradict the relationship of inverse proportion between static knowledge and self-description we observed in Section 3.1. When we rely on generic tools, we push the interpretation of the data in a context in which its application semantics is unknown and thus must be represented within the data.

3.3 Semistructured Data

In CSV data, self-description is justified by variable-length structures. Structural variability may assume a great variety of forms, however, and some forms limit severely what can be assumed about the data in the context of reference. In these cases, structural variations are perceived as 'irregularities' and one speaks of *semistructured data* [10, 11, 71].

Data is semistructured if its structure is explicit and yet varies under the same application semantics. Data elements that are conceptually related in the problem domain may have different parts, and those they share may differ in turn. This irregularity reduces the static knowledge of the data and thus renews the requirement for self-description. Labelled graphs serve the purpose well, as they can describe arbitrary structure without imposing homogeneity constraints upon it. Essentially, the graph semantics absorbs the irregularity that would be observed under less general abstractions.

Figure 3.3 showcases the range of irregularities that may be found in semistructured data. Different employees are modelled differently within the data: employees may have zero, one, or more phone numbers and their names may or may not be structured; some employees are linked to their offices but the link is labelled differently across employees. Similar features may also be observed over time: some employees may acquire email addresses while others may acquire standard postal addresses, and the two may be differently structured; phone numbers of employees may be described indirectly as properties of their addresses.

Spatial and temporal irregularities of this kind have been observed in scientific domains characterised by long-term studies, particularly when bridging systematic data and experimental evidence, or when the data grows too quickly under advances in domain-specific technology [72].

They have been more generally linked to decentralised collation and modelling policies. Distributed information services, for example, associate irregularity with the variety of narrative styles and document types induced by autonomous authoring; services that integrate or mediate between content collections may find that the degree of regularity that could be assumed for each collection is considerably reduced at the time of their merging [73, 74]. The example generalises: problems of data integration and evolution affect most applications that target autonomously managed and widely distributed data.

The problem with semistructured data is that mainstream technology cannot easily handle it. Conventional programming and database systems make assumptions of homogeneity and stability of the data which allow them to generalise over its structure and semantics. The resulting types and schemas form a body of static knowledge that the systems exploit to verify, document, and optimise the use of the data. The

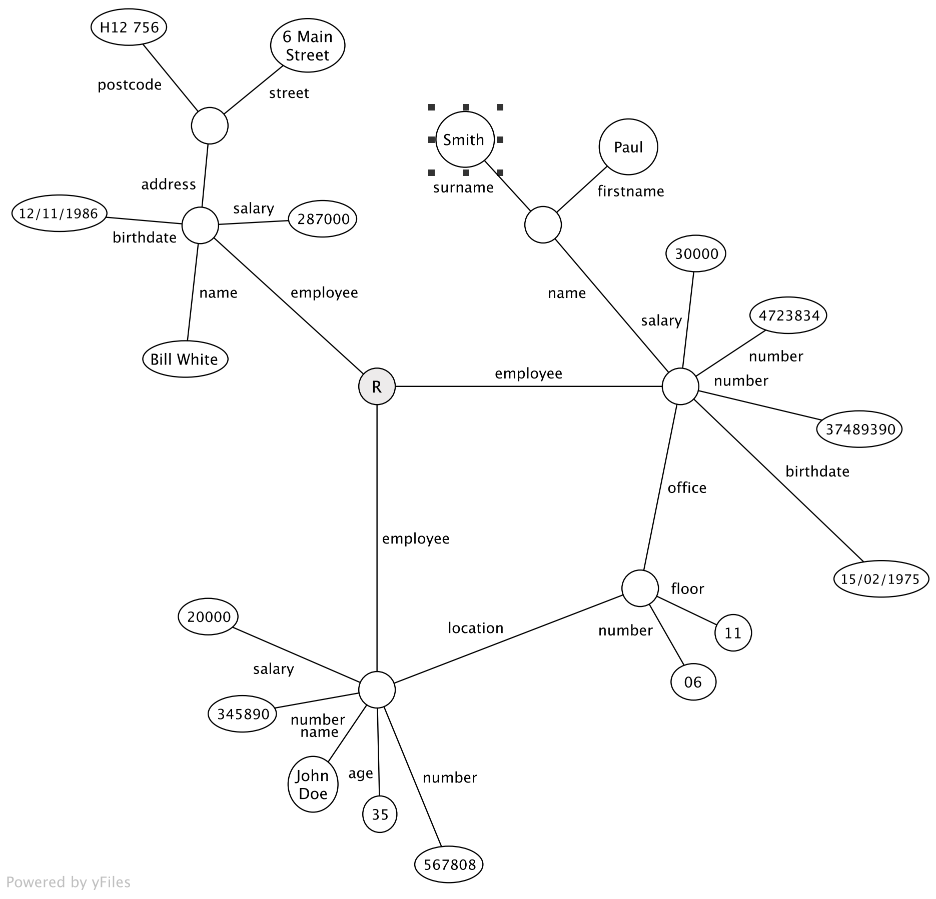

Figure 3.3: An example of semistructured data

approach pays off in proportion to the amount and specificity of the available knowledge; its effectiveness decreases as the generality of types and schemas increases. The systems populate their value space accordingly, with relations, records, objects, and collections that enforce constraints of homogeneity and uniqueness upon the data, and thus run opposite to the requirements of semistructured data. Using such structures to model graphs is a solution only insofar as user-defined algebras offer adequate programming idioms for irregular data. Expectations in this sense are high: one would want flexible constructs to distribute programming logic along structural variations in the data and, conversely, the possibility to ignore variations that bear no relationship to it. The requirement is thus for algebras that may simplify *case-based programming* and programming under *partial knowledge* of the structure of the data, ideally without excessive compromises in efficiency. To compound the problem, user-defined models of graphs sacrifice static knowledge for modelling flexibility. The trade-off may then be unappealing if the data is partly regular and its irregularities are known in advance, as it is often the case in data integration scenarios.

There is a substantial body of work that addresses this problem. Most of the work originates at the fringes of database research and investigates algebras for querying and transforming semistructured data. This is reasonable ground: irregularities are most commonly observed in long-lived data and declarative algebras with bulk evaluation semantics can yield enough transparencies to handle them. Two strategies emerge:

- an 'evolutionary' strategy, which extends the static approach of conventional technology to accommodate contained degrees of irregularity;
- a 'revolutionary' strategy, which dispenses with static knowledge for maximum generality.

The two strategies can result in similar algebras, and their integration within a single system is possible [75].

In the first strategy the value space remains populated with heterogeneous structures but the type system includes some form of finite disjunction to describe them homogeneously (cf. Section 2.4.2). In [76], the approach relies on *tagged unions* but makes them transparent to the query algebra by inferring casts and absorbing failures. This allows queries to abstract over irregularities that occur in the data. In [77], the approach relies on *untagged unions* and combines them with an algebra of case expressions based on pattern-matching. A flexible notion of subtyping automates the optimal distribution of unions and thus simplifies case-based programming.

Neither approach dispenses with the requirement for self-description; labels are used to discriminate among disjoints, explicitly for tagged unions and implicitly for untagged unions. Both approaches, however, meet the requirement opportunistically and rely on self-description only where and when the irregularities become manifest. This is appropriate when the data is mostly regular and the structural variations can be statically enumerated. It becomes progressively less appealing when the lack of static knowledge cannot be so conveniently bound.

In the second strategy the query algebra is defined directly over a value space of graphs [78, 79, 80, 81, 82, 83]. The design space here is richer and proposals vary along a number of dimensions, including:

- *scope of application*: from general purpose data management to data integration, data conversion, and website management [73, 84, 80, 81, 85];
- *choice of data model*: from trees to unconstrained graphs, from ordered to unordered structures, from node-labelled to edge-labelled structures, and from object-based to value-based structures [86, 87];
- *syntactic and semantic foundations*: from structural recursion, graph simulation, and comprehension syntax to Skolem functions and various forms of logic [88, 89, 90, 91, 92].

The resulting algebras differ in expressive power but converge on core features:

- Queries are multi-clause expressions that declare filters, joins and transformations against variable bindings, in resemblance of conventional query algebras [93, 94];
- Variable bindings are to nodes and nodes are denoted by the paths that connect them to the root. *Path expressions* replace the multiplicity of operators required to traverse deep structures in heterogeneous value spaces, and confer a 'navigational' feel to the algebra. Effectively, they identify regular subsets of the data based on the topology of the graph rather than by abstraction over the application semantics of its nodes;
- Abstractions now apply to the paths themselves: regular expressions over labels and sequence of labels accommodate irregularity of data naming and location; variables on labels and sequence of labels enable queries that inspect the structure and transform it into data [95];
- Path expressions that do not match the graph topology make no contribution to the output of queries, but they never fail them either.

The last point captures the spirit of the strategy. When one designs for unbridled irregularity, the issue of structural correctness is moot in principle: a query that appears unsound at the point of evaluation may prove correct shortly after. Equivalently, a match that fails on the structure is indistinguishable from a condition that fails on the data. This is in stark contrast with the conventional approach and its evolutions, where opposite expectations justify a clear notion of structural correctness and motivate the use of typechecking techniques for its static enforcement (cf. Section 2.2).

In practice the structure of the data is never entirely unpredictable, of course. As much as the conventional approach may concede to some irregularity, so can the opposite approach acknowledge and exploit the existence of some regularity. Static knowledge may thus be reconciled with a self-describing value space, to an extent that varies along a spectrum of expectations about the data.

At one side of the spectrum, the data is mostly irregular and the static knowledge is partial, indicative, and relatively transient. A number of type and constraint formalisms have been proposed to infer and describe this form of knowledge [96, 97, 98, 99, 100]. Exploitation here is opportunistic rather than systematic, and it does not change the underlying model of query correctness. Types are intended for query decomposition and optimisation, or as a guide to query formulation [101, 102].

Moving towards the opposite side of the spectrum, the data becomes more regular and the static knowledge is progressively less partial and more stable. Ultimately, it becomes prescriptive and the expectation of typechecking can be reintroduced.

One interesting result is that we may meet the expectation without a 'continuity jump' in the design space. Instead of reverting to a conventional value space with union types, we can directly typecheck queries over the self-describing value space, at least as long as the data can be interpreted as an ordered tree.

This requires unconventional type systems based on formal language theory, and may even require unconventional notions of type correctness [103, 18, 104, 105, 106]. The approach is immediately relevant to programming over partly-regular data, because it grants more resilience to irregularities that are problematic to handle with union types in conventional value spaces [104].

3.4 Loose Coupling

The last implication of labelled structures is more subtle. It stems from the observation that application semantics that is represented within the data does not have to be known a priori, or in its entirety, in all the different contexts in which the data may be interpreted. Within such contexts, programs may refer to it dynamically and to the precise extent which is necessary to validate and uphold *their own* interpretation of the data.

Programs that align minimally against shared data, rather than synchronise fully on a shared interpretation of the data, are *loosely coupled*. Once more, we observe the relationship of inverse proportion between self-description and static knowledge, now in relation to the broader context in which sharing occurs: we reduce shared knowledge and increase self-description in order to decouple interpretations of the data (cf. Figure 3.4).

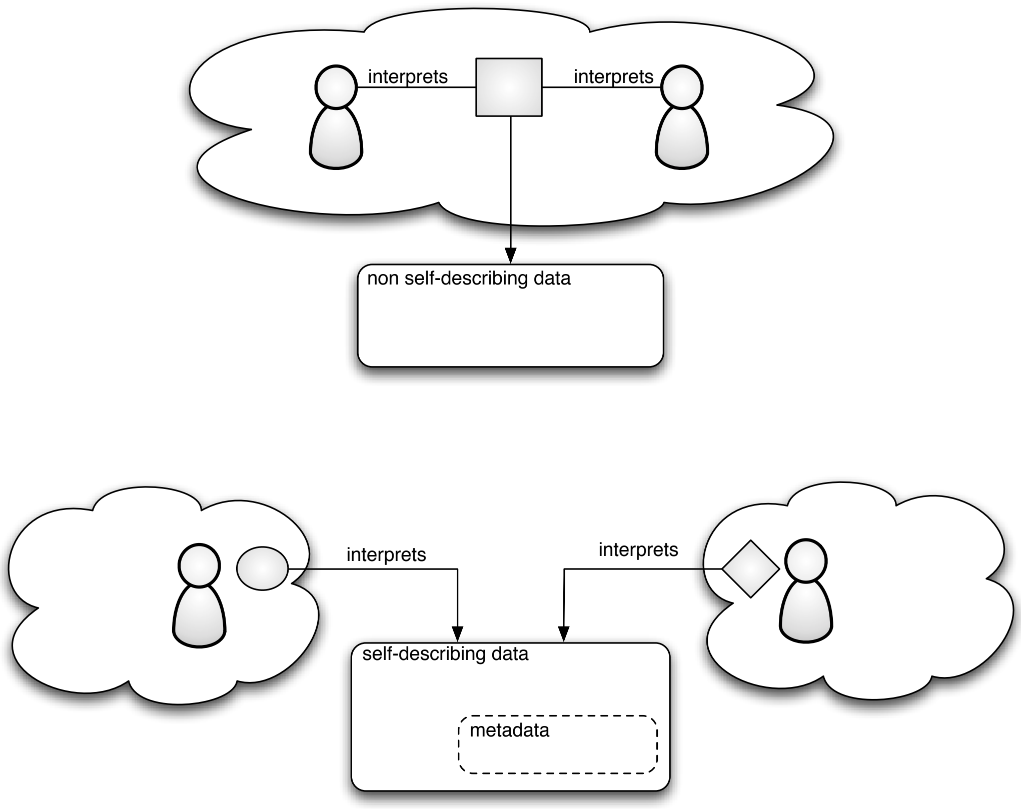

Figure 3.4: Tight and Loose Coupling

As a simple example of loose coupling and its implications, consider a program that wishes to consume some fields of interest within CSV datasets. If the datasets do not name the fields of their tuples, the program needs to access them based on position. Positional access requires exact knowledge of the structure of the dataset (up to the point where it is being accessed), and thus forces the program to interpret the data as the program that produced it. As the requirements demand only partial knowledge of it, the interpretation introduces unwanted dependencies. Over time, changes induced by the producer (e.g. additions and removals of fields) require similar changes on the program, even when these are irrelevant to its requirements. If datasets include a header that labels the fields, however, the program can directly dereference the required ones and ignore the evolution of the others. With access 'by name', the program needs no more knowledge of its input than strictly induced by its requirements.

These observations generalise, in that the relationship between labelled structures and loose coupling is more general than access by name. In the labels, different programs may find sufficient evidence of application semantics to uphold radically different interpretations of the data, not just partial ones and not solely for program resilience.

Arguably, this is the most subversive value proposition of self-description for data sharing. Traditionally, data has been shared under the assumption of predictable processing requirements. In this case, there is little or no requirement for loose coupling and labelled structures inflate the size of the data unnecessarily. Under such 'closed-world' assumptions, self-description is best avoided and, until recently, so it has been.

Closed-world assumptions characterise the definition of many formats for the exchange, persistent storage, and transient representation of structured data. We find them already in the runtime of statically typed programming languages, for example in the definition of record-like structures. As the fields of records are statically defined, their names can be factored out of the runtime representations of individual records and moved into the representation of their types. Type representations can then be used to resolve access by name into positional access, typically prior to execution.

Closed-world assumptions here contribute to optimal memory management, and overall justify the semantics of records as particularly constrained labelled structures.

Conventional middleware systems based on remote procedure calls extend the closed-world assumptions of language runtimes [8, 107]. Labels are excluded from record serialisations and are kept instead into the context of reference, here here the skeletons and proxies that govern the typed exchanged of data. Serialisations may thus remain minimal but bindings become fragile when types evolve at the producing end of the wire. Evolution typically requires synchronisation of skeletons and proxies, and thus the re-establishment of shared knowledge at both ends of the wire [108].

An increasingly large class of applications, however, make fewer expectations on where, when, by whom, and most importantly *how* their data may be consumed during its lifetime. 'Open-world' assumptions are most commonly found in large-scale computing, where programs are distributed across mutually autonomous runtimes that share data over long distances and periods of time. They may also be recognised within single runtimes, as a result of the increasing level of abstraction at which even local computations manipulate their inputs (e.g. in application frameworks).

More generally, open-world assumptions characterise application systems that are built around principles of loose coupling. Within such systems, inflating the representation of shared data is a better option than conflating the context in which sharing occurs.

3.5 Self-description and Type Projections

With loose coupling, we can finally establish a connection between self-description and typeful programming (cf. Section 2.7). Self-description encourages diverging interpretations of shared data; typeful programming requires them to converge towards local processing requirements. Self-description is thus an *enabler* of typeful programming.

With type projections, we act on this relationship in the context of a statically typed language, where types capture the interpretations of data and type safety is one of the benefits that we expect in return. External type projections exploit self-description to assert the principle of typeful programming at the boundary of the language. Internal type projections do so directly over language values.

By implementing type projections, in fact, we reflect all the implications of self-description which we have discussed in this Chapter; we write a single generic program that uses the labels embedded within the data to parametrically validate and uphold interpretations that other programs may wish to lay upon the data, including partial interpretations of regular subsets of otherwise semistructured data.

Part II

Publications

Chapter 4

Extraction Mechanisms

This Chapter reports the contents of the following publication:

> SIMEONI, F., MANGHI, P., LIEVENS, D., CONNOR, R. C. H., AND NEELY, S. An approach to high-level language bindings to XML. *Information & Software Technology 44*, 4 (2002), 217–228.

This is the first publication on type projections. It is largely motivated by the reconciliation of mainstream typed technologies and XML, particularly the type-driven 'extraction' of language values from regular subsets of potentially semistructured data.

The focus is on the formalisation of the ideas, and the main contribution is a language-independent framework for the rigorous definition of *extraction mechanisms* based on external type projections. The framework is instantiated to yield an extraction mechanism for an idealised language with a type system of purely structural abstractions, including record, collection, union, and recursive types. A high-level algorithm implements the mechanism and the implementation is shown to be correct and complete with respect to the formal definitions.

The work presents also a simple metric that quantifies the accuracy with which a type projection matches the bound data. It then reports on a first application in distributed computing based on a correspondence between the type system of the idealised language and CORBA's IDL [8]. The application is investigated further in [24].

4.1 Abstract

Values of existing typed programming languages are increasingly generated and manipulated outside the language jurisdiction. Instead, they often occur as fragments of XML documents, where they are uniformly interpreted as labelled trees in spite of their domain-specific semantics. In particular, the values are divorced from the *high-level* type with which they are conveniently, safely, and efficiently manipulated within the language.

We propose language-specific mechanisms which extract language values from arbitrary XML documents and inject them in the language. In particular, we provide a general framework for the formal interpretation of extraction mechanisms and then instantiate it to the definition of a mechanism for a sample language core **L**. We prove that such mechanism can be built by giving a sound and complete algorithm that implements it.

The values, types, and type semantics of **L** are sufficiently general to show that extraction mechanisms can be defined for many existing typed languages, including object-oriented languages. In fact, extraction mechanisms for a large class of existing languages can be directly derived from **L**'s. As a proof of this, we introduce the *SNAQue* prototype system, which transforms XML fragments into CORBA objects and exposes them across the ORB framework to any CORBA-compliant language.

4.2 Introduction

Values of existing typed programming languages are increasingly generated and manipulated outside the language jurisdiction. Instead, they often occur as fragments of XML documents [12].

This may be because the containing documents are *semistructured*, i.e. their structure is too irregular or unstable to be effectively handled by traditional programming languages or DBMSs [11, 10]). It may also occur when the document is more disciplined, but needs to be exchanged across proprietary boundaries in a standard and self-describing format.

As an example, consider the following XML document d, where some irregularities have been intentionally added to the data for sake of illustration.

```
<staff>
   <member code = "123517">
      <name>Richard Connor</name>
      <home>www.cis.strath.ac.uk/˜richard</home>
   </member>
   <member code = "123345">
      <name>Steve Neely</name>
      <ext>4565</ext>
      <project>
         <name>SNAQue</name>
      <project/>
   </member>
   <member code = "175417">
      <ext>4566</ext>
      <name>Fabio Simeoni</name>
   </member>
</staff>
```

On a much larger scale, this irregularity would prevent the document from being conveniently managed within the typed framework of conventional technology. While union types and object-oriented features may accommodate some of the irregularity, their abuse would soon degrade the performance of the system and complicate program specification and maintenance.

Consider instead the fragment d' of d shown next:

```
<staff>
   <member code = "123517">
      <name>Richard Connor</name>
   </member>
   <member code = "123345">
      <name>Steve Neely</name>
   <member code = "175417">
      <name>Fabio Simeoni</name>
   </member>
</staff>
```

For most object-oriented languages, d' may be an XML encoding of an object `staff` of class `Staff`, where

```
class Staff {
      private Member[] member;
      Member[] getMembers() {...}
      void setMembers(Member[] members) {...}
      ...}
```

and `Member` is the class:

```
class Member {
      private String name;
      private int code;
      String getName() {...}
      void setName (String n){...}
      int getCode() {...}
      void setCode (int c){...}
      ...}
```

This simple observation raises the expectation that programming over d' be as simple, safe, and efficient as programming over `staff` with existing programming languages. In particular, we require these good properties to scale, i.e. hold for generalised computations over XML fragments considerably larger than d'. Unfortunately, we believe that none of the current approaches fully satisfies such requirement.

4.2.1 Background

To date, computations over XML data can be specified in a variety of paradigms, models and languages. Two kinds of approaches, however, appear to prevail: dedicated query languages and bindings to programming languages, typically object-oriented ones.

In query languages such as [82, 79, 109, 110], queries have a familiar SQL-like structure, but contain powerful path expressions specified against the tree topology of the data. This gives the languages the flexibility required to compute over data with irregular or partially known structure. It makes them also more succinct than full-fledged programming languages for most operations of data filtering and transformation.

However, query languages are usually not Turing-complete, nor well suited to complex programming tasks over large datasets, possibly involving recursion. Furthermore, they are essentially untyped, except with respect to the tree structure of the data. While this is justified in the general case by the typeless nature of the format, potential regularity in (subsets of) the data could and should be exploited for program verification and optimisation.

Language bindings are instead defined by implementing programming interfaces to one of two possible in-memory representations of the data. In the Document Object Model interface [16], the data is organised and manipulated as a labelled tree. In the Simple API for XML [17], the data is a string of characters organised and processed along parsing events.

Beside performance-related differences, both solutions impose an interpretation of the data which generalises their structural relationships (e.g. nodes of a tree), but conveys only indirectly and too concretely their intended meaning (e.g. staff of a university department). When computations explicitly address the structural properties of the data (e.g.adding or removing a node, searching for a string in the data), this interpretation is adequate. In most cases, however, it complicates program specification, making it tedious, error-prone and hard to maintain.

Consider, for example, any computation over the names and codes of the staff members in d. In a pure implementation of the DOM interface for a Java-like language, the code may include something like:

```
int code;
String name=null;
Element staff=d.getDocumentElement();
NodeList members = staff.getElementsByTagName("member");
int memberCount = members.getLength();
for (int i=0;i<memberCount;i++) {
 Element member = (Element) members.item(i);
 code = Integer.parseInt(member.getAttribute("code"));
 NodeList children = member.getChildNodes();
 int length = children.getLength();
 for (int j=0;j<length;j++) {
  Node child = children.item(j);
  if (child.getNodeType()==Node.ELEMENT_NODE) {
   String tagName= ((Element) child).getTagName();
   if (tagName.equals("name")) name=
    ((characterData) child.getFirstChild()).getData(); }
    ...do something with name and code ...}}
```

Even for a simple task, the code is highly convoluted and inefficient. Partly, this is due to the document-oriented nature of any XML programming interface. For instance, semantically related data must be accessed with the different algebras of elements and attributes. Similarly, manipulating atomic data requires an implicit or explicit cast from the type of strings, the only available. More generally, the logic of the computation is unnecessarily expressed in an algebra of trees, while domain-specific concepts (e.g. names and codes) are relegated to the role of run-time parameters.

The same task could have been specified directly against the object `staff` of class `Staff` defined above, and as simply as:

```
Member[] members = staff.getMembers();
for (int i=0;i<members.length;i++) {
 int code = members[i].getCode();
 String name = members[i].getName();
 ...do something with name and code ...}
```

The code is now aligned to the semantics of the application. It is also more succinct and less redundant, for generic operations on staff and staff members do not have to be repeated within specific computations, but can be factored out in class declarations, thoroughly tested, and then reused.

Inadequate data abstractions also compromise static checking of computations. Correctness can be guaranteed for operations on trees and strings, but not staff members. For example, the following invocation:

```
NodeList members = staff.getElementsByTagName("mebmer");
```

where `"mebmer"` is a typo for `"member"`, would silently compile and return a `null` value only at run-time. Safety is thus responsibility of the programmer, not the system. Programmatic checks worsen readability and maintainability of the code, and are not always sufficient to guarantee correct behaviour. In the lack of some description of the data (e.g. a DTD), the typo may be interpreted as the absence of required data and thus trigger unintended behaviour. Even assuming some data description, the typo may accidentally identify some other data or, in the best case, be simply signalled at run-time.

For similar reasons, the system can optimise resources only within the limits of its static knowledge of the data. For instance, it ignores the fact that all staff members have names and codes.

4.2.2 Extraction Mechanisms

Motivated by the previous observations, we aim at defining *high-level bindings* between XML and existing programming languages, which preserve the intended semantics of the data.

Specifically, we propose language-specific mechanisms that *extract* self-describing representations of language values from arbitrary XML documents, and transform them into their counterparts within the language. Thereafter, the extracted data are computed over in a familiar, expressive, and robust environment.

To achieve this for a given language, we interpret the extraction of a value as the projection of its language type over the containing XML document. Following the previous example, the projection of class `Staff` over d would result in the extraction of the object `staff`.

More formally, let $\mathbf{D}$ be the set of XML documents, $\mathbf{L}$ a typed programming language, and $\mathbf{V}$ and $\mathbf{T}$ the value and type spaces of $\mathbf{L}$, respectively (see Figure 4.1). Let also $sd : \mathbf{V} \rightarrow \mathbf{D}$ be a self-describing interpretation of $\mathbf{L}$'s values in $\mathbf{D}$, and $\preceq$ in $\mathbf{D} \times \mathbf{D}$ a relation of 'inclusion' between XML documents.

Definition 4.2.1 *Let $v \in \mathbf{V}$. v is* extractable *from $d \in \mathbf{D}$ according to $T \in \mathbf{T}$ if: (i) v has type T, and (ii) there exists $d' \preceq d$ such that $sd(v) = d'$.*

Finally, an *extraction mechanism* for $\mathbf{L}$ takes both a document d and a type T, and returns a value v extractable from d according to T, if one exists.

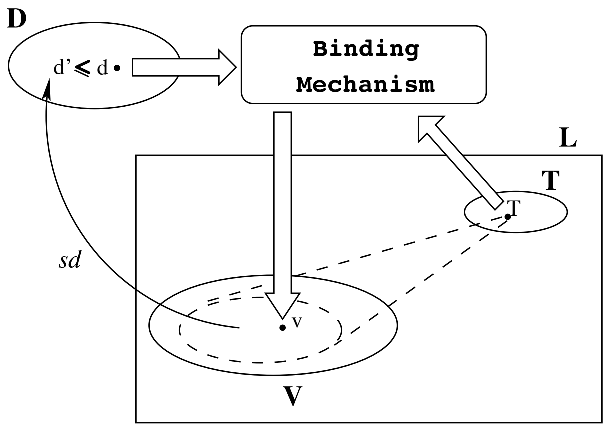

Figure 4.1: An extraction mechanism for $\mathbf{L}$

An extraction mechanism is invoked by a programmer, via some interface to the language. If no extractable value can be returned, the programmer is notified of the failure. In case of successful extraction, the returned value may still not fully satisfy the programmer's requirements.

The reason is that, when passing a type to the mechanism, the programmer may not be aware of the exact structure of the target document. Such a type will probably be defined after an eye-inspection of the document or, if available, a description of its structure. As a result, the type may be an under-specification or an over-specification of the data that are potentially relevant to the programmer.

These sort of misjudgements may be very frequent, especially in correspondence with large documents in which data have been inserted at different times and possibly by different users with a different cognition of data representation.

In general, we require that some quantification of relevance be always returned along with the value extracted by the mechanism, and Section 4.7 will present one quantification scheme in more detail. By interpreting the quantification, the programmer may conclude the inadequacy of the proposed type, refine it, and then re-invoke the mechanism in a prototyping fashion.

4.2.3 Outline

In the rest of the paper, we concentrate on the formal definition of an extraction mechanism. This is done in Section 4.5, after the definition of a simplified syntax for XML data and a typed language core in Section 4.3 and Section 4.4, respectively.

Section 4.6 completes the definition of the extraction mechanism by presenting an algorithm that implements it. The algorithm is then used to illustrate a sample quantification scheme in Section 4.7. Section 4.8 presents a prototype implementation of the mechanism, while Section 4.9 and Section 4.10 examine related work, draw conclusions, and outline further work.

4.3 A Document Syntax

In this Section, we define a syntax for XML documents that isolates the data-oriented features of the format (e.g. naming and nesting) from its document-oriented features (e.g. ordering, attributes, processing instructions, etc.). In practice, element attributes may be replaced by subelements.

Definition 4.3.1 *Let* D *be the language of* documents *defined by the following grammar:*

$$d,\, d_1,\, d_2 ::= <> \mid\; s \;\mid < l \rightarrow d > \mid\; d_1 \wedge d_2$$

where $l \in Lbl$, $s \in Str$, and the sets Lbl of labels and Str of strings are pre-defined languages over the same alphabet of characters.

A document d is atomic or complex. An atomic document is either the *empty document* $<>$ or else a string. A complex document is either the singleton document $< l \rightarrow d >$ or the concatenation $d_1 \wedge d_2$ of two complex or empty documents. Finally, we shall consider equal two documents that differ only in the ordering of components (e.g. $d_1 \wedge d_2 = d_2 \wedge d_1$).

Essentially, we interpret a document as an edge-labelled tree, in slight contrast with the standard interpretation of XML documents as node-labelled trees. This choice allows us to simplify the formal treatment but has no impact on the applicability of our results.

In the rest of the paper, we shall abbreviate $d =< l_1 \rightarrow d_1 > \wedge \ldots \wedge < l_p \rightarrow d_p >$ with $< l_1 \rightarrow d_1, \ldots, l_p \rightarrow d_p >$ and refer to $l \rightarrow d$ as to an *l-field* with name l and value d. We will also use the function FV_l which takes a document and returns the set of its l-field values.

Definition 4.3.2 *Let $l \in Lbl$. Let $FV_l : \mathbf{D} \rightarrow \wp(\mathbf{D})$ be the function:*

$$FV_l(d) = \begin{cases} \{d'\} & d = [l \rightarrow d'] \\ FV_l(d_1) \bigcup FV_l(d_2) & d = d_1 \wedge d_2 \\ \emptyset & otherwise \end{cases}$$

As an example, the following document d

$$< member \rightarrow < name \rightarrow Richard,\ age \rightarrow < > > >$$

is a rewriting of the XML syntax

```
<member>
   <name>Richard</name>
   <age/>
</member>
```

while $FV_{\texttt{member}}(d) = \{< name \rightarrow Richard, age \rightarrow < >>\}$.

4.4 A Language Core

In this Section, we present the language $\mathbf{L}$ for our sample extraction mechanism of Section 4.5. For our purposes, it suffices a language core defined around a value notation, a language of structural types, and a relationship of typing between the two. Extensions to full-fledged languages with value operators or object-oriented types do not present particular problems.

Along the way, we follow a principle of generality that allows extraction mechanisms for other typed languages to be derived from $\mathbf{L}$'s.

We have chosen for $\mathbf{L}$ a selection of the type constructs commonly found in existing programming languages. Available types are constructed from a range of atomic types $B_1, B_2, \ldots, B_N$. They include record, set, and union types, possibly recursively defined. Record constructors are the singleton record type $[l : T]$ and the concatenation $T_1 \wedge T_2$ of two disjoint record types, where two record types are disjoint when they have no field name in common. Set types are denoted by $set(T)$, where T is the member type of the set. Note that we could have chosen bag types that, differently from set types, can describe repeated sub-documents of a given document. The extension does not present particular problems but it slightly complicates the formal treatment and has been avoided here. Recursive types are denoted by $\mu X.T$, where T is the body of the type and X a type variable, and union types by $T_1 \vee T_2$, where T_1 and T_2 are the branch types of the union. Contrary to most languages, union types are untagged, as we do not need tags to describe alternatives in the structure of the data – of course, this does not exclude the use of tagged unions.

Definition 4.4.1 *Let* $\mathbf{T}$ *be the language of* types *generated by the following grammar:*

$$T, T_1, T_2 ::= B_k \mid X \mid [l : T] \mid T_1 \wedge T_2 \mid T_1 \vee T_2 \mid set(T) \mid \mu X.T$$

where $l \in Lbl, X \in Var, k \in [1, n]$.

As for documents, two types are equal if they differ only in the ordering of the components (e.g. $T_1 \wedge T_2 = T_2 \wedge T_1$ and $T_1 \vee T_2 = T_2 \vee T_1$). Similarly, we shall implicitly extend to types the abbreviations, conventions, and auxiliary functions introduced for documents.

The values of $\mathbf{L}$ include the elements of the atomic types, singleton records $[l = v]$, disjoint concatenations $v_1 \wedge v_2$ of two record values, sets $\{v_1, \ldots v_n\}$, and the empty set $\{\,\}$. To improve readability, we do not syntactically distinguish the operations of concatenation of documents, types, and values. The context shall clarify the domains of definition. In the following examples, we shall assume that $\mathbf{L}$ include integer numbers n of type int and strings $"s"$ of type $string$ among its atomic values and types.

Finally, value equality follows the same rules as document and type equality.

Definition 4.4.2 *Let* $\mathbf{V}$ *be the language of* values *generated by the following grammar:*

$$v, v_1, \ldots, v_n ::= b_k \mid [l = v] \mid v_1 \wedge v_2 \mid \{\,\} \mid \{v_1, \ldots, v_n\}$$

where $l \in Lbl$ *and* $b_k \in B_k$.

The relation of typing between values and types is standard and can be defined as follows:

Definition 4.4.3 *Let* $d \in \mathbf{D}, T \in \mathbf{T}$. d has type T *if* $d : T$, *where* $: \subseteq \mathbf{D} \times \mathbf{T}$ *is the* typing relation *inductively defined by the following rules:*

$$b_k : B_k \quad (ATM) \qquad \{\,\} : set(T) \quad (ESET)$$

$$\frac{v_1 : T, \ldots, v_n : T}{\{v_1, \ldots, v_n\} : set(T)} \quad (SET) \qquad \frac{v : T}{[l = v] : [l : T]} \quad (SREC)$$

$$\frac{v_1 : T_1 \quad v_2 : T_2}{v_1 \wedge v_2 : T_1 \wedge T_2} \quad (REC) \qquad \frac{v : T\left[{}^{\mu X.\,T}/_{X}\right]}{v : \mu X.T} \quad (RCS)$$

$$\frac{v : T_1}{v : T_1 \vee T_2} \quad (ULFT) \qquad \frac{v : T_2}{v : T_1 \vee T_2} \quad (URGT)$$

where $T\left[{}^{\mu X.\,T}/_{X}\right]$ *denotes the standard operation of (capture-avoiding) variable substitution*

4.5 An Extraction Mechanism

In this Section, we provide a self-describing interpretation of language values as well as an inclusion relation between documents.

Definition 4.5.1 *Let $sd : \mathbf{V} \rightarrow \mathbf{D}$ be the function defined as:*

$$sd(b_k) = \texttt{textify}(b_k)$$

$$sd([l = v]) = \begin{cases} <> & v = \{\,\} \\ \bigwedge_{i=1}^{n} < l \rightarrow sd(v_i) > & v = \{v_1, \ldots v_n\} \\ < l \rightarrow sd(v) > & otherwise \end{cases}$$

$$sd(v_1 \wedge v_2) = sd(v_1) \wedge sd(v_2)$$

where `textify` *is any function that returns string representations of atomic values.*

The interpretation is straightforward, except perhaps for the case of set values, which are interpreted only within record values. The reason is that set values are not directly supported in $\mathbf{D}$ and must be interpreted in correspondence with repeated field names within complex documents.

For example, the document

$<member \rightarrow< name \rightarrow Steve >,$
$\;\; member \rightarrow< name \rightarrow Fabio >>$

interprets the value

$$[member = \{[name = “Steve''], [name = “Fabio'']\}].$$

In particular, values such as $\{1, 2\}$, $v \vee \{1, 2\}$, or $[a : \{\{1, 2\}, 3\}]$ cannot be interpreted in $\mathbf{D}$ for no field label is available for their interpretation.

Inclusion of documents is susceptible of different interpretations. Here, we have followed the simple intuition according to which d' is included d if d' is syntactically contained in d, with the exception that the empty document is included in any document.

Definition 4.5.2 *Let* $d, d' \in \mathbf{D}$. d' *is* contained *in* d *if* $d' \preceq d$, *where* $\preceq : \subseteq \mathbf{D} \times \mathbf{D}$ *is the relation inductively defined by the following rules:*

$$<> \preceq d \qquad (EMP) \qquad\qquad s \preceq s \qquad (STR)$$

$$\frac{d_1 \preceq d_2}{< l \rightarrow d_1 > \preceq < l \rightarrow d_2 >} \ (SDOC) \qquad\qquad \frac{d_1 \preceq d_3 \quad d_2 \preceq d_4}{d_1 \wedge d_2 \preceq d_3 \wedge d_4} \ (DOC)$$

4.6 Extraction Algorithm

In this Section, we show the algorithm `Ext` that implements the extraction mechanism defined in Section 4.5.

Definition 4.6.1 *Let* $\mathbf{V}_\perp = \mathbf{V} \bigcup \{\perp\}$. *Let* `Ext` $: \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{V}_\perp$ *be the algorithm defined as*

```
Ext($d$, $T$) =

1  if unf($T$)=$\{T\}$
2    case of $T$
3      $B_k$:
4          { if $\exists b_k$ s.t.  textify($b_k$)$=d$
                return $b_k$
5          return $\perp$ }

6      $T_1 \wedge T_2$:  return (Ext($d,T_1$)$\wedge_\perp$Ext($d,T_2$))
```

```
    $[l:T']$, $T' = set(T'')$:
        {if $d \in Str$ return $\perp$
        $A := \emptyset$
        for each $d' \in FV_l(d)$ {
          $v =$ Ext($d', T''$)
          if $v \neq \perp$ $A = A \bigcup \{v\}$ }
        return [l=collect(A)] }

    $[l:T']$, $T' \neq set(T'')$:
        { for each $d' \in FV_l(d)$ {
          $v =$ Ext($d', T'$)
          if $v \neq \perp$ return [l=v] }
        return $\perp$ }

    otherwise:  return $\perp$

else
 { for each $T' \in$ unf($T$) {
    $v =$ Ext($d, T'$)
    if $v \neq \perp$ return $v$ }
 return $\perp$ }
```

where $\wedge_\perp : \mathbf{V}_\perp \times \mathbf{V}_\perp \to \mathbf{V}_\perp$ *is the function defined as:*

$$v_1 \wedge_\perp v_2 = \begin{cases} \perp & v_1 = \perp \text{ or } v_2 = \perp \\ v_1 \wedge v_2 & otherwise \end{cases}$$

Given $d \in \mathbf{D}$ and $T \in \mathbf{T}$, `Ext` performs a recursive analysis of both type and document and it either fails (i.e. returns $\perp$) or else derives a value $v \in \mathbf{V}$ extractable from d according to T. To achieve this, `Ext` solves two main problems.

The first is that set values must be extracted along with record values, for the same reason underlying Definition 4.5.1. This explains why `Ext` processes set types only when processing record types and fails with types such as $set(T) \vee T$ or $set(set(T))$.

The situation is complicated further by the possibility that set types do occur within record types but are 'protected' by union or recursive types. Given the type $[a : set(T) \vee T]$, for example, we cannot recursively delegate to the union case the extraction of a value of type $set(T)$, for we would lose the label a necessary to extract the value set from the document.

`Ext` solves the problem by extracting values only according to record types that are 'flattened', i.e. contain no union or recursive type fields. This is achieved with the preliminary check on line 1, which invokes an implementation of the function unf. unf is a generalisation of the standard operation of one-step unfolding of recursive types. In particular, it one-steps unfolds a union type into the set of its branches, and a record type into the set of records obtained by one-step unfolding all its non-record field values.

Definition 4.6.2 *Let* $unf : \mathbf{T} \rightarrow \mathbf{T}$ *be the function defined as:*

$$unf(T) = \begin{cases} \{T' \left[{}^{\mu X.\,T}/_{X}\right]\} & T = \mu X.T' \\ \{T_1, T_2\} & T = T_1 \vee T_2 \\ \{[l : T''] \mid T'' \in unf_R(T')\} & T = [l : T'] \\ \{T_1' \wedge T_2' \mid & \\ \quad T_1' \in unf(T_1), T_2' \in unf(T_2)\} & T = T_1 \wedge T_2 \\ \{T\} & otherwise \end{cases}$$

where $unf_R : \mathbf{T} \to \mathbf{T}$ *is the function*

$$unf_R(T) = \begin{cases} \{T\} & T = [l : T'] \\ unf(T) & otherwise \end{cases}$$

For example,

$$\begin{aligned} unf([a : int \vee set(string), b : [c : int \vee set(int)]]) = \\ \{\ [a : int, b : [c : int \vee set(int)]], \\ [a : set(string), b : [c : int \vee set(int)]]\}. \end{aligned}$$

A type T is then *unfolded* if $unf(T) = \{T\}$, otherwise is *folded*.

The second problem occurs when multiple values of type T can be extracted from d. Due to the presence of set and union types, this possibility is in fact the norm. For example, consider the document $d =< a \to 1, a \to 2, b \to 3, c \to four >$ and the type $T = T_1 \vee T_2$, where $T_1 = [a : set(int), b : int]$ and $T_2 = [a : set(int), c : string]$.

From Definition 4.2.1 and Definition 4.5.1, it is easy to see that the values $[a = \{1, 2\}, b = 3]$ and $[a = \{1, 2\}, c = "four"]$ are extractable from d according to T_1 and T_2, respectively, and thus according to T. The same is true of the values $[a = \{\ \}, b = 3]$, $[a = \{1\}, b = 3]$, $[a = \{2\}, b = 3]$, etc.

`Ext` returns one extractable value on the basis of both a *best-attempt* and a *first-attempt* policy. Specifically, it returns: (i) the largest value extractable from d according to a set type, and (ii) the first value extractable from d according to one of the branches of union types, when these are ordered from left to right (left-to-right is also the branch ordering followed by the sub-routine `unf`). In the previous example, `Ext` derives the value $[a = \{1, 2\}, b = 3]$.

The best-attempt policy is justified by an immediate *principle of maximisation* of the data contained in a correctly extracted value. The first-attempt policy is instead more arbitrary but it simplifies the definition of the algorithm and is thus adequate for the purpose of a proof of concept.

In lines 21-23, `Ext` implements its first-attempt policy and either fails or returns the first extractable value returned by a recursive execution of `Ext` with d and an unfolding of T.

In lines 2-19, `Ext` processes basic, singleton record, and concatenated record types. For a basic type B_k, `Ext` is successful only if d 'textifies' a value v of type B_k (lines 3-5).

For singleton record types $[l : T']$, `Ext` tries to derive a singleton record value $[l : v]$, where the shape of v depends on whether T' is a set type.

If T' is a set type $set(T'')$ (lines 7-13), `Ext` implements its best-attempt policy and tries to extract a value of type T'' from each l-field value of d. The extracted values are then memorised and eventually grouped into a set value v by the sub-routine `collect`. Notice that, in this case, `Ext` fails only when the document is a string (line 8). Otherwise, it returns at worst the singleton record $[l = \{\ \}]$.

If T' is not a set type (lines 14-18), `Ext` returns the first value of type T' extractable from an l-field value of d. If such value does not exist, `Ext` fails.

If T is the concatenation of two record types T_1 and T_2, `Ext` concatenates the values extracted from d according to T_1 and T_2 respectively. The operation of concatenation $\wedge_\perp$ refines standard concatenation by returning $\perp$ any time one of the operands is $\perp$. This ensures that failing the extraction according to either singleton record type fails the entire process.

`Ext` is clearly terminating, for it recursively operates on subdocuments of the input document and because `unf`, `textify`, and `collect` are trivially terminating. Note that termination holds under the standard assumption that contractive recursive types, such as $\mu X.X$, are not part of the type language (cf. [111]). Next, we shall also prove that `Ext` is sound, i.e. returns a value extractable from d according to T, and complete, i.e. it fails only when no value of type T can be extracted from d.

This completeness result is already sufficient to give users confidence in applications of `Ext`. As shown earlier, however, the first-attempt policy of the algorithm and the presence of union types prevents the user of assuming any relationship between `Ext`'s output and any other value of type T extractable from d. In some cases, this may not be satisfactory. When executed against the document $< a \to 1, b \to 2, c \to foo >$ and the type $[a : int] \vee [a : int, c : string]$, for example, `Ext` returns the value $[a = 1]$ instead of the 'larger' $[a = 1, c = "foo"]$.

4.6.1 Correctness

Lemma 4.6.3 *Let $v \in \mathbf{V}$,$T \in \mathbf{T}$. $v : T$ if and only if there exists $T' \in unf(T)$ such that $v : T'$.*

Proof. By structural induction on $\mathbf{T}$. The proof is immediate and we shall here discuss only the cases $T = T_1 \wedge T_2$.

Assume $v : T$. For typing scheme (REC), $v = v_1 \wedge v_2$, with $v_1 : T_1$, and $v_2 : T_2$. For the inductive hypothesis, $v_1 : T_1'$ and $v_2 : T_2'$, for some $T_1' \in unf(T_1)$ and $T_2' \in unf(T_2)$. For typing scheme (REC) and Definition 4.6.2, $v = v_1 \wedge v_2 : T_1' \wedge T_2' \in unf(T_1 \wedge T_2) = unf(T)$.

Vice versa, assume $v : T'$, with $T' \in unf(T)$. For Definition 4.6.2, $T' = T_1' \wedge T_2'$, with $T_1' \in unf(T_1)$ and $T_2' \in unf(T_2)$. For type scheme (REC), $v = v_1 \wedge v_2, v_1 : T_1', v_2 : T_2'$. For the inductive hypothesis and typing scheme (REC), $v = v_1 \wedge v_2 : T_1 \wedge T_2 = T$.

$\diamond$

Proposition 4.6.4 *Let $d \in \mathbf{D}$, $T \in \mathbf{T}$. If* `Ext`*$(d, T) \neq \perp$ then* `Ext`*(d, T) is extractable from d according to T.*

Proof. By induction on the height h of the execution tree of $\texttt{Ext}(d, T)$.

For the hypothesis, the case $h = 1$ corresponds to one of the cases $T = B_k$ and $T = [l = T']$, with $T' = set(T'')$. In the first case, $\texttt{Ext}(d,T)$= b_k and $sd(b_k) = d$. The thesis follows then from typing scheme (ATM), and the reflexivity of document inclusion, which can be easily proven by structural induction on $\mathbf{D}$. In the second case, $v = [l = \{\,\}]$. For Definition 4.5.1 and inclusion scheme (EMP), $sd(v) = sd([l = \{\,\}]) = <> \preceq d$. For typing schemes (SREC) and (ESET), $v : T$ and the thesis is proven.

Assume now an execution tree of height $h > 1$ and that the thesis is proven for all execution trees of height $h - 1$. Let us distinguish the cases in which T is folded or unfolded.

If T is folded, the hypothesis ensures that $\texttt{Ext}(d,T)$=$\texttt{Ext}(d,T')$= $v \neq \perp$ for some $T' \in unf(T)$. For the inductive hypothesis, $sd(v) \preceq d$ and $v : T'$. For Lemma 4.6.3, $v : T$ and the thesis is proven.

If T is unfolded, there are three cases to examine:

(i) $T = T_1 \wedge T_2$. For the hypothesis and the definition of $\wedge_\perp$, $\texttt{Ext}(d,T)$= $v = v_1 \wedge v_2$, where v_1 =$\texttt{Ext}(d,T_1)$$\neq\perp$ and v_2 =$\texttt{Ext}(d,T_2)$$\neq\perp$. For the inductive hypothesis, $sd(v_1) \preceq d$,$v_1 : T_1$,$sd(v_2) \preceq d$, and $v_2 : T_2$. For Definition 4.5.1 and inclusion scheme (DOC) , $sd(v) = sd(v_1 \wedge v_2) = sd(v_1) \wedge sd(v_2) \preceq d \wedge d = d$. For typing scheme (REC), $v : T$ and the thesis is proven.

(ii) $T = [l : T']$, with $T' = set(T'')$. For $h > 1$, $FV_l(d) \neq \emptyset$. For the hypothesis, $\texttt{Ext}(d,T)$= $v = [l = \{v_1, \ldots, v_n\}]$, where v_i = $\texttt{Ext}(d_i,T'')$$\neq\perp$, $d_i \in FV_l(d)$, for each $i \in [1,n]$ and some $n \in \mathbb{N}$. For the inductive hypothesis, $sd(v_i) \preceq d$ and $v_i : T''$. For Definition 4.5.1 and scheme (DOC) , $sd(v) = sd([l = \{v_1, \ldots, v_n\}]) = \bigwedge_{i=1}^{n} < l \rightarrow sd(v_i) > \preceq \bigwedge_{i=1}^{n} < l \rightarrow d_i > \preceq d$. For typing schemes (SREC) and (SET), $v : T$ and the thesis is proven.

(iii) $T = [l : T']$, with $T' \neq set(T'')$. For the hypothesis, $\texttt{Ext}(d,T) = v = [l = v']$, where $v' = \texttt{Ext}(d',T') \neq \perp, d' \in FV_l(d)$. In particular, $d = [l : d'] \wedge d''$, for some $d'' \in (D)$. For the inductive hypothesis, $sd(v') \preceq d'$ and $v' : T'$. For Definition 4.5.1 and inclusion scheme (DOC), $sd(v) = sd([l = v']) =< l \rightarrow sd(v') > \preceq d$. For scheme (SREC), $v : T$ and the thesis is proven.

$\diamond$

Proposition 4.6.5 *Let $d \in \mathbf{D}$, $T \in \mathbf{T}$, $v \in \mathbf{V}$. If v is extractable from d according to T then $\texttt{Ext}(d,T) \neq \perp$.*

Proof. By induction on the height h of the proof tree of $v : T$.

The case $h = 1$ corresponds to one of the case $v = b_k$ and $v = \{\}$. The second case is excluded by the hypothesis and Definition 4.5.1. In the first case, $T = B_k$, $\texttt{Ext}(d,T) = b_k \neq \perp$ and the thesis is proven.

Assume now an execution tree of height $h > 1$ and that the thesis is proven for all execution trees of height $h - 1$. Let us distinguish the cases in which T is folded or unfolded.

If T is folded, the hypothesis $v : T$ and Lemma 4.6.3 ensure that $v : T'$, for some $T' \in unf(T)$, and thus that v is extractable from d according to T'. For the inductive hypothesis, $\texttt{Ext}(d,T') \neq \perp$ and thus $\texttt{Ext}(d,T) \neq \perp$.

If T is unfolded, there are three cases to examine.

(i) $T = T_1 \wedge T_2$. For typing scheme (REC) there exist $v_1, v_2 \in \mathbf{V}$ such that $v_1 : T_1$, $v_2 : T_2$, and $v = v_1 \wedge v_2$. For the definition of $\wedge$, Definition 4.5.1, inclusion schemes (EMP) and (DOC), and the hypothesis $sd(v) \preceq d$, $sd(v_1) = sd(v_1) \wedge <> \preceq sd(v_1) \wedge sd(v_2) = sd(v_1 \wedge v_2) = sd(v) \preceq d$. Similarly, $sd(v_2) \preceq d$. For the inductive hypothesis, $\texttt{Ext}(d,T_1) \neq \perp$ and $\texttt{Ext}(d,T_2) \neq \perp$. For the definition of $\wedge_\perp$, $\texttt{Ext}(d,T) = \texttt{Ext}(d,T_1) \wedge_\perp \texttt{Ext}(d,T_2) \neq \perp$, and the thesis is proven.

(ii) $[l : T']$, with $T' = set(T'')$. The thesis follows immediately from Ext(d, T)=$[l = v']$, for some $v' \in (V)$.

(iii) $[l : T']$, with $T' \neq set(T'')$. For the hypothesis $v : T$ and scheme (SREC), $v = [l = v']$, and $v' : T'$. For the hypothesis $sd(v) \preceq d$ and Definition 4.5.1, $sd(v) = sd([l = v'] =< l \rightarrow sd(v') \preceq d$. From inclusion scheme (DOC), $d = [l : d'] \wedge d''$, for some $d', d'' \in (D)$, where $sd(v') \preceq d'$. For the inductive hypothesis, Ext(d', T')= $v'' \neq \perp$, Ext(d', T')= $[l : v''] \neq \perp$, and the thesis is proven.

$\diamond$

4.7 Relevance

In this Section, we present a simple relevance quantification scheme for the extraction mechanism defined in Section 4.5. The scheme can be easily embedded in Ext, but we give it here a separate specification to improve readability.

Let us start with a motivating example. Consider the following document d:

$< staff$

  $< member \rightarrow$

    $< name \rightarrow David >$

    $< project \rightarrow$

      $< name \rightarrow SNAQue, \ldots > \ldots >$

    $< project \rightarrow$

      $< name \rightarrow GLOSS, \ldots > \ldots > \ldots >$

  $< member \rightarrow$

    $< name \rightarrow Paolo >$

    $< project \rightarrow$

$$< name \rightarrow Tequyla, \ldots > \ldots >$$
$$< project \rightarrow$$
$$< name \rightarrow TQL, \ldots > \ldots > \ldots > \ldots >$$

and assume that the programmer requires to compute over named projects of staff members.

From an initial analysis of d, the programmer proposes the type T:

$$[staff : [member : set([project : [name : string]])]]$$

and the mechanism returns the value v:

$$[staff = [member = \{[project = [name = "SNAQue"]],$$
$$[project = [name = "TeQuyLa"]]\}]].$$

The programmer did not notice that staff members have more than one project, i.e. d contains more relevant data than T makes possible to extract.

To inform the user, we quantify the *precision* with which T describes the data in d that are relevant to the programmer. To return a readable measure, we distribute it along the singleton record types occurring in T, i.e where loss of relevant data may actually occur. The result is a set of annotations for T that may help the user to refine the type and improve extraction.

In particular, the precision of a singleton record type $[l : T]$ is measured with respect to all the documents that are processed with $[l : T]$ on a successful execution path of `Ext` (by successful execution path, we intend a path of the execution tree along which `Ext` never fails).

Let thus $D_{[l:T]}$ be the set of all such documents, and let $P_{[l:T]}$ be the set defined as:

$$P_{[l:T]} = \{ (d, [l = v]) \mid d \in D_{[l:T]},\ [l = v] = \texttt{Ext}(d, [l : T]) \}.$$

The precision $prec_{[l:T]}$ of $[l : T]$ is then calculated as:

$$prec_{[l:T]} = \frac{\sum_{p \in P_{[l:T]}} vprec(p)}{\mid P_{[l:T]} \mid}$$

where the *value precision* $vprec$ of a pair in $P_{[l:T]}$ is defined as:

$$vprec(d, [l = v]) = \begin{cases} \frac{|v|}{|FV_l(d)|} & \mid FV_l(d) \mid > 0 \\ 1 & \mid FV_l(d) \mid = 0 \end{cases}$$

and, in turn

$$\mid v \mid = \begin{cases} 0 & v = \{\} \\ n & v = \{v_1, \ldots, v_n\} \\ 1 & otherwise \end{cases}$$

Informally, $prec_{[l:T]}$ is the average of the precisions calculated at each pair $(d, [l = v]) \in P_{[l:T]}$. Each of these is in turn the ratio between the number of l-field values from which `Ext` extracted a value and the number of those from which it did not.

In particular, low precision for a singleton $T_1 = [l : set(T')]$ in T suggests that `Ext` did not extract values of type T' from many l-field values in d. A renewed analysis of the data may then reveal that the singleton $T_2 = [l : set(T' \vee T'')]$ allows to extract more relevant data from d and should thus replace T_1. Similarly, $[l : set(T')]$ may do better than a low-precision singleton $[l : T']$.

For example, the precision of type T in the previous example may be returned as the following annotation:

$$^1[staff :^1 [member : set\{^{\frac{1}{2}}[project :^1 [name : string]]\}]]$$

The user may then improve extraction by refining T into the type:

$$[staff : [member : set\{[project : set([name : string])]\}]]$$

which has precision 1 on all its singleton record types.

The problem of relevance quantification is certainly complex and identifies an interesting research topic per se. The scheme presented is fairly simple, and we have introduced it as a proof of concept. Although we have not yet gathered experimental results, we believe that the scheme can be useful with large datasets, where the exact structure of relevant data is not known when the mechanisms is first invoked.

4.8 SNAQue for CORBA

Based on the algorithm `Ext`, we have built a distributed system prototype. The system, maintained at the Computer Science Department of the University of Strathclyde, is called `SNAQue` - *the Strathclyde Novel Architecture for Querying extensible mark-up language*. Although some parts are still under development, the system is currently being tested in a number of biodiversity projects by the Palaeobiology Research Group at the University of Glasgow.

SNAQue is a CORBA application that implements an extraction mechanism for a subset of the *CORBA Interface Definition Language* [8], and therefore for any CORBA-compliant language (e.g. C, C++, Smalltalk, Java, Ada95, etc.).

The correctness of the system relies directly on the correctness of the extraction mechanism defined in Section 4.5. In particular, SNAQue receives an XML document and an IDL type description from a remote client, and maps them onto a document $d \in \mathbf{D}$ and a type $T \in \mathbf{T}$, respectively. It then invokes `Ext` and transforms the output $v \in \mathbf{V}$ of type T in a number of interrelated CORBA objects. By virtue of the mapping from IDL to $\mathbf{T}$, the objects expose interfaces corresponding to the initial IDL description.

There is always one entry point to the generated CORBA objects: the object o that corresponds to the root of the XML document. A reference to o is returned to

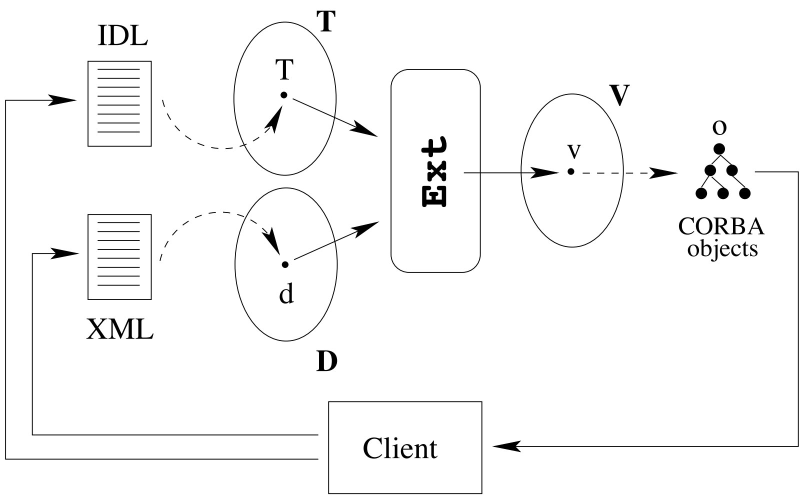

Figure 4.2: SNAQue for CORBA

the client for local binding in programs written in any CORBA-compliant language of choice. Optionally, SNAQue may register *o* with a public alias provided by the client. Then any client informed of the alias can come along and gain remote access to *o* (see Figure 4.2).

The details of the mappings from XML to $\mathbf{D}$, IDL to $\mathbf{T}$, and $\mathbf{V}$ to the corresponding CORBA objects are out of the scope of this paper. Roughly, the mapping from XML concentrates on the data-oriented features of the format, while the mapping from IDL converts interfaces, sequences, and tagged union types, into records, sets, and untagged union types in $\mathbf{T}$.

As an example, consider the document d and its fragment d' introduced in Section 4.2. Using SNAQue, it is straightforward to compute over d' with a code analogous to that shown in Section 4.2. One has only to provide SNAQue with the following IDL type description:

```
interface Staff {
   typedef sequence <Member> MemberSeq;
   attribute MemberSeq members; }

interface Member {
   attribute String name;
   attribute long code; }
```

With respect to these inputs, SNAQue will create four CORBA objects: one conforming to the `Staff` interface and three conforming to the `Member` interface. It will then return a reference to the first object to which clients can bind in their programs.

SNAQue chooses Java to implement the extracted CORBA objects. In particular, the Java classes generated by the system for the objects derived above are exactly the `Staff` and `Member` classes shown in Section 4.2. Similarly, the Java-like code suggested there could be immediately used to compute over the data.

The choice of CORBA IDL as the type language is an obvious one. It increases the applicability of the extraction mechanism for **L** and makes it distributed. However, the distributed nature raises performance issues, such as those related with every read or write operation performed on the data across the network.

We are currently investigating two ways of tackling such problems. On the one hand, we are considering the use of *value types*, which have been recently introduced into CORBA to allow objects to be passed by value, rather than by reference. This allows clients to *pull* the generated CORBA objects over the network and inject them in the local environment. On the other hand, we could push computations to the server, by allowing clients to specify additional methods in the IDL interfaces. For example, the `Staff` interface could be extended with a new method:

```
interface Staff {
   typedef sequence<Member> MemberSeq;
   attribute MemberSeq members;
   Member getMember(in String name) }
```

SNAQue can not automatically generate the implementation for this method, which has to be provided by the client. Due to the lack of space, we cannot discuss this facility in more detail.

4.9 Related Work

The differences between extraction mechanisms and existing approaches has been largely discussed in Section 4.2. It is worth noticing here that extraction mechanisms operate on arbitrary XML documents and can thus be easily coupled with untyped query languages. In an integrated environment, the convenience of the first would complement the flexibility of the second for data with a varying degree of structural regularity.

Other high-level bindings between XML and existing programming languages have been recently presented [112, 15]. They all map some form of data description (usually a DTD or an XML Schema) onto language types that capture directly the semantics intended for the data. For this reason, they operate on fairly regular XML documents and do not provide facilities for extracting regular subsets from arbitrary documents. In addition, they have been developed for specific languages (e.g. Haskell, Java) and do not generalise.

The idea of exploiting regularity in XML, and more generally, semistructured data, has also motivated a number of approaches.

Early proposals were for extending standard database technology to accommodate some degree of irregularity in the data, typically via the provision of union types. Although similar in motivation, such approaches differ from ours in their attempt to provide a total description of the data. As mentioned in Section 4.2, significantly irregular data lead to an uncontrolled use of union types, thereby progressively decreasing system performance and complicating program specification.

Later proposals assume a dedicated query language as a starting point, but differ in their data-first or type-first strategy.

Approaches of the first kind infer type information from existing datasets. In this case, type inference can be performed by the system for the entire database, automatically or semi-automatically [96, 98, 113, 102, 114]. The resulting types are mainly for users to understand the data and, to some extent, for query optimisers to improve execution [101]. Partial inference can be also performed by users, and the results then fed to the system as hints to reduce the scope of a search [97]. Overall, inference-based approaches exploit typing for resource optimisation, while computations remain essentially untyped.

Approaches of the second kind exploit static knowledge to guarantee computational safety [115, 116, 117, 81]. To achieve this against a tree-based model, they resort to low-level types for XML documents. Due to the support of regular expressions, such *tree types* are more flexible than high-level types in capturing irregularities in the data [18, 118].

To the best of our knowledge, Ozone [75] is the only attempt to seamlessly integrate structured and semistructured data in the same typed environment. The system extends the ODMG model to include semistructured data, and allows structured objects to be queried with semistructured primitives. Interestingly, it also supports a function for coercing semistructured data to structured objects according to a type and, as such, implements a simple extraction mechanism for ODMG. However, our mechanism is proved correct and returns values of a larger set of types.

4.10 Conclusion and Future Work

We have presented a novel approach to programming over XML data based on language bindings. The bindings are defined as mechanisms that identify and derive language values from subsets of arbitrary XML documents. When programming over such subsets, the approach delivers the computational advantages associated with the host language. Furthermore, the derived values preserve the semantics intended for the data, and thus facilitate program specification.

These mechanisms can be formally defined and correctly implemented, and we have done it for a sample but representative core language. In particular, we have proven the generality of the sample mechanism by deriving extraction mechanisms for all CORBA-compliant languages directly from it.

Future research directions concern both theoretical and practical aspects of the investigation. Beyond XML, we have already extended our results to more general forms of semistructured data. In particular, we are able to extract language values from graph-structured data, i.e. in the presence of cycles and sharing. The interested reader is referred to [25] for the full treatment.

Another interesting direction relates to the definition of inclusion between documents. The one we proposed follows first intuitions, but alternative definitions could be considered. As a first example example, inclusion checks may start from arbitrary elements of the target document, not necessarily the root element. This would save the user the often tedious task of describing the structure that leads from the root of the document to the data of interest. It would also give a hint of the flexibility achieved by navigational query paradigms without reducing the advantages of the approach.

The extraction algorithm has been proved sound but the belief that it is also tractable in pragmatic terms has not been supported by a formal analysis of its complexity. Although tests on large data samples have shown acceptable performance even on desktop machines, the impact of a considerable use of union types remains to be measured. Furthermore, we are currently working with back-tracking techniques towards algorithms with stronger properties of completeness.

Relevance quantification could be certainly improved over the sample scheme proposed in Section 4.7. In particular, an extraction mechanism could customise a general scheme to the programmer's specifications.

Finally, SNAQue is under continuous development, and a web interface to the system is being published at the time of writing. The research agenda is currently focusing on whether values can be virtually injected in the value space of the target language rather than materialised. Single or multiple indexes to regular subsets suggest the possibility to dynamically synchronise the interface between the language and the database under updates. At the same time, they raise the opportunity for incremental extractions.

In addition, the client/server scenario raises a number of questions related to the efficiency of the system and to the possibility of integrating data from distributed XML servers.

Investigation is needed to identify the cases in which it is more convenient to pull extracted values at the client side or else push client computations to the server. Completeness quantification could here be used by both client and server to make intelligent decisions about data or code migration.

In addition, the possibility of storing and publishing typed interfaces over the data at the server side suggests interesting data protection and data evolution policies. For example, the usage and volume of XML data referenced through the interfaces could be gathered into statistical information that may be used to assess the impact of changes to the data.

Chapter 5

Language Bindings to XML

This Chapter reports the contents of the following publication:

> SIMEONI, F., LIEVENS, D., CONNOR, R. C. H., AND MANGHI, P. Language bindings to XML. *IEEE Internet Computing 7*, 1 (2003), 19–27.

Here, the theme of typeful programming is illustrated with a systematic comparison of mainstream approaches to XML parsing [16, 17]. A new class of data binding tools enters the arena of XML processing and defines an ideal point of reference for motivating and applying type projections. A deeper understanding of typeful programming reveals the tight-coupling induced by models that are dictated by data producers. This forms the basis for discussing the advantages that type projections retain over mainstream approaches.

The prototypical application in Chapter elsevier evolves as a general-purpose data binding tool for Java, and a correspondence between the type system of the idealised language and Java's bridges the design of the tool to the formal results presented in previous work.

5.1 Abstract

In this paper, we investigate the issues that arise when binding statically typed languages to XML data. In particular, our motivation is to exploit the computational facilities of mainstream languages when computing over real-world entities encoded as XML documents or document fragments. These include completeness, strong typing, efficiency, as well as user-base and support.

We first show that standard binding solutions, such as the SAX and DOM APIs, do not preserve the semantics of such entities, and thus hinder program specification, verification, and optimisation. We then compare two novel approaches, which rely on type information to preserve semantics. The first is Sun's JAXB architecture, in which types are automatically generated from document descriptions. The second is our SNAQue architecture, where types are directly specified by binding computations. For certain classes of applications, we show that the latter offers substantial advantages in terms of simplicity and flexibility.

In previous work [119], we have formally proven that SNAQue bindings can be correctly built for a representative, canonical language. Here, we extend that work and present SNAQue/J, a binding mechanism specific to the Java language.

5.2 Introduction

The significance of the eXtensible Markup Language extends beyond the community it was originally intended to serve, namely document processing over the Web. As a standard and widely supported format for arbitrary data, XML delivers to any heterogeneous, distributed computer system the common lingo for data exchange which only closed or relatively closed systems could previously assume and enforce.

In addition, XML data contains a description of its intended meaning, either in the form of element names or, optionally, as an embedded or referenced type description (e.g. a DTD). Self-description allows programs to interpret the data dynamically, i.e. with no reliance on shared assumptions or agreements (e.g. header files or interface definition languages). In turn, this promotes system flexibility, for programs become more resilient to irrelevant changes in the data.

In its untyped form, XML is also an ideal carrier for *semistructured data*, where the structure is too irregular or instable to be effectively handled in statically typed programming languages and database management systems [10].

5.2.1 Motivations

This broad view over XML explains the proliferation of related technologies, especially programming models. To date, computations over XML data can be specified in a variety of paradigms and languages.

Most approaches, however, have so far focused on novel and dedicated solutions. These include flexible query languages, which resort to regular expressions to match data with irregular or partially known structure [109, 120, 79]. They also include Turing-complete and/or strongly typed functional languages, which exploit structural regularity to ensure correctness of arbitrary computations [69, 70, 115].

In contrast, the computational facilities of mainstream, statically typed languages have only been partially reused in this novel context. Among these, the completeness and simplicity of the procedural and object-oriented programming models, the reliability and efficiency of computations, and equally important, the large user-base and tool support.

In particular, we identify two broad scenarios where XML and typed languages could be conveniently bridged. The first accounts for most of current XML usage, namely the exchange of business data across proprietary boundaries. The second concerns semistructured documents with large fragments of more disciplined data, such as those that arise in scientific domains or from the integration of heterogeneous databases.

In the general case, the goal is to compute over representations of real-world entities which are maintained or even generated outside the language jurisdiction, and occur as fragments of (possibly semistructured) XML documents. For example, consider the following document d about staff members of a university department:

```
<staff code="123517">
   <member>
      <name>Richard Connor</name>
      <home>www.cis.strath.ac.uk/~richard</home>
   </member>
   <member code="123345">
      <name>Steve Neely</name>
      <ext>4565</ext>
      <project>
         <name>SNAQue</name>
      <project/>
   </member code="175417">
   <member>
      <ext>4566</ext>
      <name>Fabio Simeoni</name>
   </member>
</staff>
```

For the sake of illustration, a few irregularities have been added to the data: some staff members have homepage information, others have projects or phone extensions. On a much larger scale, these irregularities would prevent the document from being conveniently managed within statically typed frameworks. While union types and object-oriented hierarchies may accommodate some of the irregularity, their extensive use would soon reduce the static knowledge of the system and complicate program specification and maintenance.

However, all members in d do have a name and a code. In particular, the following fragment d' of d may be an XML representation of a standard language value:

```
<staff>
   <member code="123517">
      <name>Richard Connor</name>
   </member>
   <member code="123345">
      <name>Steve Neely</name>
   </member>
   <member code="175417">
      <name>Fabio Simeoni</name>
   </member>
</staff>
```

For most statically typed object-oriented languages, for example, d' may be an XML encoding of the state of an object `staff` of class `Staff`, where

```
class Staff {
      private Member[] member;
      Member[] getMembers() {...}
      void setMembers(Member[] members) {...}
      ...}
```

and

```
class Member {
      private String name;
      private int code;
      String getName() {...}
      void setName (String n){...}
      int getCode() {...}
```

```
    void setCode (int c){...}
    ...}
```

This simple observation raises the expectation that programming over d' be at least as simple, safe, and efficient as programming over `staff`. Furthermore, one would like these good properties to scale, i.e. hold for generalised computations over fragments of XML documents considerably larger than d'.

Motivated by such computational requirement, we advocate the importance of *language bindings* to XML, i.e. software mechanisms which transform XML data into values that programmers can access and manipulate from within their language of choice.

However, the computational facilities of the host language do not guarantee per se the convenience of a binding. It is also necessary that the resulting values support the interpretation of the data required by computations. One aim of this paper is to show that, when this is not the case, the benefits of expressiveness, strong typing, and efficiency can only be partially delivered to the programmer.

5.2.2 Overview

In this paper, we advance and motivate the claim that standard binding solutions, namely the *Simple API for XML* and the *Document Object Model API* are not well suited to compute over XML-encodings of real-world entities. In Section 5.3, in particular, we show that they do not preserve the semantics of entities when passing from the XML to the language representation.

We then compare two novel approaches. In Section 5.4, we discuss Sun's JAXB architecure [15], while in Section 5.5 we introduce SNAQue, an architecture of our own definition. Both approaches rely on type information to preserve the semantics of real-world entities. In JAXB, language types are statically generated from document descriptions, in the style of middleware solutions for heterogeneously typed frameworks. In SNAQue, they are directly specified by the binding computation, in the style of statically typed languages with dynamic types [1].

In previous work [119], we have formally proven that SNAQue bindings can be built for a representative, canonical language. Here, we extend that work and present SNAQue/J, a binding mechanism specific to the Java language . This is done is Section 5.6, while in Section 5.7 we draw some conclusions and outline further research directions.

5.3 SAX and DOM

We assume pure Java implementations of the APIs and then consider a generic computation over the names and codes of the staff members that occur in the document d of Section 5.2.

From a binding perspective, SAX transforms an XML document into a string served to the programmer as a temporal sequence of tokens. The induced programming model is one based on parser-generated events and programmer-implemented callbacks.

The SAX programmer is thus invited to share the interpretation of the data of the underlying parser, albeit at a higher level of abstraction. In particular, the programmer receives only tokens that correspond to distinguished elements of the format, such as `PCDATA` sections or element names. The following may be a SAX program that iterates over staff members to retrieve their names and codes:

```
class SaxTask extends DefaultHandler {
  private String name;
  private int code;
  private boolean inProject;
  private CharArrayWriter buffer = new CharArrayWriter();
  public void characters (char[ ] ch,int start,int length)
    {buffer.write(ch,start,length);}

  public void startElement (String uri,String name,String qName,
                                              Attributes atts){
    if (name.equals("member")) code=atts.getValue("code");
    if (name.equals("project")) inProject = true;
    buffer.reset();}

  public void endElement (String uri,String name,String qName){
     if (name.equals("project")) inProject = false;
     if (name.equals("name") && !inProject) name = buffer.toString().trim();
     ...do something with name and code...}}
```

Instead, DOM represents an XML document as a node-labelled tree where nodes correspond to the distinguished elements of the format. The DOM programmer is thus invited to interpret the data as the document in which it is contained, i.e. in terms of the 'has-a' relationships between its structural components. The following is a DOM solution to the previous task:

```
int code;
String name = null;
Element staff = d.getDocumentElement();
NodeList members = staff.getElementsByTagName("member");
int memberCount = members.getLength();
for (int i=0;i<memberCount;i++) {
    Element member = (Element) members.item(i);
    code = Integer.parseInt(member.getAttribute("code"));
    NodeList children = member.getChildNodes();
    int length = children.getLength();
    for (int j=0;j<length;j++) {
        Node child = children.item(j);
        if (child.getNodeType() == Node.ELEMENT_NODE) {
              String tagName=((Element) child).getTagName();
              if (tagName.equals("name")) name =
                    ((characterData) child.getFirstChild()).getData();}
                    ...do something with name and code...}}
```

Even for such a simple task, both programming solutions are quite convoluted. Facing problems on a much larger scale, SAX and DOM programmers have described their code as tedious, hard to read and maintain, and thus prone to errors.

Partly, this is due to the document-oriented nature of any complete programming interface to XML. Conceptually related data (e.g. names and codes of staff members) must be accessed with the different algebras of elements and attributes. Similarly, manipulating atomic data requires an implicit or explicit cast from the type of strings, the only available. The main problem, however, is that the computation is expressed in an algebra of strings and trees, while domain-specific concepts (e.g. staff names and codes) are relegated to the role of actual parameters.

In SAX, for example, one does not ask for the name of a member, but patiently collect the characters of `name` elements, and only when one has made sure it is the right `name` element. In DOM, one has first to reach a member node, then scan its children nodes to get a name node, and eventually get its string value. SAX does not even reflect the structure of the data and in order to get things right one has to coordinate the action of possibly distant methods.

Inadequate data abstractions compromise also static checking of computations. Correctness can be guaranteed for generic operations on strings and trees, not staff members. The DOM invocation:

```
staff.getElementsByTagName("mebmer");
```

where `"mebmer"` is a typo for `"member"`, would silently compile and return a `null` value at run-time. Safety becomes thus responsibility of the programmer, not the system. Programmatic checks worsen code readability and maintainability, and are not always sufficient to guarantee correct behaviour. In the lack of some document description, the typo may be interpreted as the absence of required data, and thus trigger unintended behaviour. Even assuming some document description, the typo may accidentally identify some other data or, in the best case, be simply signalled at run-time. For similar reasons, the system can optimise resources only within the limits of its static knowledge of the data. For instance, it ignores the fact that all staff members have names, codes, and ages.

Compare this with the code required to perform the same task over the object `staff` equivalent to d':

```
Member[] members = staff.getMembers();
for (int i=0;i<members.length;i++) {
 int code = members[i].getCode();
 String name = members[i].getName();
 ...do something with name and code ...}
```

The code is now aligned with the semantics of the task. It is also more succinct, for generic operations on staff and staff members can be easily factored out in class definitions, thoroughly tested, and then reused. The programmer can also count on finer-grained system type-checking to ensure code correctness: the age has the properties of an integer number, all members have one and only one name, while erroneous manipulations are detected at compile-time.

In conclusion, SAX and DOM do not preserve the semantics of XML-encodings of real-world entities. When computations need to address the syntax or the structure of the data, respectively, both solutions can be very effective. For example, SAX is efficient and simple when it comes to count the number of elements or the occurrences of a particular string within the document. Similarly, DOM is ideal when it comes to add or remove ‘has-a’ relationships between document components.

A large class of computations, however, are directly concerned with the intended meaning of the data, not its syntactic or structural properties. In these cases, the data should be represented within the language by values whose semantics reflects the intended meaning as closely as possible. When a binding does not satisfy these expectations, the programmer faces the choice of expressing computations in an awkward algebra or writing ad-hoc translation code between the available values and the desired ones.

For example, many SAX and DOM programmers would prefer to tackle the proposed task by mapping the string or tree generated by the parser into `staff`, and then compute directly over `staff`. In a statically typed language, this requires the preliminary declaration of type `Staff` as a description of the semantics intended for d'. This suggests the possibility of defining automated solutions that take types as the input to semantic-preserving bindings to XML data.

5.4 JAXB

In the *Java Architecture for XML Binding*, type information is automatically generated from document descriptions, such as DTDs. Specifically, descriptions are converted into classes with *unmarshalling* functionality, i.e. they can recursively generate their own instances from XML documents which are valid with respect to the generating descriptions.

For example, suppose the fragment d' of d shown in Section 5.2 exists as a standalone document associated with the following DTD *STAFF*:

```
<!ELEMENT staff (member*)>
<!ELEMENT member (name)>
<!ATTLIST member code CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
```

The JAXB programmer passes *STAFF* to a *schema compiler*, which generates the declarations for two classes `Staff` and `Member`. These are similar to those declared in Section 5.2 but, besides access methods to private properties, they also include a static method `unmarshal` for binding to XML-encodings of its instances.

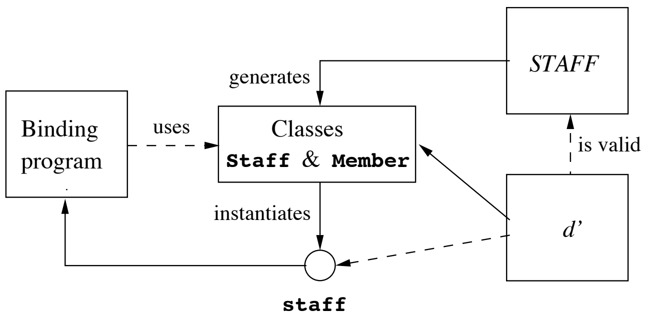

Figure 5.1: JAXB Bindings

The programmer may then add domain-specific functionality to the generated classes, and use them in programs. To compute over d', applications would then invoke the `unmarshal` method of `Staff` on d'. Assuming that `doc` denotes the file that contains d', the instantiation of d' would look like:

```
Staff staff = Staff.unmarshal(doc);
```

Thereafter, applications can manipulate `staff` with the standard facilities of the host language (cf. Figure 5.1).

Binding is successful only if the target document is valid w.r.t. the initial DTD, otherwise an exception is raised. This is guaranteed by the schema compiler, which augments the binding logic within the generated classes with programmatic checks equivalent to the constraints expressed in the DTD. Validity is thus enforced at run time, with a single dynamic check in an otherwise statically typed program.

In addition, the generated classes are capable of *marshalling* their instances back into XML documents, thereby implementing a lightweight, file-based update model.

Object marshalling is also validated, in that it generates only documents that conform to the DTD that initiated the binding.

Finally, the JAXB programmer may also feed a *binding schema* to the compiler in order to specialise its binding strategy. An XML application itself, the binding language gives control over the generation of class names, properties and methods, as well as type conversions, constructor functions, type-safe enumeration classes and even interfaces. For example, a simple binding schema may specify that `Staff` objects encapsulate arrays of `Member` objects instead of lists, and that their `code` property is an integer rather than a string.

Although JAXB succeeds in preserving data semantics, it has a number of shortcomings. Some of these are certainly related to the early implementation status, such as the instable and incomplete functionality, including the limited support for description formalisms. Others occur more noticeably in the form of loss of type information, for the constraints imposed by DTDs are enforced at the point of marshalling, rather than when they are actually violated.

The main issues, however, are directly raised by the dependency on document descriptions of the underlying architecture. Firstly, programs cannot directly bind to document fragments in order to reflect the distribution of computational tasks across the system or to access regular subsets of semistructured documents. For example, JAXB programs cannot bind directly to d' within the more irregular d introduced in Section 5.2, but they have to provide a DTD for the entire d and bind to it. Given the irregularities in d, the DTD would probably be rather convoluted and thus compile into classes that offer either poor or excessive abstraction over the actual data. Such classes would provide little static knowledge to the system and a cumbersome, inefficient, cast-based algebra to the programmer. Overall, this solution would result in the introduction of semistructured data into a statically typed language, which is notoriously an inconvenient match. Describing staff members, for example, would require either a single class with the totality of required properties, or else a rather unnatural class hierarchy of `Member`, `MemberWithHomepage`, `MemberWithPhone`, `MemberWithNameAndPhone`, etc. In both cases, computations would have to make extensive use of type casts and/or conditional statements.

Document-level bindings show also poor resilience to changes in the data. Even when change is irrelevant to existing programs, the classes generated from old document descriptions become invalid and must be regenerated. Besides the problems it may imply in loosely-coupled systems, class regeneration discourages programmers from adding domain-specific functionality to the generated classes. For example, a method `find` which takes the name of a member and returns the corresponding `Member` object should not be directly declared for `Staff` objects, for future class generations would simply make it vanish. Solutions must therefore be based on wrapper classes or extension patterns, which makes it necessary for the programmer to be aware of the internal workings of the binding architecture.

Because of the automatic class generation, this complexity is in fact propagated throughout the binding process. In order to get control on the generation, the programmer must learn a new binding language that becomes complicated in most non-trivial cases. Programming control remains partial due to the unavoidable tension between the generality of the mechanism and the specificity of the computational requirements. Finally, the generated classes remain rather opaque to the programmer and can prove difficult to integrate with legacy code.

5.5 SNAQue

In the SNAQue architecture, the *Strathclyde Novel Architecture for Querying the eXtensible Markup Language*, document descriptions play no role and type information is directly projected over XML documents by binding programs.

Specifically, a SNAQue binding is completely defined by a type and a target document. The type –which may be the result of an inspection of the document or, when available, a document description– is then projected over the document in an attempt to find a conforming document fragment. This entails a recursive match between the type structure and the names of elements in the document. If the match is successful, the conforming fragment is transformed into an instance of the projected type and finally returned to the binding program (see Figure 5.2).

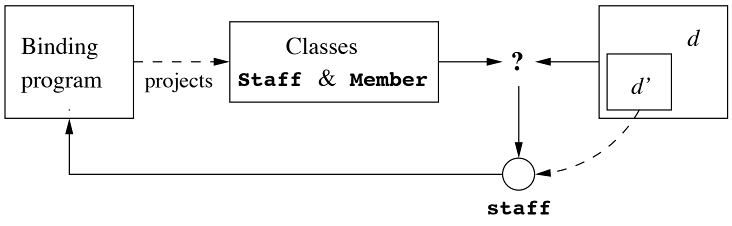

Figure 5.2: SNAQue Bindings

SNAQue is thus similar to the language mechanisms which have been extensively used in statically typed languages to guarantee type-safe access to heterogeneous or persistent data. These include the numerous language incarnations of infinite, untagged union types (e.g. the subclass structure of Simula-67 [121] or the type `dynamic` of Amber [59]), especially their implementations in persistent languages (e.g. the type `any` of Napier88 [4]). Instead of performing a dynamic type check between the type projected by the computation and the type of the data, however, SNAQue compares the first directly with the self-describing document.

The programmer has now full control on the projected types –which may have been defined for the purpose of binding or simply reused from legacy code– and the design in which they should participate. Bindings can be defined with an arbitrary degree of granularity, and thus give access to fragments of possibly semistructured XML documents. Finally, irrelevant changes to the data have no impact on projected types or binding programs.

At the time being, the SNAQue architecture focuses on bindings and does not include an update model over the bound data. However, such possibility is not precluded and, in fact, SNAQue could offer a finer grained model than JAXB. This may introduce validity issues, for the full range of document constraints cannot be captured by language types but requires programmatic checks. While JAXB automatically generates the checks from document descriptions, SNAQue would leave the responsibility to binding programs.

5.5.1 An Example

With reference to the definitions in Section 5.2, consider an object-oriented program that wants to bind to d' using SNAQue. The program declares classes `Staff` and `Member` and then projects `Staff` on d. The binding mechanism infers a structural type from `Staff` and begins a recursive analysis of both type and document in an attempt to match their structure.

Since `Staff` has only an instance variable `member` of class `Member[]`, the mechanism considers only homonymous elements immediately below the `staff` element. At the second step, it repeats the analysis between the type inferred from class `Member` and the ‘sub-documents’ of d rooted in each of the `member` elements identified at the previous step, e.g.:

```
<member code ="123345">
   <name>Steve Neely</name>
   <ext>4565</ext>
   <project>
      <name>SNAQue</name>
   <project/>
```

```
</member>
```

In particular, the mechanism looks in each sub-document for at least a `name` element and a `code` attribute. At the third step, it repeats the analysis comparing the atomic type `String` and `int` with the CDATA and PCDATA content of, respectively, the `name` elements and `code` attributes identified at the previous step. The comparisons are all successful because any PCDATA content is a string while the CDATA value of all `code` attributes can be converted into an integer.

At this point, the fragment of d that conforms to the type inferred from class `Staff` is that obtained by traversing d along the elements and attributes successfully considered by the recursive analysis, i.e. d'. The last step of the mechanism is then to convert d' into the `Staff` object `staff` shown in Section 5.2, and finally return `staff` to the binding program (see Figure 5.1(b))[1].

5.5.2 Formal Definition and Canonical Specification

The SNAQue architecture can be more formally defined as follows. Let $\mathbf{D}$ be the set of XML documents, $\mathbf{L}$ a statically typed programming language, and $\mathbf{V}$ and $\mathbf{T}$ the value and type spaces of $\mathbf{L}$, respectively (see Figure 5.3). Let $sd : \mathbf{V} \rightarrow \mathbf{D}$ be a mapping which gives an interpretation of language values as self-describing documents. Let also $\preceq$ in $\mathbf{D} \times \mathbf{D}$ a relation of containment between XML documents which gives an interpretation of the notion of document fragments.

Definition 5.5.1 *Let $v \in \mathbf{V}$. v is* extractable *from $d \in \mathbf{D}$ according to $T \in \mathbf{T}$ if: (i) v has type T, and (ii) there exists $d' \preceq d$ such that $sd(v) = d'$.*

Finally, a SNAQue *binding mechanism* for $\mathbf{L}$ takes both a document d and a type T, and returns a value v extractable from d according to T, if one exists.

[1]More detailed examples of SNAQue bindings can be found in Section 5.6.

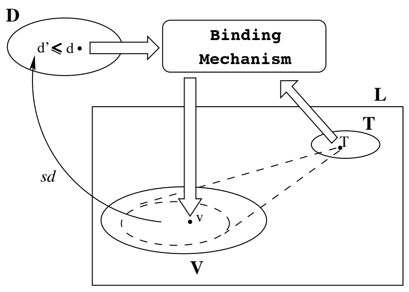

Figure 5.3: The SNAQue architecture

In [119], we have shown that the SNAQue architecture can be correctly implemented, and we have done it for a canonical *language core* $\mathbf{L}$. In doing so, we have followed a principle of generality that would allow binding mechanisms for other typed languages to be derived directly from $\mathbf{L}$'s. In Section 5.6, in particular, we will extend such canonical mechanism to one for the Java language. For later reference, we summarise here the characteristics of the canonical mechanism.

The canonical language $\mathbf{L}$ is defined around a value notation $\mathbf{V}$, a type language $\mathbf{T}$, and a relationship of typing between the two. In particular, we have chosen for $\mathbf{T}$ a selection of the first-order types commonly found in existing procedural languages. Canonical types can be recursively built from a set of atomic types, and include *record*, *set*, *union*, and *recursive* types. Specifically, a type $T \in \mathbf{T}$ is one of the atomic types B_1, B_2, …, B_N, a record type $[l_1 : T_1, \ldots, l_n : T_n]$, a set type $set(T)$, a union type $T_1 + T_2$, or a recursive type $\mu X.T$, where X is a type variable and the operator μ binds occurrences of X in T.

Canonical values mirror the available types. A value $v \in \mathbf{V}$ is an atomic value $b_k \in B_k$, a record value $[l_1 = v_1, \ldots, l_n = v_n]$, a set value $\{v_1, \ldots, v_n\}$, or the empty set $\{\}$. The typing relation is inductively defined in a standard fashion. An atomic value b_k has the corresponding type B_k, while a record $[l_1 = v_1, \ldots, l_n = v_n]$ has the type $[l_1 : T_1, \ldots, l_n : T_n]$ only if each v_i has type T_i. A set $\{v_1, v_2, \ldots, v_n\}$ has the type

$set(T)$ only if all the v_i have type T, while the empty set has the type $set(T)$ for all T. A value v has then type $T_1 + T_2$ if v has type T_1 or type T_2 and, finally, v has type $\mu X.T$ if v has the type obtained by substituting $\mu X.T$ for all the bound occurrences of X in T.

For the specification of the canonical mechanism, we have defined a fairly simple self-describing interpretation of canonical values, although the lack of XML support for set values introduces some subtlety in the definition of sd. We have then defined the relation of document containment $\preceq$ according to the intuition that a document is a fragment of another if they are both well-formed and the first is syntactically contained in the second, as in the case of d' and d of Section 5.2.

In the canonical binding mechanism, for example, the binding shown in Section 5.5.1 would require the projection of the following canonical type over d:

$$[member : set([name : String, code : int])]$$

and result in the identification of the following canonical value equivalent to d':

$$\begin{aligned}[member : \{&[name = "RichardConnor", code = 123517],\\ &[name = "SteveNeely", code = 123345],\\ &[name = "FabioSimeoni", code = 175417]\}]\end{aligned}$$

Finally, we have given an algorithm `Ext` that takes an XML document d and a canonical type T, and returns a canonical value v which is extractable from d according to T. In particular, we have proven the soundness and completeness of `Ext` with respect to the canonical specification.

5.6 SNAQue/J

In this Section, we briefly discuss a prototype implementation of the SNAQue architecture for the Java language. *SNAQue/J* consists of an API with a single public class

`SNAQueJ` which exposes a static method `bind` for binding to fragments of XML documents. In particular, `bind` takes a `Class` object and a reference to an XML document, and returns an instance of the class corresponding to the `Class` object.

Consider the example of Section 5.5.1 and assume that `clStaff` and `doc` denote the `Class` object corresponding to `Staff` and the document d, respectively. SNAQue/J can then be invoked as simply as:

```
Staff staff = (Staff) SNAQueJ.bind(clStaff,doc);
```

If the binding fails an exception is raised, otherwise the program can downcast the returned object to the projected class.

SNAQue/J is directly derived from the canonical mechanism discussed in Section 5.5.2. In particular, the projected classes are mapped onto canonical types before `Ext` is invoked. The mapping is then used to convert the canonical values returned by successful bindings into equivalent graphs of Java objects (see Figure 5.4). The same extension scheme can be used to derive implementations of the SNAQue architecture for other typed languages.

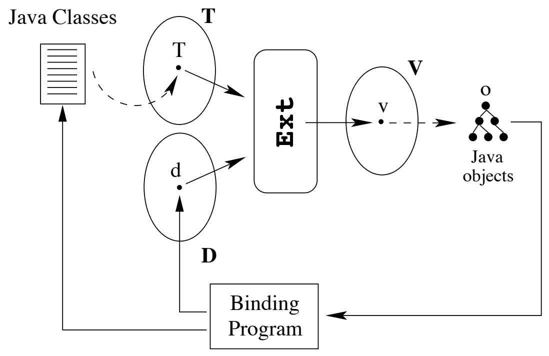

Figure 5.4: The SNAQue/J Architecture

SNAQue/J is thus completely defined by the mapping between Java classes and canonical types. The mapping is established at binding time, by a reflective analysis of the projected class. Roughly, the mapping behaves like the identity function on atomic types while it associates classes with record types, possibly recursively defined. For example, class `Member` would map onto the record type $[name : string, code : int]$.

We derive union and set types by inferring additional information at binding time. In particular, SNAQue/J maps an array class or any class that implements the `Collection` interface onto a type $set(T)$, using the names of instance variable to deduce the type T. Consider, for example, the following version of class `Member`:

```
class Member {
      String name;
      int code;
      List project;}
```

where the intended class of objects in list `project` is the following:

```
class Project {
      String name;
      Member coordinator;}
```

When `Member` is projected over an XML document, SNAQue/J uses the name `project` of the `List` variable in `Member` to infer the following recursive type:

$$\mu X.[name : String, code : int, project : set([name : String, coordinator : X])]$$

Inferring union types is less trivial, for Java –like most object-oriented languages– offers no direct support for them. Partly, this is due to the fact that much of the flexibility of union types can be achieved with inheritance. In particular, we infer *non-disjoint* union types from class hierarchies and *disjoint* union types from structurally disjoint classes which implement the same interface, possibly empty. For example, if the following class is in scope when `Member` is projected:

```
class Professor extends Member {
      Project[] supervisedProject;}
```

SNAQue/J maps class `Member` into the following type: :

$$\begin{aligned}
&\mu X.[name : String, code : int, project : set([name : String, coordinator : X])]\\
&+\\
&\mu X.[name : String, code : int, project : set([name : String, coordinator : X]),\\
&\qquad\qquad supervisedProject : set([name : String, coordinator : X])]
\end{aligned}$$

As an example of the second case, consider a binding to a semistructured XML document in which university projects are represented by `project` elements having either a `name` or a `title` sub-elements. To capture this terminological diversity, the binding program may first declare an interface `Project` with a single method `getName()`. It may then declare two classes `Project1` and `Project2` which implement `Project` using a `String` instance variable called `name` and `title`, respectively. Finally, the program may project `Project` over the target document, and let SNAQue/J infer the following type:

$$[name : String] + [title : String]$$

Depending on their content, SNAQue/J would convert the `project` elements in the document into instances of either `Project1` or `Project2`.

5.7 Conclusions

We have discussed four binding approaches to XML encodings of real-world entities and investigated the extent to which they preserve the computational advantages of the host language. The main characteristics and application areas of each approach are summarised in the following table:

	Data Structure/ Computational model	*Application domain*
SAX	string/ parsing events & callbacks	syntax-oriented processing
DOM	tree/ tree navigation	structure-oriented processing
JAXB	document-dependent (via type generation)/ generic	semantics-oriented processing mostly in *tightly-coupled* systems (regular and stable data, e.g. DTD-governed)
SNAQue	computation-dependent (via type projection)/ generic	semantics-oriented processing mostly in *loosely-coupled* systems (data with semistructured features)

Differently from SAX and DOM, JAXB and SNAQue preserve the semantics of data and thus the simplicity and safety of programming. In SNAQue, in particular, bindings are driven by partial and user-specified type projections rather than document descriptions, as in JAXB. This makes them simpler to define and maintain, especially in the presence of semistructured and rapidly evolving data.

This suggests that type projections may be fruitfully employed *within* a dedicated XML language rather than at its boundary with the file system/network. Early tests of the hypothesis appear in [26] and [122] in the context of an object-based and a mixed-paradigm (procedural/query) model, respectively. Interestingly, they bring out the similarity between partial projections over untyped but self-describing data and structural record subtyping of statically typed data (e.g. XDuce's). Besides achieving flexibility within a safe environment, however, projections enable *partial typing*, i.e typed and untyped views of the same data within the same computation.

Chapter 6

Hybrid Applications over XML

This Chapter reports the contents of the following publication:

> MANGHI, P., SIMEONI, F., LIEVENS, D., AND CONNOR, R. C. H. Hybrid applications over XML: integrating the procedural and declarative approaches. In *Fourth ACM CIKM International Workshop on Web Information and Data Management (WIDM 2002)*, pp. 9–14.

This work embraces the full implications of typeful programming and marks a shift from external to internal type projections. The focus is on metamorphic languages, and type projections are employed to decouple the components of a single program in support of typeful programming. One such language is proposed which integrates procedural and declarative paradigms for hybrid applications over XML.

6.1 Abstract

We discuss the design of a quasi-statically typed language for XML in which data may be associated with different structures and different algebras in different scopes. In *declarative scopes*, data are labelled trees and may be queried with the full flexibility

of XML query algebras. In *procedural scopes*, data have more conventional structures built out of records and sets and may be manipulated with the constructs normally found in mainstream procedural languages. By combining the facilities of declarative and procedural paradigms, the language offers integrated support for the development of hybrid applications over XML, where data change form to reflect programming expectations and enable their enforcement.

6.2 Introduction

To date, programming over XML is largely programming over trees of labelled nodes and string leaves, according to the standard interpretation of the format [123][1]. This can be done in a procedural algebra, such as the DOM [16], or in the declarative algebra of a dedicated query language, XQuery's before others [69].

Based on powerful path expressions, declarative algebras offer unrivalled flexibility for retrieving and transforming the data, in the spirit of query languages. They may also offer computational completeness and, as in XQuery, an appropriate notion of static typing. On the other hand, procedural algebras fit within a well-known computational model inclusive of update. Embedded in mainstream languages via language-specific bindings, they also promise full integration with existing computational facilities (e.g. input/output, user interface, database access, network programming, legacy code), a large user base, as well as proven and familiar development tools.

For their different qualities, the two approaches are complementary and would integrate well within, say, a single object-oriented language. In spite of their flexibility, however, labelled tree structures cannot be expected to be adequate choices for *all* computational tasks. More familiar programming abstractions - such as pairs, tuples, records, sets, relations - may better reflect the application's view of the data. Indeed,

[1]Stream-based procedural algebras such as SAX [17] are also common, but they are normally used as high-level parsers and more rarely participate of application-level programming.

the rise of XML as a universal exchange format for structured data suggests that more and more XML data will originate in mainstream programming languages and database systems. How many employees live within programs and databases as labelled trees?

6.2.1 XML and Data Structures

Ignoring for a moment issues of efficiency, inadequate data structures induce linguistic problems, for programs become soon harder to write, read, and thus maintain. As a simple example, consider the case of an employee record `e` which is encoded in XML, sent over the wire, and then materialised at destination as a labelled tree. Here, the relationship between the employee and its name becomes one between two nodes and their labels, whereas it was originally one between a record and the value of its field `name`. Accordingly, the sender accesses the name by writing the expression:

```
e.name
```

whereas the receiver using a DOM implementation must resort to something like:

```
NodeList children = e.getChildNodes();
  for (int j=0;j<children.getLength();j++) {
   Node child = children.item(j);
   if (child.getNodeType() == Node.ELEMENT\_NODE) {
     String tagName=((Element) child).getTagName();
     if (tagName.equals("name")) ...
       ((characterData) child.getFirstChild()).getData()...}}
```

It should be evident that even simple operations become unnecessarily convoluted. Instead of a simply de-referencing the `name` field of `e`, the receiver has to: (i) loop through all the children nodes of `e`, (ii) distinguish element nodes from other kinds of nodes potentially parsed from the XML encoding of the data, (iii) identify one of them as the `name` node, and eventually (iv) access yet another child in order to get the required string value. Of course language-specific APIs may simplify access to relevant nodes by better tuning the tree algebra to the type system of the host language. Perhaps more significantly, linguistic problems may be sensibly reduced by ‘subsetting’ XML and the associated algebra in order to reflect a stronger orientation towards data rather than documents. It should be clear, however, that the tree structure does not directly reflect the data semantics required by the receiver and, in this particular case, that originally intended by the sender.

When computations are statically typed, the problem is further aggravated by a loss of safety, for correctness can only be guaranteed for generic operations on strings and trees, not employees. For example, `name` was part of a record type at the sender and it has become data within the tree structure at the receiver. As such, it escapes the static knowledge of the system which may no longer ensure its correct use before program execution or, worse, not ensure it at all (e.g. when a misspelled label accidentally identifies another). Performance is a further concern for, again, the receiver’s system can optimise resources only within the limits of its static knowledge of the tree data. For instance, the system ignores now that all employees have a name.

The declarative algebras of query languages alleviate considerably these problems by ‘hiding’ the underlying tree structures under a navigational syntax, thereby achieving succinct and clear programs. XQuery shows also that fine-grained type safety is not an implication of conventional data structures but can be extended to trees, while recent developments show that this can done via extremely flexible type systems (eg. [104]). The choice of data structures, however, extends its impact on semantics: like the procedural XML programmer, the declarative one is still forced to perceive employees as trees in spite of more intuitive models of the data. Besides, the completeness of programming environments and the notion of mutability are normally outside the scope of query languages.

We believe that the problem here is not directly related to labelled trees, which remain the preferred structures for computations that interpret the data as the hierarchical pattern of relationships in which they are organised (i.e. independently of more refined, domain-specific semantics). Such 'structural computations' include most document manipulations – i.e. the original and still primary target of the format – but also flexible querying, filtering and transformation of arbitrary data. Trees are also ideal for *semistructured* data management , for their generality can easily accommodate spatial and temporal irregularities in the data which would otherwise defeat or strain the more rigid structures of conventional typed technologies (cf. [74]).

We argue instead that problems similar to those discussed above would surface with any structure imposed by the wire format and, more generally, with any incarnation of the one-size-fits-all approach to data modelling.

The example of the SAX programming model and algebra is here appropriate. With SAX, the XML data are served to applications as a stream of textual tokens, where each token identifies the occurrence in the data of some distinguished component of the format. The associated programming model is then based on callbacks issued in correspondence with token parsing events. Overall, the SAX programmer is invited to share almost the same view of the data as the underlying parser's. Compared with DOM, this makes it slightly easier to specify computations which interpret the data as the text in which they are encoded; such 'syntactic computations' include token counts, deep queries and, most commonly, high-level parsing routines for instantiating application-specific or domain-specific data structures. Nonetheless, it is easy to see that the problems associated with DOM programs resurface unsolved, and in fact aggravated, when the application logic is directly cast in terms of SAX structures and algebra. On the other hand, SAX programming is notedly efficient in terms of memory consumption. This reminds us of the impact of data structures on performance and show us that different structures may satisfy different computational requirements.

6.2.2 A Language for Hybrid Applications

Motivated by the previous observations, we advocate the importance of applications in which the same data are subject to different structural views and are manipulated according to different algebras in order to support different programming tasks. For example, some components of such *hybrid applications*, as we may call them, may benefit from a tree view over the data and from the flexibility of a query algebra. Other components may instead be safer and simpler through, say, a view based on record and set abstractions and their associated algebras.

In practice, hybrid applications are common and yet the integration of their components is essentially unsupported. XML programmers must associate components with different tools and computational environments – such as a mainstream programming language and a dedicated query engine – and then share data between them through the file system or the network. This forces interactions between components to be ‘off-line’ (i.e. planned in advance of execution) and strictly sequential (a component’s output becomes another’s input). Lack of integration becomes also lack of efficiency, due to the unnecessary operations of input/output and parsing of the data. Updates occur only at the file level while sharing between components require ad-hoc conversions between data structures which are prone to errors and always irrelevant to application semantics. Overall, the XML programmer is entirely responsible for understanding and maintaining the mapping between the application design and its scattered implementation.

In our research, we explore the possibility of writing hybrid applications within the context of a single programming language, where the imposition of structure over the data is transparent and entirely under the programmer’s control. In this paper, in particular, we experiment with two different interpretations of the data and associate them with different scopes in the program. In *declarative scopes*, the data are labelled trees and can be manipulated with a simple query algebra based on XPath expressions [124].

In *procedural scopes*, the structure of the data is a recursive composition of records and sets and can be manipulated with conventional algebras. In the latter case, programmers may also count on a selection of the basic types and programming constructs found in most procedural languages. We believe that integrating the procedural and the declarative paradigm within the same language would satisfy most requirements for structural change associated with hybrid applications.

In such language, programs could be partitioned according to the view which is syntactically in scope, with data changing form upon entering and exiting scopes. The passage to a declarative scope would be straightforward, for the data can always be interpreted as a labelled tree. Different is the opposite case, when the *projection* of more constrained structures over trees would introduce the possibility of failure. We propose to solve these problems by resorting to a *quasi-statically typed* language and interpreting structural projections as type assertions attached to program variables. Noticeably, type assertions would be verified dynamically, when variables are bound to trees, and yet their scope within the program would be statically typechecked.

The rest of the paper is organised as follows. Section 6.3 introduces our model of type projections while Section 6.4 illustrates by way of example the design of a language core built around the projection model. Section 6.5 examines related work and, finally, Section 6.6 draws some conclusions and suggests directions for further development.

6.3 Type Projections

We have successfully investigated the problem of type projections over labelled tree data in the context of `SNAQue`, a language-independent architecture for binding quasi-statically typed programming languages to XML data which emanate from outside their context. `SNAQue` is formally defined in [119], while an implementation specific to the Java language is discussed in [125]. In the same paper, the approach is also compared with related work, including SAX and DOM and other high-level XML data binding solutions, such as Sun's JAXB [15].

With `SNAQue`, the programmer projects a type over an XML document in an attempt to instantiate the content of the second into a value of the first, and thus derive the benefits discussed in Section 6.2. Conceptually, this requires: (i) parsing the file into a labelled tree structure with string leaves, and (ii) establishing whether the parsed tree is an encoding of a value of the projected type according to a pre-defined mapping between language values and labelled trees. If this is the case, the value is materialised from the tree and may be subject to application logic, otherwise an indication of failure is returned to the programmer[1].

For generality, we have studied type projections in the formal context of a *canonical language* defined around a value notation, a type language, and a relationship of typing between the two. In particular, we have chosen a selection of structural types commonly found in existing procedural languages: built from a set of atomic types, they include include *record*, *collection*, and *disjoint union* types, possibly recursively defined. Specifically, a type T is one of a finite set of atomic types B_i, a record type $[l_1 : T_1, \ldots, l_n : T_n]$ where the l_i are drawn from some pre-defined set of labels, a collection type $coll(T)$, a union type $T_1 + T_2$, or a recursive type $\mu X.T$, where X is a type variable and the operator μ binds occurrences of X in T[2].

The choice of structural types reflects the non-procedural nature of a general-purpose and textual data format, and thus the restricted applicability of projecting function types and data abstractions built around them. However, non-functional components of abstract data types may well be projected over XML encodings of the state

[1]An optimised implementation may well prefer a stream-based view over the data and thus interleave parsing and projection to avoid materialising the tree structure. This is the strategy adopted for the latest implementation of `SNAQue`.

[2]Recursive types do not need to appear such theoretical guise within programs, but can be derived as usual from self-referencing type declarations.

of their values. An example of this can be found in [125], where we have stripped Java classes of their names and method declarations before projecting them over XML documents.

Collection types have the semantics of sets, although other bulk types preserving ordering and duplicates could have been equally considered. For example, the projection of array types is considered in [125]. Union types are important additions to the type language, for they allow to capture some of the irregularity allowed within tree structures, especially those obtained from parsing semistructured XML documents. In addition, they justify the choice of projecting recursive types in the lack of cyclic graph structures. The combination with recursive types is also useful within the language discussed in Section 6.4, where it allows the definition of finite recursive structures. In this context, our union types are similar to the *united modes* of Algol 68 [51], in that they act as typed views of more concretely typed values but do not introduce new values in the language.

Canonical values mirror the available types. A value v is an atomic value $b_k \in B_k$, a record value $[l_1 = v_1, \ldots, l_n = v_n]$, a collection value $\{v_1, \ldots, v_n\}$, or the empty collection $\{\}$. The typing relation is inductively defined in a standard fashion. An atomic value b_k has the corresponding type B_k, while a record $[l_1 = v_1, \ldots, l_n = v_n]$ has the type $[l_1 : T_1, \ldots, l_n : T_n]$ only if each v_i has type T_i. A collection $\{v_1, v_2, \ldots, v_n\}$ has the type $coll(T)$ only if all the v_i have type T, while the empty collection has the type $coll(T)$ for all T. A value v has type $T_1 + T_2$ if v has type T_1 or type T_2 and, finally, v has type $\mu X.T$ if v has the type obtained by substituting $\mu X.T$ for all the bound occurrences of X in T.

For parsing purposes, we have considered a tree interpretation of XML data which abstracts over the document-oriented features of the format (ordering, processing instructions, etc.) and concentrates on the 'data-oriented' features (e.g. naming and nesting)[1]; again, this reflects our view on XML as a generic exchange format. The interpretation is straightforward and Figure 6.1 gives a visual example of the tree resulting from parsing a sample XML document.

```
<store>
  <name> BooksRus </name>
  <book>
    <author> John Backus </author>
    <author> Peter Naur </author>
    <title>XML Does Not Care </title>
  </book>
  <book>
    <author>
       <fname> Stan </fname>
       <sname> Lee </sname>
    </author>
    <title> The Annotated Spiderman </title>
  </book>
  <book>
    <title> The Bible </title>
  </book>
</store>
```

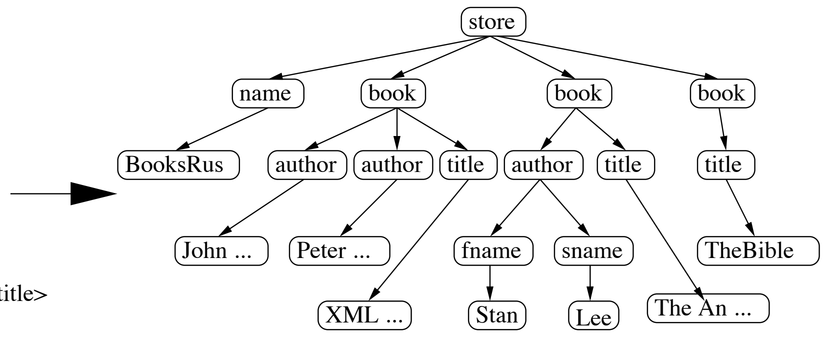

Figure 6.1: XML Parsing

The tree encoding of values is illustrated by way of example in Figure 6.2, which shows the tree corresponding to the record:

v=[a=1,b={"two,"three"},c=[d="four",e=5],f={}],

in the reasonable assumption that string and integers are among the available atomic values. Essentially, atomic values are encoded as childless nodes labelled with a textual encoding of the values themselves, while the encoding of records is built from that of their field values via nodes labelled with corresponding field names. This seems rather intuitive given that, purely from a structural perspective, records can be seen as labelled trees constrained to have no nodes with two or more equally labelled children.

An encoding of collections is instead less obvious for labelled tree structures give no direct support for aggregations of anonymous values. One possibility would be to

[1]For flexibility, we allow attributes and parse them as subelements.

follow the encoding rule of records and choose some special label, say ε, in place of field names. However, we feel that such an explicit introduction of collections within tree structures would: (i) complicate parsing of external XML documents which do not follow the convention and yet could be conveniently projected over, (ii) force early distinctions in the data between record and singleton collections, and (iii) complicate either the specification or the evaluation of queries within the integrated language (see Section 6.4). Overall, introducing and then managing ε-nodes would introduce unnecessary overhead within the projection framework.

We prefer instead to encode collections indirectly, that is only when they occur as values of record fields. In this case, we specialise the encoding rule for records by interpreting a collection-valued field as a collection of homonymous fields. For example, the collection {`"two,"three"`} is encoded in the context of the surrounding record as a collection of two string valued `b`-fields. In particular, the encoding does not preserve empty collections, such as the value of the `f`-field of v. Besides solving most of the problems raised above, we believe that this encoding reflects common practices in XML-based modelling. Its main drawback is that it forces collection types to be wrapped within record types before being projected, which may introduce some programming noise when typing the entire document (see Section 6.4).

A last problem in the encoding is introduced by the mismatch between node-labelled trees and anonymous record values, which raises the question of what labels should be used at root nodes. In previous work, we have avoided this problem by interpreting XML-data into edge-labelled tree structures, in the spirit of other data-oriented approaches. Here, we find more convenient to maintain labels on nodes in order to rely later on standard query semantics for XML. As with collections, our view is then that the label may only be provided by the context in the form of the field name of an enclosing record. For example, the root of the tree that encodes the record `[d="four",e=5]` is provided by the name of the `c`-field of v. Besides explaining the label omitted at the root node in Figure 6.2, this implies that tags of top-level document elements, such as `store` in Figure 6.1, are simply ignored for projection purposes.

Type projections can finally be illustrated with the following example, where the type $Store$:

$Store =$

`[name:string,`

`book:coll([title:string,author:coll(`$Author$`)]`

$Author =$ `string+[fname:string,sname:string]`

is projected over the data in Figure 6.1. This results in the language value $store$, where:

$store =$ `[name="BookRus",book=`$\{book1, book2, book3\}$`]`

$book1 =$

`[title="XML Does Not Care",`

`author=`$\{$`"John Backus","Peter Naur"`$\}$`]`

$book2 =$

`[title:"The Annotated Spiderman",`

`author=`$\{$`[fname="Stan",sname="Lee"]`$\}$`]`

$book3 =$ `[title="The Bible",authors:`$\{\}$`]`

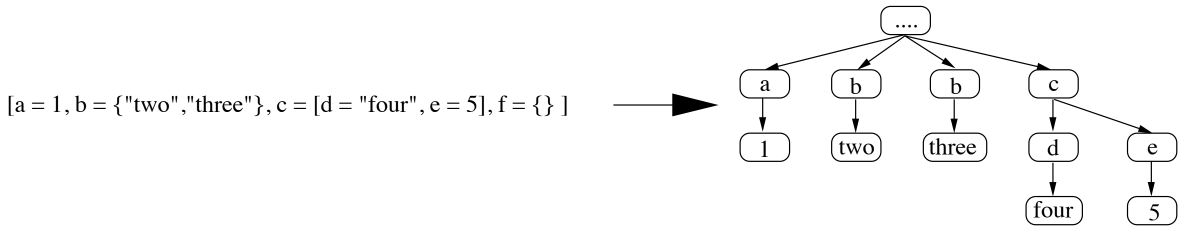
[a = 1, b = {"two","three"}, c = [d = "four", e = 5], f = {}]
....
a
b
b
c
1
two
three
d
e
four
5

Figure 6.2: Encoding example

6.4 Procedures and Queries

Technically, this work originates from the hypothesis of moving tree data and the projection model within the language, rather than at its boundary with the file system or the network. At this early stage of investigation, however, our aim is not to define and implement a complete language, rather to explore the design space induced by the type projection model. In this Section, in particular, we use a pseudo-syntax to illustrate by way of example a number of possible extensions to the canonical language.

For the procedural fragment of the language, we assume the availability of type declarations, functional and procedural abstractions, assignments, and a standard choice of control structures, atomic types, and value operators. We also assume static scoping, by-reference semantics for function and procedure application, as well as a regime of static typing in all but specific points in the programs, as shown below. Finally, type equivalence is structural to match the nature of type projections.

As a first example, the following program fragment declares and invokes a function `addBook` to increment the number of books of an existing collections:

```
type Book = [title:string,isbn:string,price:number]

proc addBook(Book book, coll(Book) booklist){
  for b in booklist
    if (b.isbn = book.isbn) remove b from booklist
  add book to booklist
}

val coll(Book) bookcoll := {}
addbook([title ="The Annotated Spiderman",
               isbn="0-567-00256-8",price =23.45])
```

The code should be self-explaining: the atomic types `number` and `string` have their standard meaning, while the operators `for..in`, `add..to`, and `remove..from` form the bulk of the collection algebra. Note also the use of anonymous types in declarations of values and formal parameters.

We then introduce XML in the language in the form of values of a type `tree` and combine it with an XPath-based query algebra into a dedicated form of functional abstraction. In particular, queries may take one or more labelled trees with string leaves in input and return a labelled tree or an atomic value in output, possibly using well-formed XML as a literal syntax.

For example, the following query `findBooks` shows the flexibility of declarative algebras in dealing with semistructured input. The query takes a tree representation of an online bookshop's catalog and returns the title, ISBN number, and price all the books scattered in the data under differently labelled nodes and possibly at different levels of depth:

```
query findBooks($catalog) {
  for $book in
        $catalog//book union $catalog/reviews/entry
  return
    <book>
      $book/title
      $book/isbn
      $book/price
    </book>}
}
```

Query semantics is standard, and in any case its details have little relevance in this context, where the focus is on the integration of programming paradigms. Accordingly, we assume the reader's familiarity with XPath and FLWR-like expressions and only point out that the query processor embeds the sequence of `book` elements produced by the `return` clause in a system-defined `ROOT` element, so as to return rooted trees to the invoking scope.

This brings us to query invocations, which represent the boundaries between procedural and declarative scopes. Actual parameters may have any type but they are in any case encoded as trees before binding to formal parameters, according to the rules described informally in Section 6.3. Return values may be implicitly projected over via assignments to typed variables and formal parameters within procedural scopes.

This is illustrated in the next example, where two trees enter the same procedural scope. The first, `read("catalog.xml")`, is parsed from a local file via the predefined function `read` but exits the scope immediately as the actual parameter of an invocation of `findBooks`. The second, `findBooks(read("catalog.xml")`, returns from the query and remains in scope with the form of a record. It is then updated before being written back to file via the predefined procedure `write`:

```
type BookList = [book:coll(Book)]
val BookList books := findBooks(read("catalog.xml"))
for b in books.book
    b.price := b.price*1.1
write(books,"books.xml")
```

The previous example captures the essence of the language, for it shows that assignments may entail type projections, implicitly. Type `BookList` is here projected over the tree output of `findBooks` with the intention of binding the resulting record to variable `books`. The model outlined in Section 6.3 and the body of `findBooks` ensure that the projection is successful; had it not been, an error would have been returned to the programmer, such as a null value or some kind of more explicit exception. The collection of books contained in `books` can then be conveniently browsed and the individual books updated to reflect an increase in price[1].

The previous example shows what we believe to be a standard programming pattern for the language: data is flexibly filtered or transformed via the query algebra to then be updated or otherwise manipulated via procedural programming abstractions and facilities. As a further example of this, the following fragment populates a listbox in some user interface with book titles, possibly in response to a user query:

```
type ListBox = ... ListBox
fun newListBox (string name) {...}
proc fillListBox (ListBox box, coll(string) data) {...}

query findTitles($catalog, $publisher, $year) {
  let $books := $catalog//book
  for $book in $books,
        $y in $books//year,
        $p in $books//publisher
  where $y = $year and $p = $publisher
  return $book/title
}

val ListBox titleBox = newListBox ("Book Titles")
val [book:coll([title:string])] books :=
```

[1]Type `coll(Book)` cannot be directed projected over the data for the encoding rules described in Section 6.3 require that a record type such as `BookList` be wrapped around it. While this does introduce some noise in the code, terseness may be regained by introducing ad-hoc projection rules. For example, collection types may be directly projected over trees in which all the children of the root have the same label (`book` in this case).

```
findTitles(read(catalog.xml"), "Addison", 2002)
val coll(string) titles = {}
for b in books.book
    add(b.title) to titles
fillListBox(titleBox, titles)
```

Listboxes and their interface are here assumed to be part of an available graphical toolkit. In the toolkit, in particular, type `ListBox` describe listboxes in terms of previously defined types, procedure `newListBox` creates a named listbox, and function `fillListBox` draws the content of a listbox from a given collection of strings. The latter is obtained as the output of the query `findTitles`, which extracts it from the catalog of books of the previous example using information received from the invoking context as a filter. .

As a final example, consider the case of a simple hybrid application that selects the cheapest price for a list of books. Assume that the list is contained in a relational database and comprises the recommended books for the courses offered by a university department. Assume also that the language is supported by a database access API and that two functions `getRecBooks` and `updateRecBooks` have already been defined as abstractions for interfacing the course database and, respectively, extract and update the list of recommended books. Finally, assume that information on a list of online stores is locally available in an XML file `"stores.xml"`:

```
type Store = [name:string,catURL:string]
type StoreList = [store:coll(Store)]

// database access
fun coll(Book) geRecBooks() {...}
proc updateRecBooks (coll(Book) rl) {...}
```

```
query getLowerPrices($list, $catalog) {
  for $b1 in list, $b2 in $catalog//book
  where $b1/isbn = $b2/isbn and $b2/price < $b1/price
  return {
    <book>
      $b2/title
      $b2/isbn
      $b2/price
    </book>
 }

val coll(Book) recList := getRecBooks()

for b in recList
  b.price := MAX_PRICE

val StoreList stores := read("stores.xml"))

for s in stores.store {
  val BookList catalog =
         getLowerPrices([book:recList],read(s.catURL))
  for b in catalog.book
    addBook(b,recList)}

updateRecBooks(recList)
```

The application is very simple and the reader should compare it with the solutions currently available. First, the list of recommended books is retrieved from the database as a collection value `recList`. Book prices in `recList` are then initialised to some large constant to ensure that an update will actually take place.

Similarly, the store list is read from`"stores.xml"` and assigned to variable `stores`. This entails the projection of type `StoreList`, which asserts that, for each store, the file contains a name and the URL of a corresponding XML catalog. Assuming the assertion is correct, the catalog of each store is fetched from its URL with an overloaded version of `read` and passed to a query `getLowerPrices`. The latter receives also the `recList`, after this has been wrapped within a record and the latter encoded as a tree upon entering the declarative scope (again, rooted in a node labelled by the system as `ROOT`)[1]. This offers a more substantial example of a conventional language value that changes its form in order to be queried.

In particular, the query filters the books in the catalog which are recommended and currently offered at a lower price. When returned to the procedural scope within the record value of `catalog`, the new offers are reflected in `recList` with repeated invocations of the function `addBook` defined above. After all the catalogs have been so processed, `recList` is eventually written back to the database.

6.5 Related Work

The language sketched in Section 3 offers what we believe to be a unique combination of: (i) a full hybrid programming paradigm which allows arbitrary computations to be expressed in either a procedural or a declarative style, according to convenience, and (ii) an unusual form of structural polymorphism whereby, for similar reasons, different structures can be imposed over the same data. It is useful to conceptually separate

[1]Similarly to the case discussed above, the record surrounding `recList` is necessary for out interpretations of collections and it may be avoided with ad-hoc encoding rules.

the two features and notice that while the first may be associated with a single data structure, the second may exist in, say, a fully procedural language.

To some degree, similar hybrid programming paradigms may be found in database languages which layer procedural constructs over the query algebra and data model of an underlying DBMS. In Oracle's own PL\SQL, for example, the programmer may specify SQL queries against a relational database, assign the resulting values to variables, and then computer over them with looping and conditional statements. Here, however, the query algebra is functional to interfacing the DBMS rather than improving programming expressiveness; in particular, the SQL algebra cannot be used over values procedurally generated or already extracted from the database.

Interestingly, a tight integration of declarative and procedural algebras is not hard to imagine in the XML scenario, where a single tree structure may offer both an XQuery and a DOM interface within the same mainstream language. Unfortunately, we are not currently aware of the availability of synchronised implementations of the two algebras.

Polymorphic behaviour, on the other hand, is far more common across mainstream and theoretical languages alike. Normally, however, it is related to type abstractions – such as those induced by subtyping relations and type parameters – which hide properties of the data that are contextually irrelevant [37]. Accordingly, data may only be subject to 'quantitative' changes and may only be manipulated under progressively restricted algebras. We propose instead a less regular, 'qualitative' form of polymorphism which increases language expressiveness in a rather different way: rather than enlarging the input of programs, perhaps to allow them to be reused over time, it exposes the features of that input that most adequately support its intended manipulation.

Interestingly, we have found this polymorphic behaviour in the dynamically typed ECMAScript standard [126], where it follows largely from the object-based paradigm and an extensive coercion scheme. In ECMAScript, structured values can be equally manipulated as objects, associative arrays or – under more restricted circumstances – heterogeneous arrays. As objects and arrays, they support familiar programming abstractions via algebras based on de-reference and indexing operator, respectively. As dictionaries, they can be more flexibly browsed with a combination of a look-up operation and a corresponding looping control structure. Building on such flexibility, we have experimented with adding types to the Javascript implementation of the standard, along with the possibility of projecting them over tree structures parsed from XML data. The reader interested in this alternative approach to the issues raised in the paper may refer to [26].

To conclude, our model of type projections extends the tradition of dynamic type-checking within statically typed languages, whereby some types are asserted statically but the soundness of those assertions is ascertained at run-time against some self-describing representation of values. This occurs when the programmer wants to force the type-checker to restrict, broaden, or otherwise revise its assumptions on the data, and is common in casting and coercing operations, as well as in operations associated with variant and union types(cf. [1]). In particular, our projections resemble those required by infinite union types – such as Amber's `dynamic` [59] and Napier's 88 `any` [4]) – and we may well make them explicit introducing an equivalent `case..of` statement. As used in SNAQue, in particular, they have also the same functionality, namely to allow type-safe access to data which emanates from outside the context of the language, such as persistent or network data. Our labelled trees, however, do not need to be type-tagged at run-time – neither through an injection operation nor implicitly – for they are sufficiently self-describing to enable dynamic type-checking. Accordingly, they offer advantages of simplicity and evolution within distributed systems with significant degree of local autonomy.

6.6 Conclusions and Further Work

We have identified a principle for XML programming and then proposed it towards the design of a quasi-statically typed XML language. The principle is that, whenever possible, the encodings of a universal exchange format should be interpreted according to the expectations of the applications that consume them rather than the intentions of those which generated them, and certainly not in the computational vacuum of the format definition. An implication of this is that, however well some data structures may satisfy *some* expectations, no single data structure may be expected to adequately reflect *all* the possible interpretations.

Accordingly, the language allows users to see data from either one of two different structural perspectives. As trees of labelled nodes and string leaves, the data may be flexibly queried, filtered, or otherwise transformed. As potentially recursive structures built from a set of atomic values via record and collection constructors, they may be more safely and directly manipulated, possibly updated, and overall injected within a complete programming environment. We have then proposed and described a model of type projections which gives data the required degree of polymorphic behaviour at the cost of introducing a controlled form of dynamic type-checking in an otherwise statically typed regime. We believe that, for the resulting integration of the declarative and procedural paradigms of computation, such a language may facilitate the development of hybrid applications over XML, where data are subject to different interpretations in different points and at different times.

Due to the novelty of the approach, it is not yet clear what language features would best support the development of hybrid applications.

Implicit type projections let us exclude the type `tree` from the syntax available to the programmer and to confine it only into more formal accounts. In particular, the programmer may overlook the fact that expressions such as `read("catalog.xml")` denote trees and rather think that the latter can only occur in declarative scopes. This increases the readability of programs and, most importantly, induces a clear separation between procedural and declarative scopes in the mind of the programmer.

However, making `tree` a 'first-class' type in procedural scopes would introduce a form of partiality in type projections and may thus be considered for the benefit of the expert programmer. The benefits are essentially those associated with an infinite union type, especially its incarnations in persistent programming languages (cf. [4]). Projected over a tree within a collection or record type, in particular, it would allow some part of the tree to remain intentionally unspecified. This may be useful as a type abstraction mechanism, where it would give some of the flexibility normally associated with parametric polymorphism. It may also be used for incremental projections over trees, whereby the complete typing of trees is defined in two of more steps and the outcome of each step determines the definition of the successive one.

In addition, the identity of data is currently preserved across procedural scopes (due the by-reference semantics of procedure and function application) but only partially when passing from procedural to declarative scopes (due to the standard by-copy semantics of query algebras). In particular, while all values can currently become trees, they cannot 'round-trip' and return to assume more constrained shapes without losing their identity. Accordingly, programs in which parts of conventional values are flexibly retrieved within declarative scopes in order to be updated later, upon returning in procedural scopes, cannot be currently written. While a by-reference semantics for queries may be considered, it would require a passage to a graph-based data model and should therefore be carefully considered.

Chapter 7

Concluding Remarks

In summary, we have presented a model for binding statically typed programs and self-describing data based on type projections. We have shown that the model exploits the potential for loose coupling which inheres in self-description in order to endorse a principle of typeful programming over shared data: the type of the data should vary in accordance with the requirements of individual programs.

We have argued that loose coupling is key between programs that execute in co-operating but autonomous runtimes. Accordingly, type projections may be employed at the boundary of typed languages to regulate the exchange of untyped data across the nodes of distributed systems. *External type projections* give rise to a class of data binding tools for mainstream languages and standard self-describing formats, such as XML.

We have also argued that loose coupling is desirable between programs that execute within a single runtime, where conventional typechecking regimes may unduly constrain interpretations of shared data. Type projections may then be employed *within* typed languages to regulate the exchange of typed data across local application components. *Internal type projections* give rise to metamorphic languages, where an unconventional 'data-first' approach to dynamic typechecking extends support for typeful programming beyond the scope of polymorphism and coercions.

Overall, we believe to have put type projections on solid conceptual ground. In the process, we have uncovered general connections between self-description, loose coupling, and typeful programming which can be readily applied to the understanding, evaluation, and design of a wide range of programming technologies. This is especially true in the context of XML processing, where our analysis yields new insights into the motivations and limitations of widely deployed tools and popular programming practices.

In particular, our analysis in the first Part of this Dissertation included:

- an account of typechecking practices which culminates with the identification of the principle of typeful programming. This qualifies and generalises the role of types in programming languages beyond the conventional viewpoint of program correctness. Most importantly, it offers the motivation that unifies all the applications of type projections.

- an account of self-describing data and its implications for general-purpose data processing. To the best of our knowledge, this represents the first systematic attempt to analyse self-description in the formal literature. Type projections then emerge as a mechanism to exploit the potential for loose-coupling which inheres in self-describing data.

- the identification of a new class of tools based on type projections for binding statically typed languages and self-describing data, and their comparison with related tools in the mainstream.

- the identification of metamorphism, a novel form of type abstraction that generalises over polymorphism and coercions under a data-first approach to dynamic typechecking.

With concepts firmly in place, we have outlined and sampled the design space induced by type projections. In particular, our contribution in the second Part of this Dissertation included:

- a formal framework for the definition of external type projections and its application to an idealised language with structural types. This serves a model of the design and implementation of binding tools for mainstream languages and standard self-describing formats.
- a report on the design and implementation of prototype binding tool for Java and XML.
- a proposal for the design of a metamorphic language that unifies procedural and declarative paradigms for computing over XML.

The formalisations and applications shown in the previous Chapters, however, served to prove the concepts. They answered the research question: *can* we build useful data binding tools and metamorphic languages based on type projections? We now face a broader question: *what* data binding tools and metamorphic languages can we build? Much work remains to be done to address this question. While some of the work has been already carried out, we have not discussed it so far. We then conclude the Dissertation with an overview of the directions in which the investigation has been and still could be pursued further.

***

For external type projections, one way forward would be to evolve the prototype presented in Chapter 5 into a full-fledged data binding tool for XML. The prototype focuses on the extraction of language values from external data, but it does not recognise or preserve the full range of XML structures, nor does it externalise language values back into self-describing forms. Similarly, the prototype falls short to meeting real-world expectations concerning the pre-compilation and customisation of bindings. However, even though there are valid research questions behind some of these functional requirements — e.g the 'impedence mismatch' between object-oriented and document-oriented structures — the issues are largely unrelated to type projections, are too focused on the target data format, and have been sufficiently explored [127]. Of more interest are instead novel applications scenarios, particularly those that require extensions to the formal framework presented in Chapter 4.

Projections over Graphs. As a first example, we note that the framework is defined over tree-based representations of data, in line with the basic definition of XML. Yet self-describing data can form graph structures, and there are conventions to encode graphs in XML which are endorsed in many applications domains. Type projections over labelled graphs may be relatively easy to implement, but we do need to extend our formal framework to guarantee the soundness and completeness of the implementations. The extension is reported in [25], and it is far from trivial. Partly, this is because natural models of cyclic structures give rise to non-wellfounded sets, which are axiomatically excluded from classic set theory [128]. To preserve such models, we must embrace hyperset theories and approach definitions and proofs with less familiar tools, such as simulation and co-induction [129]. In addition, the possibility of cycles and sharing in the data imply that multiple types may be simultaneously projected on the same node. This should be allowed only if we can extract a value from the node which has all the types projected over it. Introducing and enforcing notions of compatibility between types (such as equivalence or subtyping) is possible, but it raises its own problems: the definition of type relationships is complicated by the presence of recursive types [111, 130]; more seriously, the definition of complete algorithms is complicated by the presence of union types, and it requires expensive backtracking techniques. It is as yet unclear whether the lack of completeness is of pragmatic significance.

Projections over Streams. Issues of data complexity aside, we meet different challenges and opportunities if we lift certain assumptions on *how* data becomes available for binding. So far, we have assumed it to be accessible in its entirety and for as long as the program needs it. It is increasingly common, however, to compute over *data streams*, where random access is granted only on a small portion of the data at the time. Streamed processing may be used opportunistically, to address unfavourable ratios between the available memory and the size of the data; it may also reflect the way in which the data is generated, rapidly and continuously. In all cases, the model raises requirements of locality and efficiency against programs, which are best matched by certain classes of queries (e.g. filters). Streamed processing is extensively discussed for relational data [131], but XML streams are also widely investigated [132, 133, 134, 135]. Most approaches focus on partial translations from declarative query languages (such as XPath or XQuery) to networks of finite automata, which then consume the stream along the events of a suitable parser (typically SAX). Data binding tools normally do not enter the picture, as the expectation is that they may be used downstream, to bind query results as these become available from the automata. With type projections, however, we can integrate query execution and data binding within the context of a single high-level language. The key observation here is that many queries may be decomposed in two sub-queries: a primary filter that identifies data with relevant structure, and another query that transforms each instance of this data into a query result. This suggests a ‘progressive’ model of query evaluation in which the two sub-queries execute asynchronously and only the structural filter interfaces the stream. Type projections can support this model when the filter is relatively simple and the transformation relatively complex, as it is the case with queries that target regular subsets of semistructured data. For such queries, the transformation may be expressed in the host language and the type of its input may be projected over the stream *as* the structural filter. The approach is discussed in detail in [27], where the formalisation and implementation of type projection is appropriately cast in terms of automata theory.

Distributed Projections. Separating program execution from type projection enables applications other than streaming. In the prototype discussed in Chapter 4, the separation occurred over a network: programs executed in client runtimes while types were projected in a server runtime, locally to the data. Clients sent the required types and then computed over the values extracted and bound by the server as CORBA objects. Recently, we have migrated similar ideas to different technologies, replacing the shared abstractions of distributed objects with standard message exchanges between Web Services. This causes a shift in responsibilities: values are now bound locally to clients against self-describing data that is extracted and returned by the service as the output of successful projections. As in the streaming application discussed above, projections act as filters while the interpretation of the data occurs downstream. The implications are now in terms of reduced coupling, as clients and service need no longer to synchronise on a type language. The service may publish a type language for projection, but clients may use different types and technologies for binding, including again type projections. Furthermore, the types used at the service are not constrained by its implementation language, which may even be dynamically typed; they may be designed or chosen for their flexibility in characterising self-describing data. We have followed this approach in a service-based system that controls a large-scale, production-level infrastructure for international scientific research[1]. A key data management service in the system provides its clients with access to a wide variety of data repositories, both internal and external to the infrastructure. The service exposes a data model of edge-labelled trees with untyped text leaves, which it exchanges using a minimal subset of XML; it then delegates to plug-ins the task of adapting the tree model to the remote repositories. Clients can require trees with a given topology and given content at the leaves. These requirements are captured by 'tree types', which are projected by plug-ins over tree representations of the data returned by the repositories.

[1]`http://www.gcube-system.org.`

The parts of the trees that do not match the type are pruned, the rest are returned to clients for binding.

***

All the applications above face limited issues of language integration. The host language is characterised exclusively in terms of its type language and value space, as reflected in the formal framework of Chapter 4. This is precisely what gives wide and immediate applicability to external type projections. As we move from external projections to internal projections, however, we must contend with a larger number of variables, all of which relate to the design of the host language.

As a first implication, it is unclear whether we can still formally capture the essence of internal type projections with a useful degree of generality; rather, we plan to further the investigation in terms of concrete language designs. In addition, we have already identified cases in which type projections are in clear tension with some instantiations of these variables. We have mentioned in Section 1.3.2 the problematic interaction between type projections and update, and hinted at the scope for solutions. We also note here that issues arise with data abstraction as well, e.g. with the encapsulation of structure which is found in mainstream languages. Equally, we are as yet unclear about the interaction between type projections and other language features, such as inclusion and parametric polymorphism, or nominal typechecking.

While these issues define a clear agenda for the investigation, we note that they do not threaten the possibility of useful applications. The language proposal in Chapter 6 proves the point by showing that metamorphoses may be fruitfully introduced into a language with update and structural typechecking, but without data abstraction and polymorphism. In fact, solving the issues may be of little value *per se* if the solutions compromise the pragmatics of the language, i.e. increase its overall complexity or decrease its expressiveness. In this sense, the issues may indicate that the integration may be best pursued for languages with particular computational models and for particular applications domains, e.g. purely functional languages or languages for data modelling and integration. Alternatively, they may indicate that the integration may not be orthogonal to the design of the language; rather, it may be confined to data-oriented fragments of a general purpose language, in the spirit of [136]. The overall theme of the investigation is thus to discover the identity and character of a metamorphic language.

Partial Typing. In keeping with this theme, the very assumption of integrating type projections in a statically typed language may be questioned. We have already shown in Section 2.6 that a useful degree of static typechecking is only one of the benefits that we expect from typeful programming, hence from metamorphoses and internal type projections. We can forego it as long as we devise means other than types to capture the interpretations that we want to project over self-describing data (e.g. representative values), or as long as we use type expressions exclusively for dynamic verification. Less drastically, we can retain the assumption of static typechecking, but introduce it 'partially' within a dynamically typed language. The idea aligns with the motivations for some of the hybrid typechecking techniques that we have discussed in Section 2.5, particularly gradual typing techniques. As with gradual typing, programmers would have the option to annotate program variables with type assertions. The typechecker would verify the available assertions insofar as they give evidence to support its deductions. When they do not, most noticeably at the boundary between typed and untyped program fragments, the typechecker would prepare the runtime for dynamic verification and rely on induction to move its analysis forward. The key difference from gradual typing is in the grain and nature of dynamic verification. In gradual typing, verification occurs at the application of primitive operations, as a conventional typecheck against the types that tag the operands. In our scheme, verification can occur at the application of user-defined operations, and takes the form of type projections against the deep structure of the data. We have explored this notion of *partial typing* in [26], where we have integrated type projections in a JavaScript implementation of the Ecma262 standard [137, 126].

***

To conclude, we notice that the evolution of the field since 2004 indicates that the ideas set forth in this Dissertation remain as relevant today as they were then. True to its promise, XML has achieved wider penetration in the market, while the uptake of novel formats for the exchange of structured data (e.g. JSON[1]) has emphasised the general value of self-description in the practice of system engineering. More to the point, it has reflected the need to depart from interpretations of shared data that reflect the strength of a standard more than the significance of local processing requirements. This has served as an illustration of the benefits of typeful programming for a class of computations (e.g. for JSON, simpler, general-purpose, procedural algebras over untyped but regular data structures). However, it has fallen short of showcasing the decoupling potential of self-description, i.e. witnessing to the range of interpretations that may remain latent within the data. At the same time, loosely coupled patterns of data exchange have acquired more application within the field, as most noticeably manifested by the increasing interest in REST-based architectures over tightly-typed RPC-based architectures for distributed systems based on Web Services [138].

Across all architectures, data binding tools continue to be employed ubiquitously upstream of application protocol stacks, though standard approaches have shown relatively little innovation with respect to the assumptions discussed in Chapter 1 and in Chapter 5, as well as limited integration with enterprise technologies that have become mainstream since (e.g. Spring [139], Guice [140], or similar containers based on Dependency Injection for lifetime management of bound objects [141]). Interestingly, the

[1]cf. `http:www.json.org`

most popular tools in this context have given some recognition to the value of bindings that are driven by consumers' requirements, rather than shared types. Since 2005 for example, alongside the usual approach whereby bound types are generated from XML schemas, JAXB allows bindings over untyped data to be defined directly via annotations of bound types, in line with modern practices for metadata specifications. This mode of usage aligns well with type projections, and its anecdotal popularity among practitioners provides further evidence towards the validity of our motivations. The inconsistencies and idiosyncrasies that still mar the realisation of the approach, however, seem to point to an unclear binding model underneath, leaving the overall impression of dealing with an implementation feature rather than a design viewpoint. At the time of writing, we are not aware of binding tools that make a systematic commitment to, and offer comprehensive support for, consumer-driven bindings to XML data.

Even more than external projections, internal type projections remain a novel proposal in language design. A string of important research projects has focused on the integration of XML within the design of programming languages, thereby bringing self-description firmly within the scope of language design [115, 142, 143, 144, 136]. Some of the approaches have been experiments centred around the themes of integration, others have targeted mainstream languages and in some case have reached production stage. Approaches have ranged widely in theoretical foundations and have targeted different computational paradigms, but they have all shared a common quest for adequate programming models over tree-structured data. The adequacy of a model has been understood as an implication of a programming algebra that promotes program fluency without compromising guarantees of static type-checking, and with good integration with the rest of the language. Clearly, these goals witness to the the value of typeful programming but seek it in a restricted context, either over the subset of the value space occupied by tree structures, or else over a value space that consists solely of such structures. With internal type projections, self-description enters the language to allow a number of different models for shared data, including but not limited to tree models. The value space is still uniformly populated but its uniformity remains hidden to the users, who manipulate the same values through a variety of programming algebras. In this sense, the language designs that we have envisioned in this Dissertation exploit self-description to generally endorse typeful programming, rather than deliver typeful programming over data in standard self-describing formats. To date, we know of no language design that enables true metamorphoses of shared data to extend its support for typeful programming beyond the scope of coercions and polymorphism.

Bibliography

[1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.

[2] L Cardelli. Amber. In *Proc. of the thirteenth spring school of the LITP on Combinators and functional programming languages*, pages 21–47, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[3] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. Fibonacci: A Programming Language for Object Databases. *VLDB J.*, 4(3):403–444, 1995.

[4] R. Morrison, R. C. H. Connor, G. N. C. Kirby, D. S. Munro, M. P. Atkinson, Q. I. Cutts, A. L. Brown, and A. Dearle. The Napier88 Persistent Programming Language and Environment. In M. P. Atkinson and R. Welland, editors, *Fully Integrated Data Environments*, pages 98–154. Springer, 1999.

[5] Rick Greenwald and David C. Kreines. *Oracle in a Nutshell.* O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, 2002.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language*

Specification, The (3rd Edition) (Java Series). Addison-Wesley Professional, July 2005.

[7] ECMA. *ECMA-334: C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 2002.

[8] Alan LaMont Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[9] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[10] Peter Buneman. Semistructured Data. In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 117–121, New York, NY, USA, 1997. ACM Press.

[11] Serge Abiteboul. Querying Semi-Structured Data. In *ICDT '97: Proceedings of the 6th International Conference on Database Theory*, pages 1–18, London, UK, 1997. Springer-Verlag.

[12] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extendible Markup Language (XML) 1.0. Technical report, World Wide Web Consortium, 1998. W3C Recommendation.

[13] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[14] B. McLaughlin. *Java and XML data binding*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2002.

[15] Joe Fialli and Sekhar Vajjhala. Java Architecture for XML Binding (JAXB) 2.0. Java Specification Request (JSR) 222, October 2005.

[16] L. Wood and A. Le Hors. Document Object Model Specification (DOM). Technical report, World Wide Web Consortium, 1997.

[17] Dave Megginson. SAX 2.0: The Simple API for XML, 2000. http://megginson.com/SAX/.

[18] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. Technical Report Recommendation REC-xmlschema-1-20010502, World Wide Web Consortium, 2001.

[19] Anita K. Jones and Barbara H. Liskov. A Language Extension for Expressing Constraints on Data Access. *Commun. ACM*, 21(5):358–367, 1978.

[20] Antonio Albano. Type hierarchies and Semantic Data Models. In *SIGPLAN '83: Proceedings of the 1983 ACM SIGPLAN symposium on Programming language issues in software systems*, pages 178–186, New York, NY, USA, 1983. ACM Press.

[21] Peter Wegner and Stanley B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In *ECOOP '88: Proceedings of the European Conference on Object-Oriented Programming*, pages 55–77, London, UK, 1988. Springer-Verlag.

[22] RCH Connor and R Morrison. Subtyping Without Tears. In *15th Australian Computer Science Conference, Hobart, Tasmania*, pages 209–225, 1992.

[23] Richard Connor, David McNally, and Ronald Morrison. Subtyping and Assignment in Database Programming Languages. In *DBPL3: Proceedings of the third international workshop on Database programming languages : bulk types*

& persistent data, pages 363–382, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[24] Steve Neely. *Mobile Computations Over Distributed Semistructured Data*. PhD thesis, University of Strathclyde, 2003.

[25] Paolo Manghi. *Extracting Typed Values from Semistructured Databases*. PhD thesis, Dipartimento di Informatica, Università di Pisa - Italy, 2001.

[26] Richard Connor, David Lievens, Fabio Simeoni, and Steve Neely. Projector - a Partially Typed Language for Querying XML. In *PlanX Workshop on Programming Language Technologies for XML*, 2002.

[27] George Russell, Mathias Neumüller, and Richard C. H. Connor. TypEx: A Type Based Approach to XML Stream Querying. In *WebDB*, pages 55–60, 2003.

[28] Luca Cardelli. Typeful Programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, Berlin, 1991.

[29] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[30] Luca Cardelli. Type Systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. 1997.

[31] D. L. Parnas. A Technique for Software Module Specification with Examples. *Commun. ACM*, 15(5):330–336, 1972.

[32] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[33] David S. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

[34] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, 1995.

[35] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods. *Computer*, 28(4):56–63, 1995.

[36] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods ...Ten Years Later. *Computer*, 39(1):40–48, 2006.

[37] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[38] H. P. Barendregt. Introduction to Lambda Calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[39] H. P. Barendregt. Lambda Calculi with Types. pages 117–309, 1992.

[40] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised.* The MIT Press, May 1997.

[41] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice-Hall, 1978.

[42] Niklaus Wirth. The Programming Language Pascal. *Acta Informatica*, 1:35–63, 1971.

[43] Luis Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[44] Christopher Strachey. Fundamental Concepts in Programming Languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, 2000.

[45] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[46] Luca Cardelli. A Semantics of Multiple Inheritance. In *Proc. of the international symposium on Semantics of data types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[47] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not Subtyping. pages 497–517, 1994.

[48] Antonio Albano, Alan Dearle, Giorgio Ghelli, Chris Marlin, Ron Morrison, Renzo Orsini, and David Stemple. A Framework for Comparing Type Systems for Database Programming Languages. In *Proceedings of the second international workshop on Database programming languages*, pages 170–178, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[49] Satish Thatte. Quasi-Static Typing. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–381, New York, NY, USA, 1990. ACM Press.

[50] Fritz Henglein. Dynamic Typing: Syntax and Proof Theory. In *ESOP'92: Selected papers of the symposium on Fourth European symposium on programming*, pages 197–230, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V.

[51] A. D. McGettrick. *Algol 68, a First and Second Course.* Cambridge University Press, Cambridge, 1978.

[52] Barbara Liskov. *CLU Reference Manual.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983.

[53] B. Lampson. A Description of the Cedar Language. Technical Report CSL-83-15, Xerox Palo Alto Research Center, Palo Alto, CA, USA, 1983.

[54] Paul Rovner. Extending Modula-2 to Build Large, Integrated Systems. *IEEE Software*, pages 46–57, November 1986.

[55] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula 3 Type System. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 202–212, New York, NY, USA, 1989. ACM Press.

[56] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, second edition, 1983.

[57] DOD. *The Programming Language Ada Reference Manual.* Springer, Berlin, 1980.

[58] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. Dynamic Typing in Polymorphic Languages. *J. Funct. Program.*, 5(1):111–130, 1995.

[59] Luca Cardelli. Amber. In Cousineau, Curien, and Robinet, editors, *Combinators and Functional Programming Languages, 13th Spring School of the LITP*, volume 242 of *LNCS*, pages 21–47. Springer-Verlag, 1985.

[60] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Workshop on Scheme and Functional Programming*, September 2006.

[61] Jeremy G. Siek and Walid Taha. Gradual Typing for Objects. In *ECOOP'07: 21st European Conference on Object-Oriented Programming*, 2007.

[62] Craig Chambers. The Cecil Language. Technical Report 93-03-05, Department of Computer Science and Engineering, University of Washington, March 1993.

[63] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[64] Guido Rossum. Python Reference Manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[65] Robert Cartwright and Mike Fagan. Soft Typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.

[66] Andrew K. Wright and Robert Cartwright. A Practical Soft Type System for Scheme. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 250–262, New York, NY, USA, 1994. ACM Press.

[67] Robert Cartwright and Matthias Felleisen. Program Verification through Soft Typing. *ACM Comput. Surv.*, 28(2):349–351, 1996.

[68] Yakov Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. Internet informational RFC 4180, October 2005.

[69] XQuery 1.0: An XML Query Language. World Wide Web Consortium, Recommendation REC-xquery-20070123, January 2007.

[70] Michael Kay. XSL transformations (XSLT) version 2.0. World Wide Web Consortium, Recommendation REC-xslt20-20070123, January 2007.

[71] Dan Suciu. Semistructured data and XML. pages 743–788, 2002.

[72] Lincoln D. Stein and Jean Thierry-Mieg. AceDB: a Genome Database Management System. *Comput. Sci. Eng.*, 1(3):44–52, 1999.

[73] Sudarshan S. Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *IPSJ*, pages 7–18, 1994.

[74] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Washington, DC, USA, 1995. IEEE Computer Society.

[75] Tirthankar Lahiri, Serge Abiteboul, and Jennifer Widom. Ozone: Integrating Structured and Semistructured Data. In *DBPL '99: Revised Papers from the 7th International Workshop on Database Programming Languages*, pages 297–323, London, UK, 2000. Springer-Verlag.

[76] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 313–324, New York, NY, USA, 1994. ACM Press.

[77] Peter Buneman and Benjamin C Pierce. Union Types for Semistructured Data. In *DBPL '99: Revised Papers from the 7th International Workshop on Database Programming Languages*, pages 184–207, London, UK, 2000. Springer-Verlag.

[78] Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D Ullman, and Jennifer Widom. Querying Semistructured Heterogeneous Information. In *DOOD '95: Proceedings of the Fourth International Conference on Deductive*

and Object-Oriented Databases, pages 319–344, London, UK, 1995. Springer-Verlag.

[79] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[80] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. Web-Site Management: The Strudel Approach. *IEEE Data Eng. Bull.*, 21(2):14–20, 1998.

[81] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your Mediators Need Data Conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 177–188, New York, June 1998. ACM Press.

[82] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: a Query Language and Algebra for Semistructured Data based on Structural Recursion. *The VLDB Journal*, 9(1):76–110, 2000.

[83] Luca Cardelli and Giorgio Ghelli. TQL: a Query Language for Semistructured Data based on the Ambient Logic. *Mathematical Structures in Computer Science*, 14(3):285–327, 2004.

[84] Dallan Quass, Jennifer Widom, Roy Goldman, Kevin Haas, Qingshan Luo, Jason McHugh, Svetlozar Nestorov, Anand Rajaraman, Hugo Rivero, Serge Abiteboul, Jeff Ullman, and Janet Wiener. LORE: a Lightweight Object REpository for Semistructured Data. *SIGMOD Rec.*, 25(2):549, 1996.

[85] Bertram Ludäscher, Rainer Himmeröder, Georg Lausen, Wolfgang May, and Christian Schlepphorst. Managing Semistructured Data with FLORID: a Deductive Object-oriented Perspective. *Inf. Syst.*, 23(9):589–613, 1998.

[86] Roy Goldman, Sudarshan Chawathe, Arturo Crespo, and Jason McHugh. A Standard Textual Interchange Format for the Object Exchange Model (OEM). Technical report, Stanford, CA, USA, 1998.

[87] P. Buneman, S. Davidson, and D. Suciu. Programming Constructs for Unstructured Data. In *Proc. DBPL*, volume 13, pages 1–12. Citeseer, March 1995.

[88] Michael Kifer and Georg Lausen. F-logic: a Higher-order Language for Reasoning about Objects, Inheritance, and Scheme. *SIGMOD Rec.*, 18(2):134–146, 1989.

[89] Richard Hull and Masatoshi Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *Proceedings of the sixteenth international conference on Very large databases*, pages 455–468, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[90] Val Breazu-Tannen, Peter Buneman, and Shamim Naqvi. Structural Recursion as a Query Language. In *DBPL3: Proceedings of the third international workshop on Database programming languages : bulk types and persistent data*, pages 9–19, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[91] M R Henzinger, T A Henzinger, and P W Kopke. Computing Simulations on Finite and Infinite Graphs. In *FOCS '95: Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, page 453, Washington, DC, USA, 1995. IEEE Computer Society.

[92] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. *ACM SIGMOD Record*, 23(1):87–96, March 1994.

[93] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[94] Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[95] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: from Relations to Semistructured data and XML*. Morgan Kaufmann Pub, 2000.

[96] Svetlozar Nestorov, Jeffrey D. Ullman, Janet L. Wiener, and Sudarshan S. Chawathe. Representative Objects: Concise Representations of Semistructured, Hierarchial Data. In Alex Gray and Per-Åke Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K*, pages 79–90. IEEE Computer Society, 1997.

[97] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding Structure to Unstructured Data. In *ICDT '97: Proceedings of the 6th International Conference on Database Theory*, pages 336–350, London, UK, 1997. Springer-Verlag.

[98] Svetlozer Nestorov, Serge Abiteboul, and Rajeev Motwani. Inferring Structure in Semistructured Data. *SIGMOD Rec.*, 26(4):39–43, 1997.

[99] C. Beeri and T. Milo. Schemas for Integration and Translation of Structured and Semi-structured Data. In *Proceedings of the International Conference on Database Theory*, number 3, pages 296–313. Springer, 1999.

[100] Peter Buneman, Wenfei Fan, and Scott Weinstein. Path Constraints in Semistructured Databases. *J. Comput. Syst. Sci.*, 61(2):146–193, 2000.

[101] Mary F. Fernandez and Dan Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 14–23, Washington, DC, USA, 1998. IEEE Computer Society.

[102] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[103] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML Transformers. In *PODS '00: Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 11–22, New York, NY, USA, 2000. ACM.

[104] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[105] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML Schema Languages using Formal Language Theory. *ACM Trans. Internet Technol.*, 5(4):660–704, 2005.

[106] Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. Static Analysis for Path Correctness of XML Queries. *J. Funct. Program.*, 16(4-5):621–661, 2006.

[107] William Grosso. *Java RMI*. O'Reilly & Associates, Inc., 2002. Designing and building distributed applications.

[108] Michi Henning. The Rise and Fall of CORBA. *Commun. ACM*, 51(8):52–57, 2008.

[109] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML. Technical report, World Wide Web Consortium, August 1999. Submission to the World Wide Web Consortium.

[110] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. Lore: a Database Management System for Semistructured Data. *SIGMOD Rec.*, 26(3):54–66, 1997.

[111] Roberto M. Amadio and Luca Cardelli. Subtyping Recursive Types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.

[112] Malcolm Wallace and Colin Ranciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34-9 of *ACM Sigplan Notices*, pages 148–159, N.Y., Sept. 27–29 1999. ACM Press.

[113] Svetlozar Nestorov, Serge Abiteboul, and Rajeev Motwani. Extracting Schema from Semistructured Data. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 295–306, New York, NY, USA, 1998. ACM.

[114] R. Goldman and J. Widom. Approximate DataGuides. In *Proceedings of the second International Workshop WebDB '99, Pennsylvania*, June 1999.

[115] Haruo Hosoya and Benjamin C. Pierce. XDuce: A Statically Typed XML Processing Language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.

[116] Antonio Albano, Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. A Type System For Querying XML documents. In *Proceedings of ACM SIGIR 2000 Workshop On XML and Information Retrieval*, Athens, Greece, July 2000.

[117] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 1–25, London, UK, 2001. Springer-Verlag.

[118] Arnaud Sahuguet. Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask (Extended Abstract). In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 171–183, London, UK, 2001. Springer-Verlag.

[119] Fabio Simeoni, Paolo Manghi, David Lievens, Richard C. H. Connor, and Steve Neely. An Approach to High-Level Language Bindings to XML. *Information & Software Technology*, 44(4):217–228, 2002.

[120] Jonathan Robie. XQL (XML Query Language). Technical report, World Wide Web Consortium, August 1999. Submission to the World Wide Web Consortium.

[121] Ole-Johan Dahl. *SIMULA 67 Common Base Language, Year = 1968.*

[122] Paolo Manghi, Fabio Simeoni, David Lievens, and Richard C. H. Connor. Hybrid applications over XML: integrating the procedural and declarative approaches. In *Fourth ACM CIKM International Workshop on Web Information and Data Management (WIDM 2002), SAIC Headquaters, LcLean, Virginia, USA, November 8, 2002*, pages 9–14, 2002.

[123] John Cowan and Richard Tobin. XML Information Set. World Wide Web Consortium, Recommendation REC-xml-infoset-20011024, October 2001.

[124] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. Technical report, World Wide Web Consortium, November 1999. W3C Recommendation.

[125] Fabio Simeoni, David Lievens, Richard C. H. Connor, and Paolo Manghi. Language Bindings to XML. *IEEE Internet Computing*, 7(1):19–27, 2003.

[126] ECMA. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA:adr, third edition, December 1999.

[127] Ralf Lämmel and Erik Meijer. Revealing the X/O Impedance Mismatch: Changing Lead into Gold. In *SSDGP'06: Proceedings of the 2006 international conference on Datatype-generic programming*, pages 285–367, Berlin, Heidelberg, 2007. Springer-Verlag.

[128] P. Aczel. *Non-well-founded Sets*. CSLI Lecture Notes, 14, 1988.

[129] J. Barwise and L. Moss. *Vicious Circles*. University of Chicago Press, 1996.

[130] Michael Brandt and Fritz Henglein. Coinductive Axiomatization of Recursive Type Equality and Subtyping. *Fundam. Inf.*, 33(4):309–338, 1998.

[131] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM.

[132] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. *The VLDB Journal*, 11(4):354–379, 2002.

[133] Feng Peng and Sudarshan S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 431–442, New York, NY, USA, 2003. ACM.

[134] Y. Diao and M.J. Franklin. High-Performance XML Filtering: An Overview of YFilter. *Data Engineering*, 51:41, 2003.

[135] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams With Deterministic Automata and Stream Indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[136] E. Meijer, B. Beckman, and G. Bierman. Linq: Reconciling object, Relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[137] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., pub-ORA:adr, fourth edition, 2002.

[138] Leonard Richardson and Sam Ruby. *Restful Web Services*. O'Reilly, first edition, 2007.

[139] Craig Walls and Ryan Breidenbach. *Spring in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[140] Robbie Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress)*. APress, 2008.

[141] Dhanji R. Prasanna. *Dependency Injection*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009.

[142] Alain Frisch. OCaml + Xduce. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 192–200, New York, NY, USA, 2006. ACM.

[143] G. Bierman, E. Meijer, and W. Schulte. The Essence of Data Access in cω. *ECOOP 2005-Object-Oriented Programming*, pages 287–311, 2005.

[144] Michael Y. Levin. *Run, XTATIC, run: Efficient Implementation of an Object-oriented Language with Regular Pattern Matching*. PhD thesis, Philadelphia, PA, USA, 2005. AAI3211170.