Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis: TD-5/01

# Extracting Typed Values from Semistructured Databases

*Paolo Manghi*

December 2001

Thesis supervisor:
Prof. Giorgio Ghelli

# Abstract

To date, current investigations on *Semi-Structured Data* (SSD) have focused on query languages that operate directly on graph-based data by matching flexible path expressions against the graph-database topology. The problem with this approach is that it renounces in principle the benefits typically associated with typing information. In particular, storage and query optimisation techniques, representation of user-knowledge of the data, and validation of computations cannot be based upon static typing information. On the other hand, the various attempts to reintroduce types for SSD, while effectively returning some of the benefits of static typing, compromise the irregular nature of SSD databases, by allowing just for mild forms of irregularities.

Our investigation is motivated by the observation that, despite the inherent irregularity of the structure, many or indeed most SSD databases contain one or more subsets that present a high degree of regularity and could therefore be treated as typed values of a programming language. In this thesis we lay the formal foundations underlying a novel query methodology based on an *extraction system* that, given an SSD database and a type of a target language, results in: (*i*) a subset of the database that is semantically equivalent to a value of the given type; (*ii*) a measure that informs the user about the quality of his type with respect to the original database. The extracted subset can then be *converted* into a value of that type and injected into the language environment, where it can be computed over with all the benefits of static typing.

*To my family,*
*and especially to Carla and Rosanna,*
*whose smiles I will never forget.*

*Mountains should be climbed with as little effort as possible and without desire. The reality of your own nature should determine the speed. If you become restless, speed up. If you become winded, slow down. You climb the mountain in an equilibrium between restlessness and exhaustion. Then, when you're no longer thinking ahead, each footstep isn't just a means to an end but a unique event in itself.*

Robert M. Pirsig
"Zen and the Art of Motorcycle Maintenance".

# Acknowledgments

Many people have directly or indirectly partaken in making this thesis possible.

To begin, Prof. Giorgio Ghelli, Prof. Richard Connor, Prof. Antonio Albano, and Fabio Simeoni deserve my special gratitude. Their support, advice, and ideas have been fundamental for the development of this work. Similarly, I wish to thank Prof. Fabio Crestani and Prof. Malcolm Atkinson, whose precious suggestions have contributed to its improvement.

For the uncountable discussions on computer science and life we had in the last four years, I also wish to thank my friends and colleagues Fabio, Vincenzo, Nadia, Valentina, Giuseppe, Gabriele, Anna, Dario, Carlo, Keith, David, and Steve.

I also owe a special thanks to my closest friends, who gave me evidence that spatio-temporal distances are just a state of mind.

Finally, I wish to express my gratitude to Federica and my numerous family for their patience and support. Their invaluable contribute has been to remind me daily that personal life and computer science should have nothing to do with each other.

# Contents

# List of Figures

# Chapter 1

# Introduction

*Database Management Systems* (*DBMSs*) have proven to be extremely effective tools for the automatic management of *information sources*. Their efficacy and effectiveness relies on a number of fundamental assumptions, whose adequacy ultimately depends on specific features of the information source involved. However, when such features do not show, designers and users spend their time and energy building and using inevitably inadequate database systems. As an example of this, consider information sources with frequently changing structure. Schemas of DBMSs handling such sources should mirror these structural changes, and users should recast old data and applications to fit the new schemas.

Over the past few years, there has been an increasing interest in information sources that are too *structurally irregular* to be effectively handled by traditional DBMSs. This inadequacy has called for novel data models and query languages, and has led to the realisation of *Semi-Structured Database Management Systems* (*SSDBMSs*).

In SSDBMSs, *semistructured databases* (*SSDBs*) represent information sources as rooted, labelled graphs. The main characteristic of these *semistructured data models* is the integration of the traditionally separate concepts of schema and data into a single, flexible data structure. This way users can insert arbitrarily structured data into the database, with no concerns about a separate, pre-defined schema. In addition, data can be successively retrieved by referring to the meta-information provided by the labels.

We shall collectively refer to SSDBs as *semi-structured* data (*SSD*) or *self-describing* data, for meta-information is intermixed with data and does not appear as a separate entity.

The query methodology underlying most SSDBMSs relies on SQL-like query languages. Informally, a query specifies a set of *path expressions*, which are a set of labelled paths to be matched against the graph topology of an SSDB. It then returns the subset of the SSDB which reflects the structure identified by the path expressions.

The major drawback of these *navigational* approaches is that SSDBMSs intrinsically disown the general benefits typically associated with static typing in DBMSs. Indeed, due to the absence of a pre-defined schema, data access and query optimisation techniques cannot be supported, user knowledge of the data is harder to acquire, and query correctness cannot be guaranteed. In addition, programmers are forced to write applications in the low-level algebra of labelled graphs.

Attempts to recover some of the advantages of static typing by reintroducing the concept of schema for SSD have limited applicability. The efficacy of these techniques degrades in the presence of information sources with a considerable amount of irregularities, where the only reasonable solution seems to be the navigational approach.

Our investigation is motivated by the observation that, due to the flexibility of labelled graphs, SSDBs may also represent regular information sources, such as those typically represented by the values of a typed language. Intuitively, as illustrated in Figure 1.1, *regular* SSDBs could be thus conveniently converted into the equivalent typed values and computed over under the governance of the language's static typing regime.

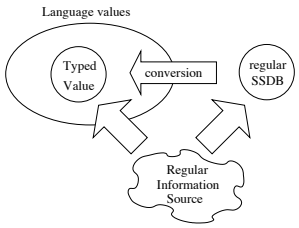



Figure 1.1: Equivalence of expressiveness between values of a typed language and SSDBs.

As SSDBs are usually adopted to represent irregular information sources, this observation is apparently of no practical value. Despite the irregularity of the structure, however, many or indeed most SSDBs "contain" one or more regular SSDBs. We are specifically interested in *identifying* the regular subsets of an SSDB that are equivalent to language values of a given type. When these subsets can be identified, we can *generate* the corresponding values and operate over them in a typed language.

Our approach is independent from the amount of irregularity of an SSDB, and aims at recovering all benefits of static typing whenever this may be convenient. Due to their complementary focus, in particular, we believe that our approach can be combined with the navigational ones in a system that offers complete support

for the management of SSDBs. Consider, for example, the management of irregular information sources with a large, structurally regular core. In our system, such sources would be represented by SSDBs to be indifferently computed over by navigational queries, for operations potentially involving data irregularities, and by typed applications, for operations regarding the regular core of the SSDB.

In its complete version, the system could be used with the further purpose of enabling typed applications to safely *update* SSDBs. The problem, not investigated in this work, is that of modifying the original SSDB according to the modifications carried out to the extracted values.

## 1.1 Thesis goals

This thesis is about the realisation of *extraction mechanisms* for typed programming languages.

First, we provide a specification for the extraction process, according to which any language can be associated with a notion of *extractability*. Extractability for a language captures the concept of value extractable from an SSDB according to a given type. Based on extractability a corresponding algorithm can thus be constructed and proved correct.

We then show the feasibility of extraction mechanisms for most typed programming language in use. To achieve this, we first define the set $S$ of SSDBs, and then characterise and implement an extraction mechanism for a representative language $L$. The definition of $L$ consists of a typing relation between a set $D$ of values and a type language $\mathbf{T}$, where $\mathbf{T}$ comprises a set of standard types: atomic, record, collection, union, and recursive types.

We believe that the generality of $L$ entails an informal proof of the feasibility of an extraction mechanism for all typed languages that support at least a subset of the types in $\mathbf{T}$. Furthermore, other types could find a suitable mapping in $S$ and more specific extraction mechanism could be devised.

Specifically, we formally characterise extractability for $L$. A value $d \in D$ is *extractable* from an SSDB $s \in S$ according to a type $T \in \mathbf{T}$, if $d$ is *of type* $T$ and there exists an $s' \in S$ *included* in $s$ such that $s'$ is *equivalent* to $d$.

Based on extractability, we provide a corresponding algorithm `Extraction` for $L$. Given $s$ and $T$, the algorithm returns a value $d$ extractable from $s$ according to $T$, if one exists, or else it fails. In case of success, `Extraction` returns also a measure of quality of the extraction process. Such measure, called *precision*,[1] quantifies the practical value of $d$ in terms of the amount of information in $s$ that is potentially relevant to $T$ but not extracted in $d$. Precision may help users at defining a better input type, and thus extract a larger and more useful subset of data.

[1] Not to be confused with the notion of precision in information retrieval.

Moreover, we formally prove that `Extraction` is *sound* with respect to extractability of $L$. This ensures that if `Extraction`$(s,T)$ returns $d$, then $d$ is extractable from $s$ according to $T$. We also show that the algorithm is not complete with respect to extractability. Accordingly, `Extraction`$(s,T)$ may fail even if there exists a value extractable from $s$ according to $T$. We discuss the consequences of this in terms of the usability of the algorithm. To conclude, we define a complete algorithm `Extraction`$_t$ for the extraction of values of $L$ from *tree-structured* SSDBs.

Finally, we shall illustrate the applicability of extraction mechanisms by presenting the system *SNAQue*. SNAQue enables CORBA-compliant languages to compute over regular subsets of XML SSDBs.

## 1.2 Thesis outline

The work is organised as follows. The first two chapters give an overview of SSD research. Specifically, Chapter 2 focuses on the main motivations behind SSD introduction, and presents semistructured data models and query languages. Chapter 3 completes the SSD survey by discussing the techniques proposed in the literature to overcome the absence of a schema in SSDBs.

The general principles underlying extraction mechanisms, along with possible applications and implementations of the mechanisms, are described in Chapter 4. The subsequent chapters define the extraction mechanism for the language $L$. In particular, Chapter 5 provides a definition of the SSD domain we shall refer to in our study. Chapter 6 defines the representative typed language $L$, providing a type language, a type equivalence relation, a set of values, and a typing relation. The mapping from language values to SSDBs is also defined here.

Chapter 7 shows an extraction algorithm `Extraction` for $L$ based on the notion of extractability for $L$. The notion of precision of extraction is defined and termination of the algorithm is proven. In Chapter 8, the proof of soundness for the algorithm `Extraction` is illustrated and the general problems behind completeness of `Extraction` are extensively discussed.

Chapter 9 shows a practical application of an extraction mechanism by reporting the results of a project called *Strathclyde Novel Architecture for Querying document Exchange format* (*SNAQue*). A prototype of SNAQue has been developed at the Department of Computer Science of the University of Strathclyde, Glasgow (UK).

# Chapter 2

# Semistructured Data

In the last few years, there has been an increasing interest in storing, handling and querying *semistructured data*. The literature does not provide a unique clear definition of semistructured data, which have been indifferently referred to as *data whose structure is not known in advance*, *data stored out of a database*, *XML files*, *data on the Web*, *data with irregular structure*, etc.

In this thesis, we refer to a definition of semistructured data which abstracts from concepts such as regularity or irregularity of the structure and only depends on the data models through which the data are represented. Moreover, we show the motivations behind the introduction of semistructured data models, which ground on the inappropriateness of *Database Management Systems* (DBMSs) to handle peculiar kind of information. We believe that separating the data models from the information sources to be handled can provide a strong reference for understanding the rational behind semistructured data research.

DBMSs offer services for the efficient handling of *information sources*. An information source is a collection of information characterised by a specific internal organisation, i.e. a *structure*, which can be exploited to identify specific portions of the current *instance* of the collection. In DBMSs, a *schema* is the representation of the structure of the generic instance of an information source, while a *database* represents a specific instance of an information source.

In this Chapter we shall explore the relation between the definition of a schema and quality and efficiency of the corresponding database, to find out that DBMSs are no longer convenient when the schema is not *optimal*. Indeed, we shall see that a useful schema describes *regular* data, thereby providing high-quality modelling of the information source involved, supporting high-performance operations over the corresponding database, and guaranteeing correctness of applications. Vice versa, a non optimal schema generally describes *irregular* data, thereby providing the same benefits with unreasonable human and system costs.

We shall define *irregular (regular) information sources* those information sources whose structure cannot (can) be described by an optimal schema in a DBMS, and

whose instances are hence represented by irregular (regular) data.

The inappropriateness of DBMSs to handle semistructured information sources, called out for novel technologies. Investigations led to the definition of SSDBMSs and *semistructured data models*, according to which any information source is represented as a rooted, directed, labelled graph carrying values on the leaves. Accordingly, in SSDBMSs a database, namely a *semistructured database* (*SSDB*), is a collection of electronic data representing both structure and instance of an information source. We shall define as *semistructured data* (*SSD*) any collection of electronic data forming an SSDB.

Note that, differently from traditional data models, which provide type languages to define expressive user-defined schemas, SSD models offer only a single predefined data structure to represent SSDBs. The choice of a graph-like structure has a twofold advantage: labelled graphs can model any sort of information source, and data can be inserted into the database at any time, with no restrictions on the structure of the data. At the same human and system costs, an SSDB may represent regular or irregular information sources.

On the other hand, the graph-structure of SSDBs inevitably impoverishes the level of interaction with the data for both users and system. Indeed, *semistructured query languages* support commands to run queries over the graph structure of a database, so as to return smaller, possibly more regular views of the original database. Hence, users are matter of factly operating only over graphs, as no other data structure is available. Most importantly, due to the lack of a schema, i.e. an explicit description of the database's content, the underlying system cannot support the typical benefits associated with static typing in DBMSs, even in presence of regular data.

## 2.1 Database construction

The main motivations behind the introduction of SSD research are to be found in the construction process of a traditional database, which leads from an information source to a correspondent database. We identify two approaches to traditional database construction, namely *schema first* and *data-first*, whose difference stays in the kind of information source involved.

- In schema-first approaches the information source is a reality to be electronically organised, namely a *problem domain*. The realisation of a database consists of a precise sequence of stages involving the *modelling* of the reality and the *implementation* of the resulting model, with particular attention to the modelling of the structure of the problem domain and its relative implementation through a database *schema*. The database, representing a particular instance of the problem domain, is populated only on a second stage.

- In data-first approaches the information sources are electronic data available before the database is constructed. Data engineering analysis must be performed, in order to realise the database schema to which the interesting subset of the external data source will conform, as well as the software required to automatically populate the database with that source.

### 2.1.1 Schema-first approaches

In schema-first approaches (see Figure 2.1) a database is realised according to a long and engaging work, which begins with the analysis and modelling of the structure of the information source of a reality, namely a *problem domain*, and ends with a *problem implementation* which satisfies at best the requirements of designers.

Usually, a generic instance of the problem domain can be conceived as a set of *entities* characterised by a specific *structure*. According to this structure, entities are associated with a set of *properties*, which are facts describing a feature of an entity, and can be classified in *categories*, each gathering entities featuring the same set of properties. Modelling, in database design, consists in identifying the set of categories of a generic instance of the problem domain by means of the *abstraction mechanisms* of a given *data model*. A *conceptual model* is the result of modelling a problem domain according to a given data model.

Consider the problem domain of a library, where the category of *books* contains entities corresponding to individual *books*, each characterised by the properties *title* and *authors*. An analyst, working with object-oriented data model abstraction mechanisms, would model this problem domain as a *class Books* of *objects* with *properties title* and *authors*. Similarly, using relational data model abstraction mechanisms, the analyst would have represented this problem domain as a *relation* with *attributes title* and *authors*.

A conceptual model becomes a database when implemented through a computer language embodying the abstraction mechanisms of a particular data model. The resulting system encompasses schema, data, and applications as described by the conceptual model and is called *problem implementation*.[1]

As mentioned above, the main issue of conceptual models is that of providing a non-ambiguous description of the structural organisation of the entities of a problem domain. On the same line, problem implementations focus on an accurate realisation of a *schema*, commonly intended as a computer description of the structure of the data that will be hosted in the database. The name schema-first is due to the fact that database population, i.e. the operation of creating the electronic data representing the current instance of a problem domain, takes place after the creation of the schema, i.e. the representation of the structure of the problem domain.

[1]Note that conceptual model and problem implementation may be based on different data models. Mappings from data models into others are available, allowing the realisation of problem implementations which are sound with respect to conceptual models based on different data models.

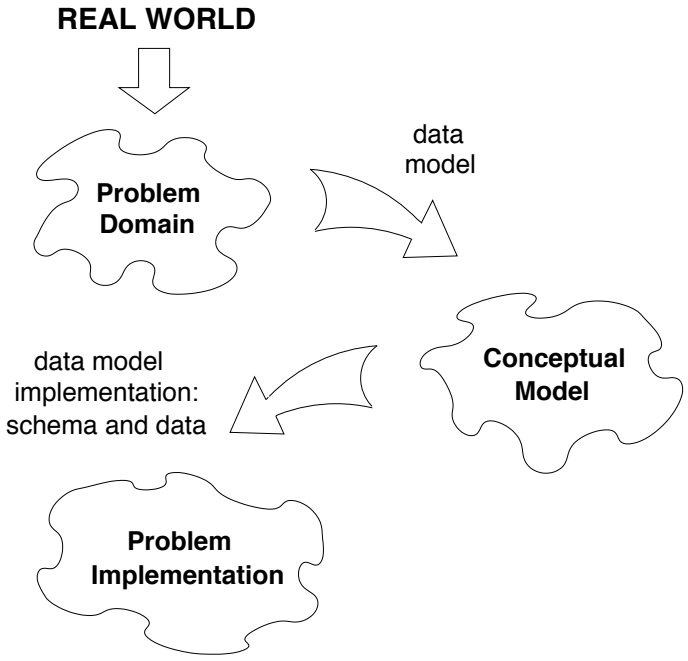



Figure 2.1: From the real world to a database

### 2.1.2 Data-first approaches

In data-first approaches, from which the name, the problem is that of developing a database to handle a subset of data stored in a set of given electronic information sources (see Figure 2.2).

These data sources can be classified as *databases* or *documents*. Databases contain data whose purpose is that of being handled with query languages, while documents contain data whose main purpose is that of being crated, modified, and eventually visualised by means of specific applications.

Unlike schema-first approaches, in data-first methodologies there is no need for a proper modelling phase as the information sources are generally characterised by either an explicit description of the structure, i.e. the schema of the database, or an implicit structure, i.e. a precise textual pattern which outlines the relevant

information in a document. Instead, an engineering analysis of the data sources involved is required, in order to define a schema mirroring the structure of the subset of data of interest.

Once the schema has been constructed, the corresponding database is ready to be populated with the relevant set of data. This operation involves the transformation of the relevant subset of the data sources into data instances conforming to the schema of the target database. Such transformation is performed by ad-hoc software, namely *wrappers* [7, 8, 69, 43, 18, 80], which retrieves the relevant subset of data from the data sources to insert it into the target database according to its new structure.

In general, the realisation of wrappers is a non-trivial and ad-hoc task. In particular, observe that wrappers moving data from a source database into a target one, rely on DBMSs primitives for both extracting and inserting data. Instead, wrappers that move data from documents into a target database, are in fact parsers which search for the data identified by a given textual pattern and then insert them into the target database through the appropriate primitives. Next, we shall see that there are various classes of documents, and point out which of these are more suitable for data-first approaches.

### Data in documents

Nowadays, a large amount of the information processed when working on a computer is stored in *documents* rather than in databases. Consider for example the World Wide Web, which is the greatest repository of information on earth. Data on the Web is stored in documents, and the only way to access the information therein is typically through visualisation and reading. Needless to tell, if inserted into a database, this information could be queried over with great advantage for system users.

To achieve this, users apply data-first approaches: they analyse their documents, identify a textual pattern which corresponds to a DBMS data structure, realise a DBMS schema according to that pattern, and develop the corresponding wrappers. However, this process is not applicable to all kinds of documents, some of which may be illegible, hence not analysable, by humans. In particular, we can identify two main categories of documents:

- *Interpreted documents*: this class gathers documents that can only be interpreted by the applications that created them. Examples are *zip* files, *ps* files, pictures, and sound, which can be modified and/or visualised and/or executed only through specific applications, such as *Win-Zip*, *Ghostview*, *Photoshop* and so on. In summary, the storage format of interpreted documents focuses on how to represent fonts, styles, characters, colors, pixels, and so on, and is hardly readable by a human.

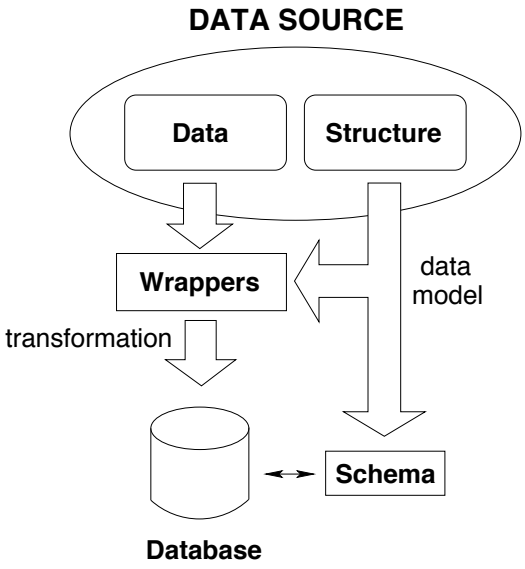



Figure 2.2: From electronic data to a database

- *Legible documents*: this class gathers documents whose main purpose is that of being modified and visualised for consultation by humans.[2] Well known instances of these documents are BibTex files, HTML files, XML files, digital libraries, on-line documentation, e-commerce database files and so on. Of course, legible documents may also be interpreted by specific applications, such as *BibTex, Latex*, and *Web browsers*, but their storage format outlines a human readable content [37, 1].

Data-first approaches are applicable to legible documents only, which, unlike interpreted documents, can be read and manipulated by humans.

Consider for example BibTex files, whose content could be queried with great advantage for documents writers. BibTex files data closely resemble relational data, as they consist of a set of entries that could be mapped onto a list of records. Hence,

[2]It is hard to trace a neat line between interpreted and legible documents. For instance, is a Word file interpreted or legible? The answer is up to the human ability of interpreting the underlying structure of a document format. However, we introduced this distinction to help the reader at understanding what sort of application domains are involved.

if the set of BibTex files at hand contains a list of equally structured entries, users could easily define a relational schema and write ad-hoc software to move the entries content into the target relational database.

As a further example, consider the HTML files generated as responses of queries run over a remote relational database through a Web interface. As the structure of the query response is presumably fixed in time, hence predictable, one may think of defining a local database to handle the regular information within such files. A programmer may design a database schema corresponding to the identified structure and develop wrappers to translate the HTML pages into data conforming to the schema.

## 2.2 Schemas

For both data-first and schema first approaches the efficiency of an overall database strictly depends on the relationships between the schema and the database, and the schema and the structure of the information source involved. In the following we capture the various facets of this relationship in the notion of *optimal schema*, and claim that DBMS databases should be used only when built around an optimal schema.

### 2.2.1 Schema benefits

DBMSs typically provide a *query language* for the specification of *queries*, which are powerful applications for the insertion, modification and retrieval of data into a database. In general, a query can be regarded as a compound of *structural* and *operational* properties. For example, an *SQL* query generally consists of a set of relations (*from* clause) plus a set of predicates over their instances (*where* clause) and a set of operations to be applied over the filtered data (*select*, *insert*, or *delete* clauses). Such queries are performed by traversing only the current instances of the specified relations, selecting the subsets of such instances which respect the given predicates, and eventually returning the result of the operations applied to the data thus filtered.

Query languages do not provide full support for the realisation of complex application systems computing over a DBMS. High-level applications, such as user interfaces, are realised by means of more sophisticated programming languages, whose type systems are compatible with the DBMS data models. Consequently, queries can be perfectly integrated with high-level applications, which otherwise could not interact with the database.

Accordingly, applications over a DBMS may be of various forms and complexity, ranging from simple queries to high-level applications. In this general context, schemas are important in DBMSs for they entail the following benefits:

- system and users have access to a short and neat description of the data *potentially* present in the database:
  - the system can check the *correctness* of the applications: an application is correct if it aims at computing over data described by the schema, i.e. data that can be potentially contained into the database, and, more generally, if run-time errors cannot occur or can be prevented;
  - users have the understanding of what is in the database and can define potentially correct applications;
- the system can optimise both space and time efficiency:
  - databases omit information that may be kept within the schema rather than being repeated in each instance of the data;
  - standard and optimised data access techniques can be devised: being the data stored according to a particular structure, if statistical information is kept up to date, a *query optimizer* can be built, as well as data structures for intelligent access to the data (indexes).

### Correctness of applications

Due to the high level of complexity of an application system constructed around a DBMS, the property of *type correctness* is of paramount importance as it ensures that no-run time errors will occur and that no data-inconsistency will arise.

Informally, an application is type correct if the correspondent computations manipulate values only with operations associated with the respective types. This property can be statically checked at compile time, before applications are executed, due to the presence of types assigned to the input and output of the applications, and to the set of pre-defined operators applicable over the values of these types.

However, DBMS applications also interact with queries, i.e. applications specifically operating over the database. Accordingly, so as to ensure that the result of a query can be correctly used by other applications, the query must be proven correct with respect to the database. *Query correctness* should hold whenever a query is executed and can be generally defined as follows:

**Definition 2.2.1** *(Query correctness in DBMS) Let $Q$ be a query. $Q$ is* correct *if* there can be *data in the database that satisfy the structural properties of $Q$.*

This property ensures the *good sense* of $Q$, which means that if $Q$ yields an empty result this can be interpreted as *no data in the database satisfied the predicates of the query.*

Finally, data is stored in a database according to a precise format, strictly related with the structure of the data as described in the schema. This means that at any

time no datum in the database may have structural properties not described by the schema. Therefore, modulo schema modification, queries can be checked for correctness, once for all, at compile time, by matching the structural properties of the query against the schema.

### 2.2.2 Optimal schemas

The *quality* of a database depends on the relationship between the schema and the structure of the information source, i.e. a problem domain or an electronic data source, to be handled with a DBMS. Moreover, DBMSs rely on a great deal of system artifacts, whose *performance* strictly depends on the relationship between the schema and the database. A schema that satisfies at best these relationships is called *optimal schema*.

**Definition 2.2.2** *(Optimal schema)*
*A schema is* optimal *in presence of:*

- good modelling*: the static irregularities of the information source to be handled by the DBMS, i.e. the properties of entities which are not common to all entities of a category, are well-described by the schema. This is the case when the relationship between schema and information source is not affected by an excessive use of* information loss *or* typing loss*, where:*
  - *information loss is adopted to produce a simpler schema: the static irregularities of the information source are not represented in the schema;*
  - *typing loss is adopted to produce a homogeneous schema: the static irregularities of the information source are extended to all entities of the category.*
- data efficacy*: the relationship between schema and data fulfills the following requirements:*
  - *the schema is a short description of the data: in presence of a schema as large as the data, optimisation techniques are totally obsolete;*
  - *the schema is quite stable in time: modification to the schema are notoriously expensive in terms of human work, as they require the modification of old data and applications [74].*

DBMSs research studied how to efficiently handle *regular information sources*, which are those leading to optimal schemas. Regular information sources are characterised by a structure which is quite stable in time and is not affected by structural irregularities, i.e. in general, entities belonging to the same category feature the same set of properties.

In the following we study information sources that cannot be efficiently handled by a DBMS, and claim they are the main motivation behind the introduction of SSD research.

## 2.3 Irregular information sources

Despite of the database design approach adopted, be it schema-first or data first, when the resulting schemas are *not optimal*, traditional database methodologies turn out to be extremely inefficient. This happens for *irregular information sources* whose importance derives from the structural irregularity and instability they feature, i.e. those affected by one or both of the following irregularities:

- *static irregularities*: entities of the information source that belong to the same category feature different properties; the structure of the generic instance of such information source is inherently irregular, thereby good modelling or data efficacy cannot be kept over a reasonable threshold;
- *dynamic irregularities*: the structure of the information sources is unstable and frequently changes in time, hence data efficacy cannot be provided.

Information sources bearing either static or dynamic irregularities are called *irregular*, as their inherent structure does not lead to an optimal schema. However, as we shall see in the next Section, irregular information sources may be handled electronically through SSD technology. Below, we exemplify known schema-first and data-first irregular information sources.

### 2.3.1 Irregular problem domains

Today, common examples of irregular problem domains are given by problem domains of specific research fields, such as biology, palaeobiology, and similar ones.

For instance, although fossil information sources generally present a common structural pattern that could be represented in a traditional database schema, each fossil may also be associated with further peculiar, relevant information. Moreover, the incessant discoveries constantly introduce new fossils, hence new structural properties.

Such information source is certainly affected by static irregularities as the category of fossils features entities with relevant and different properties. Any attempt to model such scenario with DBMS data models would either make heavy usage of information loss, in order to keep only the properties common to all fossils, or type loss, in order to extend the properties peculiar to each fossil to the category of all fossils.

An alternative is that of considering the category of fossils as a compound of different categories, each peculiar to a limited number of fossils, i.e. those featuring

the same properties. In this case, good modelling may be provided, but data efficacy would be very low, as the schema would be almost the size of the data.

Finally, note that in such scenario, data efficacy would be quite low anyway, due to the frequently changing structure of the information source. Indeed, this would entail changes to the schema, which are likely to imply modifications to both the data and the applications.

For the reasons exemplified above, whenever a problem domain is affected by static and dynamic irregularities, DBMSs cannot be generally adopted and other tools should be employed instead [87].

### 2.3.2 Irregular data sources

In data-first approaches the structure of relevant data plays an important role as it identifies interesting data in the source documents or database, and it may implicitly define a schema for the target database. However, the resulting schema may not be optimal. Indeed, blending different data sources together and storing them in a DBMS database, or either moving document information into databases, may lead to irregular collections of data. Next, we discuss these common scenarios.

#### Integration of different information sources

A challenging issue in database research is that of the integration of heterogeneous information sources, in order to query them together in the same database system.

The problem is that of blending data deriving from various information sources, such as relational or object-oriented databases, the Web, file systems, and others, in an integrated data repository, so as to query them all together. Data in such a repository could be modelled in the general framework of object-oriented data, but the overall structure is likely to be irregular. Indeed, some objects may have missing attributes, others may have multiple occurrences of the same attribute, the same attribute may have different types in different objects, semantically related information may be represented differently in various objects. The resulting data is therefore inherently irregular, and cannot be efficiently stored in a DBMS database.

#### Legible documents

Generally, the structure of information stored within documents is not optimal. Indeed, optimal structure is typically associated with data whose main purpose is that of being queried over, while the purpose of documents is that of being read by humans and interpreted by specific applications.

For example, in BibTex files it is customary to find compulsory entry fields missing. Furthermore, while some fields have meaningful structure, e.g. *author*, there are complex features, such as abbreviations or cross references that are not easy to describe in some database systems (cf. [1]). Due to these static irregularities,

the application of data-first techniques to BibTex files is likely to lead to an inefficient database.

In fact, the structure of data stored in legible documents is often irregular, unknown in advance, and even when it is known, it may change without notice. For such reasons, documents constitute a potential source of irregular information.

#### Data on the Web and eXtensible Markup Language

The success of the World Wide Web is largely due to the adoption of *HTML* (*Hyper Text Markup Language*) [83], and, more recently, to the introduction of *XML* (*eXtensible Markup Language* [24]). Both markup languages have been proposed as international standards for publishing Web documents, providing a common, simple and human legible format for documents. Since Web-documents favored quick and easy information exchange, people from the Web community started to re-design or convert their data onto Web documents so as to make them available to an increasingly wider community.

In particular, due to its flexibility and expressiveness, nowadays XML incessantly plays the role of a standard *data-exchange* format rather than that of a standard *document-exchange* format. XML documents are explicitly intended as information to be queried over, this fact generating an enormous demand for XML-as-database technology.

For example, consider the Home pages of the academics of a Computer Science Department. These pages may contain some similar information, such as *name*, *e-mail*, *photo*, *age* and *address*. However, some of this information may be missing in some pages, while extra information may be present in others. Transferring this information into a database may result in a quite inefficient setting, due to the bad definition of the corresponding schema. In fact, since all the information should be preserved, the schema would be affected by typing loss or information loss, or lead to a schema that is almost a copy of the data. In addition, once the mapping from the HTML (XML) source onto the database schema is designed and the correspondent wrappers are written, there is no certainty for future HTML (XML) pages to fit with the current schema. Accordingly, HTML documents, as well as XML documents, are often taken as examples of irregular information sources.

## 2.4 Semistructured Data

We have seen that irregular information sources are characterised by a structure that is either:

- too variable to be represented by a stable schema;
- too irregular to be represented by a short and clear schema.

These observations led to the definition of *semistructured data models*, according to which databases are self-describing collections of data represented as rooted, directed, labelled trees or graphs [25]. Note that, unlike traditional data models, which provide a set of abstractions to be appropriately combined to define the conceptual model of a database, SSD models offer just one abstraction mechanism. Accordingly, system developers are not concerned with the creation of a schema, which is implicitly defined by the data model and describes all possible *semistructured databases* (*SSDBs*) as labelled trees or graphs. In this work we define *semistructured data* as data stored in an SSDB.

The most representative semistructured data model is the *Object Exchange Model* (OEM) [81]. The novelty of this data model is in the fact that it implicitly defines a schema for all possible databases, which are represented as rooted, labelled, directed graphs with values into the leaves. In particular, OEM consists only of the definition of the set of SSDBs by means of the following BNF:

$$
\begin{aligned}
db &::= Node \\
Node &::= < id, Label, Value > \\
Value &::= Atomic \mid \{Node, \ldots, Node\} \mid id
\end{aligned}
$$

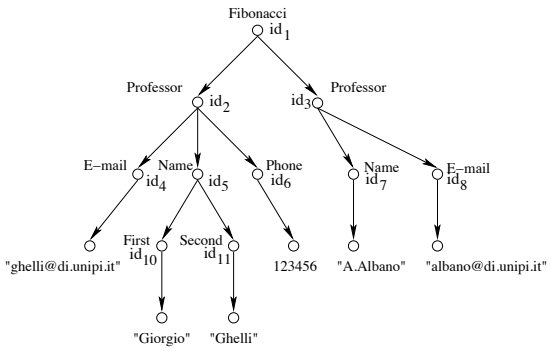
where the *identifiers id* have unique identity in the model.

From the modelling point of view, unlike traditional data models, OEM provides a conceptual model with the unique category of graphs and, by the grammar above, implicitly defines the corresponding schema. Thus, the representation of the structure of an information source is left to the expressivity of the single database, which can only represent entities and properties, i.e. associations, between entities.

By definition, entities of an instance, which are represented by identifiers in the database, are possibly related with other entities for being their property values; properties are uniquely represented in the database by pairs $(label, id)$, where $id$ is a target node. Note that identifiers are considered as independent individuals and not as members of a class or a relation as in traditional data models.

Figure 2.3 graphically exemplifies an OEM collection. Each identifier is associated with a label which provides its description, and may have an arbitrary number of children that may be equally named. Furthermore, a child may be an ancestor of its parent.

Note how the data model consents the definition of databases featuring any sort of static irregularity. Entities that belong to the same category, such as the entities labelled as *Professor* in the example, may be feature different properties, e.g. *Phone*. Furthermore, there is no restriction on the names of the properties of an entity, which can be repeated as for *Professor* for the entity named as *Fibonacci*. Finally, other entities, with arbitrary structural properties, may be added to the actual database at any time.




```
<FIBONACCI>
    <PROFESSOR>
        <E-MAIL>
           ghelli@di.unipi.it
        </E-MAIL>
        <NAME>
           <FIRST>
              Giorgio
           </FIRST>
           <SECOND>
              Ghelli
           </SECOND>
        </NAME>
        <PHONE>
           123456
        </PHONE>
    </PROFESSOR>
    <PROFESSOR>
        <NAME>
           A.Albano
        </NAME>
        <E-MAIL>
           albano@di.unipi.it
        </E-MAIL>
    </PROFESSOR>
</FIBONACCI>
```

Figure 2.3: OEM and XML SSDBs of the Fibonacci's group pages

Different authors considered specific restrictions or representations for their data models, depending on the issues they were seeking. Typical examples are, narrowing the analysis to trees, so as to avoid tedious reasoning about cycles and shared nodes, and considering labelled edges rather than labelled nodes. A notable case is that of XML documents, which are a widely accepted representation for SSDBs due to the hierarchical structure of the format (cf. [4, 88]). For instance, consider the XML document corresponding to the OEM collection in Figure 2.3.

An SSDB is a representation of the structure and of the instance of an information source. This integration makes SSD technology extremely flexible as it gives to the user the ability of:

- freely inserting or deleting data representing portions of the information source at any time;
- modifying the database in correspondence of changes to the information source at any time with very low costs.

In practice, these models have been applied to quite a few research prototypes, working on the areas of data integration and conversion (cf. [36, 39, 45]), Web Site management (cf. [86, 54, 59]), general purpose management of semistructured data (cf. [29, 51]), and XML data management, which we shall discuss in the following.

## 2.5 Semistructured data manipulation

SSD models provide the flexibility required to represent SSD. On the other hand, system and users are provided only with:

1. the semantic information given by the labels;
2. the implicit structural information derived from the hierarchical structure of the database.

Consequently, the data can only be accessed and manipulated as a labelled graph with unknown topology. These features considerably limit the range of possible operations over the data, which can only be treated as graphs, and inevitably reduces the number of system facilities, such as query optimisation and the usage of indexes over the data. In this Section we introduce the general feature of the most common semistructured query languages.

### 2.5.1 Query languages

*Semistructured Query Languages* (*SSDQLs*) are fundamentally tools to identify possibly more regular, smaller, views of large SSDBs. Views are the result of the execution of queries, which specify a set of assumptions over the structure of the data to be queried.

For instance, a query over the collection in Figure 2.3, written in *Lorel* [3], the language defined by the *TSIMMIS* project group, looks like:

```
select result:x.Second
from Fibonacci.*.Professor x
where ''ghelli'' in x.E-mail
```

The query searches the surname of those professors whose e-mail contains the string *ghelli*, and returns a tree-structured SSDB with one edge *result* for each of the surnames found in the process. Moreover, even though this does not apply to our example, the clause $*$ states that the node labelled *Professor* could have been at arbitrary depth in the database graph, as long as it was a descendant of the root node labelled *Fibonacci*.

An SSDQL query consists of three parts, to be executed separately to get a result: *binding*, *filtering*, and *constructing*.

First a set of candidate nodes is identified, by providing the structure through which they must be reached from the root of the graph, i.e. the entry point of the database. Subsequently, the resulting set is bound with a variable, such as $x$ in the sample query above.

The structure for the identification of the candidate nodes is generally given in terms of a set of *paths* of the form $l_1, l_2, \ldots, l_n$ to be matched against the SSDB.

The result of applying such paths to an SSDB graph, is the set of nodes $e_n$ such that there exist an edge $(r, l_1, e_1), (e_1, l_2, e_2), \ldots, (e_{n-1}, l_n, e_n)$ in the SSDB, where $r$ is the entry point of the database.

A query specifies a set of paths by means of *Generalised Path Expressions* (*GPEs*, cf. [38, 40]), which are expressions of the form $p_1.p_2, \ldots .p_n$, where the $p_i$'s may be a label $l$ or particular symbols, known as *wild cards*, of the form $*$, ?, |. For example, the GPE $l_1, l_2, \ldots, l_n$ represents only the relative path, while the GPE $l_1.*.l_2$ represents the infinite set of paths which begin with a label $l_1$ and reach a label $l_2$ after encountering an arbitrary number of labels.

The process of querying an SSDB is based on pattern matching techniques exploiting the inherent graph structure of the given data model and the semantic information coded by the GPEs.

Afterwards, the set of nodes can be filtered through a set of predicates applied over the corresponding binding variables, such as `''ghelli'' in x.E-mail` in the query above. The set of nodes triggered by the predicates is in turn applied to a set of constructors, such as `result: x.Second`, which return an SSDB as a result of the query.

The expressive power of SSDQLs is measured on the base of the kind of queries the user can express (cf. [28]). Most languages provide the basic operations required by relational query languages, such as joins and grouping, plus some form of restructuring, i.e. the ability of creating a new SSDB from another one.

Well known query languages are *Lorel* [3, 82] developed by the *TSIMMIS* group, *StruQL* [55], *UnQL* [29], developed by the UnQL group, *YATL* [39] developed by the Verso group, *TQL* [33], and others. These languages typically provide SQL-like constructs, through which programmers may create and query SSDBs.

Furthermore, in the last few years the *World Wide Web Consortium* (*W3C*) focused on the design of standard data models [56] and query algebras [53] for XML SSDBs, so as to provide general guidelines for commercial developers. Well known XML query languages are *Lorel* adapted to XML [63], *XQL* [84], and *XML-QL* [49], *XPATH* [40] developed by the W3C, *XDuce* [66], *Quilt* [35], and *XQuery* [34, 70]. In the next Chapter, we shall see that *XDuce*, *Quilt* and *XQuery* support types for SSDBs, and are the only example of SSDQLs with the expressive power of Turing-complete languages.

### 2.5.2 Query languages implementation

The implementation of SSDBMSs is a known and interesting problem, which embraces most of the issues of traditional database design plus others, stricly peculiar to the challenge of querying SSDBs.

Two main approaches have been explored. The first, and most researched, approach is based on the storage of data in a relational, or object-relational, DBMS, and on the translation of the queries into SQL queries [61]. *Quilt* [32] and *XML-QL*

have been implemented in this way. These implementations usually are not able to support all operations one would require in an SSDQL, due to limitations of the underlying DBMSs which are definitely intendend for different purposes [85].

Another approach, which is much more expensive but overcomes these limitations, consists in the construction of a new complete and specific system. For example, *Lorel* [71], *StruQL* [50], *XML-QL*, and *XQL* have been implemented in this way, while the implementation of *Xyleme* [9] is somehow intermediate. This approach requires the definition of a new storage model [67], a specialized query algebra, as well as the definition of adequate cost-based optimisation techniques. While some efforts have been spent in the context of *Lorel* and, to some extent, of *StruQL*, satisfactory algebras and general cost models for such implementations are still missing.

## 2.6 Semistructured data drawbacks

So far, we have shown how the choice of SSD models may overcome the problem of using otherwise inefficient DBMSs to manage irregular information sources. On the other hand, SSDBMSs are not as powerful as DBMSs as explained in the following.

- *Query optimisations and data access cannot be supported*: due to the absence of a schema the only possible execution plan is the exhaustive traversal of the whole database according to the structure specified by the GPEs; furthermore, the data cannot be stored according to particular data structures, exploiting indexes as in DBMSs, but only according to strategies for the memorisation of labelled graphs.

- *Query correctness cannot be supported*: with reference to Definition 2.2.1 of query correctness for DBMSs, in SSDBMSs query correctness cannot be statically checked as there is no notion of static schema. Indeed, as structure of the database may change at any time, query correctness should be checked whenever a query is executed. Furthermore, since in SSDBs the structure of the data is intermixed with the data, query correctness should be matched against the actual SSDB.

  In conclusion, query correctness in SSDQLs may be dynamically checked, during query execution, by verifying that all paths specified by the query found a match in the traversal of the SSDB. This way, when a query returns an empty result the user learns if this was due to the erroneous GPEs he had specified or, more interestingly, because none of the candidate nodes identified in the binding part survived the query filtering.

  Clearly, this notion of correctness cannot be a discriminating factor for the repeated execution of the query as it could be in a DBMS. Indeed, a query may turn out to be incorrect for one execution and, due to modifications to

the database, correct for the one immediately after. Furthermore, due to the size of a database and to its potentially changing topology, users cannot be aware of the precise structure of the database.

In fact, users define queries after an eye-inspection of the database content, so as to identify peculiar structural properties, and queries are performed in a *greedy* fashion, in an attempt to get back as much information as possible.

- *The language type system is poor*: the absence of a schema and, consequently, of a static form of query correctness, hampers the integration of SSDQLs with traditional typed languages, which require the specification of types for the values to be manipulated and whose type systems are usually much more sophisticated than semistructured data models. Accordingly, the level of complexity of SSDBMS applications is generally limited to the SQL-like queries presented above. Computations over the database may modify the content of an SSDB only by renaming some edges or nodes, changing values in the leaves, or adding nodes and edges to the SSDB.

- *Queries may be hard to define*: the absence of schema, hence of a short description of the data, typically hampers the definition of queries looking for specific data in the SSDB. The users may get hold of structural information only after an eye-inspection of higly-irregular graph-like databases.

There are two main trends in SSDQLs realisation. Those that do not take into account any form of meta-information and those that are based on the assumption that the SSDB comes with a schema. The former are usually a specific case of the latter when a schema is not available. In the following chapter we discuss the setbacks listed above and show how some of them have been overcome by reintroducing the concept of type for SSDBs in SSDQLs.

# Chapter 3

# Typing semistructured data

In the previous Chapter we have highlighted the differences between SSDBMSs and DBMSs. In particular, we concluded that SSDBMSs can be applied to handle any kind of information sources, while DBMSs offer notably efficient services when applied to information sources with stable and regular structure and show very poor performances when applied to irregular information sources. Consequently, for strongly irregular information sources, SSD technology seems to be the only possible solution. However, the lack of schema generates a notable distance between SSDBMSs and DBMSs, summarised by the following aspects:

- *application correctness*: correctness of applications over a database cannot be guaranteed, while in DBMSs it is verified at compile time by means of type checking mechanisms;
- *operations applicable over the data*: data manipulation is limited to queries upon a labelled graph structure, while in DBMSs it is open to sophisticated computations over various typed data structures, such as relation, records, collections, and so on;
- *system optimisations*: queries may be executed by simply sequentially traversing the whole database, while in DBMSs queries are performed by exploiting various forms of optimisations for query execution and data access;
- *user knowledge*: users may come to know the content of a database by an inspection of the overall database, while in DBMS users have access to a schema, which is a short description of the database's content.

Nevertheless, structure-less is not a proper definition for semistructured data models. Indeed, each datum in an SSDB is reachable by traversing the database according to a specific *structure*, i.e. a sequence of labels from the entry point, called *path*. In this Chapter we discuss when and how a separate description of such structure, i.e. a schema, may be statically or dynamically provided for SSDBs.

In particular, we shall see how the choice of either typing methodology for SSD is concerned with a general trade off between limiting the flexibility of SSD models and improving all of the benefits deriving from meta-information.

## 3.1 Static and dynamic typing for SSD

The observation that data in SSDBs are identified by a precise structure has motivated a number of investigations into the applicability of typing techniques to SSDBMSs. There are essentially two different research trends aiming at typing SSD, according to which a type may be provided statically or dynamically.

**Static typing** The relationship between types and data is that of conventional DBMSs, i.e. the data in the database must structurally *conform* to the schema. We shall see that different definitions of schema and conformity may be given, so as to capture irregular data and guarantee some of the benefits of static typing at the same time.

**Dynamic typing** This approach does not impose any constraint on the structure of SSDBs, which can be arbitrarily organised; readable and possibly succinct forms of meta-information are inferred from the structural information represented by the labels of SSDBs, in order to be exploited by the user to better define his queries and by the system to support forms of query optimisation.

In the following we present static and dynamic approaches together with some examples of well-known approaches.

### 3.1.1 Static typing

Static typing techniques for SSD mirror DBMSs schema-first methodologies in that a schema is provided before data population. In both scenarios, this requirement entails a strong assumption of *structural stability* of the information source at hand, whose structure must be fixed in time and known in advance. In general such assumption excludes the applications of such techniques to irregular information sources affected by dynamic irregularities, and static typing techniques are customary applied to information sources mainly affected by static irregularities.

The ability of a type language to describe data irregularities is in a trade-off with static typing benefits, i.e. the ability of the system to provide optimisations, effective definitions of query correctness, user knowledge, and sophisticated data structures. Therefore, various forms of static typing techniques for irregular information sources have been proposed, whose nature depends on the amount of irregularities to be handled by the system. So as to reveal this trade off, next we present a well known approach relying on DBMSs systems endowed with peculiar typing methods, which privileges static typing benefits to system flexibility, and some typing approaches for SSD, which gain further flexibility by loosing some of the benefits.

- *DBMS languages with union types*: mildly irregular information sources are represented as values of traditional data models endowed with union types:
  - query and applications can operate on traditional and intuitive type structures, such as relation, records, collections and others;
  - all static typing benefits can be supported.

  DBMSs and programming languages advantages can be exploited, but the amount of irregularities must be very limited due to system efficiency issues implied by union types.

- *SSDBMSs with low-level type systems*: irregular information sources are represented as SSDBs, whose content is described through an algebra of *low-level types*, which typically describe the structure of labelled graphs, rather than that of values such as records and relations:
  - applications are limited to queries operating over labelled graph structures;
  - depending on the relation of conformity between types and SSDBs, query correctness may assume different shapes or not be supported at all, while query and storage optimisation can be generally supported.

  Some benefits of DBMSs static typing cannot be supported, but low-level types and SSDQLs can generally better cope with irregularities than DBMSs with union types. As examples of low-level types for SSD, we shall show some known examples of XML typed query languages, and the peculiar typing of *UnQL* by means of *graph schemas*.

### Union types

Untagged union types are introduced in programming languages to increase application flexibility by allowing values of different types to be described by the same one. Informally, the semantics of union types states that a value $v$ conforms to a union type $union(T_1, \ldots, T_n)$ if there exists $i : 1, \ldots, n$ such that $v$ conforms to $T_i$. Due to their peculiar nature, union type values should be *projected* into their specific type of the union before they can be referred and manipulated. To this aim, programming languages support particular commands, such as,

```
typeof x is union(T1, ..., Tn)
⋮
union case x of
T1 :< operating on x as a value of type T1 >;
⋮
```

$T_n$ :< `operating on` $x$ `as a value of type` $T_n$ >;
`endcase;`

At run-time the `case` command matches the actual type of the value associated to the variable `x` with the $T_i$'s and executes the branch corresponding to the matching type. Note that, despite of the dynamic type checking required, the introduction of union types does not compromise static type checking of applications.

Sophie Cluet, in [41], observed that language type systems or traditional database data models, endowed with union types, are apt to describe some mild form of irregular information sources. To be convinced of this, recall that static irregularities derive from entities, represented by values $v$, each featuring a different set of properties, represented by the types $T_i$, which all belong to the same category. Union types can be used to represent such category as a collection of type $set(union(T_1, \ldots, T_n))$.

For example, in Figure 3.1, by means of intuitive languages for the definition of values and high-level types in a DBMS application, we provide the value `x` corresponding to the SSDB in Figure 2.3 with the corresponding high-level type `Professors`. Applications and queries operating over the elements of the collection `x` can be checked for correctness, and system and users can take advantage of all benefits usually enforced by DBMSs.

```
let profname = [First = ''Giorgio'', Second = ''Ghelli'' ]
let gg = [ Name = profname, E-mail = ''ghelli@di.unipi.it'',
           Phone = 123456 ]
let aa = [ Name = ''A. Albano'', E-mail = ''albano@di.unipi.it'' ]
let x = { aa, gg }

type Professors = set(Professor)
type Professor = union(record(E-mail:  string;
                              Name:record(First:  string;
                                          Second:  string);
                              Phone:  integer );
                       record(Name:  string; E-mail:string ); );
```

Figure 3.1: Type description for the value representing the Fibonacci's SSDB

Note how a set of OEM nodes with the same label and reachable from the same node, such as `Professor` with `Fibonacci` in our example, intuitively maps into a collection value, such as a relation in relational databases. Similarly, a node reaching differently labelled nodes intuitively maps into a record value.

This approach should be used when the number of static irregularities runs under a certain threshold and the overall data structure is quite stable in time. In

summary, the information source should be *almost* regular. Indeed, a type which accurately describes a value representing a significantly irregular information source would provide little useful abstraction over the database and compromise system's efficiency: such a type would be beyond human understanding and require expensive static and dynamic type checking controls, therefore be of no practical use.

### Integration of different SSDBs

A less obvious approach to the typing of irregular information sources has been proposed in the specific field of integration of different databases. This integration process falls in the category of DBMS data-first approaches and requires the recasting of the data sources according to a new unifying schema. However, as explained in Section 2.3.2, the integration of databases representing different information sources is likely to lead to a collection of irregular data. Such data cannot be re-cast under an optimal schema and should therefore be represented as SSD, thereby losing any static type information associated with the original structured data source.

However, sometimes databases are unified because they contain related information which should be merged to be queried over as a single repository, e.g. databases of professors from different departments. Still, although the databases may be described by very similar schemas, as long as the conceptual models were developed by distinct individuals, the resulting data may present static irregularities. In the presence of a limited number of irregularities, the resulting database could be represented by DBMS languages endowed with union types, as shown in the previous paragraph. However, applications written for the original schemas should be rewritten as they might not be correct for the new schema.

Buneman and Pierce [30] claimed that the data arising from this specific form of integration can be represented by SSDBs whose content is described by a particular type system endowed with untagged union types. Union types capture the irregularities of the resulting data, while specific type rules consent to reduce the number of modifications to the applications which are typically required in correspondence with changes to the schema.

The type system proposed by the authors is the following:

```
T ::= T1 × T2 | (records)
      T1 + T2 | (untagged unions)
      set(T)  | (sets)
      l:T       (singletons)
```

which describes SSDBs as trees, represented by nested record values, where the label of the fields cannot be repeated. In particular, record types are defined as products of singletons of the form `l:T`, but may also include elements of the form $T_1 + T_2$. Hence, a further peculiarity of this language are typed operators to access record structures such as $T_1 \times (T_2 + T_3)$, not to be described here.

Databases could be represented by means of such values, while the schemas could find a match, at least a partial match, in the type system above. Thus, SSDBs could preserve, at least partially, some of the static information provided by the schema of the original data source. Consequently, typed applications may be written to operate over the SSDB, potentially exploiting all benefits of static typing.

The type system is provided with type equivalence rules, subtyping rules and a *distribution rule* over record and union types, which defines the following type equivalence:

$$\mathtt{T}_1 \times (\mathtt{T}_2 + \mathtt{T}_3) \equiv (\mathtt{T}_1 \times \mathtt{T}_2) + (\mathtt{T}_1 \times \mathtt{T}_3)$$

Assume the SSDB is typed as $\mathtt{set}(\mathtt{T}_1 \times \mathtt{T}_2)$ and that a new data source, whose schema maps onto a type $\mathtt{set}(\mathtt{T}_1 \times \mathtt{T}_3)$, should be added to it. Their integration could be typed as $\mathtt{set}((\mathtt{T}_1 \times \mathtt{T}_2) + (\mathtt{T}_1 \times \mathtt{T}_3))$. The distribution rule states that those applications written to operate on the part common to both data sources, i.e. the type $\mathtt{T}_1$, are still correct and can therefore operate on the resulting SSDBs without being modified.

For example consider the integration of two databases constructed to model professors according to the types `db_one` and `db_two`:

```
db_one:
set((Name:  [(First:string) × (Second:string)]) ×
    (E-mail:  string) ×
    (Phone:  integer) )

db_two:
set((Name:  string) ×
    (E-mail:  string) )
```

The resulting database could be typed as `db`, i.e. a set of the union of the core types of the two original sets:

```
db:
set([ (Name:  [(First:string) × (Second:string)]) ×
      (E-mail:  string) ×
      (Phone:  integer) ] +
    [ (Name:  string) ×
      (E-mail:  string) ] )
```

According to the distribution rule, the following type equivalence holds:

```
db = set((E-mail:  string) ×
         [(Name:  (First:string) × (Second:string) ) ×
          (Phone:  integer)
```

```
 +
 (Name:  string)
])
```

Thus, applications operating only on the field `E-mail` and proved correct with respect to both `db_one` and `db_two` can be reused on the new database. In a way, the type system degrades gracefully when new data sources, with variation in structure, are added to the SSDB, while preserving the common structure of the data sources where it exists.

The authors claim that for the amount of irregularities generally encountered in this particular form of data integration, the system shows good performances; in particular, the complexity of the distribution rule algorithm is reasonable. The ideas behind this type system have been extended in subsequent works, in particular in the realisation of the XML processing functional language *XDuce* [66].

### Types for SSDQLs

SSDQLs should be adopted to handle information sources whose irregularities could not be efficiently handled by DBMSs. When dealing with SSDBs, types are abstractions as expressive as GPEs over labelled graphs, and conformity is a matching relation between the structure provided by the type and the structure of the actual database: a database conforms to a type if the paths defined by the type are present in the database according to the type semantics.

For instance, consider XML query languages [57]. The spread of standards for specifying XML meta-information, such as *DTDs* [15] at first and *XML Schema* [52, 89, 21] afterward, called out for the realisation of typed XML query languages, capable of exploiting a schema if available. DTDs and XML schemas are low-level type systems, describing the nested, tagged structure of XML documents and capturing the static irregularities of the data.

For example, a DTD for the XML document in Figure 2.3 may look like:

```
<!ELEMENT Fibonacci (Professor*)>
<!ELEMENT Professor (Name, E-mail, Phone?)>
<!ELEMENT Name ((First, Second) | ♯PCDATA)>
<!ELEMENT E-mail ♯PCDATA>
<!ELEMENT Phone ♯PCDATA>
```

This schema requires the XML documents conforming to it to have an entry point labelled as `Fibonacci`, nesting an arbitrarily long sequence of tags `Professor`. Furthermore, each of such tags should feature two tags `Name` and `E-mail`, possibly followed by an optional tag `Phone`. Finally, `Name` may nest either a simple string, denoted by the type `♯PCDATA`, or a sequence of two tags `First` and `Second`, in turn of type `♯PCDATA`.

Typed XML languages range from those capable of querying typed data to those providing Turing-complete programming paradigms. Examples of the first kind are *XSL* [91], *YATL*, *TeQuyLa* [10, 12, 11], while examples of the second kind are *XDuce*, *XQuery*, and *Quilt*.

For instance, *XDuce* is a Turing-complete typed programming language for the definition and manipulation of XML documents. The type system, which is based on the ideas exposed in [31] and reported in the previous paragraph, have the same expressive power as DTDs and describe XML documents. Language programs are second order typed functions operating over the data that can be checked for correctness before execution. Interestingly, types are used as a matching tool for the data at run-time: data dereference is performed by matching types with data at run-time, in a way that resembles that of untagged union type values. This is because the generic XML document may be described by different types and accessed according to different interpretations.

Based on types, some optimisation techniques have been designed, but not yet developed, for nested queries [44, 42, 68] and GPEs matching [73, 72, 38, 60]. Due to the variety of application contexts, however, there are no precise definitions of query correctness. Correctness is generally intended as a relation between the structure specified by the GPEs in a query and the schema of the database. For example, consider again the XML document in Figure 2.3 and the DTD given above. The query

```
select x
from Fibonacci.Professor.(Name | Fullname) x
```

searches for paths from the root of the document that match either the structure `Fibonacci.Professor.Name` or `Fibonacci.Professor.Fullname`. The query may be considered as correct or incorrect, depending on the kind of correctness policy adopted.

*Existential approaches* to correctness would establish that the query is correct because there exists a non-empty intersection between the paths defined by the schema semantics and the paths specified by the query semantics. The rational underneath such policy is that a query is correct as long as there is a chance to produce a result. The user may be possibly warned by the system about the fact that his query is searching for paths that cannot be found in the database. Such policy, which is inappropriate for traditional type checking, is quite reasonable when dealing with SSDBs.

*Universal approaches* to correctness would instead establish that the query is not correct because the query searches for paths not described by the schema. The underlying policy may state that a query is correct only if all paths in the query find a match in the schema; in other cases, the policy may be more restrictive, stating that a schema with a union type can only be accessed by queries which specify a

case for each member of the union, i.e. the query must match at least one path in the database.

Other refinements of these definitions are possible, each leading to a different notion of query correctness. In general, due to the variety of application scenarios, it is hard to find a solution which applies to all situations. Indeed, only few languages provide a definition of query correctness, with the notable exception of *XDuce*.

The drawbacks of these approaches are that applications are quite poor, as they operate on data modelled as an XML graph; furthermore, as pointed out in the introduction of the Section, these techniques are pointless in presence of strongly irregular information sources.

### UnQL and Graph schemas

In the language *UnQL* [29], SSDBs are edge-labelled graphs where values are specified as the last edges of paths. For example, the *UnQL* database corresponding to the OEM collection in Figure 2.3 is illustrated in Figure 3.2.

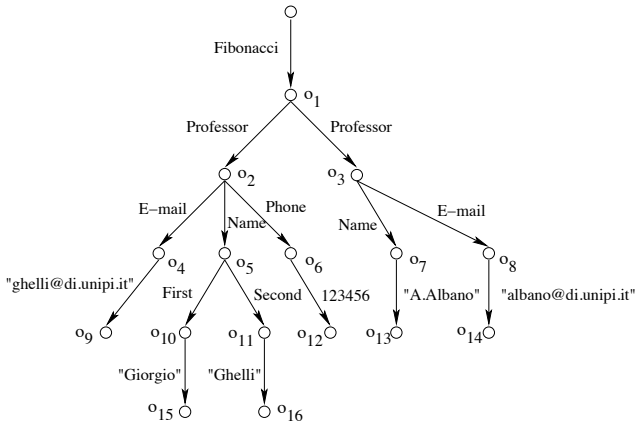



Figure 3.2: UnQL database

In a later stage of development, the language, which is similar to the SSDQLs presented in the previous Chapter, has been extended with a type system [26, 60, 58]. In *UnQL* a type is a graph, namely a *graph schema*, whose edges are marked with constraints, namely unary formulas, over the domain of SSDB labels. A SSDB *db* conforms to a graph schema $G$ ($db \preceq G$) if it is *similar* to it. Informally, this means

there exist a relation $\mathcal{R}$ between the nodes of the SSDB and the nodes of the schema such that:

1. the roots are in $\mathcal{R}$,
2. $(o_1, o'_1) \in \mathcal{R}$ if for each edge $(o_1, l, o_2)$ outgoing a node $o_1$ in the SSDB there exists an edge $(o'_1, p(x), o'_2)$ outgoing the node $o'_1$ of the schema such that:
   (a) the label associated with the former edge verifies the predicate associated with the latter edge: $p(l)$ is true;
   (b) $o_2$ and $o'_2$ are in $\mathcal{R}$.

Note that, graph schemas are not as restrictive as traditional programming language type and low-level types, in that conformity does not enforce the presence of a label outgoing a node. As a consequence of this, the empty SSDB conforms to all possible graph schemas.

Figure 3.3 illustrates an example of graph schema for the database graph in Figure 3.2. Note that, for simplicity, the predicate $p(x) \equiv$ `(`$x$ `= Fibonacci)` is simply replaced with `Fibonacci`. $G_1$ states that if there is an edge labelled *Fibonacci* outgoing the entry point of the database, such edge reaches nodes which have an outgoing edge labelled as *Phone* or others which are not equal to it. In turn, edges labelled *Phone* lead to nodes whose outgoing edges can only bear integer values.

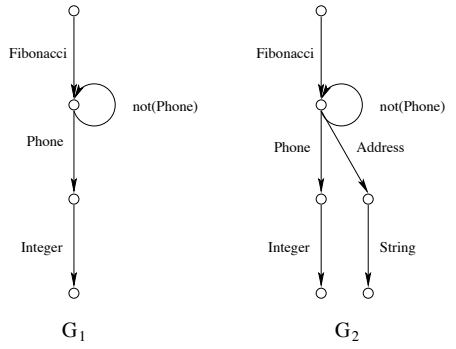



Figure 3.3: Graph schemas

Query correctness cannot be supported because the notion of schema is too loose and does not provide complete information about the data. For instance, the

database graph in Figure 2.3 conforms also to $G_2$. A query $Q$ selecting entities reachable with the path $Professor.Address.x$ should be intuitively correct with respect to $G_2$. However, its result would be empty because the data cannot satisfy the structural properties of the query.

In practice, *UnQL* exploits graph schemas for the definition of typed views over the data ($G_Q$'s), to provide user knowledge, and to improve query optimisation and decomposition.

### 3.1.2 Dynamic typing

Dynamic typing approaches exploit typing information mostly with the purpose of enabling resource optimisation and promoting user understanding of the data. Given an existing SSDB, information about its structure is automatically or semi-automatically inferred by the system.

Notable examples of dynamic typing methodologies are *Representative Objects*, *Dataguides*, and *approximate types*, which we shall discuss in the following.

#### Minimal type inference

These techniques are concerned with inferring a detailed and minimal schema from a given SSDB. *Dataguides* [90, 64] and *Representative Objects* [78] are examples of such schemas.

For example, representative objects are labelled graphs obtained from the original SSDB by keeping track of all possible paths from the root. One such schema is the minimal description of an SSDB and can be used for schema browsing, user knowledge purposes, and basic forms of query optimisations, such as avoiding the execution of queries whose result is proven to be empty. Techniques for schema maintenance have also been developed.

In the presence of SSDBs with shared nodes nested at various levels, schema inference becomes very expensive. These methodologies, however, are also doomed to fail also in presence of a sheer number of irregularities. Indeed, the size of the resulting schema would not provide useful information to the user.

#### Approximate typing

Minimal types provide useful but limited abstraction over an SSDB. In particular, when the data is extremely irregular, the number of possible paths in the SSDB becomes overwhelming and defining precise queries turns into a challenging task. This motivated the development of more sophisticated type inference techniques [77, 76], whose main aim is that of identifying some regular structure underlying very large SSDBs. Nodes in the SSDB are assigned a type, while trying to minimise the *total number of types* and the *deficit*. Deficit measures, given an association between

a node and a type, the amount of information related to the node, typically its outgoing and/or incoming labelled edges, which is disregarded by the type.

Informally, the methodology is based on (*i*) a distance function between SSDBs and (*ii*) a measure for the size of types. Given an SSDB $db$, the problem is that of finding a set of types $\tau$ and a database $db'$ such that (*i*) $db'$ is typed as $\tau$, (*ii*) the size of $\tau$ is smaller of a given threshold, and (*iii*) the distance between $db$ and $db'$ is minimised. The resulting type, together with the associations of nodes to the corresponding types, may then be passed to the user, for user knowledge purposes, and to an optimiser, in order to improve query evaluation.

The authors proved the problem of inferring a perfect type for an SSDB to be NP-complete, and their solution relies on heuristics capable of calculating the *best approximate type* for an arbitrary database. They developed two different strategies, the first based on schemas as Datalog programs, and the second resulting in a traditional structure of nested record types. The difference between the two rests in the notation used: monadic Datalog programs describe SSDB by defining classes of nodes in terms of their incoming and outgoing edges, while records characterise a sets of nodes in terms of their outgoing edges, here corresponding to record value fields.

# Chapter 4

# Extraction Mechanism

Emerging typed approaches provide languages capable of querying SSDBs with the benefits associated with static typing. As shown in Chapter 3, these approaches are profitable when SSDBs do have an irregular but known structure. However, these solutions fail at *fully* recovering the benefits of traditional typed approaches and yet preserve the modelling flexibility granted by SSD.

In this thesis, we present a radically different approach, in that we do not propose a new query language for SSD, but a system for querying SSDBs with existing, computationally complete, typed languages. The system is based on a language-dependent *extraction mechanism*, which performs the extraction of regular subsets of SSDBs that correspond to values of a given type. Accordingly, users can freely populate their SSDBs and fully recover the benefits of static typing when convenient.

We believe there exists a wide range of SSDBs that could be gainfully manipulated with our mechanism. Moreover, our approach should not be regarded as an alternative to the navigational techniques presented in Chapter 2, but rather complementary to them. In combination, the two approaches can provide complete functionality for SSDBs management.

In this Chapter, we first present the intuitions behind the idea of extracting regular subsets of SSDBs and then introduce a specification of the process of realisation of an extraction mechanism for a language. We then discuss possible application scenarios and compare our approach with other typing techniques for SSD.

## 4.1 Intuitions

In earlier chapters, we have observed that typed language values are typically used to represent quite regular information sources, while labelled graphs, i.e. SSDBs, are flexible modelling primitives that can be used to represent any information source. Accordingly, we can draw a *mapping* from the values of a programming language onto those regular SSDBs that represent the same regular information sources.

To be convinced of this, consider a language with record types. A record value,

$d =$ `[firstname = ''Michele'', surname = ''Casini'']`

of type ,

$T =$ `record[firstname: string, surname: string]`,

represents an entity of the problem domain *friends of mine*. Such entity can be equally represented by the SSDB in Figure 4.1. Hence, $d$ could *map* onto $s$, as the latter features the structural-syntactical properties, i.e. the labels, required by $d$ to be identified as a value of type $T$.

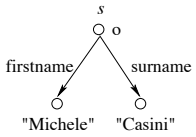



Figure 4.1: SSDB as expressive as the language value $d$.

This mapping emphasises that typed application computing over the values of the language are indirectly computing over the corresponding regular SSDBs. Accordingly, based on the mapping, the SSDB $s$ in Figure 4.1 could be conveniently converted into the corresponding value $d$ of type $T$, so as to be computed over with the benefits of static typing.

As SSDBs are usually adopted to represent irregular information sources, however, this observation is apparently of no practical value. Indeed, as explained in Chapter 2, a type that abstracts over a significantly irregular information source provides little useful information, and is therefore of no practical use.

However, many if not most SSDBs *include* one or more regular SSDBs that represent values abstracted over by *useful* types. This observation inspired the realisation of language-dependent *extraction mechanisms*. These mechanisms first *identify* the regular subsets of an SSDB that are equivalent, according to a given mapping, to values of a given type. Then, if these subsets exist, the corresponding values are *generated* to be injected into the language.

As an example of this process, consider the SSDB $s$ in Fig. 4.2. Intuitively, the regular SSDBs $s_1$, $s_2$, $s_3$ contained in $s$ fulfil the structural requirements entailed by the three language types:

$T_1 =$ `record[a:record[b:integer]]`

$T_2 =$ `coll(record[a:record[b:union(integer, record[c:string])]])`

$T_3 =$ `let rec X = record[a: record[d: X,b:record[c:string]]]`

An extraction mechanism, given $s$ and the three types above, would first identify the three regular subsets $s_1$, $s_2$, and $s_3$, and then generate language values of the form

$d_1 =$ `[a = [b = 1]]`

$d_2 =$ { `[a = [b = 1]], [a = [b = 2]], [a = [b = [c = ‘‘three’’]]` }

$d_3 =$ `x = [a = [d = x, b = [c = ‘‘three’’]]]`

such that $d_1$, $d_2$, $d_3$ map onto $s_1$, $s_2$, $s_3$, respectively. These regular SSDBs can thus be indirectly computed over by applications in $L$ operating over values of types $T_1$, $T_2$ and $T_3$, i.e. under the governance of a static typing regime.

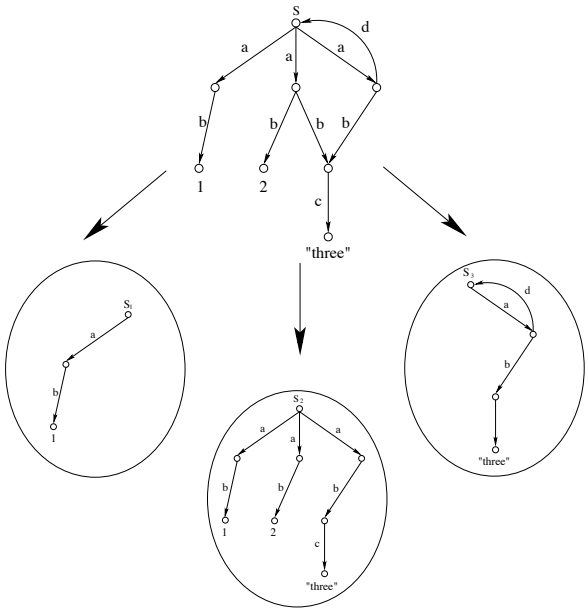



Figure 4.2: Extraction of regular subsets

## 4.2 Extraction mechanisms realisation

The realisation of an extraction mechanism for a language is a non-trivial task, which first focuses on giving a formal characterisation of extraction and then concentrates on the definition of a corresponding algorithm.

In particular, any language can be associated with a formal definition of *extractability*, which captures the concept of *value extractable from an SSDB according to a given type*. Given a definition of extractability, an extraction algorithm can thus be constructed and proved correct with respect to it.

In the following, we give a specification of extractability for the generic programming language, a specification of a corresponding algorithm, and a definition of correctness of the algorithm with respect to extractability.

### 4.2.1 Extractability

Extractability for a language characterises an extraction mechanism, specifying when a value can be classified as extractable according to a given type from a given SSDB. Intuitively, this is when:

1. the value has the given type;
2. there exists a regular SSDB such that:
   (a) the subset is *included* in the given SSDB;
   (b) the subset is *equivalent* to the given value.

Let $PL$ be a programming language with a typing relation $:\subseteq D \times \mathbf{T}$ that associates a type $T \in \mathbf{T}$ with the set of values $d \in D$ such that $d : T$ (see Figure 4.6). More formally, the notion of extractability for $PL$ is based on:

1. a relation of inclusion $<\subseteq S \times S$ between SSDBs (see Figure 4.3), which indirectly associates each SSDB $s$ with the set of SSDBs included into it.

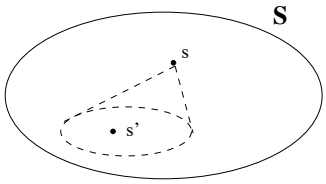



Figure 4.3: Inclusion relation

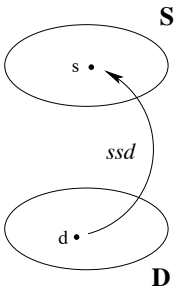



Figure 4.4: Mapping from typable values to SSDBs

2. a mapping $ssd : D \rightarrow S$, from values to SSDBs (see Figure 4.4), which justifies the equivalent expressiveness of $d$ and $s$;

Accordingly, we can provide a specification of extractability for the generic $PL$ as follows.

**Definition 4.2.1** *(Specification of extractability for PL) Let $T \in \mathbf{T}$, $s \in S$. A value $d \in D$ is* extractable *from $s$ according to $T$ if $d : T$, and there exists $s' \in S$ such that $ssd(d) = s'$ and $s' < s$.*

The notion of extractable value can be naturally extended to the notion of *set of values extractable from an SSDB according to a type.*

**Definition 4.2.2** *(Specification of extractable values) Let $s \in S$, $T \in \mathbf{T}$. The set of values extractable from $s$ according to $T$ is defined as,*

$$D_{s,T} = \{d \mid d \text{ is extractable from } s \text{ according to } T\}$$

### 4.2.2 Extraction algorithm correctness

The generic extraction algorithm,

$$\texttt{EXTR}_{PL} : S \times \mathbf{T} \rightarrow D \cup \{fail\}$$

is constructed according to a specific definition of extractability for $PL$. As illustrated in Figure 4.5, given $s$ and $T$, $\texttt{EXTR}_{PL}$ is expected to return a value extractable from $s$ according to $T$, if one exists. If $D_{s,T}$ is empty, i.e. there is no value extractable from $s$ according to $T$, the algorithm returns $fail$.

*Correctness* of $\texttt{EXTR}_{PL}$ is formally established by proving soundness and completeness with respect to extractability. Formally,

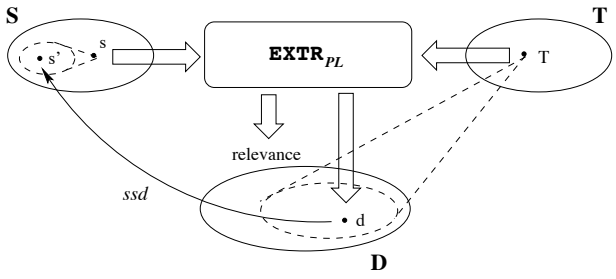



Figure 4.5: Extraction algorithm outline

**Definition 4.2.3** *(Soundness of* $\texttt{EXTR}_{PL}$*) Let* $s \in S$ *and* $T \in \mathbf{T}$*. An extraction algorithm* $\texttt{EXTR}_{PL}$ *is* sound *if, every successful execution* $\texttt{EXTR}_{PL}(s, T) = d$ *is such that:*

- $ssd(d) < s$ *(*soundness of inclusion*);*
- $d : T$ *(*soundness of typing*).*

**Definition 4.2.4** *(Completeness of* $\texttt{EXTR}_{PL}$*) Let* $s \in S$ *and* $T \in \mathbf{T}$*. An extraction algorithm* $\texttt{EXTR}_{PL}$ *is* complete *if, whenever* $D_{s,T} \neq \emptyset$*,* $\texttt{EXTR}_{PL}(s, T) = d$ *with* $d \in D_{s,T}$*.*

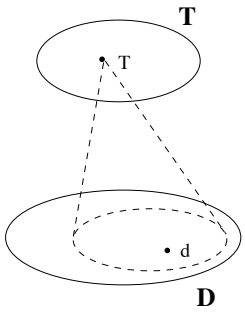



Figure 4.6: Typing relation

Completeness is here given in a very general form, so as to allow the definition of the simplest extraction algorithm. However, other definitions can be conceived,

which narrow this one by defining specific properties the extracted value should respect.

## 4.3 A new SSD query methodology

Given a correct extraction algorithm $\texttt{EXTR}_{PL}$ for a language $PL$, the query methodology consists of two phases:

- In the first phase a programmer attempts to project a given type $T$ onto a given $s$, that is $s$ and $T$ are passed as input to $\texttt{EXTR}_{PL}$. If extraction is not possible the user will be notified of the failure, otherwise an extractable value is returned.

  When successful, the extraction process also yields a quantification of *relevance*[1] of the extracted value (see Figure 4.5). Relevance should be able to provide users with information over the practical importance of the extracted value with respect to the type they have specified. By interpreting this information, users may be able to improve the input type and perform further, more useful extractions. This process may continue until users believe relevance meets proper requirements.

- The second phase is exactly that of traditional typed programming and query mechanisms: the applications and the data are known to conform to the same type, and therefore standard static properties relative to the target language environment can be assumed. Among these, application correctness is always supported, while optimisations are up to the hosting system.

Our methodology does not represent an alternative to navigation-based query approaches. On the contrary, we believe that in combination, the two of them can provide a complete support to SSD management. In other words, flexibility should be a fundamental property of SSD while regularity of typing should be recovered when advantageous.

### 4.3.1 Possible application scenarios

The main inconvenience of extraction mechanisms is the cost of traversing an SSDB in the attempt to extract a value. This operation may require the repeated visit of an entire SSDB and, according to its size, strongly degrade performance. Accordingly, our query methodology is particularly suitable in application scenarios where new extractions do not occur frequently.

In particular, new extractions are needed when new types are to be matched against the SSDB, or when types require to be matched against an up-to-date version of the data. Hence, extraction mechanisms should be employed when,

[1] Not to be confused with the notion of relevance in information retrieval.

- the SSDB involved is quite fixed in time: applications implicitly operate over a mirror of the current SSDB;
- the SSDB involved may be modified: applications do not require to operate over an up-to-date version of the data;
- applications are not subject to frequent type modifications.

For instance, in the case of user-interactive typed queries over the database, each query may require the execution of a new extraction. In this scenario, SSDQLs seems to be generally the best approach.

An example of a scenario suitable to our extraction system is that of *palaeobiological data*. These data are usually kept in very large (XML) SSDBs with a fairly regular core. Such databases would benefit from our methodology and run complex applications over the regular subsets of their highly irregular SSDBs.

### 4.3.2 Implementation notes

The extraction system may be implemented in various ways, according to the different application context and requirements. Here, we briefly discuss the main issues of materialisation of the extracted values and of distribution of the system over a network.

#### Materialisation

Given an SSDB $s$ and a type $T$, let $d$ be the value resulting from the extraction from $s$ according to $T$. In order to be accessed by applications, $d$ should be available to the run-time system of $L$.

This could be done by *materialising* $d$, i.e. transforming the corresponding subset of $s$ into an internal representation of the values of type $T$.

A further solution could be to create an index to $s$, which provides a transparent interface between the run-time environment of $L$ and the subset of $s$ corresponding to $d$. A computation accessing $d$ would directly access, through the index, the corresponding parts of $s$.

Note that, as the index would refer to the actual data in $s$, changes to $d$ may be directly reflected on $s$. Furthermore, the extraction process may be incremental, thereby handling better frequently changing SSDBs. Finally, observe that the presence of multiple indexes over the same SSDB may correspond to multiple type views over the data, and eventually result in some sort of integrated, shared and safely accessed SSDB.

#### Distribution over a network

SSDBs are becoming common on the Internet, especially in the form of XML files. We thus expect the net to be a suitable application scenario for the extraction

system. This introduces the non-trivial problem related to the location of the SSDB, the extraction system, and the consuming applications.

For example, if the extraction system is local to the consuming applications while the SSDB is remotely located, the operation of transferring and filtering the database locally could be very expensive in terms of time and bandwidth.

In Chapter 9 we present a CORBA-based implementation of the extraction mechanism, in which the system is local to the SSDB and the applications may be remotely located. Extraction requests are then sent to system, which returns handles to the resulting subsets. Applications may then prefer to materialise a copy of the data locally or to keep working with the handle on the remote extracted data via the handle.

### 4.3.3 Comparison with other typing techniques for SSD

As far as we know only the system *Ozone* [2] proposed something similar, with the purpose of integrating the data model ODMG with OEM. Ozone supports a coercion function which converts OEM collections into ODMG objects according to a type, based on intuitions similar to the ones explained in this Chapter. This solution, however, is specific to ODMG and is not justified by a complete formal treatment. Moreover, only tree-structured OEM databases are regarded, extracted according to collection and record types only.

Extraction mechanisms are generally complementary to other typical solutions to typing SSD, as we shall discuss next.

#### Static approaches

In Chapter 3, we have seen that in the presence of a large number of irregularities of the information source, traditional data models endowed with union types cannot be used. Hence, not to renounce the advantages of static typing, low-level types are introduced. These types can flexibly describe the structure of SSDBs, and statically capture static irregularities of any kind.

Low-level static types can be used to support many of the DBMSs benefits discussed in earlier chapters and yet cope with static irregularities. However, dynamic irregularities cannot be handled, as the SSDB must be populated according to a pre-defined schema.

Our system does provide the abstractions and the benefits typical of DBMSs, while keeping the full irregularity of SSDBs. Regularity is recovered when possible and gainful, while irregularities are left to SSDBMSs.

#### Dynamic approaches

As we shall see in later sections, this approach is complementary to ours and would indeed form an essential part of an integrated system.

Dynamic types do not provide query correctness, but only some form of query optimisation. On the other hand they do not limit the number of irregularities in an SSDB. These techniques may be profitably used, paired with our approach, for the purpose of extracting a subset of regular information from an SSDB. Such values may than be imported within the run-time environment of a typed programming language and gain all the benefits of static typing.

# Chapter 5

# A Semistructured Data Model

In this Chapter we provide a formal definition of SSDBs, to which we shall refer in the realisation of an extraction mechanism for the representative language $L$.

Our SSDBs are defined according to a data model that describes any problem domain in terms of two modeling primitives: *entities*, assertions about the existence of concepts or phenomena in the problem domain; and *facts*, named binary associations between entities.

Given the problem domain of my life, for example, a physical person who is a *friend of mine*, the sequence of characters *Fabio*, or the number *30* are all examples of entities. Furthermore, the first two entities may be associated by a fact *Name*, while the first and the third one may be associated by a fact *Age*.

Facts are directed associations between *source* and *target* entities: they increment the knowledge about entities beyond their simple existence by qualifying target entities as *properties* of source entities. We distinguish *unique* entities, i.e. a friend of mine, from *value* entities, or simply *values*, i.e. 30, according to whether they are of interest per-se or only as properties of other objects.

We represent problem domains modelled by these primitives as graph-structured SSDBs. In SSDBs entities and facts are represented, respectively, by *objects* and *labeled edges* between objects. In particular, we focus on SSDBs with the following properties:

*i*) there is a distinguished object called the *root* of the SSDB;

*ii*) edges are labeled and directed;

*iii*) any object in the SSDB is reachable via a path of edges from the root;

*iv*) leaves are atomic values from integers and strings.

For example, the SSDB in Figure 5.1 represents the problem domain of the authors of a paper.

In the following we formally define SSDBs, together with different forms of equivalence and inclusion relation between them, which we shall exploit in later chapters.

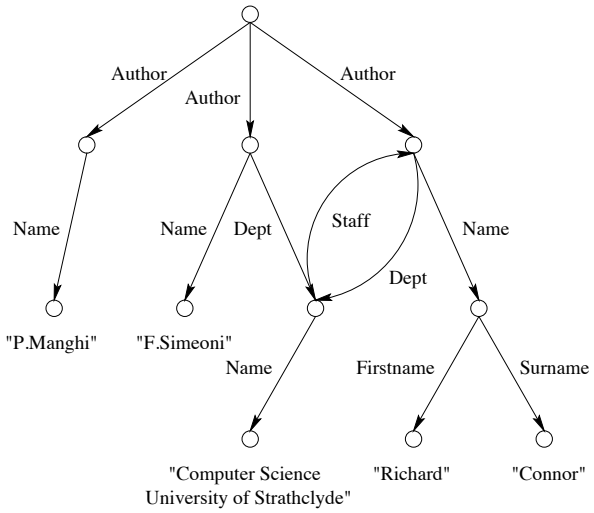



Figure 5.1: Graphical representation of a SSDB according to our data model

## 5.1 A formal definition of SSDBs

We define the domain $S$ of SSDBs as a particular subset of the domain $\Gamma$ of labeled graphs. Labeled graphs are built out of the following primitive domains: *Label, String, Integer*, and *Oid*, all abiding by standard definitions. Labels are used on edges to represent fact names, while strings and integers model atomic data in terminal objects. Object identifiers represent identity for non-terminal objects. Next, we shall use the abbreviations $Atomic = String + Integers$, and $Obj = Oid + Atomic$.

Labeled graphs are pairs $(o, E)$ where $o$ is an oid identifying the root of the graph, and $E$ is a set of *edges*, i.e. triples $< o, l, o' >$ where $o \in Oid$, $o' \in Obj$, and $l \in Label$. Formally, the set $\Gamma$ of graphs can be defined as:

$$\Gamma = Obj \times \mathcal{P}_{fin}(Oid \times Label \times Obj)$$

Note that this definition includes graphs that are not SSDBs. For example, graphs whose oids are not reachable with a path from the root. To restrict to SSDBs, we first require the following notation:

- $g \equiv (o, E) \in \Gamma$, with $g_e = E$ and $g_r = o$;
- $e \equiv < o, l, o' > \in \mathcal{P}_{fin}(Oid \times Label \times Obj)$, with $\overleftarrow{e} = o$, $\overrightarrow{e} = o'$, and $label(e) = l$.

**Definition 5.1.1** *(Objects and oids in a graph) Given a graph* $g \in \Gamma$ the set of oids in $g$ *is defined as,*

$$Oid(g) = \{o \in Oid \mid e \in g_e \ \wedge \ (\overleftarrow{e} = o \vee \ \overrightarrow{e} = o)\};$$

*similarly,* the set of objects in $g$ *is defined as,*

$$Obj(g) = Oid(g) \cup \{v \in Atomic \mid e \in g_e \wedge \ \overrightarrow{e} = v\};$$

**Definition 5.1.2** *(Operators on set of edges) Given a set of edges* $E$ *and an oid* $o \in Oid$*,* the set of edges outgoing $o$ *in* $E$ *is defined as,*

$$E(o) = \{e \in E \mid \overleftarrow{e} = o\};$$

the set of edges labeled with $l$ *in* $E$ *is defined as,*

$$E(l) = \{e \in E \mid label(e) = l\};$$

the set of source oids in $E$ *is defined as,*

$$\overleftarrow{E} = \{\overleftarrow{e} \in Oid \mid e \in E\};$$

the set of target oids in $E$ *is defined as,*

$$\overrightarrow{E} = \{\overrightarrow{e} \in Oid \mid e \in E\};$$

the set of labels of the edges in $E$ *is defined as,*

$$Label(E) = \{label(e) \mid e \in E\}.$$

**Definition 5.1.3** *(Reachable oid) Given a set of edges* $E$ *and* $o, o' \in Oid$*,* $o'$ *is* reachable *from* $o$ *in* $E$ *(*$o' \leq_E o$*) if*

1. $o' = o$*, i.e.* $o \leq_E o$*;*
2. *there exists a* path *in* $E$*, i.e. a sequence of edges* $[e_1, \ldots, e_n]$ *such that* $\overleftarrow{e}_1 = o$*,* $\overrightarrow{e}_n = o'$*, and* $\forall i : 1, \ldots, n-1.\ \overrightarrow{e}_i = \overleftarrow{e}_{i+1}$.

**Definition 5.1.4** *(SSDB)*
*The set of* SSDBs *is the set* $S \subset \Gamma$*, defined as,*

$$S = \{g \in \Gamma \mid \forall o \in Oid(g).\ o \leq_{g_e} g_r\}$$

## 5.2 Inclusion of SSDB graphs

Inclusion of SSDB can be provided according to various definitions, each depending on concepts such as identity, topology and labeling of graphs. Extant SSD approaches typically rely on well-known relations between graphs, such as isomorphism [62] and simulation [27].

The choice of a particular definition is fundamental to our extraction system, which relies on an inclusion relation to identify an extractable value. We first introduce *lt-inclusion*, which we shall adopt, followed by *simulation* and by an example of a very peculiar inclusion relation.

### 5.2.1 Inclusion by labeling and topology

Inclusion by *labeling and topology* ensures that an SSDB $s'$ is included into an SSDB $s$ if the latter has all the structural properties of the former: objects and edges in $s'$ find a one to one mapping with a subset of the object and edges of $s$.

**Definition 5.2.1** *(SSDB lt-inclusion) Given $s, s' \in S$ $s'$ is* lt-included *in $s$ ($s' < s$) if there exists a morphism $h : Oid(s') \rightarrow Oid(s)$ such that $h(s'_r) = s_r$, and $\forall e' \in s'_e$,*

1. *if $\overrightarrow{e'} \in Oid$ then $\exists e \in s_e :\ h(\overleftarrow{e'}) = \overleftarrow{e}\ \wedge h(\overrightarrow{e'}) = \overrightarrow{e}\ \wedge label(e') = label(e)$*
2. *if $\overrightarrow{e'} \in Atomic$ then $\exists e \in s_e :\ h(\overleftarrow{e'}) = \overleftarrow{e}\ \wedge\ \overrightarrow{e'} = \overrightarrow{e}\ \wedge label(e') = label(e)$*

Consequently, identity by *labeling and topology* ensures that two SSDB graphs are the same if they have the same structural properties: there exists a one-to-one correspondence between both objects and edges.

**Definition 5.2.2** *(SSDB lt-equivalence) Given $s, s' \in S$, $s'$ is* lt-equivalent *to $s$ if $s' < s$ and $s < s'$.*

In Figure 5.2 the graph $s_2$ is lt-included into the graph $s_1$, while the two graphs $s_3$ and $s_4$ are lt-equivalent.

### 5.2.2 Identity by bisimulation

A common form of inclusion is that provided by graph simulation.

**Definition 5.2.3** *(SSDB simulation) Given two SSDB graphs $s$ and $s'$, a* SSDB simulation *between them is a binary relation $\mathcal{R}_{s,s'} \subseteq Oid(s) \times Oid(s')$ such that, if $o_1 \mathcal{R}_{s,s'} o'_1$ then,*

1. *if $o_1 \in Oid$ then $\forall < o_1, l, o_2 > \in s_e$, $\exists < o'_1, l, o'_2 > \in s'_e$ s.t. $o_2 \mathcal{R}_{s,s'} o'_2$;*
2. *if $o_1 \in Atomic$ then $o_1 = o_2$.*

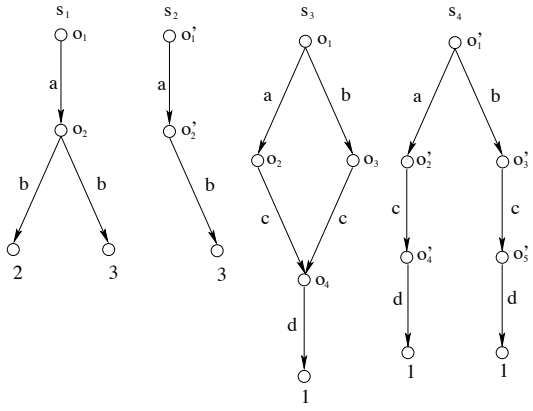



Figure 5.2: lt-included and bisimilar graphs

*The graph $s$ is* similar *to the graph $s'$ ($s <_b s'$) if there exists a graph simulation $R_{s,s'}$ such that $s_r\ R\ s'_r$.*

Identity by labeling corresponds to graph bisimulation. In Figure 5.2, we show two bisimilar graphs which are lt-included and are not lt-equal.

**Definition 5.2.4** *(SSDB b-equality) Given $s, s' \in S$, $s'$ is* bisimilar *to $s$ if $s <_b s'$ and $s' <_b s$.*

In Figure 5.2 the graphs $s_1$ and $s_2$ are bisimilar, as well as the graphs $s_3$ and $s_4$.

### 5.2.3 SSDB d-inclusion

Other, more sophisticated, forms of inclusion may be defined, giving rise to more flexible and powerful extraction mechanisms. For example, consider the *tree-structured* SSDBs in Figure 5.3.

The SSDB $s'$ is *d-included* (*depth-included*) in the SSDB $s$. Intuitively, this is true because any path in $s'$ appears in $s$ modulo some discontinuity. As suggested by the example, this form of inclusion may be particularly suitable for extracting collections of elements of the same type that are spread over the whole SSDB.

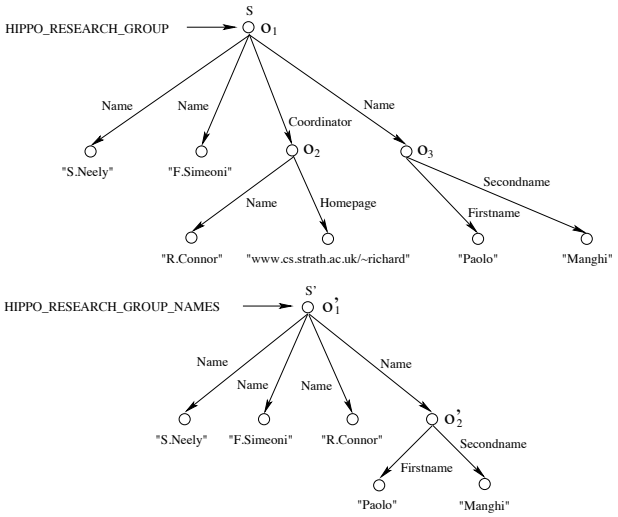



Figure 5.3: Other forms of inclusion.

# Chapter 6

# Target language

In this Chapter, we define the typed language $L$, for which we shall realise an extraction system. $L$ is characterised by a set of values and a type language based on standard constructs. As our focus is on the extraction of values of a given type from SSDBs, we are not concerned with the definition of specific value operators for $L$.

Introducing the type language, we provide a general introduction to recursive types. We then give a coinductive characterisation of type equivalence, followed by an inductive axiomatisation.

The only purpose of $L$ is providing the formal platform required for the definition of extractability for a language. Accordingly, language values are directly defined as particular labelled graphs, so as to simplify the definition of a mapping from values to SSDBs.

Finally, we formally define the typing relation and give an inductive axiomatisation for it, with a corresponding proof of soundness and completeness.

## 6.1 Type language

The type language of $L$ offers standard constructs for data description: simple base types, record, collection, and union constructors, as well as a way of introducing recursive definitions. As these are typical abstractions supported by most programming languages, we believe the results of our work can be easily adapted to specific languages and type systems. As we shall see in Chapter 9, for example, fragments of XML SSDBs may be extracted with respect to a subset of the Java type system.

Formally, we consider the subset $\mathbf{T}$ of the *well-defined* and *canonical* expressions generated by the BNF grammar $G$:

$$T \ ::= \ \mathit{int} \mid \mathit{string} \mid X \mid T_1 + T_2 \mid \mu X.T \mid [l_1 : T_1, \ldots, l_n : T_n] \mid \mathit{coll}(T)$$

where $\forall i : 1, \ldots, n.\ l_i \in \mathit{Label}$. In the union expression $T_1 + T_2$, the $T_i$'s are called *members*. The expressions $\mu X.T$, $X$ is a *recursive variable* and $T$ is called *body*.

Finally, the pairs $l_i : T_i$ of a record expression $[l_1 : T_1, \ldots, l_n : T_n]$ are called *record fields*, where $l_i$ is a *record field label* and $T_i$ is a *record field expression*.

The set of well-defined expressions generated by $G$ is defined as follows:

**Definition 6.1.1** *(Well-defined expressions)*

$$\frac{}{V \vdash \mathit{int}} \quad (\mathit{integer})$$

$$\frac{}{V \vdash \mathit{string}} \quad (\mathit{string})$$

$$\frac{((T_i \not\equiv_{\mathrm{T}} \mathit{coll}(T)) \wedge (V \vdash T_i)) \vee ((T_i \equiv_{\mathrm{T}} \mathit{coll}(T)) \wedge (V \vdash T))}{V \vdash [l_1 : T_1, \ldots, l_n : T_n]} \quad (\mathit{record})$$

*where* $i, j : 1, \ldots, n.\ j \neq i \Rightarrow l_i \neq l_j$

$$\frac{V \vdash T_1 \quad V \vdash T_2}{V \vdash T_1 + T_2} \quad (\mathit{union})$$

$$\frac{V,\ X \vdash T}{V \vdash \mu X.\ T} \quad (\mathit{rec})$$

$$\frac{X \in V}{V \vdash X} \quad (\mathit{var})$$

The judgement $\vdash T$ states that $T$ is well-defined. Proofs of well-definition of expressions are inductive-algorithmic. Indeed, rule (*rec*) gathers in $V$ the recursive variables introduced by $\mu$-expressions, and rule (*var*) validates the well-definition of the corresponding bodies at subsequent stages.

The rules are fairly standard, except for (*record*). The rule canonically requires the labels of record fields to be different, but restricts to well-defined expressions where collections can appear only as record field expressions. For example, expressions such as $[a : \mathit{coll}(T_1) + T_2]$ or $\mathit{coll}(T)$ are not well-defined.[1] We shall refer to

[1] As we better explain in later Sections, this choice is due to the fact that we shall conveniently represent the values of $L$ as particular labelled graphs, so as to ease the formalisation of the extraction process. In particular, collection values are represented as oids with a set of equally labelled outgoing edges. As a collection type should specify such label, we have chosen to refer to the record field label.

record fields as *non-collection fields* or *collection* fields, depending on the nature of the record fields expression.

The set of contractive expressions is defined as follows.

**Definition 6.1.2** *(Canonical expressions) An expression of $G$ is* canonical *if its recursive variables appear as expressions of non-collection field of a record $(l : X)$ or as expressions of collections $(l : coll(X))$.*

In later Sections we shall see that the expressions discarded by this definition do not describe interesting language values and may introduce problems in the formalisation of type equality and typing.

Finally, we can define the set of types of $L$ as,

**Definition 6.1.3** *(Type language $\mathbf{T}$) The type language $\mathbf{T}$ of $L$ comprises all well-defined and canonical expressions defined by the grammar $G$.*

### 6.1.1 Recursive types

Types are introduced in programming languages to characterise sets of values. In particular, *recursive types* are meant to characterise values whose structure consists of the arbitrarily large repetition of the same fixed pattern. As such, recursive types extend the capabilities of a type system to the abstraction over values with variable structure.

In $\mathbf{T}$ recursive types are syntactically represented by $\mu$-types. In this section, we show the rationale behind their introduction in type theory.

Let us consider the type system $\mathbf{T}^*$ of all the types in $\mathbf{T}$ except for $\mu$-types and variables. Recursive types are motivated by type equations in $\mathbf{T}^*$ of the form:

$$X = T(X), \tag{6.1.1}$$

where $X$ is a meta-variable ranging over $\mathbf{T}^*$ and $=$ is the symbol of provable equality. A solution of (6.1.1) is a type expression $\overline{T} \in \mathbf{T}^*$ that substituted to all occurrences of $X$, transforms the two members of the equation into equal types, i.e. $\overline{T} = T(\overline{T})$. Note that $\overline{T}$, called recursive type, characterises values whose structure consists of the finite repetition of the same fixed pattern $T(X)$.

To introduce such types in $\mathbf{T}^*$ a finite syntactic notation to represent the infinite solution $\overline{T} = T(T(T(T(T(...)))$ is required. The solution can be univocally characterised in terms of its structural pattern $T(X)$. Hence, the type system $\mathbf{T}^*$ is extended with recursive variables and new type expressions, $\mu$-types, of the form,

$$\mu X.T(X)$$

$\mu X.T$ is also called the *canonical solution* of (6.1.1). Intuitively, the recursive variable $X$ marks the point from which the structural pattern determined by the body can be infinitely substituted to generate the infinite solution of (6.1.1).

Note that, for this notation to be unambiguous, each recursive equation in $\mathbf{T}^*$ must have a unique solution. This is generally true, as equations of the form (6.1.1) seem to admit only the solution obtained by infinitely expanding $T(X)$. However, when $T(X)$ is $X$, all types in $\mathbf{T}^*$ could be solutions of the equation $X = X$. For this reason, we restricted $\mathbf{T}$ to canonical type expressions. This excludes types of the form $\mu X.X$ and $\mu X.X + X$, which may compromise the consistency of our type system.

## 6.2 Type equivalence

The type language $\mathbf{T}$ is similar to that studied by Amadio and Cardelli in [14] and Brandt in [22]. In these works, the focus is on the definition of type equality and subtyping, which become both intuitively and formally complex in the presence of recursive types. Here we adapt these results to formally characterise and axiomatise type equality in $\mathbf{T}$, referring to the literature for formal proofs where these are required.

In $\mathbf{T}^*$ two types $T, T' \in \mathbf{T}^*$ are equal ($T \equiv T'$) if they are syntactically equivalent, up to the ordering of the record fields and the union members. An immediate consequence of introducing recursive types is that syntactical identity is no longer sufficient to capture type equality.

In the following, we present two formalisation of type equality in $\mathbf{T}$. The first, *weak type equality*, is an inductive axiomatisation which captures only partially type equality. Inductive rules provide both a neat definition and a proof algorithm for type equality. The second, *strong type equality*, is more expressive but less intuitive. Therefore, we relate the corresponding inductive axiomatisation to a mathematical characterisation.

### 6.2.1 Weak type equality

The relation of *weak type equality* is inductively defined by a set of rules of the form,

$$\frac{x_1, \ldots, x_n}{x} \qquad n \geq 0$$

where $x, x_1, \ldots x_n$ are pairs of equivalent types $(T_1, T_2)$. Such rules state that the pair $x$ is in the relation provided that the pairs $x_1, \ldots, x_n$ are in the relation.

Specifically, $=_{\mathbf{T},w}$ is the smallest subset in $\mathbf{T} \times \mathbf{T}$ that is closed with respect to the rules in Figure 6.2.1, where we implicitly consider record types and union types equivalent up to the reordering of fields and members, respectively.

$$\frac{}{T =_{\mathbf{T},w} T} \quad (REF - EQ)$$

$$\frac{T' =_{\mathbf{T},w} T}{T =_{\mathbf{T},w} T'} \quad (SYMM - EQ)$$

$$\frac{T_1 =_{\mathbf{T},w} T_2 \quad T_2 =_{\mathbf{T},w} T_3}{T_1 =_{\mathbf{T},w} T_3} \quad (TRANS - EQ)$$

$$\frac{T_1 =_{\mathbf{T},w} T_1' \quad , \ldots , T_n =_{\mathbf{T},w} T_n'}{[l_1 : T_1, \ldots , l_n : T_n] =_{\mathbf{T},w} [l_1 : T_1', \ldots , l_n : T_n']} \quad (RECORD - EQ)$$

$$\frac{T =_{\mathbf{T},w} T'}{coll(T) =_{\mathbf{T},w} coll(T')} \quad (COLL - EQ)$$

$$\frac{T_1 =_{\mathbf{T},w} T_1' \quad T_2 =_{\mathbf{T},w} T_2'}{T_1 + T_2 =_{\mathbf{T},w} T_1' + T_2'} \quad (UNION - EQ)$$

$$\mu X.T =_{\mathbf{T},w} T \left[ {}^{\mu X.T}/_{X} \right] \quad (UNFOLD - EQ)$$

Figure 6.1: Weak type equivalence rules

Rule $(UNFOLD - EQ)$ is justified by the following observation. Since $\mu X.T$ represents the only solution of the equation $X = T(X)$, the equivalence $\mu X.T = T(\mu X.T)$ must hold as both types are trivially a solution for the equation. Syntactically, this equivalence is captured by the rule $(UNFOLD - EQ)$, where the *substitution operation* $T \left[ {}^{\overline{T}}/_{X} \right]$ replaces any occurrence of $X$ in $T$ with the type $\overline{T}$. The right member of $(UNFOLD)$ is called *one-step unfolding* of $\mu X.T$, denoted as $unfold_1(\mu X.T)$. It is obtained by substituting each occurrence of $X$ in the body $T$ with $\mu X.T$. By transitivity, we infer that $\mu X.T$ is also equivalent to its two-step unfolding $unfold_2 = T \left[ {}^{unfold_1(\mu X.T)}/_{X} \right]$) as well as to all its $n$-step unfoldings.

For example, the equation $X = [a : X]$ is associated to its canonical solution

$\mu X.[a : X]$. Therefore, the equivalence,

$$\mu X.[a : X] =_{\mathbf{T},w} [a : \mu X.[a : X]]$$

should hold, as well as the equivalence,

$$\mu X.[a : X] =_{\mathbf{T},w} [a : [a : \mu X.[a : X]]].$$

### 6.2.2 Strong Type Equality

Weak type equality is not expressive enough to capture type equivalence in **T**. To understand this, consider the equations $X = T(X)$ and $X = T(T(X))$. Obviously, the solution of the first is also a solution of the second. Their canonical solutions, $\mu X.T(X)$ and $\mu X.T\left[{}^{T(X)}/_{X}\right]$, respectively, must thus be equivalent in **T**. With weak type equality, intuitively, these types can be proven equivalent applying the rules an infinite number of times, as their unfolding expand to the same infinite type. Consequently, according to the inductive interpretation of the rules, the pair $(\mu X.T(X), \mu X.T\left[{}^{T(X)}/_{X}\right])$ cannot be included in the relation. Such recursive types are known in type theory as *non-synchronised* recursive types, for their equivalence cannot be proven with rule $(UNFOLD - EQ)$.

The literature offers two different formalisations of *strong type equality*, one by Amadio and Cardelli [14] based on semantic grounds and one by Brandt [22, 23] based on syntactical grounds. The first defines equivalence of types in terms of equality of corresponding mathematical trees. The second defines type equivalence as a syntactic property of types. These definitions offer support and justification for two different axiomatisations with which equivalence of two types can be formally proven in a finite number of steps. The two systems have been proven equivalent by Brandt in [22, 23]. In the following, we present the more intuitive formalisation suggested by Brandt and rely on it to describe type equivalence in **T**.

#### Definition of type equality for T

Brandt's intuition is that two type expressions are equivalent if and only if their structural comparison, possibly obtained by unfolding the $\mu$-types if any, does not show any syntactic diversity. If such analysis can be indefinitely protracted with no evidence of contradiction, the two type expressions must be equivalent.

Formally, this structural comparison can be captured by a simulation relation, according to which two expressions are equivalent if there exist a *type bisimulation* between them.

**Definition 6.2.1** *(Type bisimulation)*
*A bisimulation on recursive types is a binary relation $\mathcal{R}$ on* **T** *satisfying:*

1. $(\mu X.T)\ \mathcal{R}\ T' \Rightarrow (T\left[{}^{\mu X.T}/_{X}\right])\ \mathcal{R}\ T'$
2. $T\ \mathcal{R}\ (\mu X.T') \Rightarrow T\ \mathcal{R}\ (T'\left[{}^{\mu X.T'}/_{X}\right])$
3. $[l_1 : T_1, \ldots, l_n : T_n]\ \mathcal{R}\ [l_1 : T'_1, \ldots, l_n : T'_n] \Rightarrow \forall i : 1, \ldots n :\ T_i\ \mathcal{R}\ T'_i$
4. $(T_1 + T_2)\ \mathcal{R}\ (T'_1 + T'_2) \Rightarrow T_1\ \mathcal{R}\ T'_1 \wedge T_2\ \mathcal{R}\ T'_2$
5. $coll(T)\ \mathcal{R}\ coll(T') \Rightarrow T\ \mathcal{R}\ T'$
6. $string\ \mathcal{R}\ string$
7. $int\ \mathcal{R}\ int$

**Definition 6.2.2** *(Type equivalence) Let* $T, \overline{T} \in \mathbf{T}$. $T =_{\mathbf{T}} \overline{T}$ *if and only if there exists a bisimulation* $\mathcal{R}$ *such that* $T\ \mathcal{R}\ \overline{T}$.

**Definition 6.2.3** *(Type equivalence relation) Type equivalence relation* $=_{\mathbf{T}}$*, i.e. the set of all pairs of equivalent types in* $\mathbf{T}$*, is the largest type bisimulation in* $\mathbf{T}$.[2]

Brandt proves that this definition is equivalent to that given by Amadio and Cardelli, which includes also the pairs of non-synchronised type expressions. Furthermore, he gives a corresponding inductive axiomatisation of $=_{\mathbf{T}}$, in order to prove the computability of the proof of equivalence of two type expressions.

### Inductive axiomatisation for $=_{\mathbf{T}}$

Definitions as 6.2.2 above are known as *coinductive* definitions, as they are the dual of *inductive definitions*. To this regard, note that the implications $x_1, \ldots, x_n \Rightarrow x$ in the bisimulation definition are equivalent to a set of rules of the form,

$$\frac{x_1, \ldots, x_n}{x}$$

The coinductive interpretation of these rules defines the relation $=_{\mathbf{T}}$ that includes all the pairs of type expressions $T$ and $\overline{T}$ such that there exists a type rule of the form,

$$\frac{x_1, \ldots, x_n}{T =_{\mathbf{T}} \overline{T}}$$

and the the pairs in the premises $x_1, \ldots, x_n$ are again in $=_{\mathbf{T}}$.[3] The resulting relation is equivalent to the largest bisimulation $=_{\mathbf{T}}$ defined in 6.2.3 Indeed, two

[2]Note that the largest bisimulation is implicitly transitive, reflexive and symmetrical.

[3]Note that the same set of rules, if augmented with rules for reflexivity, symmetry and transitivity, reduce to the rules that define weak type equivalence. For an interesting discussion on inductive and coinductive definitions, well-founded and non-well founded sets, and relation between them, refer to [20, 19, 5, 6]

expressions may be equivalent because of an infinite application of type rules justifies so. In particular, the rules,

$$\frac{T\left[\mu X.T/X\right] =_{\mathbf{T}} T'}{\mu X.T =_{\mathbf{T}} T'} \qquad (REC-L-EQ)$$

$$\frac{T' =_{\mathbf{T}} T\left[\mu X.T/X\right]}{T' =_{\mathbf{T}} \mu X.T} \qquad (REC-R-EQ)$$

justify equivalence of non-synchronised types, thereby capturing *strong type equivalence*.

Given two non-synchronised type expressions their equivalence cannot be proven with a finite number of applications of these rules [6]. Brandt noticed, however, that coinductive proofs for this specific set of rules either terminate in a finite number of steps or else indefinitely and circularly repeat the same finite number of steps. Finite proofs could be obtained by enriching the judgements in the rules with sets of *assumptions*, that keep track of the pairs of types which have been already visited in the proof. Specifically, rules will be based on judgements of the form $A \vdash T =_{\mathbf{T}} T'$, that state that $T =_{\mathbf{T}} T'$ under the assumptions $A$.

**Definition 6.2.4** *(Type equivalence rules)*

$$A \vdash T =_{\mathbf{T}} T \qquad (REF-EQ)$$

$$\frac{A \vdash T' =_{\mathbf{T}} T}{A \vdash T =_{\mathbf{T}} T'} \qquad (SYM-EQ)$$

$$\frac{A \vdash T =_{\mathbf{T}} T' \quad A \vdash T' = T''}{A \vdash T =_{\mathbf{T}} T''} \qquad (TRANS-EQ)$$

$$A \vdash \mathit{int} =_{\mathbf{T}} \mathit{int} \qquad (INT-EQ)$$

$$A \vdash \mathit{string} =_{\mathbf{T}} \mathit{string} \qquad (STRING-EQ)$$

$$A, \mu X.\ T =_{\mathbf{T}} T' \vdash \mu X.\ T =_{\mathbf{T}} T' \qquad (HYP)$$

$$\frac{A, \mu X.\ T =_{\mathbf{T}} T' \vdash T\left[{}^{\mu X.\ T}/_{X}\right] =_{\mathbf{T}} T'}{A \vdash \mu X.\ T =_{\mathbf{T}} T'}\ \mu X.\ T = T' \notin A \qquad (REC - EQ)$$

$$\frac{A \vdash T_1 =_{\mathbf{T}} T_1' \quad , \ldots , A \vdash T_n =_{\mathbf{T}} T_n'}{A \vdash [l_1 : T_1, \ldots , l_n : T_n] =_{\mathbf{T}} [l_1 : T_1', \ldots , l_n : T_n']} \qquad (RECORD - EQ)$$

$$\frac{A \vdash T =_{\mathbf{T}} T'}{A \vdash coll(T) =_{\mathbf{T}} coll(T')} \qquad (COLL - EQ)$$

$$\frac{A \vdash T_1 =_{\mathbf{T}} T_1' \quad A \vdash T_2 =_{\mathbf{T}} T_2'}{A \vdash T_1 + T_2 =_{\mathbf{T}} T_1' + T_2'} \qquad (UNION - EQ)$$

Recursive types are the only types that entail circular proofs. Accordingly, assumption sets are enriched only by rule $(REC)$, thereby keeping track of all possible returning points for the proofs, while rule $(HYP)$ extracts assumptions from assumption sets to terminate circular proofs. For example consider the following proof of equivalence between the non-synchronised type expressions $T \equiv \mu X.[a : X]$ and $T' \equiv \mu X.[a : [a : X]]$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{TRUE}{(T =_{\mathbf{T}} T'),\ ([a : T] =_{\mathbf{T}} T'),\ (T =_{\mathbf{T}} [a : T']) \vdash T =_{\mathbf{T}} T'}\ (HYP)}{(T =_{\mathbf{T}} T'),\ ([a : T] =_{\mathbf{T}} T'),\ (T =_{\mathbf{T}} [a : T']) \vdash [a : T] =_{\mathbf{T}} [a : T']}\ (RECORD - EQ)}{(T =_{\mathbf{T}} T'),\ ([a : T] =_{\mathbf{T}} T') \vdash T =_{\mathbf{T}} [a : T']}\ (REC - EQ)}{(T =_{\mathbf{T}} T'),\ ([a : T] =_{\mathbf{T}} T') \vdash [a : T] =_{\mathbf{T}} [a : [a : T']]}\ (RECORD - EQ)}{(T =_{\mathbf{T}} T') \vdash [a : T] =_{\mathbf{T}} T'}\ (REC - EQ)}{\emptyset \vdash T =_{\mathbf{T}} T'}\ (REC - EQ)$$

These rules are not to be associated with either inductive or coinductive definitions of type equivalence. In fact, they inductively define a set of triples of the form $(A, T, T')$. Their purpose is capturing the circular, hence finite, nature of proofs of equivalence induced by the coinductive definition of type equivalence.

Indeed, Brandt showed that, whenever $\emptyset \vdash T = T'$, there exists a type bisimulation $\mathcal{R}$ such that includes $T\ \mathcal{R}\ T'$ and vice versa. Furthermore, Brandt showed that $\emptyset \vdash T = T'$ if and only if $\vdash_{AC} T = T'$, where $\vdash_{AC}$ is a judgment in the type rules of Amadio and Cardelli. As the authors proved soundness and completeness of their rules with respect to a terminating algorithm to check type equivalence, the same

holds for Brandt system. This result is very important to us, as the algorithm we propose in Chapter 7 requires a type equivalence check.

## 6.3 Value language

We represent the set of values of $L$ as a particular form of labelled graphs, so as to simplify the description of the extraction process as well as its formalisation. Specifically, values differ from SSDBs for a *mark*, i.e. a further label, assigned to each edge, and for particular leaf values, i.e. $\emptyset_T$, denoting empty collections of type $T$.

Marks will be used to distinguish between record field values and elements of a collection in the specification of the typing relation. Consider, for example, the values in Fig. 6.2.

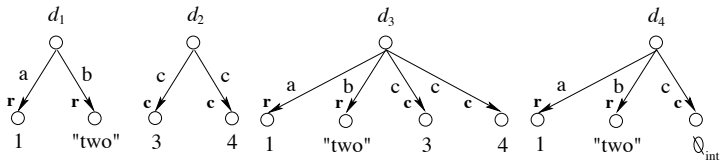



Figure 6.2: Examples of values

According to our interpretation, the values $d_1$, $d_2$, $d_3$, and $d_4$ stand for the more traditional syntactical representations $[a = 1, b ='' two'']$, $[c = \{3, 4\}]$, $[a = 1, b ='' two'', c = \{3, 4\}]$, and $[a = 1, b ='' two'', c = \{\}_{int}]$, respectively.

Observe that we introduced empty collection values because we denote collections as sets of equally labelled edges emanating from the same oid and marked with $c$. The type has been introduced to disambiguate typing, in the case an empty collection is shared by other values.

Furthermore, we represent values of $L$ as graphs with shared nodes and cycles. These mirror well-known examples of shared values in programming languages, such as *objects*, in oo-programming languages, or *explicit locations*, e.g. pointers, in imperative languages.

Overall, this representation has the advantage to be quite representative for any programming language and to be easily mapped onto SSDBs. The extraction process will transform an SSDB into a value of $L$ by simply adding the appropriate marks and the possible empty collection values.

Formally, the set of language values can be defined as a restriction over the set of *marked graphs*.

**Definition 6.3.1** *(Marked graphs) The set of* marked graphs *is defined as,*

$$\Gamma_m = Oid \times \mathcal{P}_{fin}(Oid \times Label \times M \times Obj^+)$$

*where* $M = \{r, c\}$*,* $m$ *is a meta-variable ranging over* $M$*, and,* $Obj^+ = Obj + EC$*, where*

$$EC = \{\emptyset_T \mid T \in \mathbf{T}\}.$$

$\emptyset_T$ *denotes an* empty collection *value of type* $T$*.*

As for SSDBS we introduce the following notation:

- $d \equiv (o, ME) \in D$, with $d_e = ME$ and $d_r = o$;
- $me \equiv < o, l, m, o' > \in \mathcal{P}_{fin}(Oid \times Label \times M \times Obj^+)$, with $\overleftarrow{me} = o$, $\overrightarrow{me} = o'$, $label(e) = l$, and $mark(me) = m$.

All the definitions introduced in Chapter 5 for SSDBs can be lifted to marked graphs.

**Definition 6.3.2** *(Operators on set of marked edges) Given a set of marked edges* $ME$ *and an oid* $o \in Oid$*,* the set of marked edges outgoing $o$ *in* $ME$ *is defined as,*

$$ME(o) = \{me \in ME \mid \overleftarrow{me} = o\};$$

the set of marked edges labelled with $l$ *in* $ME$ *is defined as,*

$$ME(l) = \{me \in ME \mid label(me) = l\};$$

the set of source oids in $ME$ *is defined as,*

$$\overleftarrow{ME} = \{\overleftarrow{me} \in Oid \mid me \in ME\};$$

the set of target oids in $ME$ *is defined as,*

$$\overrightarrow{ME} = \{\overrightarrow{me} \in Oid \mid me \in ME\};$$

the set of labels of the marked edges in $ME$ *is defined as,*

$$Label(ME) = \{label(me) \mid me \in ME\};$$

the set of edges marked as $r$ *in* $ME$ *is defined as,*

$$ME_r = \{me \in ME \mid mark(me) = r\}$$

the set of edges marked as $c$ *in* $ME$ *is defined as,*

$$ME_c = \{me \in ME \mid mark(me) = c\}$$

Note that in the following we shall denote the set of edges in $ME$ marked as $c$, outgoing the oid $o$, and with label $l$ as $ME(o,l)_c$.

Finally, the set of values of $L$ is the defined as follows:

**Definition 6.3.3** *(Values of L) The set of* values of $L$ *is the set* $D \subset \Gamma_m$*, defined as,*

$$
\begin{aligned}
D = \{(\overline{o}, ME) \in \Gamma_m \mid \forall o \in \overleftarrow{ME}\ .\ & \\
& (i)\ o \leq_{ME} \overline{o} \\
& (ii)\ \mid ME(o) \mid > 0 \\
& (iii)\ \forall me \in ME(o). \\
& \qquad mark(me) = r\ \Rightarrow \\
& \qquad\quad (\mid ME(o, label(me))_r \mid\ = 1\ ) \wedge \\
& \qquad\quad (\mid ME(o, label(me))_c \mid\ = 0) \\
& \qquad \overrightarrow{me} = \emptyset_T\ \Rightarrow \\
& \qquad\quad (mark(me) = c) \wedge \\
& \qquad\quad (\mid ME(o, label(me))_c \mid\ = 1)\}
\end{aligned}
$$

Clearly, marked graphs are similar to SSDBs, in that they are rooted graphs in which oids are all reachable from the root. In particular, we expect our record values not to have repeated labels and to be *unambiguously typed*. Accordingly, $D$ does not include marked graphs in which oids have:

- no outgoing edges;
- two or more outgoing edges with the same label and marked with $r$;
- a set of outgoing edges with the same label and different marks.

Furthermore, as the values $\emptyset_T$ denote empty collections, we consider only marked graphs in which these values are reached by edges $< o, l, c, \emptyset_T >$ and cannot have siblings reached by edges labelled as $l$.

## 6.4 Mapping from values of $L$ onto SSDBs

Language values can be easily mapped onto SSDBs by removing marks and the edges corresponding to empty collections. Formally,

**Definition 6.4.1** *(Mapping from D to S) Let* $erase : (Oid \times Label \times M \times Obj) \rightarrow (Oid \times Label \times Obj)$ *be the mapping,*

$$
erase(me) = \begin{cases} <\overleftarrow{me}, label(me), \overrightarrow{me}> & \overrightarrow{me} \neq \emptyset_T \\ \emptyset & \overrightarrow{me} = \emptyset_T \end{cases}
$$

*Let* $Erase : \mathcal{P}_{fin}(Oid \times Label \times M \times Obj) \to \mathcal{P}_{fin}(Oid \times Label \times Obj)$ *be the mapping*

$$Erase(ME) = \{erase(me) \mid me \in ME\}.$$

*Finally, let ssd be the mapping* $ssd : D \to S$ *such that:*

$$ssd((o, ME)) = (o, Erase(ME)).$$

Note that *ssd* does not vary the topology of its argument, as the only edges to be removed, those corresponding to empty collections, are terminal. Therefore, reachability is not compromised and its application on a value $d$ is such that $ssd(d) \in S$.

## 6.5 Definition of typing

A value $d$ *has type* $T$, or else is a *value* of $T$, if $d$ respects the structural requirements identified by $T$. This equates to say that $d$ has type $T$ if its root $d_r$ features the structural properties of $T$. Hence, typing depends on a *conformity* relation between a type and the edges emanating from the oids of a value. Specifically, conformity can be formally defined as the following mathematical relation.

**Definition 6.5.1** *(Conformity relation) Let* $T \in \mathbf{T}$ *and* $ME \subseteq \mathcal{P}_{fin}(Oid \times Label \times M \times Obj^+)$. $\mathcal{R}_{ME} \subseteq Obj^+ \times \mathbf{T}$ *is an* ME-conformity relation *if:*

1. $o \; \mathcal{R}_{ME} \; \mu X.T \;\Rightarrow\; o \; \mathcal{R}_{ME} \; T\left[{}^{\mu X.T}/_{X}\right]$;
2. $o \; \mathcal{R}_{ME} \; T_1 + T_2 \;\Rightarrow\; (o \; \mathcal{R}_{ME} \; T_1) \vee (o \; \mathcal{R}_{ME} \; T_2)$
3. $o \; \mathcal{R}_{ME} \; int \;\Rightarrow\; o \in Integer$
4. $o \; \mathcal{R}_{ME} \; string \;\Rightarrow\; o \in String$
5. $\emptyset_{T'} \; \mathcal{R}_{ME} \; T \;\Rightarrow\; T =_{\mathbf{T}} T'$
6. $o \; \mathcal{R}_{ME} \; [l_1 : T_1, \ldots l_n : T_n] \;\Rightarrow$
   (a) $Label(ME(o)) = \{l_1, \ldots, l_n\}$;
   (b) $\forall i : 1, \ldots, n. \forall me \in ME(o, l_i)$
      i. $T_i \not\equiv_{\mathbf{T}} coll(U) \;\Rightarrow\; (mark(me) = r \wedge \vec{me} \; \mathcal{R}_{ME} \; T_i)$
      ii. $T_i \equiv_{\mathbf{T}} coll(U) \Rightarrow \; (mark(me) = c \wedge \vec{me} \; \mathcal{R}_{ME} \; U)$.

First of all, note that conformity associates each oid $o$ only with record types that have the same labelling. We shall see that this assumption notably simplifies the formalisation of the extraction process, but excludes any sort of polymorphism, such as record type subtyping. The extraction algorithm, however, could be adapted to a extract according to a typing that supports subtyping.

Secondly, recursive types characterise sets of values whose structure equates the repetition of a certain structural pattern $T(X)$ for an arbitrary, finite number of times. In other words, the type expression $\mu X.T$ describes values whose structure conforms to $T$, where every occurrence of $X$ in $T$ stays for $\mu X.T$ again.

**Definition 6.5.2** *(ME-conformity of objects) Let $o \in Obj^+$, $T \in \mathbf{T}$, and $ME \subseteq \mathcal{P}_{fin}(Oid \times Label \times M \times Obj^+)$). $o$* ME-conforms *to $T$ ($o \rhd_{ME} T$) if there exists a conformity relation $\mathcal{R}_{ME}$ such that $o\mathcal{R}_{ME}T$.*

Finally, we provide the definition of typing in terms of the previous definition.

**Definition 6.5.3** *(Typing) Let $d \in D$, $T \in \mathbf{T}$.* $d$ has type $T$ *($d \rhd T$) if and only if $d_r \rhd_{d_f} T$.*

For example, the value $d$ in Fig. 6.3 is of type $T \equiv_{\mathbf{T}} \mu X.[a : x]$, i.e. its root $o$ conforms to $T$. Indeed, according to the recursive type interpretation given above, for $o$ to conform to $T$, $o$ should conform to the type $[a : T]$. As we consider a strict interpretation of record types, $o$ should have only one outgoing edge, labelled as $a$ and marked as $r$. Since this is true, we have to make sure that the target node $o'$ of such edge is of type $T$, and so on. This reasoning suggests a relation $\mathcal{R}_{d_e} = \{(o, \mu X.[a : X]), (o, [a : \mu X.[a : X]]), (o', \mu X.[a : X]), (o', [a : \mu X.[a : X]])\}$, which proves conformity of $o$ to $T$.

### 6.5.1 Observation about typing

Observing the conformity relation above, while rules 1 to 4 capture the intuition directly, rule 5 requires a deeper explanation, which we shall give in the following.

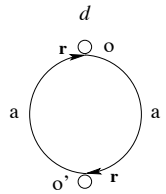



Figure 6.3: Value $d$

Furthermore, we shall also show some interesting differences between the set of values for $L$ and SSDBs, thus justifying the introduction of marked graphs to describe values. Indeed, relying directly on $S$ would have notably simplified our work but would have led to an ambiguous typing relation.

**Collection types**

Traditionally, collection types $coll(U)$ are treated independently from record types $[l_1 : T_1, \ldots, l_n : T_n]$. Intuitively, in our settings $coll(U)$ should abstract over oids with a set of outgoing edges which target oids conforming to the type $U$. $[l_1 : T_1, \ldots, l_n : T_n]$ should abstract over oids with $n$ outgoing edges, exactly one for each field $l_i$ and labelled with $l_i$, and targeting an oid $o'$ conforming to $T_i$.

However, note that this interpretation of collection types is far too loose in our context, where, due to extraction purposes, we expect also collection types to be associated to a specific labelling of the edges.

One possible solution is that of introducing *explicitly labelled collection types*, i.e. types of the form $coll_{label}(U)$. For example, the type $coll_{child}(string)$ would abstract over oids with a set of outgoing edges labelled as *child* which target atomic values conforming to the type *string*.

The problem with this solution is that it is based on an interpretation of the data typical of programming languages. Here, records and collections values are usually separate syntactical entities. Instead, SSDBs are populated with no concern about separating oids intuitively representing collections values from those that intuitively represent records values. For example, consider the two possible instances $s'$ and $s''$, in Fig. 6.4, of SSDBs representing the family of *Adam* and *Eve*.

Assume developers are interested in running applications written in $L$ over the language value $d$ representing the man and the children of the family represented in $s'$. In this case there is no type that abstracts over this subset of $s'$: record types are useless because there are two labels *child* outgoing $o$; collection types are useless because the edge labelled as *man* compromises the interpretation of $o$ as a collection value.

Note that the same extraction applied over $s''$ would succeed. $s''$ could be typed as $[man : string,\ children : coll_{child}(string)]$. Unfortunately, while SSDBs of the form of $s''$ are possible, but not frequent, SSDBs of the form of $s'$ are natural in a semistructured data model. This inconvenience could be avoided by refining the semantics of explicitly labelled collection types. Collection types $coll_l(U)$ appearing within a record field $l'$ have two possible interpretations: if $l = l'$, the structural requirements of $coll_l(U)$ should be applied to the oid that is being associated to the record type. This way, the record $[man : string,\ child : coll_{child}(string)]$ could type $s'$.

Facing this cumbersome semantics of collection types, which relies on label matching within record types, we preferred to introduce the less intuitive but uni-

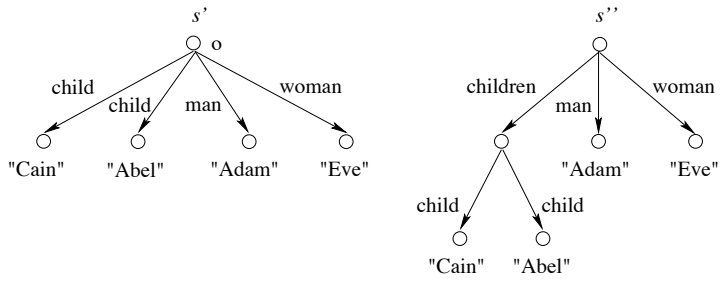



Figure 6.4: SSDBs of Adam and Eve's family

form definition presented by rule 5 in the definition of conformity. Collections types are still labelled, but their label is that specified by the record field within which they appear. This has the drawback of forcing the usage of collection types within record types. SSDBs such as $s = (o, \{< o, child, "Cain" >, < o, child, "Abel" >)$, which clearly represents a collection value of type *string*, can only be extracted and typed as values of type $[child : coll(string)]$. On the other hand, both $s'$ and $s''$ in Fig. 6.4 can be successfully extracted with respect to the types $[man : string,\ child : coll(string)]$ and $[man : string,\ children : [child : coll(string)]]$.

### Marked edges and empty collections

Note that so far, while discussing extraction and typing in relation with collections types, we did not mention marks and empty collection values. We have directly dealt with SSDBs as if they could represent the values of $L$. This would considerably simplify the formalisation of an extraction mechanism for $L$, where extraction would simply consist of the identification of a regular subset of the original SSDB. Here, we show that, due to lack of self-description of the edges and to the presence of shared oids, SSDBs cannot represent the values of $L$.

Consider the definition of conformity as defined between types and SSDBs rather than between types and values. As for the values of $L$, typing could only be checked according to the self-description provided by labelled graphs, i.e. the labels of edges emanating from an oid. Therefore, the SSDB $\overline{s}$ in Fig. 6.5 could be typed as:

$$
\begin{array}{l}
[ \\
\quad man : [children : [child : coll(string)]], \\
\quad woman : [children : [child : string]] \\
].
\end{array}
$$

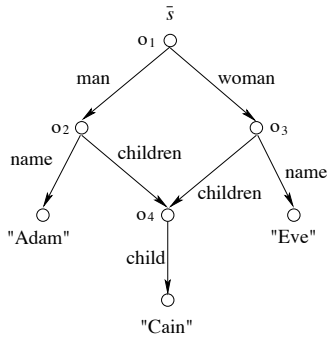



Figure 6.5: Ambiguous typing

Indeed, the edge $< o_4, child, \text{“}Cain\text{”} >$, $o_4$ would conform to both

$$[child : coll(string)] \text{ and } [child : string].$$

That is, SSDBs provide no way to differentiate an oid representing a record with a collection field from one representing a record with a non-collection field. Due to such lack of self-description, SSDBs may lead to an ambiguous typing, as in the case of $o_4$. For example, at run-time one could access $o_4$ as a collection value and drop the only element therein. This would cause an inconsistency whenever $o_4$ is accessed as a record in later stages.

Marked edges provide the degree of self-description required by the edges, i.e. by the graphs, to disambiguate typing. Extraction becomes then the transformation of an SSDB into a value by adding the marks required by the type at hand.

Now, consider conformity between types and values, the latter deprived of empty collection values. The only possible way to represent an empty collection of type $[l : coll(T)]$ would be to consider an oid $o$ with no outgoing edges labelled as $l$. This definition, however, would be too loose and lead to ambiguous typing when shared oids are involved. Consider, for example, the typing of the value $d$ in Figure 6.6 with respect to the type,

$$\begin{array}{l}[\\ \quad man : [children : [child : coll(string)]],\\ \quad woman : [children : [child : coll(int)]]\\ ].\end{array}$$

The oid $o_4$ could be typed as both $[child : coll(string)]$ and $[child : coll(int)]$. Of course, the addition of elements to either collection would cause a run-time

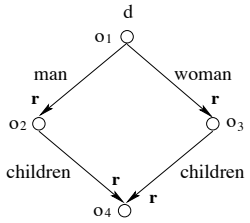



Figure 6.6: Ambiguous typing due to empty collections

type inconsistency. The introduction of typed empty collection values prevents this anomaly and always ensures an unambiguous typing.

## 6.6 Axiomatisation of typing

In this Section, we provide an inductive/algorithmic axiomatisation of typing, which we shall prove sound and complete with respect to the definition of typing ($\triangleright$). This axiomatisation proves the usability of our type language and provides the foundation for the proof of soundness of the extraction algorithm for $L$.

**Definition 6.6.1** *(Axiomatisation of typing) Let $d \in D$, $T \in \mathbf{T}$. $d$* has type *$T$ ($d : T$) if and only if $\emptyset; d_e \vdash d_r :: T$, where:*

$$A;\, ME \vdash o :: int \qquad o \in Integer \tag{INT}$$

$$A;\, ME \vdash o :: string \qquad o \in String \tag{STRING}$$

$$A \cup (o,T);\, ME \vdash o :: T \tag{HYP}$$

$$\frac{T =_{\mathbf{T}} T'}{A;\, ME \vdash \emptyset_{T'} :: T} \tag{EMPTYCOLL}$$

$$\frac{A;\, ME \vdash o :: T\left[\mu X.\, T/_X\right]}{A;\, ME \vdash o :: \mu X.\, T} \tag{REC}$$

$$\frac{\begin{array}{l} Label(ME(o)) = \{l_1, \ldots, l_n\} \wedge \\ (\forall i : 1, \ldots, n. \forall me \in ME(o, l_i). \\ \quad (T_i \not\equiv_{\mathbf{T}} coll(U) \;\Rightarrow\; mark(me) = r \;\wedge\; A \cup (o, \overline{T});\, ME \vdash \overrightarrow{me} :: T_i)) \wedge \\ \quad (T_i \equiv_{\mathbf{T}} coll(U) \;\Rightarrow\; mark(me) = c \;\wedge\; (A \cup (o, \overline{T});\, ME \vdash \overrightarrow{me} :: U)) \end{array}}{A;\, ME \vdash o :: \overline{T} \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n]} \; (RECORD - COLL)$$

$$\frac{A;\, ME \vdash o :: T_1}{A;\, ME \vdash o :: T_1 + T_2} \; (UNION - L)$$

$$\frac{A;\, ME \vdash o :: T_2}{A;\, ME \vdash o :: T_1 + T_2} \; (UNION - R)$$

*where rule* $(UNION - L)$ *has precedence over the rule* $(UNION - R)$.

The judgement $A;\, ME \vdash o :: T$ states that $o$ conforms to $T$ according to $ME$ and under the assumptions $A$. This axiomatisation grounds on the same principles of the type equivalence rules given in Section 6.2.2. In particular, assumptions are enriched with record types rather than with $\mu$-types as in type equivalence. This does not compromise termination and consistency of the rules, as the restriction to canonical types (see Definition 6.1.2) ensures that all $\mu$-types in $\mathbf{T}$ include at least one record type. Moreover, rule $(HYP)$ plays the same role.

The rules provide a tool for algorithmically proving conformity of an oid to a type with respect to a set of marked edges, but do not directly define typing. Indeed, the rules define a larger set of tuples $< A, ME, o, T >$, such that $A;\, ME \vdash o :: T$. Typing relation is instead defined by the subset of tuples $< \emptyset, ME, o, T >$, such that $(o, ME)$ is a value of $L$, associated with the judgements $\emptyset;\, ME \vdash o :: T$.

Next, we prove this axiomatisation is sound and complete with respect to the definition of typing.

**Theorem 6.6.2** *(Soundness and completeness of typing) Let* $d \in D$, $T \in \mathbf{T}$.

$$d \rhd T \;\Leftrightarrow d : T$$

Do do so, we shall rely on the definition of derivation tree,

**Definition 6.6.3** *(Derivation Tree) Given a judgement* $A;\, ME \vdash o :: T$ *a* derivation tree *is a term of the following grammar:*

$$DT ::= \frac{DT_1, \ldots, DT_n}{A;\, ME \vdash o :: T} \mid T =_{\mathbf{T}} T' \mid o \in Integer \mid o \in String \mid fail$$

*where* $n \geq 0$.

### 6.6.1 Completeness

Completeness states that given $d \in D$ and $T \in \mathbf{T}$, $d \rhd T$ entails $d : T$. To prove completeness we rely on an algorithm `Proof` (see Figure 6.7) that given $o \in Obj^+$, a set of edges $ME$, and a canonical type $T$, returns the derivation tree for the judgement $\emptyset; ME \vdash o :: T$. We shall prove that this algorithm terminates and than that if $o \rhd_{ME} T$ then $\texttt{Proof}(\emptyset, ME, o, T)$ returns a valid derivation for $\emptyset; ME \vdash o :: T$. Consequently, since $d \rhd T$ implies that $d_r \rhd_{d_e} T$, we can prove prove that the judgement $\emptyset; d_e \vdash d_r :: T$ is valid, from which, by definition, $d : T$.

Note that a call $\texttt{Proof}(A, ME, o, T)$ returns the derivation tree relative to the judgement $A; ME \vdash o :: T$. When no rule is applicable, hence a judgement cannot be proven, the algorithm returns $fail$; hence, all derivation trees generated by the algorithm, whose leaves are different from *fail*, are valid.

#### Syntactic properties of types

In this section we point out an interesting property of types in $\mathbf{T}$, which will be fundamental in the proof of termination of the algorithm `Proof`. $\mathbf{T}$ is a well founded set with respect to the following relation of *syntactic inclusion*, and each type $T$ has a finite number of syntactical subterms.

**Definition 6.6.4** *(Syntactic subterms)*
*Let $T, T' \in \mathbf{T}$. $T$ is said to be a* syntactic subterm *of $T'$ if $T \sqsubseteq T'$, where $\sqsubseteq$ is defined as follows:*

$$\frac{}{T \sqsubseteq T} \qquad (REF)$$

$$\frac{\exists i : 1, \ldots, n.\ ((T \sqsubseteq T_i) \wedge (T_i \not\equiv coll(U))) \vee ((T \sqsubseteq U) \wedge (T_i \equiv_{\mathbf{T}} coll(U)))}{T \sqsubseteq [l_1 : T_1, \ldots, l_n : T_n]} \qquad (RECORD - COLL)$$

$$\frac{T \sqsubseteq T_1}{T \sqsubseteq T_1 + T_2} \qquad (UNION - L)$$

$$\frac{T \sqsubseteq T_2}{T \sqsubseteq T_1 + T_2} \qquad (UNION - R)$$

$$\frac{T \sqsubseteq U\left[{}^{\mu X.\ U}/_{X}\right]}{T \sqsubseteq \mu X.\ U} \qquad (UNFOLD)$$

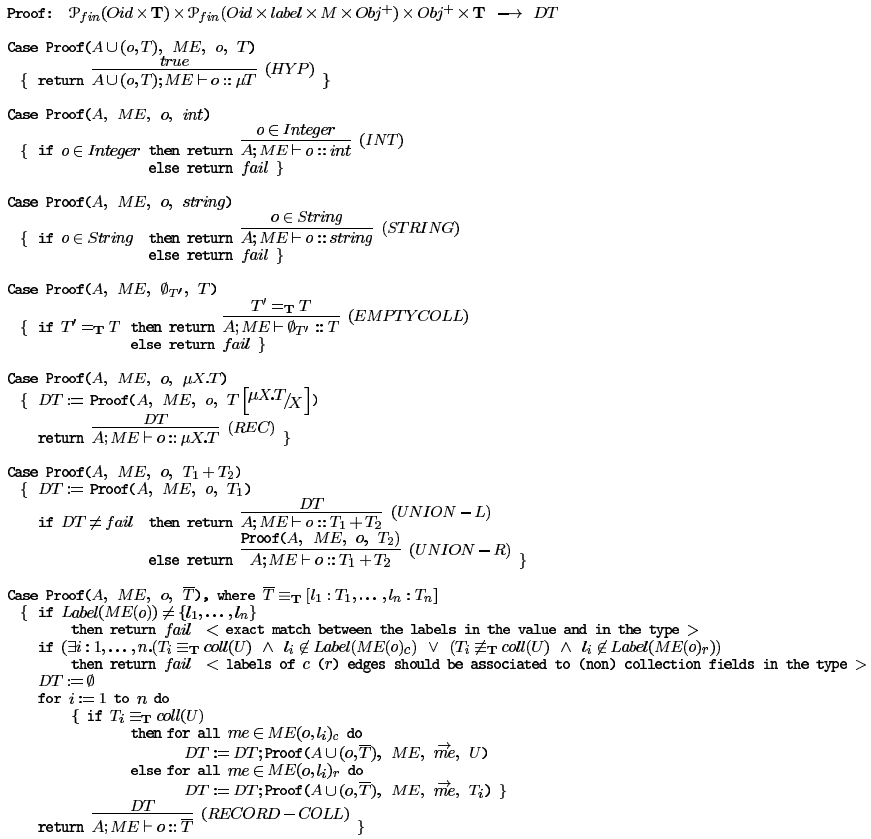



Figure 6.7: Algorithm Proof

**Lemma 6.6.5** *The relation $\sqsubseteq$ is transitive, i.e. $\forall U, V, Z \in \mathbf{T}$ if $U \sqsubseteq V$, $V \sqsubseteq Z$, than $U \sqsubseteq Z$.*

**Proof.** We prove this statement by induction on the generic derivation $V \sqsubseteq Z$ of syntactic inclusion. First we prove it for the axiom rule (*REF*), and then for the other rules, by assuming that thesis holds for the premises of each rule.

**Case** (*REF*)**:** we know that $U \sqsubseteq V$ and that $V \equiv_{\mathbf{T}} Z$, thus, $U \sqsubseteq V \equiv_{\mathbf{T}} Z$.

**Case** (*RECORD*)**:** we know that $U \sqsubseteq V$ and that, by the premises of the rule, for $i : 1, \ldots, n$:

if $T_i \equiv_{\mathbf{T}} coll(U')$ and $V \sqsubseteq U'$;
if $T_i \not\equiv_{\mathbf{T}} coll(U')$ and $V \sqsubseteq T_i$.

By Induction Hypothesis (from now on IH) we know that:

if $T_i \equiv_{\mathbf{T}} coll(U')$ then $U \sqsubseteq U'$;
if $T_i \not\equiv_{\mathbf{T}} coll(U')$ then $U \sqsubseteq T_i$.

Accordingly, an application of rule (*RECORD*) gives $U \sqsubseteq [l_1 : T_1, \ldots, l_n : T_n]$.

**Case** $(UNION - L)$**:** we know that $U \sqsubseteq V$ and that, by the premises of the rule, $V \sqsubseteq T_1$. By IH $U \sqsubseteq T_1$, hence an application of rule $(UNION - L)$ gives $U \sqsubseteq T_1 + T_2$.

**Case** $(UNION - R)$**:** similar to the previous case.

**Case** (*UNFOLD*)**:** we know that $U \sqsubseteq V$ and that, by the premises of the rule, $V \sqsubseteq T\left[\mu X.T/X\right]$. By IH $U \sqsubseteq T\left[\mu X.T/X\right]$, hence an application of rule (*UNFOLD*) gives $U \sqsubseteq \mu X.T$.

■

The next step is the definition of the *subterm closure* of a type, which is a function that returns the set of all syntactic subterms of a type.

**Definition 6.6.6** *(Subterm closure)*
*The* subterm closure *of $T \in \mathbf{T}$ is denoted as $T^*$ and defined as:*

$X^* = \{X\}$

$$(\mu X.T)^* = \{\mu X.T\} \cup T^* \left[{}^{\mu X.T}/_{X}\right]$$

$$(T_1 + T_2)^* = \{T_1 + T_2\} \cup T_1^* \cup T_2^*$$

$$int^* = \{int\}$$

$$string^* = \{string\}$$

$$([l_1 : T_1, \ldots, l_n : T_n])^* =$$

$$\{[l_1 : T_1, \ldots, l_n : T_n]\} \cup \bigcup_{i=1 T_i \equiv_{\mathbf{T}} coll(U)}^{n} U^* \cup \bigcup_{i=1 T_i \not\equiv_{\mathbf{T}} coll(U)}^{n} T_i^*$$

*where substitution is applied element wise to sets of recursive types.*

We then require the definition of the following property to be able to prove that the relation of subterm closure is sound with respect to syntactic subterm definition.

**Lemma 6.6.7** *Subterm closure commutes with substitution, i.e.*

$$(T' \left[{}^{T}/_{X}\right])^* = (T')^* \left[{}^{T}/_{X}\right] \cup T^*$$

*where* $X \in fv(T')$.

**Proof.** We prove this statement by induction on the structure of $T'$ such that $X \in fv(T')$.

**Case** $T' = X$**:** the left hand side of the equation evaluates to

$$(X \left[{}^{T}/_{X}\right])^* = T^*$$

while the left hand side yields,

$$(X)^* \left[{}^{T}/_{X}\right] \cup T^* = \{X\} \left[{}^{T}/_{X}\right] \cup T^* = \{T\} \cup T^* = T^*$$

**Case** $T' = \mu Y.U$**:** we can assume that $Y \notin fv(T)$. By IH we know that,

$$(U \left[{}^{T}/_{X}\right])^* = U^* \left[{}^{T}/_{X}\right] \cup T^*$$

therefore,

$$
\begin{aligned}
&((\mu Y.U)\left[T/X\right])^* = \\
&= (\mu Y.U\left[T/X\right])^* = \\
&= \{\mu Y.U\left[T/X\right]\} \cup (U\left[T/X\right])^* \left[\mu Y.U\left[T/X\right]/Y\right] = \\
&\quad \{\text{IH}\} \\
&= \{\mu Y.U\}\left[T/X\right] \cup (U^*\left[T/X\right] \cup T^*)\left[\mu Y.U\left[T/X\right]/Y\right] = \\
&= \{\mu Y.U\}\left[T/X\right] \cup (U^*\left[T/X\right])\left[\mu Y.U\left[T/X\right]/Y\right] \cup T^*\left[\mu Y.U\left[T/X\right]/Y\right] = \\
&\quad \{\, Y \notin fv(T) \text{ and } \left[\mu Y.U\left[T/X\right]/Y\right] = \left[\mu Y.U/Y\right]\left[T/X\right]\} \\
&= \{\mu Y.U\}\left[T/X\right] \cup U^*\left[\mu Y.U/Y\right]\left[T/X\right]) \cup T^* = \\
&= (\{\mu Y.U\} \cup U^*\left[\mu Y.U/Y\right])\left[T/X\right] \cup T^* = \\
&= (\mu Y.U)^*\left[T/X\right] \cup T^*
\end{aligned}
$$

**Case** $T' = T_1 + T_2$**:** since $X \in fv(T')$, than it must occur free in either $T_1$ or $T_2$. We assume that $X \in fv(T_1)$ and $X \notin fv(T_2)$, and, by IH,

$$(T_1\left[T/X\right])^* = T_1^*\left[T/X\right] \cup T^*$$

Therefore,

$$
\begin{aligned}
&((T_1 + T_2)\left[T/X\right])^* = \\
&= (T_1\left[T/X\right] + T_2\left[T/X\right])^* = \\
&= \{(T_1\left[T/X\right] + T_2\left[T/X\right])\} \cup (T_1\left[T/X\right])^* \cup (T_2\left[T/X\right])^* = \\
&\quad \{\text{ IH and } X \notin fv(T_2)\ \} \\
&= \{T_1 + T_2\}\left[T/X\right] \cup T_1^*\left[T/X\right] \cup T^* \cup T_2^*\left[T/X\right] = \\
&= (\{T_1 + T_2\} \cup T_1^* \cup T_2^)\left[T/X\right] \cup T^* = \\
&(T_1 + T_2)^*\left[T/X\right] \cup T^*
\end{aligned}
$$

The evaluation when $X \in fv(T_2)$ or $X \in fv(T_2) \cap fv(T_1)$ is similar.

**Case** $T' = [l_1 : T_1, \ldots, l_n : T_n]$**:** since $X \in fv(T')$, than there exists $i : 1, ..., n$ such that, $X \in fv(T_i)$**.** By IH we than know that for such $i$,

if $T_i \equiv_{\mathbf{T}} coll(U)$ then

$$(U\left[{}^{T}/_{X}\right])^* = U^*\left[{}^{T}/_{X}\right] \cup T^*$$

if $T_i \not\equiv_{\mathbf{T}} coll(T)$ then

$$(T_i\left[{}^{T}/_{X}\right])^* = T_i^*\left[{}^{T}/_{X}\right] \cup T^*$$

In the following we assume that there exists only one such $i$, namely $i = 1$, and prove the equality for both cases listed above. From the proof it is clear that the result is not affected by the number of $T_i$'s in which $X$ occurs free.

$\boxed{i = 1, T_1 \equiv_{\mathbf{T}} coll(U)}$

$$\begin{array}{l}
([l_1 : T_1, \ldots, l_n : T_n]\left[{}^{T}/_{X}\right])^* = \\
= ([l_1 : T_1\left[{}^{T}/_{X}\right], \ldots, l_n : T_n\left[{}^{T}/_{X}\right]])^* = \\
\{\ T_1 \equiv_{\mathbf{T}} coll(U)\ \} \\
= \{[l_1 : T_1\left[{}^{T}/_{X}\right], \ldots, l_n : T_n\left[{}^{T}/_{X}\right]]\} \cup (U\left[{}^{T}/_{X}\right])^* \cup \\
\quad \bigcup_{2 \le i \le n,\ T_i \equiv_{\mathbf{T}} coll(U')} (U'\left[{}^{T}/_{X}\right])^* \cup \\
\quad \bigcup_{2 \le i \le n,\ T_i \not\equiv_{\mathbf{T}} coll(U')} (T_i\left[{}^{T}/_{X}\right])^* = \\
\{\ \text{IH}\ \} \\
= \{[l_1 : T_1, \ldots, l_n : T_n]\}\left[{}^{T}/_{X}\right] \cup U^*\left[{}^{T}/_{X}\right] \cup T^* \\
\quad \bigcup_{2 \le i \le n,\ T_i \equiv_{\mathbf{T}} coll(U')} U'^*\left[{}^{T}/_{X}\right] \cup \\
\quad \bigcup_{2 \le i \le n,\ T_i \not\equiv_{\mathbf{T}} coll(U')} T_i^*\left[{}^{T}/_{X}\right] = \\
= (\{[l_1 : T_1, \ldots, l_n : T_n]\} \cup U^* \cup \bigcup_{2 \le i \le n,\ T_i \equiv_{\mathbf{T}} coll(U')} U'^* \cup \\
\quad \bigcup_{2 \le i \le n,\ T_i \not\equiv_{\mathbf{T}} coll(U')} T_i^*)\left[{}^{T}/_{X}\right] \cup T^* = \\
= ([l_1 : T_1, \ldots, l_n : T_n])^*\left[{}^{T}/_{X}\right] \cup T^*
\end{array}$$

$\boxed{i = 1, T_1 \not\equiv_{\mathbf{T}} coll(U)}$

$$\begin{array}{l}
([l_1 : T_1, \ldots, l_n : T_n]\left[{}^{T}/_{X}\right])^* = \\
= ([l_1 : T_1\left[{}^{T}/_{X}\right], \ldots, l_n : T_n\left[{}^{T}/_{X}\right]])^* = \\
\quad \{\ T_1 \not\equiv_{\mathbf{T}} coll(U)\ \} \\
= \{[l_1 : T_1\left[{}^{T}/_{X}\right], \ldots, l_n : T_n\left[{}^{T}/_{X}\right]]\} \cup (T_1\left[{}^{T}/_{X}\right])^* \cup
\end{array}$$

$$
\begin{array}{l}
\bigcup_{2\leq i\leq n,\ T_i\equiv_{\mathbf{T}} coll(U')}(U'\left[{}^{T}/_{X}\right])^*\cup \\
\bigcup_{2\leq i\leq n,\ T_i\not\equiv_{\mathbf{T}} coll(U')}(T_i\left[{}^{T}/_{X}\right])^* = \\
\{\text{ IH }\} \\
= \{[l_1:T_1,\ldots,l_n:T_n]\}\left[{}^{T}/_{X}\right]\cup T_1^*\left[{}^{T}/_{X}\right]\cup T^*\cup \\
\bigcup_{2\leq i\leq n,\ T_i\equiv_{\mathbf{T}} coll(U')} U'^*\left[{}^{T}/_{X}\right]\cup \\
\bigcup_{2\leq i\leq n,\ T_i\not\equiv_{\mathbf{T}} coll(T)} T_i^*\left[{}^{T}/_{X}\right] = \\
= \{[l_1:T_1,\ldots,l_n:T_n]\cup T_1^*\cup \\
\bigcup_{2\leq i\leq n,\ T_i\equiv_{\mathbf{T}} coll(T)} U'^*\cup \\
\bigcup_{2\leq i\leq n,\ T_i\not\equiv_{\mathbf{T}} coll(T)} T_i^*\}\left[{}^{T}/_{X}\right]\cup T^* = \\
= ([l_1:T_1,\ldots,l_n:T_n])^*\left[{}^{T}/_{X}\right]\cup T^*
\end{array}
$$

■

Now we have the tools to prove that $\{U \mid U \sqsubseteq T\} \subseteq T^*$.

**Lemma 6.6.8** *if* $T \sqsubseteq U$ *then* $T \in U^*$

**Proof.** We prove this statement by induction on the generic derivation $T \sqsubseteq U$.

**Case** $(REF)$**:** the result follows by simply observing that $T \in T^*$.

**Case** $(UNION - L)$**:** we know that $T \sqsubseteq T_1$ and that, by IH, $T \in T_1^*$. Note that,

$$T \subseteq (T_1 + T_2)^* = \{T_1 + T_2\} \cup T_1^* \cup T_2^*$$

from which $T \in (T_1 + T_2)^*$;

**Case** $(UNION - R)$**:** the same as above;

**Case** $(RECORD)$**:** we know that there exists $i : 1, \ldots, n$ such that either $T_i \not\equiv_{\mathbf{T}} coll(U)$ and $T \sqsubseteq T_i$ or $T_i \equiv_{\mathbf{T}} coll(U)$ and $T \sqsubseteq U$**.** By IH, we know that either $T \in U$ for $T_i \equiv_{\mathbf{T}} coll(U)$ or $T \in T_i$ for $T_i \not\equiv coll(U)$. In both cases we observe that,

$$
\begin{array}{l}
T \subseteq ([l_1:T_1,\ldots,l_n:T_n])^* = \\
= \{[l_1:T_1,\ldots,l_n:T_n]\} \cup \bigcup_{i=1\, T_i\equiv_{\mathbf{T}} coll(U)}^{n} U^* \cup \bigcup_{i=1\, T_i\not\equiv_{\mathbf{T}} coll(U)}^{n} T_i^*
\end{array}
$$

Therefore, we get that $T \in ([l_1 : T_1, \ldots, l_n : T_n])^*$;

**Case** (*UNFOLD*)**:** we know that $T \sqsubseteq U\left[{}^{\mu X.U}/_{X}\right]$. Moreover, by IH, we know that $T \in (U\left[{}^{\mu X.U}/_{X}\right])^*$. Since substitution and syntactic closure commute (Lemma 6.6.7), we can conclude that,

$$T \in U^*\left[{}^{\mu X.U}/_{X}\right]) \cup (\mu X.U)^*$$

By definition of subterm closure,

$$U^*\left[{}^{\mu X.U}/_{X}\right] \subseteq (\mu X.U)^*$$

hence $T \in (\mu X.U)^*$.

■

The next step is proving that the number of types in a subterm closure is finite. On the base of soundness of subterm closure with respect to the syntactic subterm relation, we can then prove that the number of subterms of a type is finite.

**Lemma 6.6.9** *For all* $T \in \mathbf{T}$, $\mid T^* \mid < \infty$

**Proof.** We prove this statement by induction on the structure of $T$.

**Case** $T = int$**:** $\mid int^* \mid = \mid \{int\} \mid = 1 < \infty$.

**Case** $T = string$**:** $\mid string^* \mid = \mid \{string\} \mid = 1 < \infty$**.**

**Case** $T = X$**:** $\mid X^* \mid = \mid \{X\} \mid = 1 < \infty$.

We assume that the cardinality of the subterm closure of the subterms of a type is finite.

**Case** $T = \mu Y.U$**:** by IH $\mid U^* \mid < \infty$, hence,

$$\mid \mu Y.U^* \mid = \mid \{\mu Y.U\} \cup U^*\left[{}^{\mu Y.U}/_{Y}\right] \mid = 1 + \mid U^*\left[{}^{\mu Y.U}/_{Y}\right] \mid < \infty$$

**Case** $T = T_1 + T_2$**:** by IH $\mid T_1^* \mid < \infty$ and $\mid T_2^* \mid < \infty$, hence,

$$\mid (T_1 + T_2)^* \mid = \mid \{T_1 + T_2\} \cup T_1^* \cup T_2^* \mid = 1 + \mid T_1^* \cup T_2^* \mid < \infty$$

**Case** $T = [l_1 : T_1, \ldots, l_n : T_n]$**:** by IH we know that for all $i : 1, \ldots, n$, $\mid T_i^* \mid < \infty$ if $T_i \not\equiv_{\mathbf{T}} coll(U)$ and $\mid U^* \mid < \infty$ if $T_i \not\equiv_{\mathbf{T}} coll(U)$. Therefore,

$$\begin{array}{l} \mid [l_1 : T_1, \ldots, l_n : T_n]^* \mid = \\ = \mid \{[l_1 : T_1, \ldots, l_n : T_n]\} \mid + \mid \bigcup_{i=1 T_i \equiv_{\mathbf{T}} coll(U)}^{n} U^* \mid + \mid \bigcup_{i=1 T_i \not\equiv_{\mathbf{T}} coll(U)}^{n} T_i^* \mid = \\ = 1 + \mid \bigcup_{i=1 T_i \equiv_{\mathbf{T}} coll(U)}^{n} U^* \mid + \mid \bigcup_{i=1 T_i \not\equiv_{\mathbf{T}} coll(U)}^{n} T_i^* \mid < \infty \end{array}$$

■

**Corollary 6.6.10** *For all* $T \in \mathbf{T}$, $\mid \{T' | T' \sqsubseteq T\} \mid < \infty$.

**Proof.** From Lemma 6.6.8 we can infer that,

$$\{T' | T' \sqsubseteq T\} \subseteq T^*$$

and $\mid T^* \mid < \infty$ by Lemma 6.6.9, from which the thesis follows.

■

### Properties of SSDBs

To prove the termination of the algorithm `Proof`, given in Figure 6.7, we also require to show that the number of oids of a value that can be visited by `Proof` is finite. From the definition of $D$ we can directly infer the following lemma.

**Lemma 6.6.11** *Let* $d \in D$ *and* $o \in Oid(d)$, *then*

$$\mid \{o' \in Oid(d) \mid o' \leq_{de} o\} \mid < \mid Oid(d) \mid < \infty$$

**Proof.** The proof follows from the definition of $D$. The number of edges in a value is finite, hence the number of oids is finite. Besides, the set of oids reachable from an oid $o \in Oid(d)$ is clearly a subset of the possible oids in $d$, i.e. those reachable with a path from $o$.

■

### Algorithm execution properties

Next, we formalise some aspects of the computational behavior of `Proof` so as to prove its termination. Each algorithm call `Proof` recursively issues a sequence of subcalls in chronological order. Each subcall in the sequence is invoked once the previous one has terminated and so on. Accordingly, each call can be associated with an ordered tree of all the subcalls recursively generated by it.

**Definition 6.6.12** *(Call tree) Given an algorithm* `fun`, *the* call tree *of the recursive computation* $\texttt{fun}(x)$ *is a multi branched, node-labelled tree defined by,*

$$CT(\texttt{fun}(x)) = \\ < \texttt{fun}(x), [CT(\texttt{fun}_1(x_1)), \ldots, CT(\texttt{fun}_n(x_n))] >$$

*where the* $\texttt{fun}_i(x_i)$*'s, with* $i : 1, \ldots, n$, *are algorithm calls, in chronological order, recursively issued by* $\texttt{fun}(x)$.

**Definition 6.6.13** *(Call path)*
*Given an algorithm* `fun`, *a* call path *of the computation* $\texttt{fun}(x)$ *is a list of calls corresponding to the labels along a tree path in* $CT(\texttt{fun}(x))$.

**Definition 6.6.14** *(Call chain)*
*Given an algorithm* `fun`*, a* call chain *of the computation* $\texttt{fun}(x)$ *is a list of nodes corresponding to the preorder traversal of* $CT(\texttt{fun}(x))$*, i.e. the list in chronological order of all calls to* `fun` *caused by the computation* $\texttt{fun}(x)$*,*

$$[\texttt{fun}_1(x_1), \ldots, \texttt{fun}_i(x_i), \ldots]$$

Observe that, if an algorithm does not terminate, its call trees may be infinitely deep as they would result in infinite call paths and chains.

**Theorem 6.6.15** *Let* $d \in D$*,* $o_1 \in Obj^+(d)$*,* $T_1 \in \mathbf{T}$*,* $ME \subseteq d_e$*, and* $A_1$ *an assumption set. If*

$$[\texttt{Proof}(A_1, ME, o_1, T_1), \ldots, \texttt{Proof}(A_n, ME, o_n, T_n), \ldots]$$

*where for all* $1 \leq i \leq n$*,* $o_i \in Obj^+(d)$ *and* $T_i \in \mathbf{T}$*, is a call path relative to the call* $\texttt{Proof}(A_1, ME, o_1, T_1)$*, then for all* $1 \leq i \leq n$*:*

$$o_i \leq_{ME} o_1 \wedge T_i \sqsubseteq T_1$$

**Proof.** We prove this statement by induction on the number $n$ of nodes we regard in a call path.

$\boxed{n=1}$ Trivial, from reflexivity of $\leq_{ME}$ and $\sqsubseteq$.

$\boxed{n>1}$ We assume that the thesis holds for the first $n-1$ nodes in the call path and perform a case analysis of the algorithm cases generating the $n$'th node in the call path. Note that if $n > 1$ then for $i : 1, \ldots, n-1$. $o_i \in Oid(d)$.

**Case** $\texttt{Proof}(A, ME, o, \mu X.T)$**:** the $n$'th execution step is then,

$$\texttt{Proof}(A,\ ME,\ o,\ T\left[\mu X.T/X\right]).$$

By IH we know that $o \leq_{ME} o_1$ and that $\mu X.T \sqsubseteq T_1$. By 6.6.5, we get that,

$$\dfrac{\dfrac{\dfrac{}{T\left[\mu X.T/X\right] \sqsubseteq T\left[\mu X.T/X\right]}\,(REF)}{T\left[\mu X.T/X\right] \sqsubseteq \mu X.T}\,(UNFOLD) \qquad \dfrac{}{\mu X.T \sqsubseteq T_1}\,(IH)}{T\left[\mu X.T/X\right] \sqsubseteq T_1}\,(TRANS)$$

**Case** $\texttt{Proof}(A, ME, o, U_1 + U_2)$**:** two $n$'th steps may arise from this case, symmetrical to each other; either

$$\texttt{Proof}(A, ME, o, U_1)$$

or

$$\texttt{Proof}(A,\ ME,\ o,\ U_2).$$

Let's assume we are in the case $\texttt{Proof}(A,\ ME,\ o,\ U_1)$, then by IH we know that $U_1 + U_2 \sqsubseteq T_1$, and directly that $o \leq_{ME} o_1$. Rule $(UNION - L)$ tells us that $U_1 \sqsubseteq U_1 + U_2$, hence by transitivity we can conclude that $U_1 \sqsubseteq T_1$.

**Case** $\texttt{Proof}(A,\ ME,\ o,\ [l_1 : T'_1, \ldots, l_n : T'_n])$**:** there are two kinds of $n$'th step that may arise from this case. The first one is of the form,

$$\texttt{Proof}(A',\ ME,\ o',\ T'_i)$$

for all $T'_i \not\equiv_{\mathbf{T}} coll(U)$; the second one is of the form,

$$\texttt{Proof}(A',\ ME,\ o',\ U)$$

for all $T'_i \equiv_{\mathbf{T}} coll(U)$, where $A' = A \cup (o, [l_1 : T'_1, \ldots, l_n : T'_n])$. From rule $(RECORD - COLL)$ we know that $T'_i \sqsubseteq [l_1 : T'_1, \ldots, l_n : T'_n]$ and $U \sqsubseteq [l_1 : T'_1, \ldots, l_n : T'_n]$, respectively.

By IH we know that $[l_1 : T'_1, \ldots, l_n : T'_n] \sqsubseteq T_1$, hence by transitivity, we can conclude, in both cases, that $T'_i \sqsubseteq T_1$ and $U \sqsubseteq T_1$.

By IH we know that $o \leq_{ME} o_1$, hence that there exists a path $p_o = [< o_1, l'_1, m_1, o'_1 >, < o'_1, l'_2, m_2, o'_2 > \ldots, < o'_n, l'_n, m_n, o >]$ in $d_e$. The call $\texttt{Proof}(A',\ ME,\ o',\ T'_i)$, as well as $\texttt{Proof}(A',\ ME,\ o',\ U)$, is issued because there exists an edge $< o, l_i, m_i, o' > \in ME(o)_{l_i}$. Therefore, $o' \leq_{d_e} o_1$ because $[p_o, < o, l_i, m_i, o' >]$ is a path in $d_e$.

■

**Lemma 6.6.16** *If* $\texttt{Proof}(A_0,\ ME,\ o_0,\ T_0), \ldots, \texttt{Proof}(A_i,\ ME,\ o_i,\ T_i), \ldots$ *is a call path of* $CT(\texttt{Proof}(A_0,\ ME,\ o_0,\ T_0))$*, then,*

$$A_0 \subseteq A_1 \subseteq \ldots \subseteq A_i \subseteq \ldots$$

**Proof.** Call $i+1$ occurs at a deeper level in the path than call $i$. Before invoking a subcall $i+1$ the assumption set can only be expanded or left untouched, hence $A_i \subseteq A_{i+1}$. Indeed, the proof follows directly from the observation that during the execution of `Proof` the assumption set can only be expanded and never contracted.

■

**Lemma 6.6.17** *Let $d \in D$, $o \in Obj^+(d)$, $T \in \mathbf{T}$, $A_0$ an assumption set, and*

$$[\texttt{Proof}(A_0,\ ME,\ o_0,\ T_0), \ldots,\ \texttt{Proof}(A_i,\ ME,\ o_i,\ T_i), \ldots]$$

*is a call path for* $\texttt{Proof}(A_0,\ ME,\ o_0,\ T_0)$*, then,*

$$\exists N\ \forall i \geq 0 :\ (o_i,\ T_i) \in \bigcup_{0 \leq j \leq N} \{(o_j,\ T_j)\}$$

*Modulo type equivalence and simple equality of oids, every call path is composed by the potentially infinite repetition of the same sequence of labels.*

**Proof.** This statement is proven by contradiction, assuming that

$$\forall N\ \exists i \geq 0 :\ (o_i,\ T_i) \notin \bigcup_{0 \leq j \leq N} \{(o_j,\ T_j)\}$$

A direct implication of this assumption is that $\bigcup_\infty \{(o_j,\ T_j)\}$ is an infinite set. According to Lemma 6.6.15, we know that

$$\bigcup_\infty \{(o_j,\ T_j)\} \subseteq (\bigcup_\infty o_j) \times (\bigcup_\infty T_j) \subseteq \{o' \mid o' \leq_{ME} o_0\} \times \{T' \mid T' \sqsubseteq T_0\}$$

From the corollaries 6.6.10 and 6.6.11, we can infer that,

$$|\bigcup_\infty \{(o_j,\ T_j)\}| \leq |\{o' \mid o' \leq_{ME} o_0\}| \cdot |\{T' \mid T' \sqsubseteq T_0\}| \leq \infty$$

contradicting the fact that $\bigcup_\infty \{(o_j,\ T_j)\}$ is an infinite set. Consequently, our assumption was false and the proposition true.

■

**Theorem 6.6.18** *(Termination of* `Proof`*) If $d \in D$, $o \in Obj^+(d)$, $T \in \mathbf{T}$, $ME \subseteq d_e$, and $A$ is an assumption set, than the call* $\texttt{Proof}(A,\ ME,\ o,\ T)$ *terminates.*

**Proof.** We proceed by contradiction, by assuming that $\texttt{Proof}(A,\ ME,\ o,\ T)$ does not terminate. Consequently, $CT(\texttt{Proof}(A,\ ME,\ o,\ T)$ has an infinite path $p$. By Lemma 6.6.17, there exists $N$ such that:

$$\forall i \geq 0 :\ (o_i,\ T_i) \in \bigcup_{0 \leq j \leq N} \{(o_j,\ T_j)\}$$

There exists $i > N$ such that $(o_i,\ T_i) \equiv (o,\ [l_1 : T_1, \ldots, l_n : T_n])$ in $p$. Indeed, had not this pair existed, all calls would be relative to union types or $\mu$-types, which is not possible as $\mathbf{T}$ defines canonical types (see Definition 6.1.2).

As $i > N$, there exists $(o_k, T_k) \equiv (o_i, [l_1 : T'_1 \ldots, l_n : T'_n])$ such that $k \leq N$ and $T_i =_{\mathbf{T}} T_k$. This pair is relative to a call $\texttt{Proof}(A_k,\ ME,\ o_k,\ T_k)$ such that $k < i$. Thus, by Lemma 6.6.16, the assumption set $A_i$ contains the pair $(o_k, T_k)$. Therefore, according to the record case of the algorithm, the call $\texttt{Proof}(A_i,\ ME,\ o_i,\ T_i)$ must terminate and $p$ cannot be infinite.

■

**Lemma 6.6.19** *Let $d \in D$, $T \in \mathbf{T}$, $o \in Obj^{+}(d)$, $ME \subseteq d_e$, and $A$ an assumption set, then:*

$$o \rhd_{ME} T \;\Rightarrow\; \texttt{Proof}(A,\ ME,\ o,\ T) = DT$$

*where $DT$ is a valid derivation for $A; ME \vdash o :: T$.*

**Proof.** Induction on the finite number $i$ of recursive calls required for the termination of the call $\texttt{Proof}(A,\ ME,\ o,\ T)$.

$\boxed{i = 0}$

**Case** $\texttt{Proof}(A \cup (o,T),\ ME, o, T)$**:** the algorithm returns the valid derivation

$$(HYP)\ A \cup (o,T); ME \vdash o :: T;$$

**Case** $\texttt{Proof}(A,\ ME, o, \mathit{int})$**:** the algorithm returns a derivation $(INT)\ A; ME \vdash o :: \mathit{int}$, which is valid because from $o \rhd_{ME} \mathit{int}$ we know that $o \in \mathit{Integer}$;

**Case** $\texttt{Proof}(A,\ ME, o, \mathit{string})$**:** the same as above;

**Case** $\texttt{Proof}(A,\ ME, \emptyset_{T'}, T)$**:** the same as above, but with the condition $T' =_{\mathbf{T}} T$.

$\boxed{i > 0}$ We assume the thesis holds for the calls of `Proof` that require less than $i$ steps to terminate.

**Case** $\texttt{Proof}(A,\ ME, o,\ \mu X.T)$**:** in this case the $i-1$'th step is

$$\texttt{Proof}(A,\ ME, o,\ T\left[\mu X.T/_X\right]).$$

As we know that $o \rhd_{ME} \mu X.T$, we also know that $o \rhd_{ME} T\left[\mu X.T/_X\right]$ from which, by IH, we get that $\texttt{Proof}(A,\ ME, o,\ T\left[\mu X.T/_X\right])$ generates a valid derivation for $A; ME \vdash o :: T\left[\mu X.T/_X\right]$. Therefore, by rule $(REC)$,

$$\texttt{Proof}(A,\ ME, o, \mu X.T) = \frac{A; ME \vdash o :: T\left[\mu X.T/_X\right]}{A; ME \vdash o :: \mu X.T}\ (REC)$$

is a valid derivation.

**Case** $\texttt{Proof}(A,\ ME, o,\ T_1 + T_2)$**:** in this case the $i-1$'th step is either

$$\texttt{Proof}(A,\ ME, o,\ T_1) \text{ or } \texttt{Proof}(A,\ ME, o,\ T_2).$$

From $o \rhd_{ME} T_1 + T_2$ we can infer either $o \rhd_{ME} T_1$ or $o \rhd_{ME} T_2$. If $o \rhd_{ME} T_1$, by IH, $\texttt{Proof}(A,\ ME, o,\ T_1)$ returns a valid derivation for $A; ME \vdash o :: T_1$. Therefore, by rule $(UNION - L)$,

$$\texttt{Proof}(A,\ ME, o, T_1 + T_2) = \frac{A; ME \vdash o :: T_1}{A; ME \vdash o :: T_1 + T_2}\ (UNION - L)$$

is a valid derivation.

**Case** $\texttt{Proof}(A,\ ME, o, [l_1 : T_1 \ldots, l_n : T_n])$**:** $o \rhd_{ME} [l_1 : T_1 \ldots, l_n : T_n]$ ensures that:

1. since the constraints in `Proof` and in the definition of ME-conformity $\rhd$ are the same, this call does not return $fail$;
2. for each $me \in ME(o)$, this call recursively invokes one call of the general form $\texttt{Proof}(A \cup (o, [l_1 : T_1 \ldots, l_n : T_n]),\ ME, \vec{me},\ T'_{me})$, which terminates in at most $i-1$ steps;
3. $\forall me \in ME(o).\ \vec{me} \rhd_{ME} T'_{me}$.

By IH we can infer that

$$\forall me \in ME(o).\ A \cup (o, [l_1 : T_1 \ldots, l_n : T_n]); ME \vdash \vec{me} :: T'_{me}$$

Therefore, by rule $(RECORD - COLL)$ we can state that,

$\texttt{Proof}(A,\ ME, o, [l_1 : T_1 \ldots, l_n : T_n]) =$

$$\frac{\forall me \in ME(o).\ A \cup (o, [l_1 : T_1 \ldots, l_n : T_n]); ME \vdash \vec{me} :: T'_{me}}{A; ME \vdash o :: [l_1 : T_1 \ldots, l_n : T_n]}$$

is a valid derivation.

■

**Theorem 6.6.20** *(Completeness) Let $d \in D$, $T \in \mathbf{T}$, then,*

$$d \rhd T \ \Rightarrow\ d : T$$

**Proof.** By definition we know that $d \rhd T$ implies that $d_r \rhd_{d_e} T$. By Lemma 6.6.19 we then know that `Proof` returns a valid derivation for $\emptyset; d_e \vdash d_r :: T$, therefore, by definition of typing, $d : T$. ■

### 6.6.2 Soundness

We prove soundness exploiting the principle of coinduction. The proof technique consists in demonstrating that for all $d \in D$ and $T \in \mathbf{T}$ such that $d : T$ there exists a $d_e$-conformity relation $R$ such that $d\,R\,T$. In the following we infer $R$ from the derivation tree relative to the proof of $d : T$.

**Definition 6.6.21** *(Derivation Tree Relation) Let $d \in D$, $T \in \mathbf{T}$, such that $\emptyset; d_e \vdash d_r :: T$ has a valid derivation tree $DT$. We call* derivation tree relation *the relation yielded by the smallest function $Rel : DT \to \mathcal{P}_{fin}(Obj^+ \times \mathbf{T})$ such that:*

$$Rel(DT) = \begin{cases} \{(o, T)\} \cup \bigcup_{i=1}^{n} Rel(DT_i) & \frac{DT_1, \ldots, DT_n}{A; ME \vdash o :: T} \\ \\ \epsilon & \textit{otherwise} \end{cases}$$

*Note that Rel is well defined since any derivation tree $DT$ is finite.*

**Theorem 6.6.22** *(Soundness) Let $d \in D$, $T \in \mathbf{T}$, then,*

$$d : T \;\Rightarrow\; d \rhd T$$

**Proof.** We prove that for all $d$ and $T$ such that $d : T$ there exists a $d_e$-conformity relation $R$ such that $d_r\,R\,T$, from which we can conclude that $d \rhd T$.

By definition of : we know that $d : T$ implies $\emptyset; d_e \vdash d_r :: T$. We dub $DT_{d,T}$ the derivation tree corresponding to that judgement and consider $R = Rel(DT_{d,T})$. First of all note that, by definition of *Rel*, $d_r\,R\,T$ is true. Than we prove that $R$ is a $d_e$-conformity relation by showing that it respects the given conditions:

**Case** $o\,R\,int$**:** if $(o, int) \in R$ there must be a judgment $A; d_e \vdash o :: int$ in $DT_{d,T}$. Since this is a valid judgement, $o \in Integer$;

**Case** $o\,R\,string$**:** the same as above;

**Case** $\emptyset_{T'}\,R\,T$**:** the same as above, but with condition $T' =_{\mathbf{T}} T$;

**Case** $o\,R\,\mu X.T$**:** if $(o, \mu X.T) \in R$ there must be a judgement $A; d_e \vdash o :: \mu X.T$ in $DT_{d,T}$. Since this is a valid judgement, by rule $(REC)$ the judgement $A; d_e \vdash o :: T\left[{}^{\mu X.T}/_{X}\right]$ is in $DT_{d,T}$ too. Therefore, by definition of *Rel*, $o\,R\,T\left[{}^{\mu X.T}/_{X}\right]$;

**Case** $o\,R\,T_1 + T_2$**:** if $(o, T_1 + T_2) \in R$ there must be a judgement $A; d_e \vdash o :: T_1 + T_2$ in $DT_{d,T}$. Since this is a valid judgement, by rules $(UNION - L)$ and $(UNION - R)$ there must be either a judgement $A; d_e \vdash o :: T_1$ or a judgement $A; d_e \vdash o :: T_2$ in $DT_{d,T}$ too. Therefore, by definition of *Rel*, $o\,R\,T_1$ or $o\,R\,T_2$;

**Case** $o\,R\,[l_1 : T_1 \ldots, l_n : T_n]$ **:** if $(o, [l_1 : T_1 \ldots, l_n : T_n]) \in R$ there must be a judgement $A$; $d_e \vdash o :: [l_1 : T_1 \ldots, l_n : T_n]$ in $DT_{d,T}$. Since this is a valid judgement, by rule $(RECORD - COLL)$ the judgements $A'$; $d_e \vdash o_j :: T'_j$, where $0 \leq j \leq n$, $n \geq 0$, and $A' = A \cup (o, [l_1 : T_1 \ldots, l_n : T_n])$, are in $DT_{d,T}$ too. By definition of $Rel$ $\forall j : 0, \ldots, n.\, o_j\, R\, T'_j$ as expected. Since the conditions of rule $(RECORD - COLL)$ are the same as the conformity relation, the thesis holds.

■

# Chapter 7

# Extraction algorithm

In this Chapter we first provide a definition of extractability for $L$, then we give an algorithm `Extraction` based on this definition. We then describe in detail the behaviour of `Extraction` and prove its termination. Moreover, we define relevance of extraction in terms of the measures of *precision of extraction* and *data capturing*.

## 7.1 Extractability for $L$

As shown in Chapter 4, extractability for a language depends on the definition of a specific mapping from values to SSDBs and a definition of inclusion. In the case of $L$, we define extractability by means of the mapping *ssd* defined in Section 6.4 and by lt-inclusion defined in Section 5.2.

**Definition 7.1.1** *(Extractability for L) Let $\overline{s} \in S$, $\overline{T} \in \mathbf{T}$. A value $d$ is* extractable *from $\overline{s}$ according to $\overline{T}$ if $d : \overline{T}$ and there exists $s \in S$ such that $ssd(d) = s$ and $s < \overline{s}$.*

Note that extractability provides extraction system designers and application developers with a precise specification of the extraction algorithm's behaviour. Indeed, `Extraction` has been realised following this definition. Furthermore, algorithm users specify types for extraction according to the current database content, but essentially relying on the definition of *ssd* and $<$.

## 7.2 The code

The extraction algorithm, shown in Figure 7.1, consists of two parts, `Extraction` and `Extract`.

`Extraction` receives an SSDB $\overline{s}$ and a type $\overline{T}$, and yields back either $fail$, if no extraction can be performed, or an extractable value $d = (\overline{s}_r, ME) \in D$. The extraction of $d$ consists of *identifying* a subset of edges in $\overline{s}_e$ and, by appropriately marking them, *generating* a set of marked edges $ME$ such that $\emptyset; ME \vdash \overline{s}_r :: \overline{T}$.

```
Extraction:  S × T → D ∪ {fail}
Extraction(s, T) =
  { (ΔA, ME):= Extract(∅, s_e, s_r, T)
    if ME = fail then return fail
                 else return (s_r, ME)
    endif }

Extract: P_fin(Oid × T) × P_fin(Oid × Label × Obj) × Obj × T →
         P_fin(Oid × T) × P_fin(Oid × Label × M × Obj+) ∪ {fail}
case Extract(A, E, o, μX.T)
  { return Extract(A, E, o, T[μX.T/X]) }

case Extract(A, E, o, int)
  { if o ∈ Integer then return (∅, ∅)
                 else return (∅, fail)
    endif }

case Extract(A, E, o, string)
  { if o ∈ String then return (∅, ∅)
                 else return (∅, fail)
    endif }

case Extract(A, E, o, T1 + T2)
  { (ΔA, ME) := Extract(A, E, o, T1)
    if ME = fail then return Extract(A, E, o, T2))
                 else return (ΔA, ME)
    endif }

case Extract(A, E, o, T), where T ≡_T [l1 : T1, ... , ln : Tn]
  { if ∃T̄ ∈ T. (o, T̄) ∈ A
       then if T =_T T̄ then return (∅, ∅)
                       else return (∅, fail)
            endif
       else { (ΔA, ME) := ({(o,T)}, ∅)   <insertion of extractable oids>
              FAILED := false
              i := 1
              while i < n + 1 and not(FAILED) do
                if T_i ≡_T coll(U)   <EXTRACTION FOR COLLECTION FIELDS>
                  then { EMPTY := true
                         for all e ∈ E(o,l_i) do
                           (ΔA_t, ME_t) :=Extract(A ∪ ΔA, E, e→, U)   <search for selected edges>
                           if ME_t ≠ fail then
                            { ME := ME ∪ ME_t ∪ { <e←, l_i, c, e→>}   <generation of a marked edge>
                              ΔA := ΔA ∪ ΔA_t   <insertion of extracted oids>
                              EMPTY := false }
                           endif
                         endfor
                         if EMPTY then ME := ME ∪ {<o,l_i,c,∅_U >} endif }
                  else { FAILED := true   <EXTRACTION FOR NON-COLLECTION FIELDS>
                         for all e ∈ E(o,l_i) do   <collection field case>
                             (ΔA_t, ME_t) := Extract(A ∪ ΔA, E, e→, T_i)   <search for selected edges>
                             if ME_t ≠ fail then
                              { ME := ME ∪ ME_t ∪ { <e←, l_i, r, e→>}   <generation of a marked edge>
                                ΔA := ΔA ∪ ΔA_t   <insertion of extracted oids>
                                FAILED := false
                                exitfor }
                             endif
                         endfor }
                endif
              endwhile
    endif
    if FAILED then (ΔA, ME) := (∅, fail) endif
    return (ΔA, ME) }

case Extract(A, E, o, T), where o and T do not match any of the cases above
  { return (∅, fail) }
```

Figure 7.1: Extraction algorithm for SSDBs

In particular, $ME$ is generated by calling the recursive algorithm `Extract`, which takes as input:

- an assumption set $A$,
- an oid $o$,
- the set of edges $\overline{s}_e$,
- and a type $T_o$.

At each step, the task of `Extract` is to extract from $\overline{s}_e$ a set of marked edges $ME_o$ that verifies the judgement $A; ME_o \vdash o :: T_o$. We shall see that in the presence of shared oids the generation of marked edges is a non-trivial task and relies on a particular usage of assumption sets.

An example of extraction is illustrated in Figure 7.2. Note how the set $ME$ of edges in the resulting $d$ corresponds to a subset of the edges of $\overline{s}$.

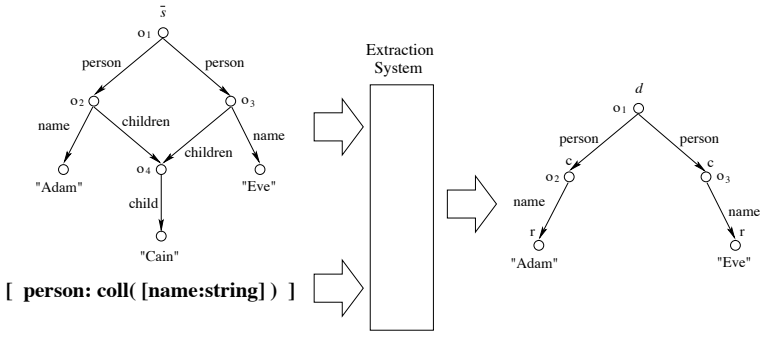



Figure 7.2: Example of extraction

In the following we discuss marked edges generation and assumption sets handling, separately. Finally, for the sake of code readability, we describe some properties characterising the algorithm's behaviour.

### 7.2.1 Generation of marked edges

`Extract` mirrors the algorithm `Proof` defined in Chapter 6, in that it provides a *case* for all possible pairs $(o, T_o)$ identified by the typing rules. However, while `Proof` checks whether $d_e(o)$ is the set of marked edges required for $o$ to have type $T_o$, `Extract` attempts to extract from $\overline{s}_e(o)$ the set of marked edges $d_e(o)$ required for $o$ to have type $T_o$. In particular,

- in the case of an atomic type *int* or *string*, `Extract` simply checks if $o$ belongs to the corresponding domain. If so, the call returns an empty set of marked edges, otherwise returns $fail$;
- in the case of union types $T_1 + T_2$, `Extract` returns the marked edges required for $o$ to have type $T_1$ or, if this is not possible, type $T_2$. This is done by recursively invoking `Extract` with $o$ and $T_1$, and, if necessary, with $o$ and $T_2$;
- in the case of recursive types $\mu X.T$, the algorithm returns the set of marked edges required to type $o$ as the unfolding $T\left[\mu X.T/X\right]$.

Except for the case of atomic types, the cases of `Extract` discussed above dispatch the extraction of marked edges to further recursive calls. The only case that identifies edges from an SSDB and generates marked edges is the record type's. Here, the algorithm extracts the edges $ME_o$ emanating from $o$ that match the structural requirements imposed by the fields of the record type $T_o \equiv [l_1 : T_1, \ldots, l_n : T_n]$. If any of the fields cannot be matched, the call fails.

In particular, the call sequentially processes the fields from 1 to $n$, generating at each step $i$ the marked edges $ME_o$ required for $o$ to conform to the record type $[l_1 : T_1, \ldots, l_i : T_i]$. Indeed, $ME_o$ collects the marked edges generated to satisfy the structural properties of the fields $l_j : T_j$, with $j : 1, \ldots, i$. For each field $l_i : T_i$, the algorithm:

1. determines the set of *candidate edges* $\overline{s}_e(o, l_i)$;
2. *selects* in $\overline{s}_e(o, l_i)$ the edges that satisfy the structural requirements of $T_i$; in particular, the algorithm searches for one edge for each non-collection field and one or more edges for each collection field:
   - $T_i \neq_{\mathbf{T}} coll(U)$: checks if there is one candidate edge $< o, l_i, o' >$ whose target object $o'$ satisfies the structural requirements of the type $T_i$; this is done by recursively applying `Extract` to $o'$s and $T_i$ under the assumptions $A_{o'}$;
   - $T_i =_{\mathbf{T}} coll(U)$: checks if there are candidate edges $< o, l_i, o' >$ whose target object $o'$ satisfies the structural requirements of the type $U$; this is done by recursively applying `Extract` with $U$ to all $o'$s of candidate edges, under the assumptions $A_{o'}$; if no edge can be selected, `Extract` generates an empty collection edge $< o, l_i, c, \emptyset_U >$;
3. if the structural requirements of $T_i$ cannot be satisfied, the algorithm returns $fail$;
4. assume the structural requirements of $T_i$ can be satisfied, then:

- $T_i \neq_{\mathrm{T}} coll(U)$: one recursive call, corresponding to a selected edge $< o, l_i, o' >$, has successfully returned a set of marked edges $ME_i = ME_{o'}$; the algorithm generates from the selected edge the set of marked edges $\overline{ME}_i = \{< o, l_i, r, o' >\}$;
- $T_i =_{\mathrm{T}} coll(U)$: more than one recursive call, each corresponding to a selected edge $< o, l_i, o' >$, has successfully returned a set of marked edges $ME_{o'}$; this yields the set $ME_i = \bigcup_{\forall < o, l_i, o' >} ME_{o'}$; the algorithm generates the set $\overline{ME}_i = \bigcup_{\forall < o, l_i, o' >} \{< o, l_i, c, o' >\}$ of all marked edges extracted from $o$ for the field $l_i : T_i$;

note that the sets $ME_{o'}$ are the marked edges required to satisfy $A_{o'}$; $ME_{o'} \vdash o' :: T_i$ or $A_{o'}$; $ME_{o'} \vdash o' :: U$;

5. $ME_o$ contains all marked edges generated so far for the record fields $l_j : T_j$ with $j : 0, \ldots, i-1$, hence it is such that $A_o$; $ME_o \vdash o :: [l_1 : T_1, \ldots, l_i - 1 : T_i - 1]$; the algorithm adds to $ME_o$ all marked edges extracted for the current record field, that is the union between the marked edges $\overline{ME}_i$ generated for $l_i : T_i$ and the marked edges $ME_i$ generated by the corresponding subcalls. Accordingly, $A_o$; $ME_o \vdash o :: [l_1 : T_1, \ldots, l_i : T_i]$ is now a valid judgement.

If the structural requirements of all pairs $l_i : T_i$ of the record type can be fulfilled by $o$, the algorithm returns the set of marked edges

$$ME_o = (\bigcup_{i=1}^{n} \overline{ME}_i) \cup (\bigcup_{i=1}^{n} ME_i)$$

which verifies $A_o$; $ME_o \vdash o :: T_o$.

In conclusion, `Extraction`$(\overline{s}, \overline{T})$ invokes `Extract`$(\emptyset, \overline{s}_e, \overline{s}_r, \overline{T})$, which returns the set of marked edges $ME$ required to prove $\emptyset$; $ME \vdash \overline{s}_r :: \overline{T}$, and, by definition of typing, $(\overline{s}, ME) : \overline{T}$.

Consider again the example in Figure 7.2, where $\overline{T} \equiv_{\mathrm{T}} [person : coll([name : string])]$. The call `Extraction`$(\overline{s}, \overline{T})$ invokes `Extract`$(\emptyset, \overline{s}_e, o_1, \overline{T})$. Since the record type is of the form $[person : coll(U)]$, `Extract` searches for the subset of candidate edges $\overline{s}_e(o_1, person) \subseteq \overline{s}_e$. Before the algorithm can effectively mark these edges with $c$, it must verify that their target objects, namely $o_2$ and $o_3$, are in turn extractable according to the type $U$ of the collection values. To this aim `Extract` issues two recursive calls, to match $o_2$ and $o_3$ with $U$. These calls generate the sets $ME_{o_2} = \{< o_2, name, r, "Adam" >\}$ and $ME_{o_3} = \{< o_3, name, r, "Eve" >\}$. Therefore, the algorithm can generate the marked edges $< o_1, person, c, o_2 >$ and $< o_1, person, c, o_3 >$ and return $ME = ME_{o_2} \cup ME_{o_3} \cup \{< o_1, person, c, o_2 >, < o_1, person, c, o_3 >\}$. The resulting $d = (o_1, ME)$ is clearly of type $\overline{T}$.

### 7.2.2 Assumption sets

It is interesting to consider type checking in the case of *cyclic* values and values with *shared oids*. A value is *cyclic* if it contains an oid $o$ that is not trivially reachable from itself; an oid is instead *shared* if it is not the root and is reachable from the root by at least two paths $p_1$ and $p_2$, where $p_1$ is not an extension of $p_2$ and vice versa.

Consider the value $d$, the type $\overline{T}$, and the call tree $CT$ of the computation `Proof`($\emptyset$, $d_e$, $d_r$, $\overline{T}$). We know that in the generic call `Proof`($A$, $d_e$, $o$, $T$), the assumption set $A$ has the only purpose of avoiding infinite loops in correspondence with cyclic values and $\mu$-types. In particular, this is done by informing each invoked recursive call about the pairs $< o$, record type $T_o >$ visited so far in the call path. For the definition of call trees, paths and chains, see Section 6.6.1.

Similarly, `Extract` keeps track in $A$ of the pairs $< o$, record type $T_o >$ visited in the current call path. However, note that the *test for the termination of a call path* takes place within the record type case, rather than in an independent algorithm case, and entails a type equivalence check.

These differences are due to the fact that `Extract` does not simply check for the conformity of $o$ with respect to $T_o$, but must generate an $o$ that respects this property. This is not a trivial task in the presence of shared oids, where the algorithm may reach $o$ through different call paths and attempt to extract from it according to different record types; $o$ could be extractable according to different record types and, without a proper termination test, the algorithm would return an inconsistent set of marked edges.

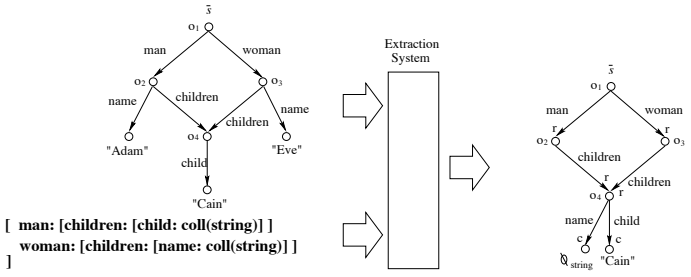



Figure 7.3: Example of wrong extraction due to loose termination test.

To be convinced of this, assume `Extraction` adopts the termination test of `Proof`, that is the test terminates an execution path when the pair $< o$, record type $T_o >$ is encountered for the second time. The application of `Extraction`

in the example shown in Figure 7.3 results in the value $d$ shown in the picture, which is not typable according to the input type. Indeed, the oid $o_4$ is visited by two independent call paths of the algorithm, respectively extracting according to the types $[name : coll(string)]$ and $[child : coll(string)]$. Both $\overline{s}_e(o_2, name)$ and $\overline{s}_e(o_2, child)$ contain edges that satisfy the structural requirements of these types, Hence, the algorithm terminates returning the value $d$. This result is not sound with respect to extractability since both $A; ME \vdash o_4 :: [name : coll(string)]$ and $A; ME \vdash o_4 :: [child : coll(string)]$ are not valid judgements.

This problem is solved by enriching the assumption set of `Extract` also with the pairs relative to the oids that have been extracted so far in the call chain. Thus, the generic assumption set $A$ may contain pairs $(o, [l_1 : T_1, \ldots, l_n : T_n])$ relative to two kinds of oids:

- *extractable oids*: the oids visited in the current call path: the algorithm is currently checking whether these oids can be extracted or not; their presence in $A$ is to avoid circular reasoning;
- *extracted oids*: the oids visited in the current call chain which are not into the current call path: these are the oids that have been successfully extracted so far; their presence in $A$ is to avoid the redundant visit of shared portions of the SSDB.

To ensure correctness, the *record case* of the algorithm should first check whether the input $o$, to be extracted according to $T_o$, is an extractable or an extracted oid in a pair $(o, T'_o)$ of $A$. In this case, if $T_o =_{\mathrm{T}} T'_o$, the algorithm returns an empty set of marked edges; this is because the call that had first visited $o$, and added $(o, T'_o)$ to $A$, is in charge of the generation of the correspondent marked edges. If $T'_o \neq_{\mathrm{T}} T_o$ the algorithm returns $fail$, as the call was trying to extract $o$ according to a different record type.

Thus, when the extraction algorithm successfully terminates, we know that all marked edges relative to an oid $o$ were generated by only one record case call according to $T_o$, that is $o$ conforms to $T_o$.

In particular, assumption sets are enriched by record type case calls. Consider the record case call $\texttt{Extract}(A, E, o, T_o)$. The node in the call tree corresponding to this call may have the following, ordered, set of children:

$$\texttt{Extract}(A_1, E, o_1, T_{o_1}) \ldots, \texttt{Extract}(A_n, E, o_n, T_{o_n})$$

First, the algorithm ensures that for all $i : 1, \ldots, n. A \cup (o, T_o) \subseteq A_i$. This informs the call $\texttt{Extract}(A_i, E, o_i, T_{o_i})$ about all extractable oids visited so far in the call path. Call paths originating from $\texttt{Extract}(A_i, E, o_i, T_{o_i})$ can thus terminate when an extractable oid is encountered.

Moreover, the successful execution of the call $\texttt{Extract}(A_i, E, o_i, T_{o_i})$, returns a pair $(ME_i, \Delta A_i)$. The set $\Delta A_i$ contains the pairs $(o', T_{o'})$ relative to the successful

record case calls that contributed to the generation of the set $ME_i$. The assumption set relative to the next call in the call chain, $\texttt{Extract}(A_{i+1}, E, o_{i+1}, T_{o_{i+1}})$, is thus enriched with the extracted oids in $\Delta A_i$. Call paths originating from such call can thus terminate when an extracted oid is encountered.

Finally, if the record case call is successful, the algorithm returns $(\Delta A, ME)$, where

$$\Delta A = (o, T_o) \cup \bigcup_{i=1}^{n} \Delta A_i.$$

This set can be passed to further extractions to ensure the consistency of typing.

As an example of use of the correct termination test, consider again the extraction in Figure 7.3. The call relative to the visit of $o_1$ with the type $[man : T_1, woman : T_2]$, first issues the subcall,

$$\texttt{Extract}(\{(o_1, [man : T_1, woman : T_2])\}, \overline{s}_e, o_2, T_1)$$

relative to the only candidate edge $< o_1, man, o_2 >$. This call successfully terminates, and returns the pair $(\Delta A_{o_2}, ME_{o_2})$,

$$\Delta A_{o_2} = \{(o_2, T_1), (o_4, [child : coll(string)])\}$$

$$ME_{o_2} = \{< o_2, children, r, o_4 >, < o_4, child, c, \text{“}Cain\text{”} >\}$$

Hence, the algorithm issues the call

$$\texttt{Extract}(A_{o_3}, \overline{s}_e, o_3, T_2)$$

relative to the only candidate edge $< o_1, woman, o_3 >$, where $A_{o_3} = \{(o_1, [man : T_1, woman : T_2])\} \cup A_{o_2}$. Recursively, this call will issue a record case call

$$\texttt{Extract}(A_{o_3} \cup \{(o_3, T_2)\}, \overline{s}_e, o_4, [child : coll(string)])$$

This call will fail, hence the overall extraction, because $o_4$ is currently an extracted value associated with a type $[child : coll(string)]$ that is not equivalent to $[name : coll(string)]$.

### 7.2.3 Reading the algorithm

To be able to better read the code of the algorithm, it is worth pointing out the following observations. Consider the generic call

$$\texttt{Extract}(A, E, o, T) = (\Delta A, ME)$$

where $ME \neq fail$, then:

- The objects *potentially reachable* by the call chain originating from this call are those reachable from $o$ with edges in $E$. In particular, the termination test excludes from this set the objects that are reachable from $o$ only passing through oids $\overline{o}$ such that there exists $(\overline{o}, \overline{T}) \in A$. The set $O_{A,o,E}$ of the objects *potentially extractable* by this call, is the set of the objects potentially reachable, deprived of the oids appearing in the pairs of $A$. Indeed, we have shown that such oids are to be extracted by the call that first inserted them into the assumption set.

- $\Delta A$ contains the pairs relative to those record case calls which effectively contributed in the generation of the marked edges in $ME$. Specifically, if $(\overline{o}, \overline{T})$ is in $\Delta A$, then $\overline{o} \in O_{A,o,s_e}$ and $\overline{o} \notin A$. Moreover, $\forall(\overline{o}, \overline{T}) \in \Delta A$ the call chain originating from this call contains a successful call

$$\texttt{Extract}(A_{\overline{o}}, E, \overline{o}, \overline{T}) = (\Delta A_{\overline{o}}, ME_{\overline{o}})$$

  such that $(\overline{o}, \overline{T}) \notin A_{\overline{o}}$, $ME_{\overline{o}} = ME(o)$ and $A_{\overline{o}} \subseteq \Delta A$.

- $\forall(\overline{o}, \overline{T}) \in \Delta A$. $ME(\overline{o}) \neq \emptyset$. This is because typing for records always requires at least one marked edge and `Extract` must generate edges accordingly.

- If $me \in ME$ then it can be either that there exists $e \in E$ such that $e = <\overleftarrow{me}, label(me), \overrightarrow{me}>$ or that $\overrightarrow{me} = \emptyset_U$ and $E(\overleftarrow{me}, label(me)) = \emptyset$;

- `Proof`$(A, ME, o, T)$ returns a valid derivation tree. Observe that, however, by dropping $A$ the algorithm may fail. This is not the case for judgements derived from `Proof`, where the assumption set can be generated from any point of the proof, and has only termination purposes. Indeed, the judgement corresponding to a successful application of `Extract`, generally features only a subset of the marked edges required to prove its validity. The rest of the marked edges is represented by pairs $(\overline{o}, \overline{T})$ relative to extracted and extractable oids. Only in a later stage will the recursive unfolding replace these pairs with the corresponding sets of marked edges and unify them with $ME$.

## 7.3 Termination

The termination of `Extraction` strictly depends on the termination of `Extract`. The latter can be proven by observing that, thanks to the assumption sets, all call paths relative to a computation `Extract`$(A, E, o, T)$ must be finite.

**Lemma 7.3.1** *If* `Extract`$(A_0, E, o_0, T_0), \ldots,$ `Extract`$(A_i, E, o_i, T_i), \ldots$ *is a call path of the call tree of* `Extract`$(A_0, E, o_0, T_0)$*, then,*

$$A_0 \subseteq A_1 \subseteq \ldots \subseteq A_i \subseteq \ldots$$

**Proof.** Call $i+1$ occurs at a deeper level in the path than call $i$. Before invoking a subcall $i+1$ the assumption set can only be expanded or left untouched, hence $A_i \subseteq A_{i+1}$. Indeed, the proof follows directly from the observation that during the execution of `Extract` the assumption set can only be expanded and never contracted. ∎

**Theorem 7.3.2** *(Termination of* `Extract`*) Given* $T \in \mathbf{T}$, $A \subseteq \mathcal{P}_{fin}(Oid \times \mathbf{T})$, $s \in S$ *and* $o \in Obj(s)$, *the call* `Extract`$(A,\ s_e,\ o,\ T)$ *terminates in a finite number of steps.*

**Proof.** We prove this statement by contradiction, assuming `Extract`$(A, s_e, o, T)$ does not terminate. If this is the case, an infinite call path $p$ originates from the call tree associated with this call. As by definition of $S$ we know that $\mid Oid(s) \mid < \infty$, we can infer that there exists an oid $\overline{o} \in Oid(s)$ visited infinite times by the calls in $p$.

By definition of `Extract`, the input types $T'$ of the calls in $p$ are all such that $T' \sqsubseteq T$. As $p$ is infinite, $\overline{o}$ is visited infinite times, and $T$ is canonical, there exists $[l_1 : T_1, \ldots, l_n : T_n] \sqsubseteq T$, such that `Extract`$(\overline{A},\ s_e,\ \overline{o},\ [l_1 : T_1, \ldots, l_n : T_n])$ is in $p$. By definition of `Extract` and by Lemma 7.3.1, we know that the assumptions sets relative to the calls following this one in $p$ will contain $\overline{A} \cup \{(\overline{o}, [l_1 : T_1, \ldots, l_n : T_n])\}$.

As $[l_1 : T_1, \ldots, l_n : T_n]$ is canonical too, for the same reasons, there must exists a further call `Extract`$(\overline{A}',\ s_e,\ \overline{o},\ [l'_1 : T'_1, \ldots, l'_n : T'_n])$ in $p$. Since $\overline{A} \cup \{(\overline{o}, [l_1 : T_1, \ldots, l_n : T_n])\} \subseteq \overline{A}'$, by definition of `Extract` this call terminates. Its result would be either $(\emptyset, \emptyset)$, if $[l_1 : T_1, \ldots, l_n : T_n] =_{\mathbf{T}} [l'_1 : T'_1, \ldots, l'_n : T'_n]$, or $(\emptyset, \mathit{fail})$ otherwise. Therefore, $p$ cannot be infinite and `Extract` terminates in a finite number of steps. ∎

**Theorem 7.3.3** *(Termination of* `Extraction`*) Let* $T \in \mathbf{T}$ *and* $s \in S$. *The call* `Extraction`$(s,\ T)$ *terminates in a finite number of steps.*

**Proof.** Directly from theorem 7.3.2, as `Extraction`$(s,\ T)$ depends on the result of the call `Extract`$(\emptyset,\ s_e,\ s_r,\ T)$. ∎

## 7.4 Relevance

When defining a type $\overline{T}$, so as to extract interesting data from an SSDB $\overline{s}$, the user may not be aware of the exact overall structure of $\overline{s}$. Indeed, we can generally assume that $\overline{T}$ is defined after an eye-inspection of $\overline{s}$ or, if available, of a representation of its structure.

As a consequence, before writing long-life applications over values of type $\overline{T}$ one would be able to make sure that all interesting data in $\overline{s}$ are embraced by the extraction according to $\overline{T}$. The same requirement surfaces when long-life applications are already available and running over extracted values of type $\overline{T}$. As discussed in

Chapter 4, extraction may be repeated after long intervals of time to synchronise with the rare but possible updates of $\overline{s}$. It may well be the case that some of these updates were interesting to the user but do not fall in the subset of $\overline{s}$ captured by $\overline{T}$. Therefore, the extant applications would run over obsolete extracted data.

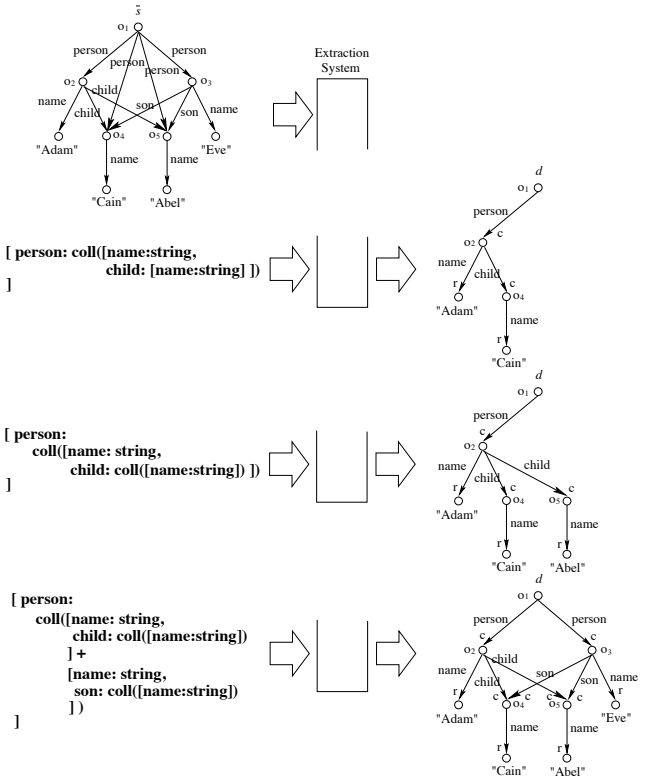



Figure 7.4: Example of extraction with a low-relevance type

Consider the picture in Figure 7.4, where the extraction system is passed the SSDB $\overline{s}$. Assume the user is interested to compute over the subset of $\overline{s}$ involving *persons* with a *name* and a *child*, which in turn has a *name*. From an initial analysis

of the collection the user comes up with the type,

$$[person : coll([name : string, child : [name : string]])]$$

shown in the first extraction in the picture. Observing the resulting value $d$, we note that $\overline{s}$ contains more data that may be potentially interesting to the user. For example, differently from the user's request, an element of the collection *person* may feature more than one *child*; moreover, there are other subsets of $\overline{s}$ that semantically correspond to elements of *persons* but are discarded because their label is *son* rather than *child*.

These sort of misjudgments may be very frequent, as we aim at dealing with the general scenario of SSDBs featuring a large number of paths, inserted at different times, possibly by different users with a different cognition of data representation. Therefore, after extraction, our algorithm provides the user with information relative to the degree of *relevance* of his current typing, with the specific purpose of minimising the misjudgments exemplified above.

We measure relevance in terms of *data capturing* and *precision*.

Data capturing is simply the ratio between the edges effectively extracted from $\overline{s}$ and the edges in $\overline{s}$.

**Definition 7.4.1** *(Data capturing) Let $\overline{T} \in \mathbf{T}$ and $\overline{s} \in S$. Given the successful call,*

$$\texttt{Extraction}(\overline{s}, \overline{T}) = d$$

data capturing *of this extraction is calculated as:*

$$dataCapt(\overline{s}, d) = \frac{\mid Erase(d_e) \mid}{\mid \overline{s}_e \mid}$$

This measure may help the user, who knows the size of $\overline{s}$, to figure out whether our query methodology is convenient in a given application context: a low level of data capturing may suggest the existence of an extremely small regular core, which may be better handled by SSDQLs as those presented in Chapter 2.

By *lost information* for $o$ we mean the edges in $\overline{s}(o)$ that have not been extracted due to the structural constraints of the extraction type $\overline{T}$, but were regarded as candidate for extraction by `Extract`. Consider again the first extraction in Figure 7.4: the edges $< o_2, child, o_5 >$, $< o_1, person, o_3 >$, $< o_1, person, o_4 >$ and $< o_1, person, o_5 >$ fall in this category. Indeed, the first edge was a possible candidate for `Extract`, because $< o_2, child, o_5 > \in \overline{s}(o_2)_{child}$; however, it was discarded because the first edge considered by `Extract` was $< o_2, child, o_4 >$, which fully satisfied the structural requirements of the record field *child*.

Precision of extraction measures how well a record type $\overline{T}$ minimises the quantity of lost information for a specific oid $o \in Oid(\overline{s})$. In particular, precision is the ratio between the number of edges outgoing $o$ correctly extracted by the algorithm to

satisfy the typing for $\overline{T}$, and the number of edges outgoing $o$ that were candidate for that extraction. The overall precision of a record type is calculated in terms of two separate forms of precision, *non-collection field precision* and *collection field precision*:

**Non-collection field precision:** it is related with the extraction of edges relative to record fields that are not associated to collection types. This is the case for *child* in the first extraction exemplified in Figure 7.4. The structural requirements of a non-collection record field entail the extraction of the first edge satisfying the typing, in the example $< o_2, child, o_4 >$. However, there is another edge labelled as *child*, which is therefore likely to be relevant for the user. The system associates with this field a precision of $\frac{1}{2}$, meaning that, due to structural constraints, the extraction returned only *one* edge out of *two* interesting edges.

The user, by observing that this measure is related with a non-collection record field, may try to improve precision of his extraction by *adding a collection type* to $\overline{T}$, thereby turning it into,

$$[person : coll([name : string, child : coll([name : string])])]$$

which is the type relative to the second extraction illustrated in Figure 7.4.

**Collection field precision:** it is related with the extraction of edges relative to record fields that are associated to collection types. This is the case for *person* in the second extraction exemplified in Figure 7.4. The structural requirements of a collection record field entail the extraction of all edges satisfying the typing, in the example $< o_1, person, o_2 >$. However, there are other edges outgoing $o$ labelled as *person*, which are therefore likely to be relevant for the user. The system associates with this field the precision $\frac{1}{4}$, meaning that, due to structural constraints, the extraction returned only *one* edge out of *four* potentially interesting edges. The user, by observing that this measure is related with a collection record field, may try to improve precision of his extraction by *adding a union type* to $\overline{T}$, thereby turning it into,

$$\begin{array}{l}[person : coll([name : string, child : coll([name : string])]+ \\ \qquad\qquad [name : string, son : coll([name : string])]) \\ ]\end{array}$$

which is the type relative to the third extraction illustrated in Figure 7.4.

Given $d \in D$ extracted from $\overline{s} \in S$ according to $\overline{T} \in \mathbf{T}$, we can formally give the following definitions:

**Definition 7.4.2** *(Non-collection field precision) Let* $T \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n] \sqsubseteq \overline{T}$ *and* $o \in Oid(d)$*. For all field* $l_i$ *such that* $T_i \not\equiv coll(U)$*, the* non-collection field precision *in* $s$ *for* $l_i$ *and* $o$ *is:*

$$ncprec_s(o, l_i) = \frac{1}{\mid s_e(o, l_i) \mid}$$

**Definition 7.4.3** *(Collection field precision) Let* $T \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n] \sqsubseteq \overline{T}$ *and* $o \in Oid(d)$*. For all field* $l_i$ *such that* $T_i \equiv_{\mathbf{T}} coll(U)$*, the* collection field precision *in* $s$ *for* $l_i$ *and* $o$ *is:*

$$cprec_{s,d}(o, l_i) = \begin{cases} \frac{|d_e(o, l_i)|}{|s_e(o, l_i)|} & \mid s_e(o, l_i) \mid \neq 0 \\ 1 & \mid s_e(o, l_i) \mid = 0 \end{cases}$$

*Observe that the precision of a collection field with respect to an oid* $o$ *is certainly* 1 *if its extraction involved an empty collection.*

Finally, we are able to provide a measure of the total precision of a record type with respect to a given oid. This measure is obtained in terms of the measures of precision for the individual fields of the record type.

**Definition 7.4.4** *(Record precision) Let* $T \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n]$ *in* $\overline{T}$ *and* $o$ *an oid in* $s'$ *that has been extracted according to* $T$*. The record loss for* $\overline{T}$ *in* $s$ *with* $o$ *is:*

$$rprec_{s,d}(o, T) = \frac{1}{n} \cdot \left( \sum_{i=1, T_i \not\equiv_{\mathbf{T}} coll(U)}^{n} ncprec_s(o,\ l_i) \ + \sum_{i=1, T_i \equiv_{\mathbf{T}} coll(U)}^{n} cprec_{s,d}(o,\ l_i) \right)$$

This measure provides minimal information about every single extraction of an oid with respect to a record type. If properly combined, these precisions may provide extremely interesting information to the user.

For example, each record type $T$ appearing in an input type $\overline{T}$ could be associated with a *total precision*. That is the ratio between the sum of the record precisions for oids in $d$ conforming to $T$, and the number of oids in $d$ conforming to $T$.

$$totrprec_{s,d}(T) = \frac{\sum_{o \in Oid(d).\ \alpha :: T} rprec_{s,d}(o, T)}{\mid \{o \in Oid(d) \mid o :: T\} \mid}$$

However, different measures of precision could be conceived, such as providing weights for different fields, or simply returning the total sum of the individual losses for each record, or calculating the loss of an extraction with respect to a type in terms of the loss of the related subextractions.

It is interesting to note that in practice relevance will be available to programs. Hence, its first use may be by the programs themselves, for example providing some thresholds.

With reference to *rprec*, after a successful extraction of $d$ from $\overline{s}$ and $\overline{T}$, the user is presented the list of the pairs $(o, T)$ where $T \sqsubseteq \overline{T}$ is a record type. Each record type comes along with its precision, as well as the precision of the single fields. These measures draw a picture of the extraction process that may help the user to improve its extraction.

For instance, the first extraction shown above produces the following measures of precision:

| *Record type* | *Oid* | *Fields Precision* | *Total Precision* |
|---|---|---|---|
| $[person : coll(T_1)]$ | $o_1$ | $(person, \frac{1}{4})$ | $\frac{1}{4}$ |
| $[name : string, child : T_2]$ | $o_2$ | $(name, 1), (child, \frac{1}{2})$ | $\frac{3}{4}$ |
| $[name : string]$ | $o_4$ | $(name, 1)$ | 1 |

The relatively low total precision of the first two record types suggests to explore the precision of the related fields. Note that, as a general rule, a low non-collection field precision suggests the addition of a collection type, while a low collection field precision suggests the introduction of a union type. Following this reasoning, we reach the third extraction exemplified in Figure 7.4. The measures of precision for that extraction are:

| *Record type* | *Oid* | *Fields Precision* | *Total Precision* |
|---|---|---|---|
| $[person : coll(T_1)]$ | $o_1$ | $(person, \frac{1}{2})$ | $\frac{1}{2}$ |
| $[name : string, child : T_2]$ | $o_2$ | $(name, 1), (child, 1)$ | 1 |
| $[name : string, son : T_2]$ | $o_3$ | $(name, 1), (son, 1)$ | 1 |
| $[name : string]$ | $o_4$ | $(name, 1)$ | 1 |
| $[name : string]$ | $o_5$ | $(name, 1)$ | 1 |

The developers or the programs may be satisfied with an extraction even if it does not report a precision 1.

## 7.5 Cost

An execution of `Extraction` may potentially require the repeated traversal of a whole SSDB. This undesirable feature is quite typical in semistructured data research, where applications are often faced with the problem of the traversal of a tree or a graph. Generally, however, these applications show extremely bad performances

for worst-case scenarios and are well-behaved in real scenarios. In other words, an application's worst case complexity has not the same relevance as the experimental results.

In this Section, we show the potentially exponential nature of `Extraction`, in order to provide the reader with a better understanding of the algorithm's behaviour. We shall measure cost in terms of the size of the SSDB $\overline{s}$ to be traversed by `Extract`, i.e. in relation to the number of edges in $\overline{s}$. We shall see that the algorithm entails exponential execution costs when applied to specific, simple types and *tree-structured SSDBs*. However, we shall also debate the general practical value of these kind of calculations and claim that the cost for a real application extraction is averagely linear on the number of edges.

Due to the structural mismatch between record fields and edges emanating from oids, the average execution of `Extract` may not traverse arbitrarily large subsets of the tree-structured $\overline{s}$. Accordingly, if no call of `Extract` fails, the worst case is when the input $\overline{s}$ and $\overline{T}$ are such that $\overline{s}$ is in a one-to-one mapping with a value $d$ of type $\overline{T}$. The cost of this extraction is that of a depth-first visit of the SSDB graph, namely $O(m)$ where $m = \mid \overline{s}_e \mid$ is the number of edges in $\overline{s}$.

Due to the failure of specific call paths, however, the same edges may be visited more than once. By observing `Extract`, this may happen when a call issued by a union type case `Extract`$(A, E, o, T_1 + T_2)$ fails. Indeed, the failure of the call `Extract`$(A, E, o, T_1)$ causes the invocation of the call `Extract`$(A, E, o, T_2)$, which may traverse again the edges outgoing $o$ and the edges reachable from $o$. Note that the same problem may arise when selecting candidate edges for a record field.

In the following, we show the exponential behaviour of the algorithm when applied to a union type and a specific SSDB, which maximises the number of failures and visit of edges. In particular, we apply `Extract` to the SSDB $\overline{s}$ with $n$ levels and fan-out $f = 1$ illustrated in Figure 7.5, and the $\mu$-type:

$$T \equiv_{\mathrm{T}} \mu X.U,$$

where $U = T_1 + \ldots + T_k$ and $T_i = [a : X]$ for $i : 1, \ldots, k$. In its execution, `Extract` visits all oids in $o \in Oid(\overline{s})$ with a union type $T_1 + \ldots + T_k$, and fails after having tried the extraction of $o$ according to all members $T_i$. In particular, each record case call with $o$ and $[a : T]$, invoked by a union case, causes the visit of the whole path from $o$ to *ciao* in $\overline{s}_e$ and fails only when visiting that string. This failure propagates back again to the union case call, which invokes an identical record case call relative to the next member of the union. Intuitively, the same path may be thus visited an exponential number of times. We give a formal proof of this in the following proposition.

**Proposition 7.5.1** *The execution cost of* `Extraction`$(\overline{s}, U)$ *is* $O(k^m)$.

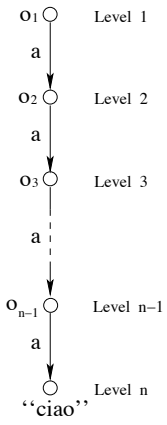



Figure 7.5: Worst case SSDB and type

**Proof.** First, note that if we consider the root of $\overline{s}$ at level 1, we have that

$$m = \left|\, \overline{s}_e \,\right| = n - 1$$

As the cost of `Extraction` is the cost of `Extract`, we prove the statement for the latter, by showing that the number of edges visited by the call `Extract`$(A, \overline{s}_e, \overline{s}_r, T)$ is $k^n$, i.e. $k^m$.

This statement can be proven by induction on the number $i$ of levels of $\overline{s}$ to be traversed by the generic union case call `Extract`$(A, E, o_{n-i}, U\left[{}^{T}/_{X}\right])$, which we shall denote as `Extract`$_{n-i}$.

$\boxed{i=1}$ `Extract`$_{n-1}$ issues a call `Extract`$(A, E, o, [a : T])$ in order to extract from $o_{n-1}$ according to the first member of the union type. The only candidate edge for the label $a$ is $< o_{n-1}, a,$ *"ciao"* $>$. Accordingly, the call will invoke a further $\mu$-type case call with *"ciao"* and fail, because *"ciao"* is not an oid as required by $\mu$-types.

This failure propagates back to `Extract`$_{n-1}$, which will try the extraction of $o_{n-1}$ with the second member of the union. As all members of the union will fail, this process is repeated $k$ times, then the union case call fails. Each time the union case call tries to extract according to one of its members, the edge $< o_{n-1}, a,$ *"ciao"* $>$ is visited once. Hence, the call visits the edge a total of $k^1 = k$ times.

$\boxed{i>1}$ By IH we know that all calls `Extract`$_{n-j}$, with $j : 1, \ldots, i-1$, fail after visiting $k^j$ edges.

The call $\texttt{Extract}_{n-i}$ issues a call $\texttt{Extract}(A,\ E,\ o_{n-i},\ [a:T])$ in order to extract from $o_{n-i}$ according to the first member of the union type. Such a call identifies the only candidate edge outgoing $o_{n-i}$, namely $< o_{n-i}, a, o_{n-i+1} >$, and issues the call $\texttt{Extract}(A,\ E,\ o_{n-i+1},\ T) = \texttt{Extract}_{n-i+1}$. The latter is a union case call at level $i-1$. By IH we know that this call traverses $k^{i-1}$ edges before failing. This failure propagates back to $\texttt{Extract}_{n-i}$, which attempts the extraction of $< o_{n-i}, a, o_{n-i+1} >$ with the second member of the union type. This process is repeated $k$ times, then $\texttt{Extract}_{n-i}$ fails. The candidate edge $< o_{n-1}, a, o_{n-i+1} >$ is visited $k$ times. In addition, each of this visits caused the visit of $k^{i-1}$ edges, for a total of $k^i$ edges visited.

In conclusion, extracting an SSDB as simple as $\bar{s}$ with a type as simple as $T$, entails the visit of $k^n$ edges. Since $m = n - 1$ we can state that the execution cost for `Extraction` is $O(k^m)$.

■

Note that the scenario we constructed represents a particularly extreme situation. For $k = 2$ and $m = 30$, for example, the total of edges visited by the algorithm would be close to a billion.

In real applications, we expect the user to extract by means of a reasonable number of union types, not to fall in the category of non optimal types. Besides, only a portion of the oids in the collection would be extracted according to them, and we do not expect all edges reachable from such oids to satisfy all members of a union type. Furthermore, due to the mismatch between the labels of the record fields and the labels of the edges, portions of the SSDB may not be traversed at all.

Consider the extraction of a generic SSDB with $m$ edges with the type:

$$[Dept : [Emp : coll([Name : string + [FirstName, SecondName : string])]].$$

This type cannot possibly give rise to an explosion of complexity. The algorithm will not visit the same edges twice, as the members of the union type are different and there are no objects in the SSDB that may match both. For this quite common scenario, the algorithm may traverse at most all $m$ edges.

# Chapter 8

# Extraction Algorithm Correctness

In this Chapter we prove correctness of the algorithm `Extraction` with respect to extractability of $L$ as defined in Chapter 7. In particular, according to the definition of correctness in Chapter 4, we shall prove:

**Soundness:** every successful execution $\texttt{Extraction}(\overline{s}, \overline{T}) = d$ is such that $d$ is *extractable* from $\overline{s}$ according to $\overline{T}$;

**Completeness:** if the set of values $D_{\overline{s},\overline{T}} \subseteq D$ which are *extractable* from $\overline{s}$ according to $T$ is not empty, the call $\texttt{Extraction}(\overline{s}, \overline{T})$ is successful and returns $d \in D_{\overline{s},\overline{T}}$.

We shall show that `Extraction` is sound with respect to extractability, but not generally complete. Indeed, the algorithm is complete whenever its application is restricted to tree-structured SSDBs. We shall also observe, however, that incompleteness is typically due to particularly critical scenarios, whose exceptionality does not generally compromise the usability of `Extraction` to all SSDBs. Furthermore, on the basis of the examples of incompleteness, we shall suggest how `Extraction` could be modified in order to decrease its degree of incompleteness.

## 8.1 Soundness

In this Section we prove the following soundness theorem.

**Theorem 8.1.1** *(Soundness of* `Extraction`*) Let* $\overline{s} \in S$ *and* $\overline{T} \in \mathbf{T}$. *Every successful execution of the algorithm* $\texttt{Extraction}(s, \overline{T}) = d$ *is such that* $ssd(d) < \overline{s}$ *and* $d : \overline{T}$.

To this aim we require the notion of *success tree* of `Extract`, which is the tree obtained by the call tree eliminating the nodes relative to failing calls and all nodes which are reachable from these. Formally,

**Definition 8.1.2** *(Failing node) A* failing node *of a call tree is a node corresponding to a failing call, i.e. a call of* `Extract` *that returns a pair* $(\emptyset, fail)$.

**Definition 8.1.3** *(Success tree of* `Extract`*) Let* $\overline{s} \in S$, $\overline{T} \in \mathbf{T}$, `Extraction`$(\overline{s},\overline{T}) = d$ *be a successful execution of the algorithm, and* $CT$(`Extract`$(\emptyset,\ \overline{s}_r,\ \overline{s}_e,\ \overline{T})$) *be the call tree corresponding to such execution.*

*The* success tree $ST$(`Extract`$(\emptyset,\ \overline{s}_r,\ \overline{s}_e,\ \overline{T})$) *of this execution is the tree obtained from the call tree by dropping all failing nodes and all nodes reachable with a call path originating from a failing node. We denote as* $ST_N$ *the nodes in the success tree* $ST$.

Furthermore, in the following we shall denote as $\overline{ME}$ the set of marked edges generated by a record case call.

**Definition 8.1.4** *(Marked edges generated by a record case call) Let* $ST$ *be a success tree,* $s \in S$, $o \in Oid(s)$, $A$ *an assumption set,* $T \equiv [l_1 : T_1, \ldots, l_n : T_n] \in \mathbf{T}$. *The node* `Extract`$(A,\ s_e,\ o,\ T) \in ST_N$ *has a finite set of children,*

$$Sub = \{\texttt{Extract}(A_1,\ s_e,\ o_1,\ T'_1) \ldots, \texttt{Extract}(A_k,\ s_e,\ o_k,\ T'_k)\}$$

*such that* $k \geq 0$. *By definition of* `Extract`, *we know that each of these calls was invoked after identifying a corresponding candidate edge* $< o, l, o_i >$. *The* set of marked edges generated by `Extract`$(A,\ E,\ o,\ T)$ *is the set*

$$\overline{ME} = (\bigcup_{i=1}^{k} \{< o, l, m, o_i >\}) \cup (\bigcup_{T_i \equiv coll(U) \wedge s(o,l_i)=\emptyset} \{< o, l_i, c, \emptyset_U >\})$$

*obtained by appropriately decorating the corresponding selected edges and adding the empty collection edges where necessary.* `Extract`$(A,\ E,\ o,\ T)$ *will return the set of marked edges*

$$ME = \overline{ME} \cup \bigcup_{i=1}^{k} ME_i$$

*where* $ME_i$ *is the set of marked edges returned by* `Extract`$(A_i,\ E,\ o_i,\ T'_i)$.

### 8.1.1 Soundness of inclusion

**Lemma 8.1.5** *Let* $A$ *be an assumption set,* $E$ *a set of edges,* $o \in Obj$, $T \in \mathbf{T}$, *and* $ST$ *the success tree of a call of* `Extract`. *If,*

$$\texttt{Extract}(A,\ E,\ o,\ T) = (\Delta A,\ ME).$$

*and,*

$$\texttt{Extract}(A,\, E,\, o,\, T) \in ST_N$$

*then,*

$$ssd((o,\, ME)) < (o,\, E)$$

**Proof.** We prove this statement by induction on the finite depth $i$ of the success tree $ST(\texttt{Extract}(A,\, E,\, o,\, T))$ and adopting the identity $id$ as the morphism required to prove lt-inclusion.

$\boxed{i = 0}$

**Case** $\texttt{Extract}(A \cup \{(o, T)\},\, E,\, o,\, T)$**:** the call corresponding to this node returns the pair $(\emptyset,\, \emptyset)$; in general for all set of edges $E$, $(o,\, \emptyset) < (o,\, E)$ as $ssd((o,\, \emptyset)) = (o,\, \emptyset)$ and $id(o) = o$;

**Case** $\texttt{Extract}(A,\, E,\, o,\, int)$**:** the same as for the case above;

**Case** $\texttt{Extract}(A,\, E,\, o,\, string)$**:** the same as for the case above.

**Case** $\texttt{Extract}(A,\, E,\, o,\, [l_1 : T_1, \ldots, l_n : T_n]) = (\Delta A,\, ME)$**:** where

$$\Delta A = \{(o, [l_1 : T_1, \ldots, l_n : T_n])\} \text{ and } ME = \bigcup_{i=1}^{n} \{< o, l_i, c, \emptyset_{U_i} >\};$$

this case occurs when

$$\forall i : 1 \ldots n.\, (T_i \equiv_{\mathrm{T}} coll(U_i)) \;\wedge\; (\mid E(l_i) \mid = 0).$$

Note that $Erase(\bigcup_{i=1}^{n} \{< o, l_i, c, \emptyset_{U_i} >\} = \emptyset$, hence the proof is trivial as above.

$\boxed{i > 0}$ We assume that the thesis holds for all nodes of $ST$ with depth less than $i$. Since $i > 0$, by definition of $ST$, for all $\texttt{Extract}(A,\, E,\, o,\, T) \in ST_N$ there exists a finite set of children,

$$Sub = \{\texttt{Extract}(A_1,\, E,\, o_1,\, T_1) \ldots, \texttt{Extract}(A_k,\, E,\, o_k,\, T_k)\}$$

such that $k \geq 1$ and their depth is $i - 1$. Furthermore, the call corresponding to the node $\texttt{Extract}(A,\, E,\, o,\, T)$ returns a pair $(\Delta A,\, ME)$ and the calls corresponding to the set of children above return pairs $(\Delta A_j,\, ME_j)$ with $1 \leq j \leq k$. By observing the algorithm we notice that for all $j : 1, \ldots, k$, $ME_j \subseteq ME$.

When $T \equiv_{\mathbf{T}} T_1 + T_2$ or $T \equiv_{\mathbf{T}} \mu X.T$, then $k = 1$ and $ME_1 = ME$. As in both situations it also holds that $o = o_1$, the thesis can be inferred directly from the IH $ssd((o_1, ME_1)) < (o_1, E)$.

When $T \equiv_{\mathbf{T}} [l_1 : T'_1, \ldots, l_n : T'_n]$, by observing the algorithm, we find that $ME = \overline{ME} \cup \bigcup_{j=1}^{k} ME_j$. By IH we know that

$$\forall j : 1, \ldots, k.\, ssd(o_j, ME_j)) = (o_j, Erase(ME_j)) < (o_j, E)$$

hence, in order to prove our thesis, we remain to prove that,

$$\begin{array}{l} \forall e \in Erase(\overline{ME}). \\ \quad \vec{e} \in Oid \;\Rightarrow\; \exists e' \in E.\, (id(\overleftarrow{e}) = \overleftarrow{e'} \;\wedge\; id(\vec{e}) = \vec{e'} \;\wedge\; label(e) = label(e')) \\ \quad \vec{e} \in Atomic \;\Rightarrow\; \exists e' \in E.\, (id(\overleftarrow{e}) = \overleftarrow{e'} \;\wedge\; \vec{e} = \vec{e'} \;\wedge\; label(e) = label(e')) \end{array}$$

Applying identity $id$ as the morphism to prove inclusion, this becomes,

$$\forall e \in Erase(\overline{ME}).\, \exists e' \in E.\, (\overleftarrow{e} = \overleftarrow{e'} \;\wedge\; \vec{e} = \vec{e'} \;\wedge\; label(e) = label(e'))$$

that is,

$$\forall e \in Erase(\overline{ME}).\, e \in E$$

This can be trivially proven by simply observing the algorithm. Indeed, $\forall me \in \overline{ME}$ with $\vec{me} \in Obj$ ($\vec{me} \notin EC$, i.e. is not an empty collection value), there exists $\texttt{Extract}(A_j,\ E,\ o_j,\ T_j)$, with $j : 1, \ldots, k$ such that

$$\begin{array}{l} \exists i : 1, \ldots, n. \\ \quad (< o, l_i, o_j > \in E(o, l_i) \wedge\; \overleftarrow{me} = o \;\wedge\; label(me) = l_i \wedge\; \vec{me} = o_j) \end{array}$$

From these conditions we can derive that

$$\forall me \in \overline{ME}.\, (\vec{me} \in Obj \Rightarrow\; \exists e \in E.\, erase(me) = e)$$

which implies, by definition of $Erase$,

$$\forall e \in Erase(\overline{ME}).\, e \in E$$

Therefore,

$$ssd((o, ME)) = (o, Erase(\overline{ME}) \cup \bigcup_{j=1}^{k} Erase(ME_j))) < (o, E)$$

■

**Theorem 8.1.6** *(Soundness of inclusion) Let $\overline{s} \in S$, $\overline{T} \in \mathbf{T}$, $\texttt{Extraction}(\overline{s},\ \overline{T}) = d$. Then, $ssd(d) < \overline{s}$.*

**Proof.** We know that

$$\texttt{Extraction}(\overline{s}, \overline{T}) = d = (\overline{s}_r,\ ME)$$

where

$$\texttt{Extract}(A,\ \overline{s}_e,\ \overline{s}_r, \overline{T}) = (\Delta A,\ ME).$$

By Lemma 8.1.5 we obtain

$$ssd((\overline{s}_r,\ ME)) < (\overline{s}_r, \overline{s}_e) = \overline{s}.$$

■

### 8.1.2 Soundness of typing

The following judgement is an extension of the typing judgement $A; ME \vdash o :: T$ to the typing of a set $A$ of pairs $(o, T)$:

**Definition 8.1.7** *($A_1; ME \vdash^{\cdot} A_2$) Given two assumption sets $A_1$ and $A_2$, a set of marked edges $ME$,*

$$A_1; ME \vdash^{\cdot} A_2 \ \Leftrightarrow\ \forall (o, T) \in A_2.\, A_1; ME \vdash o :: T$$

We prove soundness of typing as a direct consequence of the following theorem, which states a strong invariant for the algorithm `Extract`.

**Theorem** *(Invariant for* `Extract`*) Let $A$ be an assumption set, $E$ a set of edges, $o \in Obj$, $T \in \mathbf{T}$. Then,*

$$\texttt{Extract}(A, E, o, T) = (\Delta A,\ ME)\ \wedge\ ME \neq fail\ \Rightarrow\ A; ME \vdash^{\cdot} \Delta A.$$

We prove this statement by induction on the finite depth $i$ of the success tree associated with the generic call $\texttt{Extract}(A, E, o, T)$, by proving the invariant for each case of `Extract`. The hardest part of the proof is the one involving the record type case call,

$$\texttt{Extract}(A, E, o, T) = (\Delta A, ME)$$

where $T \equiv_{\mathbf{r}} [l_1 : T_1 :, \ldots, l_n : T_n]$. Such call may recursively issue $1 \leq j \leq k$ calls,

$$\texttt{Extract}(A_j,\ E,\ o_j, T_j) = (\Delta A_j, ME_j)$$

which are all located at depth $i-1$ in the success tree. The results $(\Delta A_j, ME_j)$ are combined together to yield $(\Delta A, ME)$ in the following way:

$$\Delta A = (o,T) \cup \bigcup_{j=1}^{k} \Delta A_j \qquad ME = \overline{ME} \cup \bigcup_{j=1}^{k} ME_j$$

Accordingly, in order to prove that $A;\ ME \vdash^{\cdot} \Delta A$ is an invariant for the record type case call above, we should prove that,

$$A;\ \overline{ME} \cup \bigcup_{j=1}^{k} ME_j \vdash^{\cdot} (o,T) \cup \bigcup_{j=1}^{k} \Delta A_j$$

is a valid judgement. By induction hypothesis, we can assume that the $k$ $\vdash^{\cdot}$-judgements

$$A_j;\ ME_j \vdash^{\cdot} \Delta A_j$$

are valid; as by observing the algorithm we can infer,

$$A_j = A \cup (o,T) \cup \bigcup_{t=1}^{j-1} \Delta A_t$$

this equates to say that,

$$(A \cup (o,T) \cup \bigcup_{t=1}^{j-1} \Delta A_t);\ ME_j \vdash^{\cdot} \Delta A_j.$$

are valid $\vdash^{\cdot}$-judgements. Consider the $\vdash^{\cdot}$-judgement for $j = k$,

$$(A \cup (o,T) \cup \bigcup_{t=1}^{k-1} \Delta A_t);\ ME_k \vdash^{\cdot} \Delta A_k.$$

From it we can directly produce another valid $\vdash^{\cdot}$-judgement, which almost proves our final statement for the record type case:

$$(A \cup (o,T) \cup \bigcup_{t=1}^{k-1} \Delta A_t);\ ME_k \vdash^{\cdot} (o,T) \cup \bigcup_{t=1}^{k} \Delta A_t.$$

Indeed, to prove our thesis we show that there exists a *dependency* between the $\Delta A_j$'s and the $ME_j$'s. Informally, this dependency states that if a pair $(\overline{o}, \overline{T})$ can be proven correct with a judgement of the form $\Delta A_j \cup A;\ ME \vdash \overline{o} :: \overline{T}$ than it can be proven correct also with a judgement of the form $A;\ ME \cup ME_j \vdash \overline{o} :: \overline{T}$. Intuitively, this relation allows to replace the $\Delta A_j$'s with the $ME_j$'s in the judgement above, thereby leading to our final proof of statement.

Before we proceed with the proof, we introduce the following formal tools.

**Definition 8.1.8** *(Converging and diverging assumption sets) Given an assumption set $A \in \mathcal{P}_{fin}(Oid \times \mathbf{T})$, and $o \in Oid$, $A$* converges *on $o$ ($A(o)\downarrow$) if and only if there exists a type $T \in \mathbf{T}$ such that $(o,T) \in A$. Vice versa $A$* diverges *on $o$ ($A(o)\uparrow$) if and only if there is no type $T \in \mathbf{T}$ such that $(o,T) \in A$.*

**Definition 8.1.9** *(Independent assumption sets) Two assumption sets* $A_1, A_2 \in \mathcal{P}_{fin}(Oid \times \mathbf{T})$ *are* independent *(*$A_1 \asymp A_2$*) if and only if:*

- $\forall o \in Oid.\, (A_1(o)\downarrow \Rightarrow A_2(o)\uparrow)$;
- $\forall o \in Oid.\, (A_2(o)\downarrow \Rightarrow A_1(o)\uparrow)$.

**Definition 8.1.10** *(Compatible sets of marked edges) Two sets of marked edges* $ME_1, ME_2 \in \mathcal{P}_{fin}(Oid \times Label \times M \times Obj^+)$ *are* compatible *(*$ME_1 \doteq ME_2$*) if and only if:*

$$\forall o \in \overleftarrow{ME}_1 \cap \overleftarrow{ME}_2 \,.\, ME_1(o) = ME_2(o)$$

A first interesting property of a successful call

$$\texttt{Extract}(A,\ E,\ o,\ T) = (\Delta A,\ ME)$$

is that the sets $\Delta A$ and $A$ are independent.

**Lemma 8.1.11** *Let* $E$ *be a set of edges,* $T \in \mathbf{T}$, $A \in \mathcal{P}_{fin}(Oid \times \mathbf{T})$, $o \in Obj$, *and* $ST$ *the success tree of a call of* `Extract`. *If,*

$$\texttt{Extract}(A, E, o, T) = (\Delta A,\ ME).$$

*and,*

$$\texttt{Extract}(A, E, o, T) \in ST_N$$

*then,*

$$A \asymp \Delta A$$

**Proof.** This statement can be proved by simply observing that $\Delta A$ contains the pairs $(\overline{o}, \overline{T})$ relative to calls

$$\texttt{Extract}(\overline{A}, E, \overline{o}, \overline{T}) \in ST_N$$

in the call chain originating from $\texttt{Extract}(A, E, o, T)$. As $(\overline{o}, \overline{T}) \in \Delta A$ these calls effectively generated marked edges from $E(\overline{o})$. Accordingly, $A(\overline{o})\uparrow$ otherwise, as $A \subseteq \overline{A}$ the termination test would have prevented these extractions.

On the other hand, if $A(\overline{o})\downarrow$ all calls

$$\texttt{Extract}(\overline{A}, E, \overline{o}, \overline{T}) \in ST_N$$

in the call chain originating from $\texttt{Extract}(A, E, o, T)$ would be such that $A \subseteq \overline{A}$, thus $\overline{A}(\overline{o})\downarrow$. Accordingly, none of these calls could have extracted edges from $E(\overline{o})$, from which $\Delta A(\overline{o})\uparrow$.

■

Note that, given a successful call $\texttt{Extract}(A, E, o, T) = (\Delta A, ME)$, the oids of the edges extracted by the call, namely the set $\overleftrightarrow{ME}$, are a subset of the oids in $O_{A,o,E}$. Hence, intuitively, $A$ converges only on the oids $\overline{o} \in \overleftrightarrow{ME}$ such that $\mid ME(\overline{o}) \mid = 0$, while $\Delta A$ converges on the remainder. Indeed, $\mid ME(\overline{o}) \mid = 0$ only if the extraction with respect to $\overline{o}$ could not be performed due to $A(\overline{o})\downarrow$, which means that an earlier call in the call chain had successfully performed the extraction of the edges outgoing $\overline{o}$. To our purposes we require related but simpler results, which we prove in the following Lemma.

**Lemma 8.1.12** *Let $A$ be an assumption set, $E$ a set of edges, $o \in Obj$, $T \in \mathbf{T}$, and $ST$ the success tree of a call of* `Extract`*. If,*

$$\texttt{Extract}(A, E, o, T) = (\Delta A, ME).$$

*and,*

$$\texttt{Extract}(A, E, o, T) \in ST_N$$

*then,*

$$\forall \overline{o} \in \overleftarrow{ME} \,.\, \Delta A(\overline{o})\downarrow$$

**Proof.** We prove this statement by induction on the finite depth $i$ of the success tree $ST$.

$\boxed{i = 0}$

**Case** $\texttt{Extract}(A \cup \{(o, T)\}, E, o, T) = (\emptyset, \emptyset)$**:** trivial;

**Case** $\texttt{Extract}(A, E, o, int) = (\emptyset, \emptyset)$**:** trivial;

**Case** $\texttt{Extract}(A, E, o, string) = (\emptyset, \emptyset)$**:** trivial;

**Case** $\texttt{Extract}(A, E, o, [l_1 : T_1, \ldots, l_n : T_n]) = (\Delta A, ME)$**:** where,

$$\Delta A = \{(o, [l_1 : T_1, \ldots, l_n : T_n])\} \text{ and } ME = \bigcup_{i=1}^{n} \{< o, l_i, c, \emptyset_{U_i} >\};$$

this case occurs when

$$\forall i : 1 \ldots n. (T_i \equiv_{\mathbf{T}} coll(U_i)) \ \wedge \ (\mid E(l_i) \mid = 0).$$

the conclusion is trivial.

$\boxed{i > 0}$ We assume that the thesis holds for all nodes of $ST$ with depth less than $i$. Since $i > 0$, by definition of $ST$, for all $\texttt{Extract}(A, E, o, T) \in ST_N$ there exists a finite set of children,

$$Sub = \{\texttt{Extract}(A_1, E, o_1, T_1) \dots, \texttt{Extract}(A_k, E, o_k, T_k)\}$$

such that $k \geq 1$ and their depth is $i - 1$. Furthermore, the call corresponding to the node $\texttt{Extract}(A, E, o, T)$ returns a pair $(\Delta A, ME)$ and the calls corresponding to the set of children above return pairs $(\Delta A_j, ME_j)$ with $1 \leq j \leq k$.

When $T \equiv_{\mathbf{T}} T_1 + T_2$ or $T \equiv_{\mathbf{T}} \mu X.T$, then $k = 1$, $A_1 = A$, and $\Delta A_1 = \Delta A$. Therefore, the thesis can be inferred directly from the IH for $(i)$ and $(ii)$.

When $T \equiv_{\mathbf{T}} [l_1 : T'_1, \dots, l_n : T'_n]$, by observing the algorithm, we note that,

$$\Delta A = A \cup \bigcup_{t=0}^{k} \Delta A_t$$

where $\Delta A_0 = \{(o, [l_1 : T_1, \dots, l_n : T_n])\}$. By IH, we know that,

$$\forall j : 1, \dots k. \forall \overline{o} \in \overleftarrow{ME_j} \,.\, \Delta A_j(\overline{o}).$$

Given $\overline{o} \in \overleftarrow{ME}$, either,

- $\exists j : 1, \dots, k. \overline{o} \in \overleftarrow{ME_j}$: by IH we know that $\Delta A_j(\overline{o}) \downarrow$, from which, as $\Delta A_j \subseteq \Delta A$, we can conclude

$$\Delta A(\overline{o}) \downarrow;$$

- $\overline{o} \in \overleftarrow{ME}$: then $\overline{o} = o$, and, as $(o,\ [l_1 : T_1, \dots, l_n : T_n]) \in \Delta A$, we can trivially conclude

$$\Delta A(o) \downarrow .$$

■

The following Lemma describes under which conditions a set of marked edges can be added to a judgement without compromising its validity.

**Lemma 8.1.13** *Given two sets of marked edges $ME_1$ and $ME_2$, an assumption set $A$, $o \in Oid$, and $T \in \mathbf{T}$,*

$$(ME_1 \dot{\asymp} ME_2 \ \wedge\ (A; ME_1 \vdash o :: T) \ \Rightarrow\ A; ME_1 \cup ME_2 \vdash o :: T$$

**Proof.** The judgement $A; ME_1 \vdash o :: T$ may be valid because $A(o)\downarrow$, $o \in \mathit{Atomic}$ and $o$ is a value of $T$, or because $ME(o)$ satisfies $T$. In the first case, any set of edges could be added on to $ME_1$, as rule $(HYP)$ has precedence over the other rules. Similarly, in the second case, the addition of edges to $ME_1$ would not compromise the typing of $o$ with the atomic type $T$. In the third case, the addition of edges could compromise the typing. Indeed, the addition of outgoing edges to an oid may no longer satisfy the requirements of $T$. The proof follows directly from observing that the hypothesis of compatibility between $ME_1$ and $ME_2$ ensures that the set of edges outgoing all oids in $\overleftarrow{ME}_1$ cannot be altered by the union with $ME_2$. ∎

We show that the conditions above apply to the sets of marked edges $ME_j$ resulting from the subcalls $\texttt{Extract}(A_j, E, o_j, T_j)$ issued by a record type case call of `Extract`.

**Lemma 8.1.14** *Let ST be a success tree for* `Extract`*, with* $\texttt{Extract}(A, E, o, T) \in ST_N$*, with* $T \equiv_\tau [l_1 : T_1, \ldots, l_n : T_n]$ *and*

$$\texttt{Extract}(A, E, o, T) = (\Delta A,\ ME).$$

*Let*

$$\texttt{Extract}(A_j,\ E,\ o_j,\ T'_j) = (\Delta A_j,\ ME_j)$$

*with* $1 \leq j \leq k$ *be the children of this node in* $ST_N$*, then,*

$$\forall 1 \leq j_1 < j_2 \leq k.\, ME_{j_1} \dot{\asymp} ME_{j_2}.$$

**Proof.** We prove this statement by contradiction, assuming that there are $1 \leq j_1 < j_2 \leq k$ and $o \in \overleftarrow{ME}_{j_1} \cap \overleftarrow{ME}_{j_2}$ such that $ME_{j_1}(o) \neq ME_{j_2}(o)$. According to Lemma 8.1.12, $\Delta A_{j_1}(o)\downarrow$ and $\Delta A_{j_2}(o)\downarrow$; but this is absurd, as we know that $\Delta A_{j_1} \subseteq A_{j_2}$ and, by Lemma 8.1.11 $A_{j_2}(o)\downarrow \Rightarrow\ A_{j_2}(o)\uparrow$.

∎

Notice that a valid judgement $A_1; ME_1 \vdash^{\cdot} A_2$ indirectly establishes a sort of dependency between the edges $ME_1$ and the assumption set $A_1$ on one side, and the assumption set $A_2$ on the other side: the pairs $(o, T)$ in $A_2$ are all verified for typing by the edges in $ME_1$ and the pairs in $A_1$. Accordingly, we expect to be able to:

1. *replace* the assumption set $A_2$ with the edges $ME_1$ in any $\vdash^{\cdot}$-judgement $A_1 \cup A_2; ME_2 \vdash^{\cdot} A_3$, without compromising its validity: $A_1; ME_2 \cup ME_1 \vdash^{\cdot} A_3$ is still valid;
2. *add* the assumption set $A_2$ to the right side of the judgement: indeed, as $A_1; ME_1 \vdash^{\cdot} A_2$ and $A_1; ME_2 \cup ME_1 \vdash^{\cdot} A_3$ are valid, the judgement $A_1; ME_2 \cup ME_1 \vdash^{\cdot} A_3 \cup A_2$ is also valid.

The following Lemmas prove that these intuitions are true, but only under particular conditions.

**Lemma 8.1.15** *Let $A$ be an assumption set and $A_1$; $ME \vdash^{\cdot} A_2$ be a valid judgement. Then,*

$$A_1 \cup A;\ ME \vdash^{\cdot} A_2$$

*is a valid judgement.*

**Proof.** The proof comes directly from the type rules in Definition 6.6.1. An assumption $(o, T)$ is extracted with $(HYP)$ from an assumption set $A$ only to terminate valid branches of potentially infinite proofs. Accordingly, if the judgement $A_1$; $ME \vdash^{\cdot} A_2$ is valid, the addition of new assumptions cannot compromise its validity, but only potentially shorten the corresponding proof of validity. ■

**Lemma 8.1.16** *(Replacement) Given the assumption sets $A_1$, $A_2$, and $A_3$, the sets of marked edges $ME_1$ and $ME_2$, $o \in Obj$, and $T \in \mathbf{T}$, if*

1. $ME_1 \dot{\asymp} ME_2$,
2. $A_1$; $ME_1 \vdash^{\cdot} A_2$, *and*
3. $A_1 \cup A_2$; $ME_2 \vdash^{\cdot} A_3$;

*then*

$$A_1;\ ME_1 \cup ME_2 \vdash^{\cdot} A_3 \cup A_2;$$

*we can* replace *$A_2$ with $ME_1$ and add it to the right side without compromising the validity of the judgement.*

**Proof.** We first prove that $A_1$; $ME_1 \cup ME_2 \vdash^{\cdot} A_3$ is a valid judgement, by induction on the finite depth of the derivation tree of the valid judgement $A_1 \cup A_2$; $ME_2 \vdash o :: T$ where $(o, T) \in A_3$. Afterward, as $A_1$; $ME_1 \vdash^{\cdot} A_2$ and $ME_1 \dot{\asymp} ME_2$, we can trivially claim that $A_1$; $ME_1 \cup ME_2 \vdash^{\cdot} A_3 \cup A_2$ is a valid judgement.

$\boxed{i = 0}$

**Case** $(HYP)$**:** that is $(o, T) \in A_1 \cup A_2$. It can be either,

- $(o, T) \in A_1$: thus $A_1$; $ME_2 \vdash o :: T$ is a valid judgement; by hypothesis 1 and Lemma 8.1.13 we derive $A_1$; $ME_1 \cup ME_2 \vdash o :: T$;
- $(o, T) \in A_2$: by hypothesis 2 we know that $A_1$; $ME_1 \vdash o :: T$; by hypothesis 1 and Lemma 8.1.13 we derive $A_1$; $ME_1 \cup ME_2 \vdash o :: T$;

**Case** $(INT)$**:** trivial, as assumption sets and marked edges are not required to verify the premises of the rule;

**Case** $(STRING)$**:** trivial, as assumption sets and marked edges are not required to verify the premises of the rule;

**Case** $(EMPTY - COLL)$**:** trivial, as assumption sets and marked edges are not required to verify the premises of the rule.

$\boxed{i > 0}$ by induction assuming that the thesis holds for the premises of the rule at hand.

**Case** $(REC)$**:** that is $A_1 \cup A_2$; $ME_2 \vdash o :: \mu X.T$; the premise of this rule is the judgement $A_1 \cup A_2$; $ME_2 \vdash o :: T\left[{}^{\mu X.T}/_{X}\right]$. By IH we know that $A_1$; $ME_1 \cup ME_2 \vdash o :: T\left[{}^{\mu X.T}/_{X}\right]$. Applying $(REC)$ we derive $A_1$; $ME_1 \cup ME_2 \vdash o :: \mu X.T$ as expected.

**Case** $(UNION - L)$**:** that is $A_1 \cup A_2$; $ME_2 \vdash o :: T_1 + T_2$; the premise of this rule is the judgement $A_1 \cup A_2$; $ME_2 \vdash o :: T_1$. By IH we know that $A_1$; $ME_1 \cup ME_2 \vdash o :: T_1$. Applying $(UNION - L)$ we derive $A_1$; $ME_1 \cup ME_2 \vdash T_1 + T_2$ as expected.

**Case** $(UNION - R)$**:** as for case $(UNION - L)$;

**Case** $(RECORD - COLL)$**:** that is $A_1 \cup A_2$; $ME_2 \vdash o :: [l_1 : T_1, \ldots, l_n : T_n]$. Note that, the premises of this judgement are $1 \leq j \leq k$ judgements

$$A_1 \cup A_2 \cup \{(o, [l_1 : T_1, \ldots, l_n : T_n])\};\ ME_2 \vdash o_j :: T'_j.$$

By Lemma 8.1.15, the judgement

$$A_1 \cup \{(o, [l_1 : T_1, \ldots, l_n : T_n])\};\ ME_1 \vdash^{\cdot} A_2$$

is valid. Therefore, by IH we can now state that $\forall j : 1, \ldots, k$:

$$A_1 \cup \{(o, [l_1 : T_1, \ldots, l_n : T_n])\};\ ME_1 \cup ME_2 \vdash o_j :: T'_j$$

Since by hypothesis 1 we know that $ME_2(o) = (ME_1 \cup ME_2)(o)$, $(REC)$ can be applied again so as to achieve:

$$A_1;\ ME_1 \cup ME_2 \vdash o :: [l_1 : T_1, \ldots, l_n : T_n]$$

■

So far, we have formally shown that the subcalls

$$\texttt{Extract}(A_j,\, E,\, o_j,\, T_j) = (\Delta A_j,\, ME_j),$$

with $1 \leq j \leq k$, of the record type case of the algorithm do verify the condition

$$\forall 1 \leq j_1 < j_2 \leq k.\, ME_{j_1} \dot{\asymp} ME_{j_2}.$$

The next Lemma applies lemma $\vdash^{\cdot}$-replacement, on the base of these conditions, so as to to prove the validity of a judgement that *almost* proves the *invariant* for the record type case call.

**Lemma 8.1.17** *Let $ST$ be a success tree for* `Extract`*, with* $\texttt{Extract}(A,\, E,\, o,\, T) \in ST_N$*, with* $T \equiv_{\mathrm{r}} [l_1 : T_1, \ldots, l_n : T_n]$ *and*

$$\texttt{Extract}(A,\, E,\, o,\, T) = (\Delta A,\ ME).$$

*Let*

$$\texttt{Extract}(A_j,\ E,\ o_j,\ T'_j) = (\Delta A_j,\ ME_j)$$

*with $1 \leq j \leq k$ be the children of this node in $ST$, then $\forall t : 1, \ldots k$:*

$$A_t;\, ME_t \vdash^{\cdot} \Delta A_t \ \Rightarrow\ A \cup \{(o, T)\};\, \bigcup_{j=1}^{t} ME_j \vdash^{\cdot} \bigcup_{j=1}^{t} \Delta A_j.$$

**Proof.** We prove this statement by induction on the finite number $k$ of subcalls invoked by the record type case call.

$\boxed{k = 1}$ by observing the algorithm, we know that $A_1 = A \cup \{(o, T)\}$; by hypothesis we know that $A_1;\, ME_1 \vdash^{\cdot} \Delta A_1$, which corresponds to what we need to prove, namely $A \cup \{(o, T)\};\, ME_1 \vdash^{\cdot} \Delta A_1$.

$\boxed{k > 0}$ if we consider that:

1. $(\bigcup_{j=1}^{k-1} ME_j) \dot{\asymp} ME_k$: direct consequence of Lemma 8.1.14;
2. by IH,

$$A \cup \{(o, T)\};\, \bigcup_{j=1}^{k-1} ME_j \vdash^{\cdot} \bigcup_{j=1}^{k-1} \Delta A_j$$

and,

3. by hypothesis, $A_k$; $ME_k \vdash^{\cdot} \Delta A_k$, which is

$$A \cup \{(o, T)\} \cup \bigcup_{j=1}^{k-1} \Delta A_j;\ ME_k \vdash^{\cdot} \Delta A_k$$

we can apply Lemma 8.1.16, obtaining

$$A \cup \{(o, T)\};\ \bigcup_{j=1}^{k} ME_j \vdash^{\cdot} \bigcup_{j=1}^{k} \Delta A_j$$

■

As mentioned above, the result of the previous Lemma *almost* works out the invariant for the record type case of `Extract`. The following Theorem, on the basis of such result, provides a complete proof for our invariant.

**Theorem 8.1.18** *(Invariant for* `Extract`*) Let $A$ be an assumption set, $E$ a set of edges, $o \in Obj$, $T \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n] \in \mathbf{T}$, and $ST$ the success tree of a call of* `Extract`*. If,*

$$\texttt{Extract}(A,\ E,\ o,\ T) = (\Delta A,\ ME).$$

*and,*

$$\texttt{Extract}(A,\ E,\ o,\ T) \in ST_N$$

*then,*

$$A,\ ME \vdash^{\cdot} \Delta A$$

**Proof.** We prove this statement by induction on the finite number $i$ of recursive calls issued by $\texttt{Extract}(A,\ E,\ o,\ T)$ to terminate its execution.

$\boxed{i = 0}$

**Case** $\texttt{Extract}(A \cup \{(o,T)\},\ E,\ o, T) = (\emptyset,\ \emptyset)$**:** $A, \emptyset \vdash^{\cdot} \emptyset$ is trivially true;

**Case** $\texttt{Extract}(A,\ E,\ o,\ int) = (\emptyset,\ \emptyset)$**:** as for the case above;

**Case** $\texttt{Extract}(A,\ E,\ o,\ string) = (\emptyset,\ \emptyset)$**:** as for the case above;

**Case** $\texttt{Extract}(A,\ E,\ o,\ [l_1 : T_1, \ldots, l_n : T_n]) = (\Delta A,\ ME)$**:** where

$$\Delta A = \{(o, [l_1 : T_1, \ldots, l_n : T_n])\} \text{ and } ME = \bigcup_{i=1}^{n} \{< o, l_i, c, \emptyset_{U_i} >\};$$

this case occurs when

$$\forall i : 1 \ldots n. (T_i \equiv_{\mathrm{T}} coll(U_i)) \ \wedge \ (\mid E(l_i) \mid = 0).$$

The conclusion comes directly from the definition of `Extract`, which ensures that $A;\ ME \vdash o :: [l_1 : T_1, \ldots, l_n : T_n]$, from which $A;\ ME \vdash^{\cdot} \Delta A$.

$\boxed{i > 0}$

**Case** $\texttt{Extract}(A,\ E,\ o,\ \mu X.T) = (\Delta A, ME)$**:** the $i-1$'th step of execution is then

$$\texttt{Extract}(A,\ E,\ o,\ T\left[{}^{\mu X.T}/_{X}\right]) = (\Delta A, ME).$$

Thus, the thesis follows directly from the IH;

**Case** $\texttt{Extract}(A,\ E,\ o,\ T_1 + T_2) = (\Delta A, ME)$**:** the $i-1$'th step is then either

$$\texttt{Extract}(A,\ E,\ o,\ T_1) = (\Delta A, ME)$$

or

$$\texttt{Extract}(A,\ E,\ o,\ T_2) = (\Delta A, ME).$$

In either case, the thesis follows directly from the IH;

**Case** $\texttt{Extract}(A,\ E,\ o,\ [l_1 : T_1, \ldots, l_n : T_n]) = (\Delta A, ME)$**:** this call invokes $1 \leq j \leq k$:

$$\texttt{Extract}(A_j,\ E,\ o_j,\ T'_j) = (\Delta A_j,\ ME_j)$$

calls, with $ME_j \neq fail$. All these calls require less than $i$ calls to terminate. By IH we can assume that $\forall 1 \leq j \leq k. (A_j,\ ME_j \vdash^{\cdot} \Delta A_j)$ and, by the application of Lemma 8.1.17, obtain the following valid judgement:

$$A \cup \{(o,\ [l_1 : T_1, \ldots, l_n : T_n])\};\ \bigcup_{j=1}^{k} ME_j \vdash^{\cdot} \bigcup_{j=1}^{k} \Delta A_j$$

Since $\overleftarrow{\overline{ME}} = \{o\}$ and $o \notin \bigcup_{j=1}^{k} \overleftarrow{ME_j}$ (because $A_j(o) \downarrow$ for all $j : 1, \ldots, k$), we can deduce that $\overline{ME} \mathrel{\dot{\asymp}} (\bigcup_{j=1}^{k} ME_j)$ and apply Lemma 8.1.13 to achieve the valid $\vdash^{\cdot}$-judgement

$$A \cup \{(o,\ [l_1 : T_1, \ldots, l_n : T_n])\};\ ME \vdash^{\cdot} \bigcup_{j=1}^{k} \Delta A_j \tag{8.1.1}$$

where $ME = \overline{ME} \cup \bigcup_{j=1}^{k} ME_j$.

Note that we can state that,

$$A \cup \{(o, [l_1 : T_1, \ldots, l_n : T_n])\}; ME \vdash o_j :: T'_j; \tag{8.1.2}$$

for all $j : 1, \ldots, k$. The proof of this is quite intuitive, but we give it below for completeness of presentation. Consider the generic subcall

$$\texttt{Extract}(A_j, E, o_j, T'_j) = (\Delta A_j, ME_j);$$

by definition of `Extract` it can be that:

- $(o_j, T'_j) \in A_j$: $o_j$ has been already extracted by an earlier call; since

$$A_j = A \cup \{(o, [l_1 : T_1, \ldots, l_n : T_n])\} \cup \bigcup_{t=1}^{j-1} \Delta A_t$$

  if $(o_j, T'_j) \in A \cup \{(o, [l_1 : T_1, \ldots, l_n : T_n])\}$ the conclusion derives from the application of rule $(HYP)$ to the $\vdash$-judgement 8.1.2; if $(o_j, T'_j) \in \bigcup_{t=1}^{j-1} \Delta A_t$, the conclusion comes directly from the $\vdash^{\cdot}$-judgement 8.1.1;
- $T'_j \in \{int, string\}$ and $o_j \in Atomic$: in this case the conclusion is trivial, as both rule $(INT)$ or rule $(STRING)$ apply to the $\vdash$-judgement 8.1.2; in particular, as by hypothesis $ME_j \neq fail$, we know that $o_j$ satisfies the atomic type $T'_j$;
- $(o_j, T'_j) \in \Delta A_j$: if none of the two cases above apply, since $ME_j \neq fail$, then $o_j$ must have been extracted by the subcall; therefore, by definition of `Extract`, $(o_j, T'_j) \in \Delta A_j$; the conclusion derives directly from the $\vdash^{\cdot}$-judgement 8.1.2.

Therefore, as by definition of the algorithm the marked edges $\overline{ME}$ verify the requirements of $[l_1 : T_1, \ldots, l_n : T_n]$, rule $(RECORD-COLL)$ can be applied to get the valid judgement:

$$A; ME \vdash o :: [l_1 : T_1, \ldots, l_n : T_n]$$

which in turn clearly implies that

$$A; ME \vdash^{\cdot} \{(o, [l_1 : T_1, \ldots, l_n : T_n])\}.$$

Considering that,

1. $\bigcup_{j=1}^{k} ME_j \dot{\asymp} ME$: trivial, as

$$ME = \overline{ME} \cup \bigcup_{j=1}^{k} ME_j$$

and

$$\overline{ME} \dot{\asymp} \bigcup_{j=1}^{k} ME_j.$$

2. $A$; $ME \vdash^{\cdot} \{(o, [l_1 : T_1, \ldots, l_n : T_n])\}$; and
3. $A \cup \{(o, [l_1 : T_1, \ldots, l_n : T_n])\}$; $\bigcup_{j=1}^{k} ME_j \vdash^{\cdot} \bigcup_{j=1}^{k} \Delta A_j$.

by Lemma 8.1.16, we can state that

$$A;\ ME \vdash^{\cdot} \Delta A$$

where $\Delta A = \{(o, [l_1 : T_1, \ldots, l_n : T_n])\} \cup \bigcup_{j=1}^{k} \Delta A_j$, is a valid judgement.

■

As we shall see in the following Theorem, soundness of typing is a trivial consequence of the invariant proven above.

**Theorem 8.1.19** *(Soundness of typing) Let $s \in S$, $T \in \mathbf{T}$, $\texttt{Extraction}(s, T) = d$. Then, $d : T$.*

**Proof.** By definition of extraction,

$$\texttt{Extraction}(s, T) = d = (s_r, ME)$$

where $\texttt{Extract}(\emptyset, s_e, s_r, T) = (\Delta A, ME)$ and $ME \neq fail$. By Lemma 8.1.18 we know that $\emptyset$; $ME \vdash^{\cdot} \Delta A$. By definition of `Extract` it can be that:

1. $(s_r, T) \notin \Delta A$: this is when $T \in \{int, string\}$ and $s_r \in Atomic$; as $ME \neq fail$, we know that $s_r$ satisfies the atomic type $T$; hence

$$\emptyset; \emptyset \vdash s_r :: T$$

is a valid judgement, from which

$$d = (s_r, \emptyset) : T$$

2. $(s_r, T) \in \Delta A$: by definition of $\vdash^{\cdot}$-judgement,

$$\emptyset;\ ME \vdash s_r :: T$$

is a valid judgement, from which

$$d = (s_r, ME) : T$$

■

## 8.2 Completeness

As shown in Chapter 4, completeness of an extraction algorithm with respect to extractability depends on the definition of the set of *values extractable from an SSDB with respect to a given type*. In the case of $L$ this set is defined as follows.

**Definition 8.2.1** *(Extractable values) Let $\overline{s} \in S$, $\overline{T} \in \mathbf{T}$. The set of values extractable from $\overline{s}$ according to $\overline{T}$ is defined as,*

$$D_{\overline{s},\overline{T}} = \{d \mid d \textit{ is extractable from } \overline{s} \textit{ according to } \overline{T}\}$$

The algorithm `Extraction` is not complete with respect to extractability. In other words, given $\overline{s} \in S$ and $\overline{T} \in \mathbf{T}$ such that $D_{\overline{s},\overline{T}}$ is not empty, the algorithm does not guarantee a successful termination.

In the following we provide examples of incompleteness and discuss their impact in the practical usage of our extraction system. Afterward, we show that `Extraction` is complete whenever its application is restricted to the domain of tree-structured SSDBs.

### 8.2.1 Cases of Incompleteness

We shall find that incompleteness is due to the presence of shared oids in the SSDB to be traversed by the algorithm, in combination with one of the following:

- the determinism introduced in the union type case of `Extract`;
- the determinism introduced in the extraction of one edge out of multiple candidate edges in the record type case of `Extract`;

or to the over-restrictive termination test in the record type case of `Extract`.

#### Union and record types determinism

Consider the example in Figure 8.1, where the SSDB $\overline{s}$ is extracted according to the type $\overline{T}$,

$$\begin{array}{l}[\\ \quad \mathit{man} : [\mathit{children} : [\mathit{age} : \mathit{int}] + [\mathit{child} : \mathit{string}]],\\ \quad \mathit{woman} : [\mathit{children} : [\mathit{child} : \mathit{string}]]\\ ].\end{array}$$

`Extract` begins its execution visiting $o_1$ with the record type $\overline{T}$. Since $o_1$ verifies the required properties, the algorithm invokes two recursive calls,

$$\texttt{Extract}(A_1, \overline{s}_e, o_2, [\mathit{children} : [\mathit{age} : \mathit{int}] + [\mathit{child} : \mathit{string}]])$$

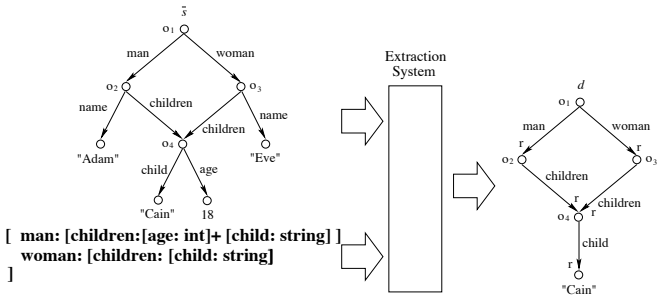



Figure 8.1: Example of incompleteness due to union types

where $A_1 = \{(o, \overline{T})\}$ and,

$$\texttt{Extract}(A_1 \cup \Delta A_1, \overline{s}_e, o_3, [children : [child : string]])$$

where $\Delta A_1$ represents the result of the extraction of the previous call.

The first recursive call recursively issues the union type case of `Extract` with $o_4$ and $[age : int] + [child : string]$. Thus, the algorithm invokes a call of `Extract` with $o_4$ and the *first* type in the union, namely $[age : int]$, which will be successfully terminated, extracting the edge labelled as *age*. In summary, the call path of the first recursive call returns the pair $(\Delta A_1, ME_1)$ such that,

$$\Delta A_1 = \{(o_2,\ [children : [age : int] + [child : string]]),\\ (o_4,\ [age : int])\}$$

$$ME = \{< o_2, children, r, o_4 >, < o_4, age, r, 18 >\})$$

The second recursive call recursively visits $o_3$ with $[children : [child : string]]$, and then $o_4$ with $[child : string]$, under the assumptions $A_2$. The latter call will fail, because $(o_4, [age : int]) \in A_2$ and $[age : int] \not\equiv_{\mathrm{T}} [child : string]$.

Nevertheless, as shown in Figure 8.1, the algorithm could have extracted the value $d \in D_{\overline{s},\overline{T}}$. The failure is due to the fact that the algorithm does not support the non-determinism encountered when extracting according to the members of a union type. To do so, rather than fail and terminate, the algorithm should roll back to the first visit of $o_4$ with a union type and try the next member of the union.

As an example of the second kind of non-determinism, consider the scenario

illustrated in Figure 8.2, where the SSDB $\overline{s}$ is extracted according to the type $\overline{T}$,

$$
\begin{array}{l}
[ \\
\quad \mathit{child} : [\mathit{name} : \mathit{string}], \\
\quad \mathit{favouritechild} : [\mathit{age} : \mathit{string}]] \\
].
\end{array}
$$

`Extract` begins its execution extracting $o_1$ with the record type $\overline{T}$, which requires the existence of an edge labelled as *favouritechild* and of an edge labelled as *child*. Note that, $o_1$ features two edges that, being labelled as *child*, are candidate for the extraction; `Extraction` fails if the edge to be selected first is $< o_1, \mathit{child}, o_3 >$. Indeed, the extraction according to this edge would succeed and eventually lead to the extraction of $o_3$ with respect to $[\mathit{name} : \mathit{string}]$. Afterward, the call relative to the label $\mathit{favouritechild}$ will then try to extract $o_3$ with respect to the type $[\mathit{age} : \mathit{int}]$, thereby causing the failure of `Extraction`.

However, observe that had `Extract` first selected the edge $< o_1, \mathit{child}, o_2 >$, `Extraction` would have successfully returned the value $d$ as shown in the picture. As in the case of union types, the algorithm does not support the non-determinism brought into play by the edges candidate for an extraction. In the example above, the algorithm should have rolled back to the extraction of $o_1$ with the record type and try with another candidate edge labelled as *child*.

Non-determinism could be enforced by endowing the algorithm with backtracking techniques. The successful extraction of $o$ with a record type $T$, which is a subterm of a union type $U$, should be memorised in a separate environment $B$ as a tuple $(o, T, o_U, U, A_U)$, where $o_U$ is the oid that was extracted according to $U$, through which $o$ was extracted, and $A_U$ is the assumption set passed to the call with $o_U$ and $U$. Whenever a subsequent record type case call fails on $o$ because of a mismatch with $T$, the algorithm should roll-back to the extraction environment relative to the extraction with $U$ and try the extraction of $o$ with the next member of the union.

The same technique could be applied to the edges with the same label of a record type call. If the call fails, as exemplified above, and there are other non-tried candidate edges available, the algorithm should roll back and re-apply the extraction.

### Termination test

Incompleteness is also due to the termination test in the record type case of `Extract`, which is too strong with respect to $L$'s relation of typing. For example, the SSDB $\overline{s}$ in Figure 8.3 could be extracted according to the type,

$$
\begin{array}{l}
[ \\
\quad \mathit{man} : [\mathit{children} : [\mathit{child} : [\mathit{name} : \mathit{string}] + T]], \\
\quad \mathit{woman} : [\mathit{children} : [\mathit{child} : [\mathit{name} : \mathit{string}]]] \\
].
\end{array}
$$

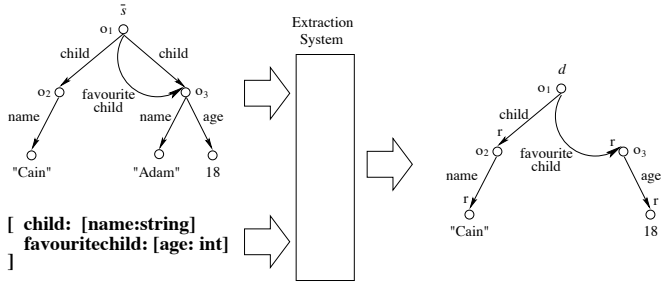



Figure 8.2: Example of incompleteness due to edges with the same label

returning the value $d$ shown in the picture. Indeed, the oid $o_5$ could be extracted both with $[name : string] + T$, for all $T \in \mathbf{T}$, and with $[name : string]$.[1]

Unfortunately, the algorithm's termination test is so restrictive that it forbids this extraction before it can take place. When visiting for the second time the shared node $o_4$ with the type $[child : [name : string]]$ the termination test fails for the oid had been previously extracted according to the type $[child : [name : string] + T]$ and,

$$[child : [name : string]] \neq_{\mathbf{T}} [child : [name : string] + T].$$

Observe that this happens whenever a generic oid ($o_4$ in the example above) is extracted according to record types differing for the union types they feature as subterms.

A partial solution to this problem consist in defining a relation of *subtyping* based on the rule $T_1 <_{\mathbf{T}} T_1 + T_2$. By means of this relation the termination test could be refined to check whether an oid falls in the category exemplified above and can therefore be extracted. For instance, the algorithm would successfully visit $o_4$ as $[child : [name : string]] <_{\mathbf{T}} [child : [name : string] + T]$.

Note that, to be effective, the test should be combined with the backtracking techniques mentioned above. Indeed, the the algorithm may first successfully visit and extract $o_4$ with $[child : [name : string] + T]$. Accordingly, the second visit with $[child : [name : string]]$ would not satisfy the subtyping check. Backtracking techniques over union types and equally labelled edges would make sure that the algorithm will try the other way around, thereby returning the correct extraction.

Enriched with subtyping test and backtracking techniques, `Extraction` would capture a larger set of extractable values, but would not yet be complete. There are

[1] Note that, with reference to the union type case anomaly discussed above, this extraction may be performed because, luckily, $[name : string]$ is the first matching member of the union type.

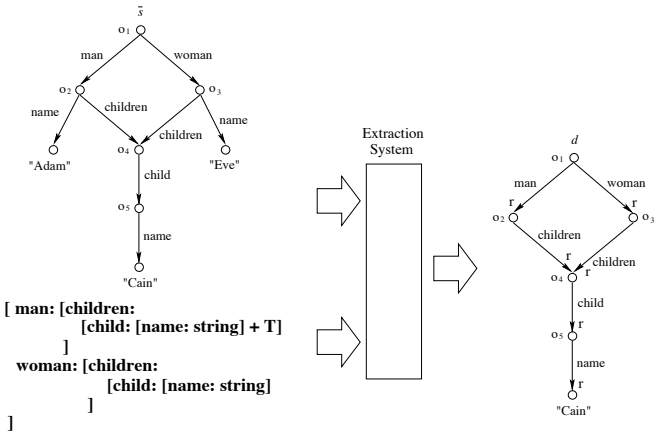



Figure 8.3: Example of incompleteness due to over-restrictive termination test

other two main problems, whose solution would be too expensive to be implemented: the order of the labels of a record type, which imposes an order of visit of the edges emanating from an oid, and the presence of particular forms of shared oids. These oids are reached by different call paths of the algorithm with record types $T_1, \ldots, T_n$ that are not in subtyping relation and have a non-empty intersection, that is there exists $T$ such that $\forall i : 1, \ldots, n : .T <_{\mathbf{T}} T_i$. In particular, these oids can be extracted according to $T$, and should therefore be extractable according to all $T_i$'s, but the algorithm fails, as the types are not in subtyping relation. As an example of this scenario, consider again $o_4$ in the example above, and the two types $[child : [name : string] + T_1]$ and $[child : [name : string] + T_2]$, where $T_1 \not<_{\mathbf{T}} T_2$ and $T_2 \not<_{\mathbf{T}} T_1$. Clearly, $d$ is extractable according to both types, but the algorithm fails.

### Observations

The anomalies presented above prove that our algorithm is not complete with respect to extractability. However, a more complex algorithm could be developed, which, heavily exploiting backtracking techniques and a refined termination test, could provide the level of non-determinism required to capture almost all extractable values for any input $\overline{s}$ and $\overline{T}$.

Nonetheless, the anomalies shown above are quite peculiar and may occur only in presence of shared oids and cycles, which are in turn quite rare in SSDBs. In addition, their manifestation does not violate the semantic of the programming

language $L$, but only hamper the extraction of an extractable value. Therefore, we believe the system can be successfully used also in presence of shared oids and cycles.

Finally, as we shall discuss in the next section, our algorithm can be proven complete with respect to tree-structured SSDBs. This restriction does not compromise the importance of our approach as most of the scenarios in the real world do not necessarily feature shared oids and cycles and fall in this category. For instance, the majority of semistructured databases available are represented as XML documents, which are based on tree-structured models.

### 8.2.2 Case of completeness

`Extraction` is not complete with respect to the definition of extractability, due to the extraction of typed values from SSDBs containing shared entities and cycles. However, if we restrict our attention to a smaller set of SSDBs, completeness can be proven.

**Definition 8.2.2** *(Non-shared oids) Let $\overline{s} \in S$. $o \in Oid(\overline{s})$ is* non-shared *if there exists only one path from $\overline{s}_r$ to $o$.*

**Definition 8.2.3** *(Tree-structured SSDBs) The set of* Tree-structured SSDBs *is defined as,*

$$S_t = \{s \in S \mid \forall o \in Oid(s).\ o \text{ is non-shared}\}$$

Restricting to the set $S_t$ ensures that the algorithm `Extraction` cannot successfully visit the same oid twice in the same call chain. Accordingly, if an oid is successfully extracted according to a record type, no termination test can cancel such extraction and the anomalies described earlier cannot take place. As the $\Delta A$'s and the termination test are no longer necessary, the algorithm `Extraction` in Figure 7.1 can be significantly simplified, to become the algorithm `Extraction`$_t$ shown in Figure 8.4.

Consider $d \in D_{\overline{s},\overline{T}}$. As $d$ has no shared oids, the proof of conformity of $d_r$ to $\overline{T}$ requires to check conformity of each $o \in Oid(d)$ with a record type $[l_1 : T_1, \ldots, l_n : T_n]_o$ *only once*. Accordingly, as $ssd(d) < \overline{s}$, we also know that each oid $o \in Oid(ssd(d))$ *can* be reached by `Extract` with the record type $[l_1 : T_1, \ldots, l_n : T_n]_o$. In particular, as $o$ conforms to $[l_1 : T_1, \ldots, l_n : T_n]_o$ and $Erase(d(o)) \subseteq \overline{s}(o)$, we know that this call can be performed successfully. Moreover, once extracted, $o$ cannot be visited again in the call chain of `Extract`. Thus, the successful extraction of $o$ cannot affect other extractions in the call chain.

In conclusion, `Extract`$_t(\overline{s}_e, \overline{s}_r, \overline{T})$ cannot fail and `Extraction`$_t$ is complete.

```
Extraction_t :  S_t × T → D ∪ {fail}
Extraction_t(s̄, T̄) =
  { ME := Extract_t(s̄_e, s̄_r, T̄)
    if ME = fail then return fail
                 else return (s̄_r, ME) }


Extract_t : P_fin(Oid × Label × Obj) × Obj × T →
        P_fin(Oid × Label × M × Obj^+) ∪ {fail}
case Extract_t(E, o, μX.T)
  { return Extract_t(E, o, T[μX.T/X]) }

case Extract_t(E, o, int)
  { if o ∈ Integer then return ∅
                 else return fail }

case Extract_t(E, o, string)
  { if o ∈ String  then return ∅
                 else return fail }

case Extract_t(E, o, T_1 + T_2)
  { ME := Extract_t (E, o, T_1)
    if ME = fail then return Extract_t(E, o, T_2))
                 else return ME

case Extract_t(E, o, T̄), where T̄ ≡_T [l_1 : T_1, ..., l_n : T_n]
  { ME := ∅
    for i := 1 to n do
    {   FAILED := true
        if T_i ≡_T coll(U)
                then if E(o, l_i) = ∅
                       then { ME := ME ∪ {<e←, l_i, c, ∅_U >}  <introduces an empty collection edge>
                              FAILED := false }
                       else for all e ∈ E(o, l_i) do
                              { ME_t := Extract_t(E, e→, U)
                                if ME_t ≠ fail then
                                { ME := ME ∪ ME_t ∪ { <e←, l_i, c, e→>}
                                  FAILED := false } }
                else for all e ∈ E(o, l_i) do
                       { ME_t := Extract_t(E, e→, T_i)
                         if ME_t ≠ fail then
                           {  ME := ME ∪ ME_t ∪ { <e←, l_i, r, e→>}
                              FAILED := false
                              exitfor; } }
        if FAILED then  <one record's label could not be satisfied by E(o)>
                { ME := fail
                  exitfor } } }
    return ME }

case Extract_t(E, o, T), where o and T do not match any of the cases above
  { return fail }
```

Figure 8.4: Extraction algorithm for tree-structured SSDBs

# Chapter 9

# SNAQue

In this Chapter we describe a novel and complementary approach to static typing approaches for querying XML, based on the extraction mechanisms presented in Chapter 4.

We present a prototype of the system *SNAQue* – the *Strathclyde Novel Architecture for Querying Extensible markup language* (cf. [46, 47]) – currently under development at Strathclyde University, Glasgow (UK). SNAQue will enable programming languages interlaced with CORBA to be used in querying the information represented in XML format.

SNAQue is based on an extraction algorithm $\texttt{EXTR}_{IDL}$ for the *Interface Definition Language* (*IDL*) of CORBA (cf. [79]), which extracts Java objects corresponding to regular subsets of XML SSDBs. Given an IDL definition and an XML SSDB, our system returns a CORBA object that serves the extracted value across the ORB framework.

In describing the system, we assume the reader has a some understanding of concepts such as *skeletons* and *stubs* in CORBA.

## 9.1 The prototype

CORBA IDL is an interface definition language based on the syntax of the type language of *C*. IDL describes the most common type abstractions of programming languages, plus some forms of behaviour, such as functions and procedures. In particular, specific mappings from IDL definitions to the types of programming languages (Java, C, and many others) are available, which implicitly define a typing relation between the values of the language and IDL definitions.

The definition of extractability underlying the system SNAQue is based on the mapping `JIDL` from IDL to Java classes, according to which Java objects are in a typing relation with the IDL definitions that map onto their corresponding classes. Hence, a Java object is extractable from an XML SSDB according to an IDL definition if:

- it *conforms* to the IDL definition: the object is of the class to which maps the IDL definition;
- the object *maps* onto an SSDB *included* into the XML SSDB.

Due to the quite straightforward mappings:

- `IDLtoT`: from a subset of IDL, that does not include interfaces, onto the type system **T**;
- `XMLtoS`: from XML SSDBs to tree-structured SSDB $S_t$;
- `JAVAtoD`: from Java objects to the values $D$;

we formally defined extractability based on the definition of extractability for $L$.

**Definition 9.1.1** *(Extractability for IDL and Java) A Java object $j$ is extractable from an XML SSDB xml according to an IDL definition idl if there exists $d \in D$ such that* $\texttt{JAVAtoD}(j) = d$*, and $d$ is extractable from* $\texttt{XMLtoS}(xml)$ *according to* $\texttt{IDLtoT}(idl)$.

Similarly, the algorithm $\texttt{EXTR}_{IDL}$, sound with respect to this definition, exploits the algorithm $\texttt{Extraction}_t$ defined in Chapter 7.[1] Specifically, given the input $xml$ and $idl$, $\texttt{EXTR}_{IDL}$ performs the following steps:

1. relying on parsers `IDLtoT` and `XMLtoS` that translate from CORBA IDL to **T** and from XML documents to $S_t$, calculates a type $T = \texttt{IDLtoT}(idl)$ and an SSDB $s = \texttt{XMLtoS}(xml)$;
2. executes the call $\texttt{Extraction}_t(s, T)$, which here we assume to be successful; the call yields a value $d$ such that $d : T$; due to the mapping `IDLtoT`, $d$ *structurally and statically conforms* to *idl* too;
3. generates a Java object $j$ corresponding to $d$ such that $j$ conforms to *idl*.

Note that the equivalence between values of different languages identified by CORBA is similar in principle to the one identified with a mapping between the values of a language and SSDBs. Hence, given the mapping $ssd(\texttt{JAVAtoD}(x))$ from Java objects to SSDBs, any CORBA-compliant language may transparently access the extracted value via the given IDL definition. SNAQue (see Figure 9.1) makes this possible by serving the Java object resulting from $\texttt{EXTR}_{IDL}$ across the ORB. Specifically, the system,

[1] Recall that $\texttt{Extraction}_t$ takes as input a tree-structured SSDB $s$ and a type $T$ from the type system **T** of the language $L$; the result of the extraction is a value $d$ of type $T$, this value corresponding to a subset $s'$ of $s$.

1. generates the CORBA interface $I_{idl}$ corresponding to the simple CORBA definition *idl*: $I_{idl}$ describes a CORBA object that has a method `getData` returning an object of type *idl*;

2. defines the CORBA object, by providing a Java implementation for $I_{idl}$, that is a Java class whose method `getData` returns the Java object $j$.

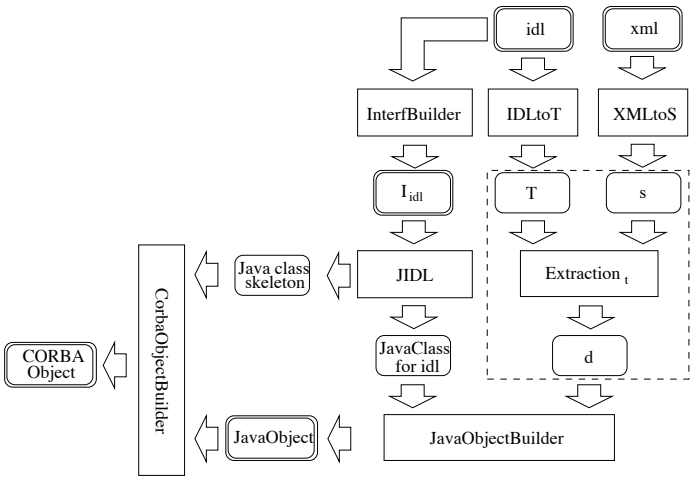



Figure 9.1: SNAQue: extraction of CORBA objects from XML SSDBs.

As described in Chapter 4, the approach consists of two phases. First, a programmer attempts to project an arbitrary XML document onto a given IDL type with SNAQue. If the extraction can be performed, the system creates a CORBA object whose unique method `getData` returns the representation of the extracted value. Secondly, programs in a general-purpose CORBA compliant language can be written with respect to this object, and refer to the object extracted value with a call to the method `getData`.

In the following sections, we formally prove the correctness of SNAQue by providing a mapping from XML documents onto the set $S_t$ and a mapping from IDL definitions onto the types in **T**. Moreover, we shall observe how the Java object $j$ conforming to *idl* is generated according to a one-to-one mapping from the extracted value $d$.

Finally, we shall see how $j$, i.e. the representation of the extracted value, is served to consuming distributed applications via a CORBA object corresponding to $I_{idl}$.

## 9.2 From XML documents to SSDBs

There are strong similarities between the logical part of the XML format and the graph-based data models for SSD. Indeed, as illustrated in Chapter 2 (see Figure 2.3), an XML document can be modelled by an SSDB in which oids correspond to document elements and the edge relation corresponds to the nesting relation for elements. In particular, the prototype of SNAQue focuses on mapping from the core subset of the XML format, namely that relative to the logical structure of documents, onto the set $S_t$ of tree-structured SSDBs. To achieve this, XML documents are not considered under the influence of a schema, thus interpreting attribute values as being of type `STRINGTYPE`. This excludes the interpretation of XML documents as graphs through the attribute types `ID` and `IDREF`. The reason for this is that we aim at giving a proof of soundness of the prototype and not a complete data model for XML.

In the following we shall describe the specific design choices characterising the mapping from XML to SSDBs underlying SNAQue. For simplicity, we shall illustrate this mapping by means of graphic representations of graphs rather than relying on the syntactic representation for SSDBs in $S_t$.

### 9.2.1 Ordering

Elements of XML documents are ordered according to their position in the text. Ordering of elements is in contrast to most SSD models where two SSDBs are equal regardless of the order of siblings (see Section 5.2). Although order can be easily captured in SSD data models, it complicates the semantics of the corresponding query languages and reduces their performance [88]. The distinction, however, is of no real importance for our purposes, as we can always replace collection of edges with lists of edges in the definition of $S_t$, and collection types with list types in the type system.

### 9.2.2 Attributes

XML makes a clear distinction between elements and attributes, although they are similar forms of labelled representation.

An attribute appears always in relation to an element, and has a unique name in the scope of that element. Attribute values are always enclosed in quotation marks, and have a different meaning depending on the type with which they are declared in the document's schema, if one exists. Specifically, attributes can be of type:

- `STRINGTYPE` or `ENUMERATION`: their value is interpreted respectively as one or many tokens of free text;
- `ID`, `IDREF` and `IDREFS`: they are used to enhance the representation of relationships between data beyond simple nesting (e.g. shared and cyclic data); an attribute of type `ID` specifies a name for the element that is unique across the document and can be referenced by attributes of type `IDREF` (single reference), or `IDREFS` (multiple references).

As to the modelling of attributes, most approaches distinguish according to the type. An attribute of type `STRINGTYPE` is usually associated with the oid that models the associated element; it is then lexically distinguished from an element, in query expressions of related languages. Attributes of type `ENUMERATION` are usually ignored.

Matching attributes of type `ID` and `IDREF/IDREFS` instead, are interpreted together as edges between matching vertices. Such edges originate from the oid associated with the referencing attribute, are labelled with the name of the attribute, and reach the oid associated with the element containing the reference attribute. Of course, the correct interpretation of attributes of type `ID`, `IDREF`, and `IDREFS` depends on the availability of a documents DTD, or the like. Without a schema, `ID` attributes are usually discarded and `IDREF/IDREFS` attributes can only be interpreted as strings, as the *literal mode* of [62].

In our prototype, we adopt the latter strategy and avoid reference/referencing attributes. Moreover, we view attributes of type `STRINGTYPE` and `ENUMERATION` as separate, childless elements. We do this because we are not constrained by the need of recovering the XML document from the associated SSDB. For an example of the mapping, see Figure 9.2.


Figure 9.2: Example of mapping for attribute.

### 9.2.3 Elements with mixed content

An XML element has either an element content, when it contains sub-elements but not free text, or a mixed content, when it contains free text optionally intermixed

with other elements.

The mapping of elements with mixed content requires some arbitrary decisions to be made. However, although such elements appear quite often in XML documents, the issue is usually ignored in most approaches. Consider, for example the simple fragment of XML in Figure 9.3.

```
<AUTHOR> Richard Connor
<DEPT> Computer Science </DEPT>
</AUTHOR>
```

Figure 9.3: Example of mixed elements

Intuitively, we would associate the text `Richard Connor` with a childless oid and draw and edge to this oid from the oid associated with the element `AUTHOR`. The problem arises when we consider the label of such an edge, as we do not have one. We solve this problem by labelling these problematic edges with the name of the element associated with their source oid. Such choice can be justified on a semantic ground, as element names are meant to identify the meaning of the content they surround.

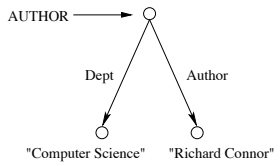



Figure 9.4: Example of graph representation for mixed elements.

For example, the XML fragment in figure 9.3 is modelled by the graph in figure 9.4. Intuitively, in terms of our SSD model, the XML fragment describes an entity corresponding to an author: the entity qualifies itself with name `Richard Connor` and with associated department that of `Computer Science`.

Notice that the same situation occurs when an element has some attributes of type `STRINGTYPE` or `ENUMERATION` as well as a content of sole text (see Figure 9.5).

### 9.2.4 Empty Elements

XML allows elements that have no content at all, i.e. empty elements. These are denoted by either a succession of start and end tags, such as `<author> </author>`, or by a single empty-element tag, namely `<author/>`. Due to the commitment to flexibility of XML, empty elements have an ambiguous semantics, where possible

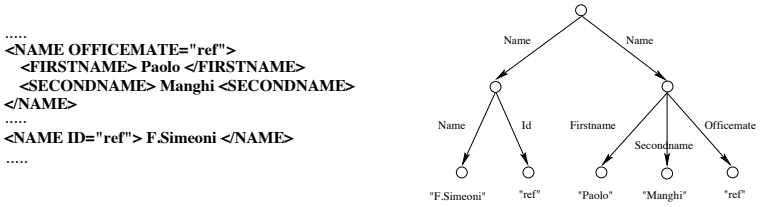



Figure 9.5: Example of mapping for attribute of text elements.

interpretations are that they describe truth or stand as placeholders in partial information. While the first interpretation implies a misuse of modelling concepts, as meta-data would be used as data, the second recalls *null* values in relational databases and is at least questionable in this scenario. Although, we could not represent empty elements altogether, we found more appropriate to our mapping to choose the second interpretation and model empty elements as childless vertices with an associated value of *undefined*. This requires the non-influential addition of this new value to *Atomic* in the definition of $S_t$. In particular, *undefined* belongs to all types in $\mathbf{T}$. As a result, `<author/>` maps onto the SSDB illustrated in Figure 9.6.

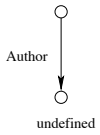



Figure 9.6: Example of graph representation for empty elements

Finally, Figure 9.7 shows the modelling of a fragment of XML that includes all the problematic features discussed above.

## 9.3 From CORBA IDL to types of $L$

The prototype of SNAQue performs extraction of Java objects with respect to the subset of IDL that maps onto the types in $T$.

The mapping onto atomic, record, and collection types is relatively straightforward. Indeed, our type language supports only integers and strings as atomic types, which map from *long* and *string* respectively in IDL. Moreover, record and collection types map from *struct* and *sequence* types respectively:

```
<HIPPO_RESEARCH_GROUP>
  <NAME>
    <FIRSTNAME> Paolo </FIRSTNAME>
    <SECONDNAME> Manghi <SECONDNAME>
    <HOMEPAGE></HOMEPAGE>
  </NAME>
  <NAME> F.Simeoni </NAME>
  <NAME> S.Neely </NAME>
  <COORDINATOR HOMEPAGE="www.cs.strath.ac.uk/~richard">
    <NAME> R.Connor </NAME>
  </COORDINATOR>
</HIPPO_RESEARCH_GROUP>
```

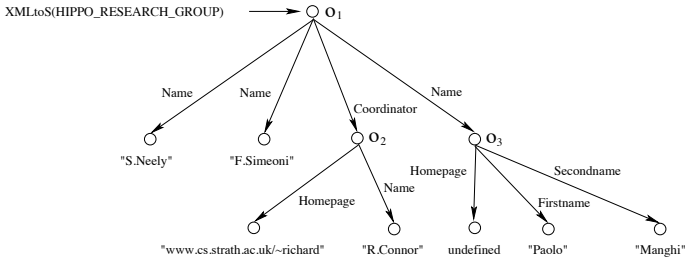



Figure 9.7: An example of our mapping from XML onto graphs.

$\texttt{IDLtoT(}\ \texttt{long}\ \texttt{)} = Int$
$\texttt{IDLtoT(}\ \texttt{string}\ \texttt{)} = String$
$\texttt{IDLtoT(}\ \texttt{struct l \{T}_1\ \texttt{l}_1\texttt{; ...; T}_n\ \texttt{l}_n\texttt{\}) } = [l_1 : T_1, \ldots, l_n : T_n]$
$\texttt{IDLtoT(}\ \texttt{sequence T l)} = [l : coll(T)]$

Union types pose a minor problem, as we made the decision to model with untagged unions in **T**, whereas IDL provides tagged unions. This led to a decision to either add tags to our own union types, which would present no special problem, or else to make some arbitrary decisions in the mapping. The latter course was chosen for purely pragmatic reasons.

We do not support mappings from IDL interface definitions. The purpose of such definitions is to model object-oriented classes, incorporating both data and behaviour. We do not, at present, expect behaviour definitions to be included within the source XML, and there is no sense in which they could be mapped to our data-description type language. [2]

[2]Notice however that, in object-oriented host languages, *struct* definitions are translated by IDL compilers into classes. The behaviour associated with these classes is simply to update and retrieve the values of instance variables corresponding to the given *struct* fields.

Finally, we require our IDL definitions to end with a `typedef` declaration, which declares a single type variable, and take this as the *main definition* for our purposes. This definition may rely upon others in the sequence; any in the sequence that are redundant with respect to this purpose are ignored. An example of SNAQue-valid IDL definition is illustrated in Figure 9.8.

```
struct bustype { string coachbuilder; long noOfSeats };
struct cartype { string make; long topSpeed };
typedef union { bustype bus; cartype car } vehicle;
```

Figure 9.8: Example of main definition.

Note that, once a value is extracted according to a main definition, SNAQue needs to define an IDL interface to serve it across the ORB. As earlier mentioned, this interface will simply introduce a method `getData` that returns an object of the type defined in the main definition.

### 9.3.1 Sequences and Unordered Collections

As mentioned previously, there is an important semantic difference in the treatment of collection types in the core system and both XML and CORBA IDL. The collection type of the core system represents an unordered collection, in keeping with the graph-based semantic model. In XML, the order in which tags appear within the text is significant, giving a repeated tag the semantics of an ordered collection or list. Equally, the sequence type constructor in IDL translates in various languages to ordered bulk constructors, for example to array in Java.

However, we need take no further account of this in the implementation. Although our core system has an unordered semantics, it is implemented on a machine by a representation which has an implicit ordering. So long as our algorithms for parsing and extracting the data work from beginning to end in a consistent manner, then the representation of the extracted subset will preserve the same order as the XML input. When this data is lifted and placed in a Java array behind the generated interface, once again the order is preserved, giving a total preservation of order from the original XML through to the CORBA interface, and then through whatever language is used to access it.

### 9.3.2 Unions

There are two possible semantic mappings from the labelled unions of IDL to XML data.

The first, which we adopt, assumes that the labels are not significant within the original data, but are instead only used as a programming aid to test for the structure of the underlying data. This interpretation allows a direct mapping from

IDL labelled unions onto the untagged unions of our type system, as the retrieved data perfectly matches the semantics of both types.

The second is that the labels are significant entities within the XML, and the union construct can then be used to abstract over different tags, rather than different structures.

For example, the main definition `vehicle` in Figure 9.8, could be expected to describe either of the XML examples:

```
<VEHICLE>
<COACHBUILDER>Alexander</COACHBUILDER>
<NOOFSEATS>43</NOOFSEATS>
</VEHICLE>

<VEHICLE>
<BUS>
<COACHBUILDER>Alexander</COACHBUILDER>
<NOOFSEATS>43</NOOFSEATS>
</BUS>
</VEHICLE>
```

Our choice, which is purely arbitrary, is the first:

$$\texttt{IDLtoT}(\texttt{union}(\texttt{T}_1\ \texttt{l}_1;\ \texttt{T}_2\ \texttt{l}_2)) = \texttt{T}_1{+}\texttt{T}_2$$

The identifier `bus` is then significant in the final query code, where it is used as a shorthand for the expected structure.

## 9.4 CORBA object instantiation

In the event of a successful call to $\texttt{Extraction}_t$, with $\texttt{XMLtoS}(xml)$ and $\texttt{IDLtoT}(idl)$, returning a value $d$, $\texttt{EXTR}_{IDL}$ must create a Java object $j$ such that $\texttt{JAVAtoD}(j) = d$. Once $j$ is created, SNAQue generates a CORBA object that serves the extracted value across the ORB framework.

Both operations, illustrated in Figure 9.1, depend on the generation of an IDL interface $I_{idl}$ corresponding to $idl$. $I_{idl}$, which has an automatically generated unique name, defines a single method `getData`, whose return type is $idl$. For example, the IDL definition:

```
struct person {string name};
struct data {long age};
typdef sequence <person> people;
```

extracts a value $d$ of the form illustrated in Figure 9.9, and translates in the interface:

```
struct person {string name};
typdef sequence <person> people;
interface Getting_Data28103
{
  people getData()
}
```

where the IDL structure `data` has been removed as it was not reachable from the main definition.

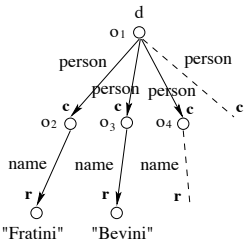



Figure 9.9: Extracted value

Based on $I_{idl}$, SNAQue instantiates the CORBA object as follows:

1. compiles $I_{idl}$ with *JIDL*, a compiler from IDL to Java, and gets the Java class skeleton for $I_{idl}$ and the Java classes for *idl*;
2. creates a Java object $j$, an instance of the classes for *idl*, that corresponds to $d$;
3. creates the CORBA object corresponding to $I_{idl}$: that is a Java object with a unique method `getData` returning the Java object $j$.

With reference to our example, the interface `Getting_Data28103` defines a sequence `people` of structures `person`, which in turn have one property `name`. When this interface is JIDLed, i.e. processed by a JIDL compiler, three Java source code files are produced along with a number of *helper* classes: the class skeleton `Getting_Data28103`, and the classes `peopleType.java` and `personType.java`.

The class `Getting_Data28103` corresponds to the interface and declares the method `getData()`, returning an object of type `peopleType`. The `peopleType` class contains a constructor and methods to access the elements of the sequence. The `personType` class contains a constructor and one attribute `name` of type string.

This Java source code is generated, manipulated, compiled, and executed by SNAQue to construct first $j$ and then the related CORBA object.

Achieving this requires dynamic class source code creation, method/class name lookup, and compilation. Programs must examine themselves and manipulate internal properties. Programming languages that support *reflection* can achieve this. Reflection is a language mechanism which enables computations to generate language code, compile it and execute it at run-time.

SNAQue uses the Java reflection API, which provides the means for determining the fields, constructors and methods of Java objects and classes, for:

- *creating the Java object*: Java reflection API offers the methods `forName`, `getMethod`, and `getReturnType`, which are used by the system to discover the return type of the `getData` method of the skeleton. In our example this type is the class `peopleType`. The constructors of this class, and arguments to these constructors, are also determined by reflection. The object $j$ is generated by recursively running through the structure of $d$, in conjunction with the structure of each class definition, and calling the appropriate class constructors as the recursion unwinds.

  This process is based on the static constraint governing the nature of $d$ so that it conforms in structure to *idl*, hence to the Java classes. In our example, the system gets and uses the constructors of the classes `peopleType` and `personType`. A new element of an array of class `peopleType` is created for each oid reachable from the root of $d$ (see Figure 9.9) with a label *person*, i.e. $o_2$, $o_3$, and alike. Furthermore, each of these oids must have an outgoing edge labeled as *name*, which corresponds to the instance variable `name` of objects of the class `personType`. Such variable can then be instantiated with the target values of these edges, i.e. *Fratini*, *Bevini* and so on.

- *creating the CORBA object*: the CORBA object is generated by compiling and instantiating the Java class skeleton corresponding to the given IDL interface, where the method `getData` is filled in by SNAQue with Java code that simply returns the newly created object $j$. In our example, `getData` is completed with code that returns the Java object of class `peopleType` constructed according to the process exemplified above.

Once the CORBA object has been instantiated, any CORBA client can access it through the corresponding stub and operate over the extracted value by invoking the method `getData`.

At the current stage of development, the execution of SNAQue is requested by an invocation to a CORBA object. In particular, both the XML data and the IDL definition reside on the server side of the CORBA gateway. A client program makes a call, through a stub to this object, to a distinguished method at the server. The server executes this method, which performs the projection of the XML data onto

the given IDL definition. Future versions of SNAQue, will provide a Web interface through which users can specify an XML URI, an IDL definition, and perform the extraction they need.

## 9.5 Experience with the prototype

The initial data set was chosen from the numerous biodiversity database projects undertaken by the Palaeobiology Research Group at the University of Glasgow. A test load, documenting the major groups of marine organisms that colonised Tropical America over the last 30 million years, has especially been identified as suitable for experimentation within the approach proposed. This data includes several thousand genera and is insufficiently regular to allow the use of conventional database technology to address the range of queries required. Whilst the bulk of the data is highly structured taxonomic information, there are significant amounts of less structured data that should be included. For example, morphological characteristics, which vary widely in the different groups of organisms, and other crucial data such as population densities, images, life habits and habitats. The data as we received it is in the form of a treatise: it has no explicit structure in terms of tags or labels but it does have a high degree of implicit structure. This implicit structure exists because of a standard format developed by palaeobiologists, over several hundred years, for documenting their findings. Given an understanding of this standard format it was possible to write a parser that converted the plain text into XML. A section of the plain text follows:

> **?Apsilingula** WILLIAMS, 1977, p. 403 [*A. parkesensis; OD]. Elongate oval with subparallel lateral margins; dorsal and ventral pseudointerareas poorly known; both valves strongly thickened posteriorly, with deeply impressed muscle scars; ventral visceral area extending to midvalve; transmedian scars possibly

A portion of the corresponding XML, as produced by the parser, is shown in Figure 9.10. The resulting XML is structured but irregular; for example, not every genus has a *shell*. wInformation of this kind is of vital importance to the study of biodiversity, in terms of analysing extinction patterns, and is still normally gathered by manual inspection of the text of the treatise.

Note, however, that every genus has a *name*, an *author* and a *date*. We use these fields for an example application, which is to print a list of all these attributes for each genus in order of the recorded dates. Operations of higher complexity could be performed over these data, which include the insertion in a DBMS and/or their combination with data stored in other DBMSs or SSDBs.

As a first step, we illustrate the process of extraction of a regular subset of our XML SSDB corresponding to an object with IDL,

```
<FAMILY>
   <GENUS>
      <NAME confirmed="false">Apsilingula</NAME>
      <AUTHOR>WILLIAMS</AUTHOR>
      <DATE>1977</DATE>
      <PAGE>p.  403</PAGE>
      <SYNONYM>[*A. parkesensis; OD]</SYNONYM>
      <DESCRIPTION>
      <SHELL>Elongate oval with subparallel lateral margins</SHELL>
      <VALVES>both valves strongly thickened posteriorly </VALVES>
      ⋮
   </GENUS>
   ⋮
</FAMILY>
```

Figure 9.10: XML source produced by the parser.

```
struct nametype {string confirmed; string name;};
struct genustype {nametype name; string author; long date;};
typedef sequence <genustype> genus;
```

which translates in the type,

```
[ genus:  coll([author:  string;
                name:    [confirmed:  string; name:  string];
                date:    int ])
]
```

Figure 9.11 shows the value $d$ extracted from the SSDB in Figure 9.10.

SNAQue returns a CORBA object, implemented in Java as an object of class `Getting_Data10`, that serves the value $d$ across the ORB. Client programmers that want to use this CORBA object, must first compile its IDL interface, in order to get a Java class stub to the object. Then, around the stub, they can write and compile the code of consuming applications.

For simplicity, we give an example of a Java client in Figure 9.12.[3] We assume the IDL interface has been compiled, and that *xmlObj* (line 1) is the name reference to the CORBA object across the ORB framework.

The client coerces, i.e. *narrows*, the reference *xmlObj* to a corresponding Java object of type `Getting_Data10` (line 1). In particular, *xmlObj* is narrowed to $p$, which becomes the reference to the CORBA object for the consuming applications.

[3]Note that our client was written in Java just for simplicity, while *xmlObj* could have been coerced into an object of any CORBA-compliant language.

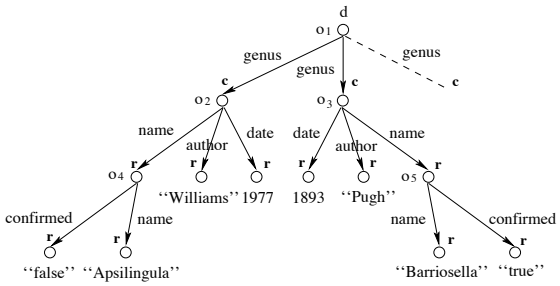



Figure 9.11: Extracted value $d$

Indeed, calling the `getData` method (line 2) of $p$ returns an array of `genustype` objects as defined by the IDL and in the mapping from CORBA to Java.

The array of `genustype` objects contains Java objects corresponding to the two structs in the IDL. The code outputs all the data in the array (line $3-5$) and then uses a quicksort algorithm to sort them by date (line 6). The sorted array is then displayed (lines $7-9$).

```
1) Getting_Data10 p = Getting_Data10Helper.narrow(xmlObj);
2) getting10.Getting_Data10Package.genustype[] obj_data = p.getData();
3) System.out.println("Name, Author, Year");
4) for (int i=0; i<obj_data.length; i++)
5)     System.out.println(obj_data[i].name.name + ", " +
                          obj_data[i].author + ", " +
                          obj_data[i].date + ".\n");
6) QuickSort(obj_data, 0, obj_data.length-1);
7) System.out.println("\nSorted data:");
8) for (int i=0; i<obj_data.length; i++)
9)     System.out.println(obj_data[i].name.name + ", " +
                          obj_data[i].author + ", " +
                          obj_data[i].date + ".\n");
```

Figure 9.12: Client query in Java.

SNAQue identifies a novel concept that we believe to be of significant importance within the field of querying XML data. Rather than adopting the more common approach of designing a special-purpose query language for the domain in particular, SNAQue provides a semantic model through which the information encoded in XML

can be exported into the domain of traditional programming systems. We do not claim that this approach is in any sense better, but that it is complementary in that it is stronger for certain classes of problem. In particular it seems to be well-suited to querying tasks which require generalised computation, especially over large sets of data.

Efficiency aspects of the approach are also clearly important; we believe in pragmatic terms that the subset creation is quite tractable. However, we are determined to go through a deep analysis of performance as long as SNAQue will reach a usable version, so as to provide precise results and claims about usability and effectiveness of our approach.

There are many future directions for this research topic. CORBA has been chosen as a convenient proof of concept mechanism, but the same approach can be used for any programmable domain that is based around a static typing discipline. For example it seems an interesting idea to use a similar mechanism to translate from XML into a relational database.

# Chapter 10

# Conclusions and Future Issues

The main contribution of this thesis is to have laid the foundations of mechanisms enabling applications of typed programming languages to indirectly compute over regular subsets of SSDBs. In itself, this result is a step forward towards the integration of SSD with structured data and typed programming languages.

Known approaches for typing SSDBs either statically impose a type and constrain data population or infer a type from the database after data population. The first offer the advantages of static typing at the cost of limiting the number of irregularities in the database. The second, while enabling unconstrained database population, offer only part of these advantages. Our approach is complementary to these, as it fully recovers the benefits of static typing after data population. We believe that in combination with navigational approaches, extraction mechanisms may provide complete functionality for SSD.

In particular, this work provides a specification for the realisation of an extraction mechanism in a typed programming language. Any language can be associated with a notion of extractability, which captures the concept of value extractable from an SSDB according to a given type. Based on extractability, an extraction algorithm can thus be constructed. Moreover, algorithm's correctness can be proven verifying specific properties of soundness and completeness with respect to extractability.

In order to show the feasibility of extraction mechanisms for most programming languages currently in use, we have realised an extraction mechanism for a representative typed language $L$ featuring a set of standard types. First, we have defined extractability of $L$ and then given a corresponding algorithm `Extraction`. Given an SSDB and a type of $L$, `Extraction` returns an extractable value, along with the measures of precision and data capturing. By interpreting these measures, users may be able to improve the results of their subsequent extractions.

`Extraction` is proven to be sound with respect to extractability for $L$, but not complete. While soundness is fundamental for extraction algorithms, however, we have shown that in many application contexts the incompleteness of `Extraction` has no relevant impact. This is the case for tree-structured SSDBs, for which we

provide a specific complete algorithm $\texttt{Extraction}_t$.

Finally, as a practical example of an extraction mechanism, we presented the system SNAQuE. SNAQuE extracts Java objects from XML SSDBs and is built around the algorithm $\texttt{Extraction}_t$. Indeed, the system focuses on XML documents interpreted as tree-structured SSDBs and a subset of Java types that matches the types of $L$. A prototype of SNAQuE has been developed and is currently under improvement at the Computer Science Department of Strathclyde University.

We believe that important and immediate enhancements to our work should be the following:

**Complete algorithm** Compared with others, which typically disregard these issues, the query methodology derived from the usage of `Extraction` enables applications to indirectly operate over SSDBs with shared oids. On the other hand, the presence of shared oids is the principal cause of algorithm's incompleteness. We have seen that incompleteness is due to quite specific combination of types and SSDBs and does not generally represent a problem for users. However, providing a reliable version of the algorithm would certainly be a stronger result. Therefore, an important future issue is that of realising a version of `Extraction`, only suggested in this work, which proves to be complete with respect to extractability.

**Formal definition of extractability for a language** Extractability is based on a mapping from values of the language to SSDBs and on a definition of inclusion between SSDBs. While the mapping is strictly related with the target language, the inclusion relation is language-independent. Indeed, the behaviour of an extraction mechanism for a specific language varies depending on the form of inclusion adopted.

An algebra for the definition of inclusion relations could be a useful tool for extraction mechanism designers. Such an algebra, for instance, may support operators for the definition of equivalences between different sets of labels or for the construction of particular structural associations between SSDBs. Hence, designers could first define a mapping from SSDBs onto the language values, and then construct a customised definition of inclusion depending on the specific SSD scenario.

**Measures** When successful, `Extraction` returns also the measures of precision and data capturing. We have seen that their importance is that of disclosing to users the backstage of the extraction process. As a further improvement, other measures may be conceived to provide different forms of feedback to the users. For example, users may be interested in a single quantity qualifying the overall precision, rather than relying on a list of record-specific quantifications.

**Experiments** Due to the absence of a working implementation, the need for extraction systems stands on a strong claim only. Thus, once completed, the SNAQuE system will be deeply tested so as to provide evidence for the usability and effectiveness of our approach.

In summary, extraction mechanisms bridge the SSD domain to typed programming language domains. Accordingly, they become useful tools in many application contexts, as both extensions to extant programming environments or platforms for the construction of new SSD-based programming systems. In the following we show some ideas which may give rise to novel research avenues.

## 10.1 Customised extraction mechanisms

The benefits of an extraction mechanism depend on the host language. In this work, for example, we have described the system SNAQuE, which extracts Java objects from SSDBs. This gives users the immediate advantage of writing type-correct Java applications over subsets of SSDBs, as well as specifying computations over high-level types rather than over simple labelled graphs. A further peculiarity of SNAQuE is that Java objects can be served as a CORBA service across the CORBA ORB framework.

Moreover, through the appropriate interfaces with various DBMSs, Java applications may become the means to populate a DBMS with regular subsets of SSDBs. This way, SSDBs could be queried with the benefits of DBMSs. However, extraction mechanisms could be devised to directly inject SSD into relational DBMSs. Here, other issues come into play, such as how to extract relationships, specified via external keys, from SSDBs.

## 10.2 Persistence and extraction mechanisms

In *Persistent Programming Languages* [13, 17, 75, 48, 16], computations store values together with their types into a *persistent store*. Values thus survive the applications that created them and may be retrieved by any run-time computation by performing an *intern* operation. Such operation takes a reference to the value, i.e. a name, and a type. If the value type matches the given type, the values is safely imported into the interested computation.

Similarly, an extraction mechanism checks for the existence of a regular SSDBs that corresponds to a value of a given type, and possibly returns such value. Accordingly, extraction mechanisms could be naturally hosted in a persistent language, where programmers are used to accessing values through dynamic controls.

A more ambitious project is that of realising a persistent language around a *semistructured persistent store*. Here, a persistent store would become an SSDB

whose data can be retrieved by high-level applications exploiting an extraction mechanism. Similarly, applications would persistently store the data they create according to a corresponding SSD representation. The persistent store could thus be queried over by means of SSDQLs. In particular, by adopting an XML representation of the store, persistent data could be also accessed via HTTP and visualised with a browser.

This idea suggests the further realisation of a persistent language integrating under the same binding mechanism, high-level commands with commands for the manipulations of SSDBs in a navigational fashion.

## 10.3 Databases data-first design

*Schema-first* techniques for DBMSs design (see Chapter 2) are not well suited to the creation of databases for complex application domains, such as scientific, geographical, financial, and multimedia domains. This is due to the fact that major structural changes to the schema of a populated database may be turn out to be critical.

Extraction mechanisms suggest a novel design technique, which may be apt to these particular scenarios. Data collections are built in an untyped fashion as SSDBs and become constrained by a schema only at a later stage. Precisely, users populate an SSDB with their relevant data and incrementally project relational or object-oriented schemas assumptions onto the database. Such process terminates when and if a satisfactory binding between schema and the SSDB can be reached.

These techniques are currently under investigation at the Department of Computing and Information Science of Strathclyde University.

# Bibliography


[1] S. Abiteboul. Querying semi-structured data. *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece*, pages 1–18, January 8-10 1997.

[2] S. Abiteboul, R. Goldman, T. Lahiri, J. McHugh, and J. Widom. Ozone: Integrating structured and semistructured data. Technical report, Stanford University Database Group, 1998.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistuctured Data. *Journal of Digital Libraries, 1(1)*, pages 68–88, April 1997.

[4] S. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[5] P. Aczel. An introduction to inductive definitions. *In Handbook of Mathematical Logic J. Barwise ed.*, pages 739–782, 1977.

[6] P. Aczel. *Non-well-founded sets.* CSLI Lecture Notes. Stanford, CSLI Publications 14, 1988.

[7] B. Adelberg. NoDoSE: A tool for semi-automatically extracting structured and semi-structured data from text documents. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 283–294, New York, June 1–4 1998. ACM Press.

[8] B. Adelberg and M. Denny. Nodose version 2.0. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMod-99)*, volume 28,2 of *SIGMOD Record*, pages 559–561, New York, June 1–3 1999. ACM Press.

[9] V. Aguilera, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying xml documents in xyleme. In *Proceedings of ACM SIGIR 2000 Workshop On XML and Information Retrieval*, Athens, Greece, July 2000.

[10] A. Albano, D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. A type system for querying XML documents. *SIGIR00-XML ACM Workshop held in conjunction with 23rd International SIGIR Conference on Research and Development in Information Retrieval, Athens, Greece*, 2000.

[11] A. Albano, D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. A typed query language for XML documents. Preliminary Draft, 2000.

[12] A. Albano, D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. A text retrieval query language for XML documents. *Special Topic Issue of JASIS on XML and Information Retrieval*, August 2001.

[13] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *Journal of Very Large Data Bases*, 4(3):403–444, 1995.

[14] G. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems(TOPLAS), 15(4)*, pages 575–631, 1993.

[15] Arnaud Sahuguet. Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask. In *WebDB-2000*, 2000.

[16] M. Atkinson and M. Jordan. A review of the rationale and architectures of pjama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform. Technical report, Sun Microsystems Laboratories, CA, USA, Last revised Sept. 2000.

[17] M. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[18] P. Atzeni and G. Mecca. Cut and Paste. In *Sixteenth ACM SIGMOD International Symposium on Principles of Database System (PODS '97)*, 1997.

[19] J. Barwise and J. Etchemendy. *The Liar. An Essay on Truth and Circularity*. Oxford, Oxford University Press, 1987.

[20] J. Barwise and L. Moss. *Vicious Circles. On the Mathematics of Non-Wellfounded Phenomena*. CSLI Lecture Notes Number 60. Stanford, CSLI Publications, 1996.

[21] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes. Technical report, World Wide Web Consortium, Dec. 1999. W3C Working Draft.

[22] M. Brandt. Recursive subtyping: Axiomatisations and computational interpretations. *Master Thesis*, 1997.

[23] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae XX*, pages 1–24, 1998.

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

[25] P. Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 117–121. ACM Press, 1997.

[26] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. *Lecture Notes in Computer Science*, 1186:336–350, 1997.

[27] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimisation techniques for unstructured data. *In Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[28] P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proceedings of 5th International Workshop on Database Programming Languages*, Gubbio, Italy, September 1995.

[29] P. Buneman, M. Fernandez, and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.

[30] P. Buneman and B. Pierce. Union types for semistructured data. In *Internet Programming Languages*. Springer, Sept. 1998. Proceedings of the International Database Programming Languages Workshop. LNCS 1686.

[31] P. Buneman and B. Pierce. Union types for semistructured data. Technical report, University of Pennsylvania, 1999.

[32] J. Campbell, D. Grossman, and A.-M. Popescu. Quilt2Sql - An XML Storage Schema and Query Engine for the Quilt Query Language. 2000.

[33] L. Cardelli and G. Ghelli. A query language for semistructured data based on the ambient logic. Manuscript, April 2000.

[34] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, jun 2001. W3C Working Draft.

[35] D. Chamberlin, D. Florescu, and J. Robie. Quilt: an XML query language for heterogeneous data sources. In *Proceedings of WebDB*, Dallas, TX, May 2000.

[36] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, 1994.

[37] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 313–324. ACM Press, 1994.

[38] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of theACM SIGMOD International Conference on Management of Data*, volume 25, 2 of *ACM SIGMOD Record*, pages 413–422, New York, June 4–6 1996. ACM Press.

[39] V. Christophides, S. Cluet, and J. Siméon. Semistructured and Structured Integration Reconciled: YAT += Efficient Query Processing. Technical report, INRIA, Verso database group, November 1998.

[40] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Technical report, World Wide Web Consortium, Nov. 1999. W3C Recommendation.

[41] S. Cluet. Modeling and querying semi-structured data. *SCIE '97*, 1997.

[42] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In M. Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*, pages 383–392. ACM Press, 1992.

[43] S. Cluet, C. Delobel, J. Siméon, and K. 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.

[44] S. Cluet and G. Moerkotte. Nested queries in object bases. In C. Beeri, A. Ohori, and D. Shasha, editors, *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993*, Workshops in Computing, pages 226–242. Springer, 1993.

[45] S. Cluet and J. Siméon. Data integration based on data conversion and restructuring. Technical report, INRIA, Verso database group, October 1997.

[46] R. Connor, D. Lievens, P. Manghi, S. Neely, and F. Simeoni. Extracting typed values from XML databases. *OOPSLA'01 Workshop on Objects, <XML> and Databases, Tampa Bay, Florida, USA*, October 2001.

[47] R. Connor, D. Lievens, P. Manghi, S. Neely, and F. Simeoni. An Approach to High-Level Language Bindings for XML. *Elsevier Journal on Information and*

*Software Technology, Special Issue on Object, XML and Databases*, To appear on spring 2002.

[48] R. Connor, K. Sibson, and P. Manghi. On the unification of persistent programming and the world wide web. *Workshop on the Web and Data Bases (WebDB98) held in conjunction with EDBT'98 (Springer Verlag, Vol. 1590, LNCS series)*, 1998.

[49] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. Technical report, World Wide Web Consortium, Aug. 1998. Submission to the World Wide Web Consortium.

[50] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMod-99)*, volume 28,2 of *SIGMOD Record*, pages 431–442, New York, June 1–3 1999. ACM Press.

[51] R. Durbin and J. T. Mieg. *ACeDB – A C. elegans Database: Syntactic definitions for the ACeDB data base manager*, 1992. `http://probe.nalusda.gov:8000/acedocs/syntax.html`.

[52] D. C. Fallside. XML Schema Part 0: Primer. Technical report, World Wide Web Consortium, Oct. 2000. W3C Candidate Recommendation.

[53] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. The XML Query Algebra. Technical report, World Wide Web Consortium, Dec. 2000. W3C Working Draft.

[54] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. STRUDEL: A Web site management system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):549–560, 1997.

[55] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a Web-site management system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(3):4–11, September 1997.

[56] M. Fernandez and J. Robie. XML Query Data Model. Technical report, World Wide Web Consortium, May 2000. W3C Working Draft.

[57] M. Fernandez, J. Siméon, and P. Wadler. XML Query Languages: Experiences and Exemplars, December 1999. Manuscript available from `http://www-db.research.bell-labs.com/user/simeon/xquery.ps`.

[58] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas (full version), February 1997. Manuscript available from `http://www.research.att.com/~{mff,suciu}`.

[59] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the Boat with Strudel: Experiences with a Web-Site Management System. In Haas and Tiwary [65], pages 414–425.

[60] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 14–23. IEEE Computer Society, 1998.

[61] D. Florescu and D. Kossmann. Storing and querying XML data using an rdbms. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[62] R. Goldman, S. Chawathe, A. Crespo, and J. McHugh. A Standard Textual Interchange Format for the Object Exchange Model (OEM). Technical report, Stanford University, 1995.

[63] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proceedings of the second International Workshop WebDB '99, Pennsylvania*, June 1999.

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

[65] L. M. Haas and A. Tiwary, editors. *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. ACM Press, 1998.

[66] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report), May 2000. WebDB workshop.

[67] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proceedings of the 16th International Conference on Data Engineering, 28 February - 3 March, 2000, San Diego, California, USA*, 2000.

[68] W. Kim. On optimizing an SQL-like nested query. *TODS*, 7(3):443–469, 1982.

[69] N. Kushmerick, D. S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 729–737, San Francisco, Aug. 23–29 1997. Morgan Kaufmann Publishers.

[70] A. Malhotra, J. Robie, and M. Rys. XML syntax for XQuery 1.0 (XQueryX). Technical report, World Wide Web Consortium, June 2001. W3C Working Draft.

[71] J. McHugh and J. Widom. Compile-time path expansion in Lore. Technical report, Stanford University Database Group, November 1998.

[72] G. Mecca, A. O. Mendelzon, and P. Merialdo. Efficient Queries over Web Views. Technical Report 2, Araneus Project Working Report, AWR-2-99 (version 1.0 - March, 16, 1999.

[73] Milo and Suciu. Index structures for path expressions. In *ICDT: 7th International Conference on Database Theory*, 1999.

[74] F. D. Monte. Meccanismi per l'evoluzione dello schema nel linguaggio Fibonacci,. Master's thesis, Universitá degli Studi di Pisa, 1995.

[75] R. Morrison, R. Connor, G. Kirby, D. Munro, M. Atkinson, Q. Cutts, A. Brown, and A. Dearle. The Napier88 persistent programming language and environment. In *Fully Integrated Data Environments*, pages 98–154. Springer, 1999.

[76] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(4):39–52, 1997.

[77] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In Haas and Tiwary [65], pages 295–306.

[78] S. Nestorov, J. Ullman, J. Weiner, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchical data. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 79–90, Birmingham, UK, Apr. 1997. IEEE.

[79] OMG. CORBA OMG IDL text file - The Object Management Group, 1999. ftp://ftp.omg.org/pub/docs/formal/99-04-01.txt.

[80] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. *Lecture Notes in Computer Science*, 1013:161–180, 1995.

[81] Y. Papakonstantinou, J. Widom, and H. Molina. Object exchange across heterogeneous information sources. *Proceedings of IEEE Int. Conference on Data Engineering, Birmingham, England*, 1996.

[82] D. Quass, A. Rajaraman, Y. Sagiv, and J. Ullman. Querying semistructured heterogeneous information. *Lecture Notes in Computer Science*, 1013:319–331, 1995.

[83] D. Raggett, A. L. Hors, and I. Jacobs. HTML 4.01 specification. Technical report, World Wide Web Consortium, Dec. 1999. W3C Recommendation.

[84] J. Robie, J. Lapp, D. Schach, M. Hyman, and J. Marsh. XML Query Language (XQL). Note to the W3C XSL Working Group, Sept. 1998.

[85] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[86] J. Siméon and S. Cluet. Using yat to build a web server. In *International Workshop on the Web Database (WebDB '98), Valencia, Spain*, March 1998.

[87] L. D. Stein and J. Thierry-Mieg. AceDB: A genome database management system. *Computing in Science and Engineering*, 1(3):44–68, May/June 1999.

[88] D. Suciu. From Semistructured Data to XML, 1999. VLDB Tutorial.

[89] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelson. XML Schema Part 1: Structures. Technical report, World Wide Web Consortium, Dec. 1999. W3C Working Draft.

[90] R. G. J. Widom. Enabling query formulation and optimization in semistructured databases. pages 436–445. VLDB, 1997.

[91] XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`.