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.

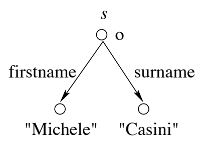

Figure 4.1: SSDB as expressive as the language value d.

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.

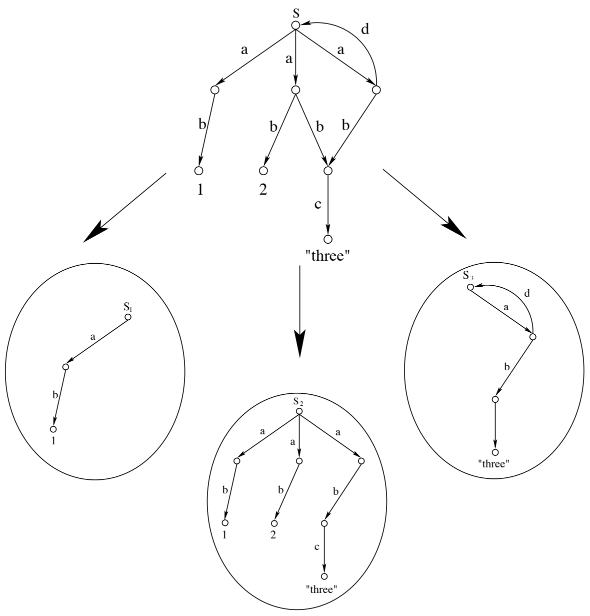

Figure 4.2: Extraction of regular subsets

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.

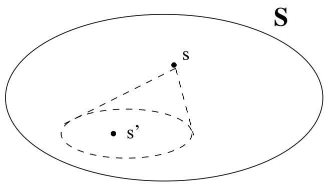

Figure 4.3: Inclusion relation

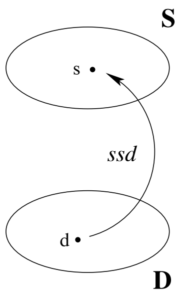

Figure 4.4: Mapping from typable values to SSDBs

2. a mapping $ssd : D \to S$, 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,*

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

4.2.2 Extraction algorithm correctness

The generic extraction algorithm,

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

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,

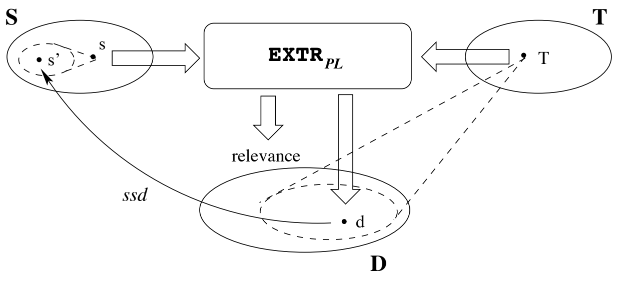

Figure 4.5: Extraction algorithm outline

Definition 4.2.3 *(Soundness of* EXTR$_{PL}$*) Let $s \in S$ and $T \in \mathbf{T}$. An extraction algorithm* EXTR$_{PL}$ *is* sound *if, every successful execution* 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* EXTR$_{PL}$*) Let $s \in S$ and $T \in \mathbf{T}$. An extraction algorithm* EXTR$_{PL}$ *is* complete *if, whenever $D_{s,T} \neq \emptyset$,* EXTR$_{PL}(s, T) = d$ *with $d \in D_{s,T}$.*

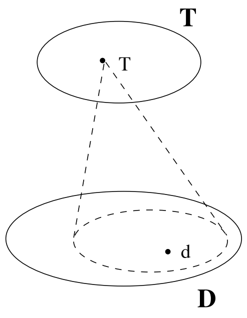

Figure 4.6: Typing relation

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.

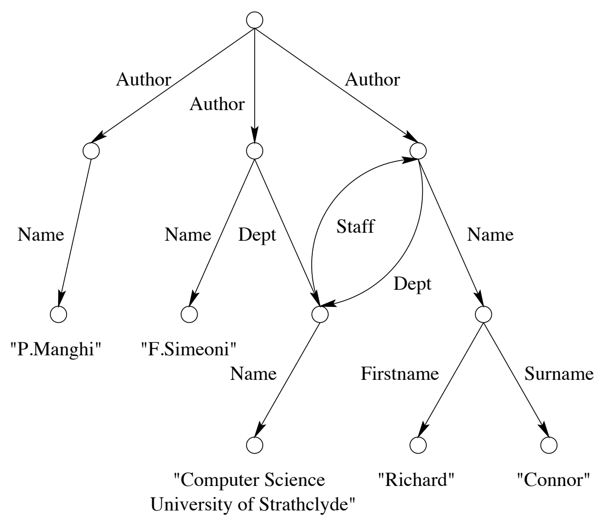

Figure 5.1: Graphical representation of a SSDB according to our data model

5.1 A formal definition of SSDBs

We define the domain S of SSDBs as a particular subset of the domain Γ 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 Γ 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 $\vec{e}' \in Oid$ then $\exists e \in s_e :\ h(\overleftarrow{e}') = \overleftarrow{e}\ \wedge h(\vec{e}') = \vec{e}\ \wedge label(e') = label(e)$*
2. *if $\vec{e}' \in Atomic$ then $\exists e \in s_e :\ h(\overleftarrow{e}') = \overleftarrow{e}\ \wedge\ \vec{e}' = \vec{e}\ \wedge label(e') = label(e)$*

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 $o_1 = o_2$.*

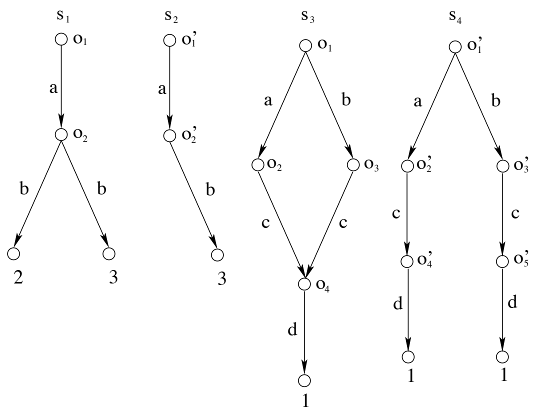

Figure 5.2: lt-included and bisimilar graphs

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.

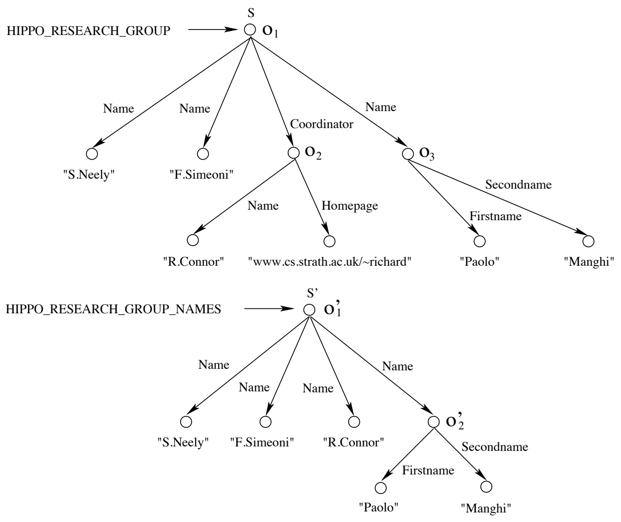

Figure 5.3: Other forms of inclusion.

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:

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

where $\forall i : 1, \dots, n.\ l_i \in Label$. 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)*

$$V \vdash int \qquad (integer)$$

$$V \vdash string \qquad (string)$$

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

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

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

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

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

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 μ-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 $[a : coll(T_1) + T_2]$ or $coll(T)$ 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 μ-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 μ-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, μ-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.

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

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

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

$$\frac{T_1 =_{\mathbf{T},w} T'_1 \quad , \dots , T_n =_{\mathbf{T},w} T'_n}{[l_1 : T_1, \dots , l_n : T_n] =_{\mathbf{T},w} [l_1 : T'_1, \dots , l_n : T'_n]} \qquad (RECORD-EQ)$$

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

$$\frac{T_1 =_{\mathbf{T},w} T'_1 \quad T_2 =_{\mathbf{T},w} T'_2}{T_1 + T_2 =_{\mathbf{T},w} T'_1 + T'_2} \qquad (UNION-EQ)$$

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

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 $\mathbf{T}$. 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 $\mathbf{T}$. 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 $\mathbf{T}$.

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 μ-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* $\mathbf{T}$ *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. $[l_1 : T_1, \dots, l_n : T_n]\ \mathcal{R}\ [l_1 : T_1', \dots, l_n : T_n'] \ \Rightarrow\ \forall i : 1, \dots n :\ T_i\ \mathcal{R}\ T_i'$
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 $x_1, \dots, x_n \Rightarrow x$ in the bisimulation definition are equivalent to a set of rules of the form,

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

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,

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

and the the pairs in the premises $x_1, \dots, x_n$ 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,

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

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

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 $A \vdash T =_{\mathrm{T}} T'$, that state that $T =_{\mathrm{T}} T'$ under the assumptions A.

Definition 6.2.4 *(Type equivalence rules)*

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

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

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

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

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

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

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

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

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

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

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 $T' \equiv \mu X.[a : [a : X]]$:

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

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 $\mathfrak{R}$ such that includes $T \ \mathfrak{R} \ T'$ 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.

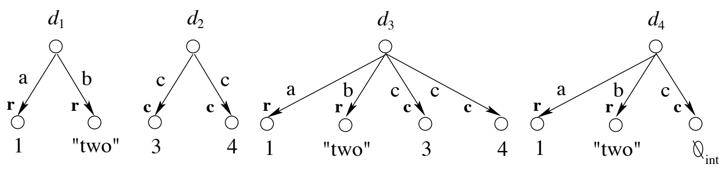

Figure 6.2: Examples of values

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

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

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. $T_i \not\equiv_{\mathbf{T}} coll(U) \ \Rightarrow\ (mark(me) = r \wedge \overrightarrow{me}\ \mathcal{R}_{ME}\ T_i)$
      ii. $T_i \equiv_{\mathbf{T}} coll(U) \Rightarrow\ (mark(me) = c \wedge \overrightarrow{me}\ \mathcal{R}_{ME}\ U)$.

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.

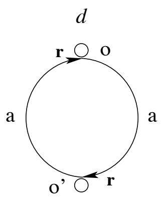

Figure 6.3: Value d

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 $[l_1 : T_1, \dots, l_n : T_n]$. Intuitively, in our settings $coll(U)$ should abstract over oids with a set of outgoing edges which target oids conforming to the type U. $[l_1 : T_1, \dots, l_n : T_n]$ 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-

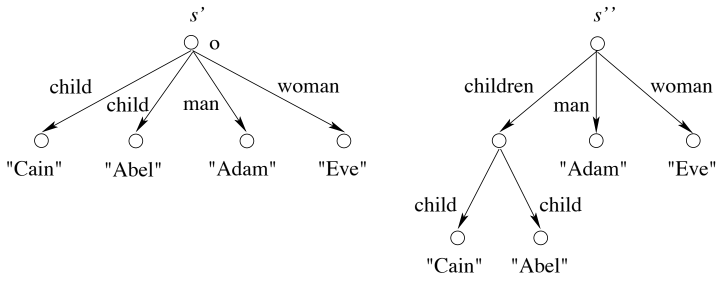

Figure 6.4: SSDBs of Adam and Eve's family

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 $s = (o, \{< o, \mathit{child}, ''\mathit{Cain}'' >, < o, \mathit{child}, ''\mathit{Abel}'' >)$, which clearly represents a collection value of type *string*, can only be extracted and typed as values of type $[\mathit{child} : \mathit{coll}(\mathit{string})]$. On the other hand, both s' and s'' in Fig. 6.4 can be successfully extracted with respect to the types $[\mathit{man} : \mathit{string},\ \mathit{child} : \mathit{coll}(\mathit{string})]$ and $[\mathit{man} : \mathit{string},\ \mathit{children} : [\mathit{child} : \mathit{coll}(\mathit{string})]]$.

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:

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

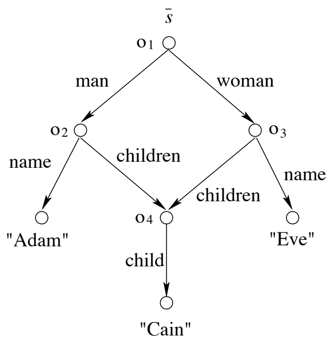

Figure 6.5: Ambiguous typing

Indeed, the edge $< o_4, child, ``Cain'' >$, 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

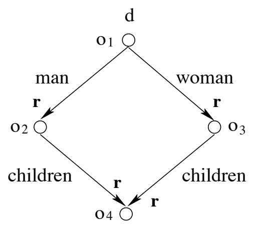

Figure 6.6: Ambiguous typing due to empty collections

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:*

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

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

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

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

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

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

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

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

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 μ-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 μ-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:*

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

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 `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 `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:*

$$T \sqsubseteq T \qquad (REF)$$

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

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

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

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

```
Proof: $\mathcal{P}_{fin}(Oid \times \mathbf{T}) \times \mathcal{P}_{fin}(Oid \times label \times M \times Obj^{+}) \times Obj^{+} \times \mathbf{T} \longrightarrow DT$

Case Proof($A \cup (o,T)$, $ME$, $o$, $T$)
  { return $\frac{true}{A \cup (o,T); ME \vdash o :: \mu T}$ $(HYP)$ }

Case Proof($A$, $ME$, $o$, $int$)
  { if $o \in Integer$ then return $\frac{o \in Integer}{A; ME \vdash o :: int}$ $(INT)$
                       else return $fail$ }

Case Proof($A$, $ME$, $o$, $string$)
  { if $o \in String$ then return $\frac{o \in String}{A; ME \vdash o :: string}$ $(STRING)$
                      else return $fail$ }

Case Proof($A$, $ME$, $\emptyset_{T'}$, $T$)
  { if $T' =_{\mathbf{T}} T$ then return $\frac{T' =_{\mathbf{T}} T}{A; ME \vdash \emptyset_{T'} :: T}$ $(EMPTYCOLL)$
                     else return $fail$ }

Case Proof($A$, $ME$, $o$, $\mu X.T$)
  { $DT :=$ Proof($A$, $ME$, $o$, $T\left[{}^{\mu X.T}/_{X}\right]$)
    return $\frac{DT}{A; ME \vdash o :: \mu X.T}$ $(REC)$ }

Case Proof($A$, $ME$, $o$, $T_1 + T_2$)
  { $DT :=$ Proof($A$, $ME$, $o$, $T_1$)
    if $DT \neq fail$ then return $\frac{DT}{A; ME \vdash o :: T_1 + T_2}$ $(UNION - L)$
                      else return $\frac{\texttt{Proof}(A,\ ME,\ o,\ T_2)}{A; ME \vdash o :: T_1 + T_2}$ $(UNION - R)$ }

Case Proof($A$, $ME$, $o$, $\overline{T}$), where $\overline{T} \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n]$
  { if $Label(ME(o)) \neq \{l_1, \ldots, l_n\}$
        then return $fail$  < exact match between the labels in the value and in the type >
    if $(\exists i : 1, \ldots, n.(T_i \equiv_{\mathbf{T}} coll(U) \ \wedge\ l_i \notin Label(ME(o)_c) \ \vee\ (T_i \not\equiv_{\mathbf{T}} coll(U) \ \wedge\ l_i \notin Label(ME(o)_r))$
        then return $fail$  < labels of $c$ ($r$) edges should be associated to (non) collection fields in the type >
    $DT := \emptyset$
    for $i := 1$ to $n$ do
        { if $T_i \equiv_{\mathbf{T}} coll(U)$
              then for all $me \in ME(o, l_i)_c$ do
                      $DT := DT$;Proof($A \cup (o, \overline{T})$, $ME$, $\vec{me}$, $U$)
              else for all $me \in ME(o, l_i)_r$ do
                      $DT := DT$;Proof($A \cup (o, \overline{T})$, $ME$, $\vec{me}$, $T_i$) }
    return $\frac{DT}{A; ME \vdash o :: \overline{T}}$ $(RECORD - COLL)$ }
```

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:*

$X^* = \{X\}$

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

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

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

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

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

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

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 $T' = [l_1 : T_1, \dots, l_n : T_n]$: 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.

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

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

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

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

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

■

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 $(\mathit{UNION} - L)$**:** 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 $(\mathit{UNION} - R)$**:** the same as above;

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

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

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,

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

■

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*

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

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* `fun(`x`)` *is a multi branched, node-labelled tree defined by,*

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

where the $\texttt{fun}_i(x_i)$'s, with $i : 1, \dots, n$, are algorithm calls, in chronological order, recursively issued by `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* $\mathtt{fun}(x)$ *is a list of nodes corresponding to the preorder traversal of* $CT(\mathtt{fun}(x))$*, i.e. the list in chronological order of all calls to* `fun` *caused by the computation* $\mathtt{fun}(x)$*,*

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

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*

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

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 $\mathtt{Proof}(A_1,\ ME,\ o_1,\ T_1)$*, 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 $\mathtt{Proof}(A,\ ME,\ o,\ \mu X.T)$**:** the n'th execution step is then,

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

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,

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

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

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

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 $\texttt{Proof}(A,\ ME,\ o,\ [l_1 : T'_1, \dots, l_n : T'_n])$**:** 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 $T'_i \not\equiv_{\mathrm{T}} coll(U)$; the second one is of the form,

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

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

By IH we know that $[l_1 : T'_1, \dots, l_n : T'_n] \sqsubseteq T_1$, 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 $p_o = [< o_1, l'_1, m_1, o'_1 >, < o'_1, l'_2, m_2, o'_2 > \dots, < o'_n, l'_n, m_n, o >]$ 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* $\texttt{Proof}(A_0,\ ME,\ o_0,\ T_0), \dots, \texttt{Proof}(A_i,\ ME,\ o_i,\ T_i), \dots$ *is a call path of* $CT(\texttt{Proof}(A_0,\ ME,\ o_0,\ T_0))$*, then,*

$$A_0 \subseteq A_1 \subseteq \dots \subseteq A_i \subseteq \dots$$

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*

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

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* $\texttt{Proof}$*) 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 $(o_i,\ T_i) \equiv (o,\ [l_1 : T_1, \dots, l_n : T_n])$ in p. Indeed, had not this pair existed, all calls would be relative to union types or μ-types, which is not possible as $\mathbf{T}$ defines canonical types (see Definition 6.1.2).

As $i > N$, there exists $(o_k, T_k) \equiv (o_i, [l_1 : T'_1 \dots, l_n : T'_n])$ 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 $\texttt{Proof}(A,\ ME, o, int)$**:** the algorithm returns a derivation $(INT)\ A; ME \vdash o :: int$, which is valid because from $o \rhd_{ME} int$ we know that $o \in Integer$;

Case $\texttt{Proof}(A,\ ME, o, string)$**:** 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 `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, `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 `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 `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,
`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:  $S \times \mathbf{T} \to D \cup \{fail\}$
Extraction($s$, $T$) =
  { ($\Delta A$, $ME$):= Extract($\emptyset$, $s_e$, $s_r$, $T$)
    if $ME = fail$  then return $fail$
                    else return ($s_r$, $ME$)
    endif }


Extract: $\mathcal{P}_{fin}(Oid \times \mathbf{T}) \times \mathcal{P}_{fin}(Oid \times Label \times Obj) \times Obj \times \mathbf{T} \;\to$
         $\mathcal{P}_{fin}(Oid \times \mathbf{T}) \times \mathcal{P}_{fin}(Oid \times Label \times M \times Obj^{+}) \cup \{fail\}$
case Extract($A$, $E$, $o$, $\mu X.T$)
  { return Extract($A$, $E$, $o$, $T\left[{}^{\mu X.T}/_{X}\right]$) }

case Extract($A$, $E$, $o$, $int$)
  { if $o \in Integer$ then return ($\emptyset$, $\emptyset$)
                    else return ($\emptyset$, $fail$)
    endif }

case Extract($A$, $E$, $o$, $string$)
  { if $o \in String$  then return ($\emptyset$, $\emptyset$)
                    else return ($\emptyset$, $fail$)
    endif }

case Extract($A$, $E$, $o$, $T_1 + T_2$)
  { ($\Delta A$, $ME$) := Extract($A$, $E$, $o$, $T_1$)
    if $ME = fail$  then return Extract($A$, $E$, $o$, $T_2$))
                    else return ($\Delta A$, $ME$)
    endif }

case Extract($A$, $E$, $o$, $T$), where $T \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n]$
  { if $\exists \overline{T} \in \mathbf{T}.\,(o,\overline{T}) \in A$
          then if $T =_{\mathbf{T}} \overline{T}$then return ($\emptyset$, $\emptyset$)
                           else return ($\emptyset$, $fail$)
               endif
          else {  ($\Delta A$, $ME$) $:= (\{(o,T)\}, \emptyset)$   <insertion of extractable oids>
                 $FAILED := false$
                 $i := 1$
                 while $i < n+1$ and $not(FAILED)$ do
                    if $T_i \equiv_{\mathbf{T}} coll(U)$  <EXTRACTION FOR COLLECTION FIELDS>
                      then {  $EMPTY := true$
                              for all $e \in E(o,l_i)$ do
                                ($\Delta A_t$, $ME_t$) :=Extract($A \cup \Delta A$, $E$, $\overrightarrow{e}$, $U$)   <search for selected edges>
                                if $ME_t \neq fail$ then
                                 { $ME := ME \cup ME_t \cup$ { $<\overleftarrow{e}, l_i, c, \overrightarrow{e}>$}   <generation of a marked edge>
                                   $\Delta A := \Delta A \cup \Delta A_t$   <insertion of extracted oids>
                                   $EMPTY := false$ }
                                endif
                              endfor
                              if $EMPTY$ then $ME := ME \cup$ $\{< o, l_i, c, \emptyset_U >\}$ endif }
                      else {  $FAILED := true$   <EXTRACTION FOR NON-COLLECTION FIELDS>
                              for all $e \in E(o,l_i)$ do  <collection field case>
                                   ($\Delta A_t$, $ME_t$) := Extract($A \cup \Delta A$, $E$, $\overrightarrow{e}$, $T_i$)   <search for selected edges>
                                   if $ME_t \neq fail$ then
                                    { $ME := ME \cup ME_t \cup$ { $<\overleftarrow{e}, l_i, r, \overrightarrow{e}>$}   <generation of a marked edge>
                                      $\Delta A := \Delta A \cup \Delta A_t$ <insertion of extracted oids>
                                      $FAILED := false$
                                      exitfor }
                                   endif
                              endfor }
                    endif
                 endwhile
    endif
    if $FAILED$ then $(\Delta A, ME) := (\emptyset, fail)$ endif
    return ($\Delta A$, $ME$) }

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

Figure 7.1: Extraction algorithm for SSDBs