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

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

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

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

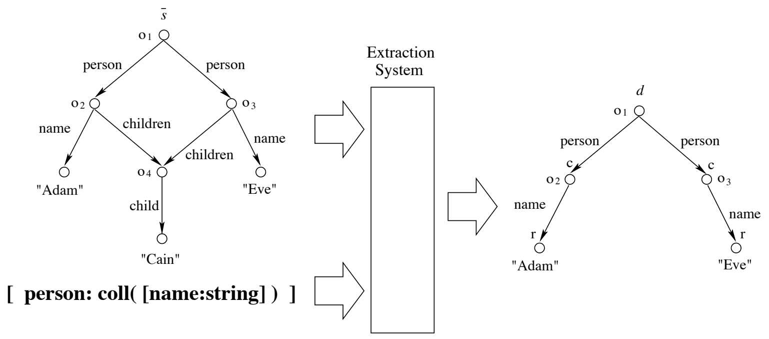


Figure 7.2: Example of extraction

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

7.2.1 Generation of marked edges

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

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

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

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

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

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

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

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

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

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

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

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

Consider again the example in Figure 7.2, where $\overline{T} \equiv_{\mathrm{T}} [person : coll([name : string])]$. The call `Extraction(`$\overline{s}$, $\overline{T}$`)` invokes `Extract(`$\emptyset$, $\overline{s}_e$, o_1, $\overline{T}$`)`. Since the record type is of the form $[person : coll(U)]$, `Extract` searches for the subset of candidate edges $\overline{s}_e(o_1, person) \subseteq \overline{s}_e$. Before the algorithm can effectively mark these edges with c, it must verify that their target objects, namely o_2 and o_3, are in turn extractable according to the type U of the collection values. To this aim `Extract` issues two recursive calls, to match o_2 and o_3 with U. These calls generate the sets $ME_{o_2} = \{< o_2, name, r, ''Adam'' >\}$ and $ME_{o_3} = \{< o_3, name, r, ''Eve'' >\}$. Therefore, the algorithm can generate the marked edges $< o_1, person, c, o_2 >$ and $< o_1, person, c, o_3 >$ and return $ME = ME_{o_2} \cup ME_{o_3} \cup \{< o_1, person, c, o_2 >, < o_1, person, c, o_3 >\}$. The resulting $d = (o_1, ME)$ is clearly of type $\overline{T}$.

7.2.2 Assumption sets

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

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

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

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

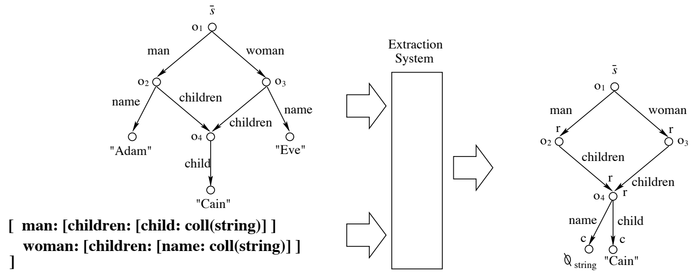


Figure 7.3: Example of wrong extraction due to loose termination test.

To be convinced of this, assume `Extraction` adopts the termination test of `Proof`, that is the test terminates an execution path when the pair $< o$, record type $T_o >$ is encountered for the second time. The application of `Extraction` in the example shown in Figure 7.3 results in the value d shown in the picture, which is not typable according to the input type. Indeed, the oid o_4 is visited by two independent call paths of the algorithm, respectively extracting according to the types $[name : coll(string)]$ and $[child : coll(string)]$. Both $\overline{s}_e(o_2, name)$ and $\overline{s}_e(o_2, child)$ contain edges that satisfy the structural requirements of these types, Hence, the algorithm terminates returning the value d. This result is not sound with respect to extractability since both $A; ME \vdash o_4 :: [name : coll(string)]$ and $A; ME \vdash o_4 :: [child : coll(string)]$ are not valid judgements.

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

- *extractable oids*: the oids visited in the current call path: the algorithm is currently checking whether these oids can be extracted or not; their presence in A is to avoid circular reasoning;

- *extracted oids*: the oids visited in the current call chain which are not into the current call path: these are the oids that have been successfully extracted so far; their presence in A is to avoid the redundant visit of shared portions of the SSDB.

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

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

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

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

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

Moreover, the successful execution of the call `Extract`(A_i, E, o_i, T_{o_i}), returns a pair $(ME_i, \Delta A_i)$. The set ΔA_i contains the pairs $(o', T_{o'})$ relative to the successful record case calls that contributed to the generation of the set ME_i. The assumption set relative to the next call in the call chain, `Extract`$(A_{i+1}, E, o_{i+1}, T_{o_{i+1}})$, is thus enriched with the extracted oids in ΔA_i. Call paths originating from such call can thus terminate when an extracted oid is encountered.

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

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

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

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

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

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

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

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

Hence, the algorithm issues the call

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

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

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

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

7.2.3 Reading the algorithm

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

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

where $ME \neq fail$, then:

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

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

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

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

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

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

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

7.3 Termination

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

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

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

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

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

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

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

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

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

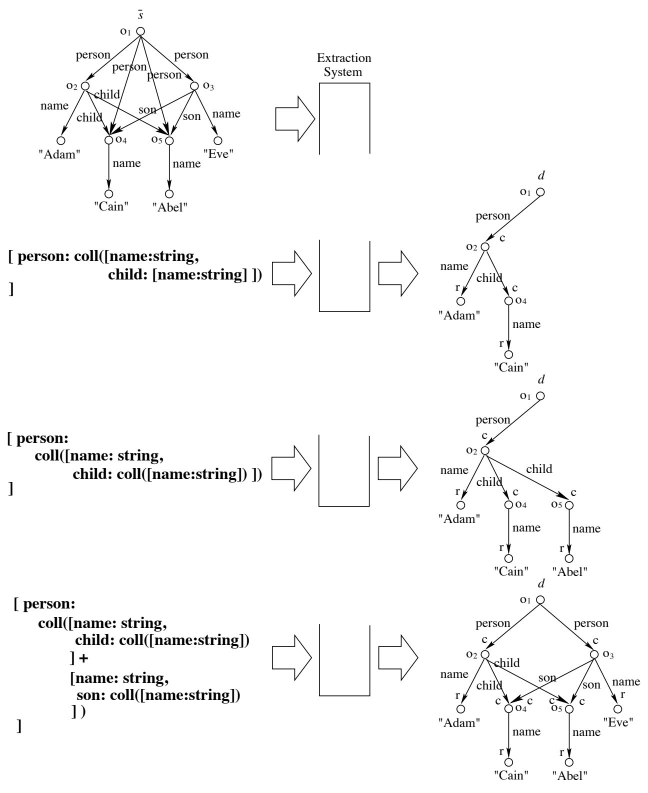
Proof. Directly from theorem 7.3.2, as `Extraction`$(s,\ T)$ depends on the result of the call `Extract`$(\emptyset,\ s_e,\ s_r,\ T)$. ∎

7.4 Relevance

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

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

Chapter 4, extraction may be repeated after long intervals of time to synchronise with the rare but possible updates of $\overline{s}$. It may well be the case that some of these updates were interesting to the user but do not fall in the subset of $\overline{s}$ captured by $\overline{T}$. Therefore, the extant applications would run over obsolete extracted data.



```
[ person: coll([name:string,
                child: [name:string] ])
]

[ person:
    coll([name: string,
          child: coll([name:string]) ])
]

[ person:
    coll([name: string,
          child: coll([name:string])
         ] +
         [name: string,
          son: coll([name:string])
         ])
]
```

Figure 7.4: Example of extraction with a low-relevance type

Consider the picture in Figure 7.4, where the extraction system is passed the SSDB $\overline{s}$. Assume the user is interested to compute over the subset of $\overline{s}$ involving *persons* with a *name* and a *child*, which in turn has a *name*. From an initial analysis of the collection the user comes up with the type,

$$[person : coll([name : string, child : [name : string]])]$$

shown in the first extraction in the picture. Observing the resulting value d, we note that $\overline{s}$ contains more data that may be potentially interesting to the user. For example, differently from the user's request, an element of the collection *person* may feature more than one *child*; moreover, there are other subsets of $\overline{s}$ that semantically correspond to elements of *persons* but are discarded because their label is *son* rather than *child*.

These sort of misjudgments may be very frequent, as we aim at dealing with the general scenario of SSDBs featuring a large number of paths, inserted at different times, possibly by different users with a different cognition of data representation. Therefore, after extraction, our algorithm provides the user with information relative to the degree of *relevance* of his current typing, with the specific purpose of minimising the misjudgments exemplified above.

We measure relevance in terms of *data capturing* and *precision*.

Data capturing is simply the ratio between the edges effectively extracted from $\overline{s}$ and the edges in $\overline{s}$.

Definition 7.4.1 *(Data capturing) Let $\overline{T} \in \mathbf{T}$ and $\overline{s} \in S$. Given the successful call,*

$$\texttt{Extraction}(\overline{s}, \overline{T}) = d$$

data capturing *of this extraction is calculated as:*

$$dataCapt(\overline{s}, d) = \frac{\mid Erase(d_e) \mid}{\mid \overline{s}_e \mid}$$

This measure may help the user, who knows the size of $\overline{s}$, to figure out whether our query methodology is convenient in a given application context: a low level of data capturing may suggest the existence of an extremely small regular core, which may be better handled by SSDQLs as those presented in Chapter 2.

By *lost information* for o we mean the edges in $\overline{s}(o)$ that have not been extracted due to the structural constraints of the extraction type $\overline{T}$, but were regarded as candidate for extraction by `Extract`. Consider again the first extraction in Figure 7.4: the edges $< o_2, child, o_5 >$, $< o_1, person, o_3 >$, $< o_1, person, o_4 >$ and $< o_1, person, o_5 >$ fall in this category. Indeed, the first edge was a possible candidate for `Extract`, because $< o_2, child, o_5 > \in \overline{s}(o_2)_{child}$; however, it was discarded because the first edge considered by `Extract` was $< o_2, child, o_4 >$, which fully satisfied the structural requirements of the record field *child*.

Precision of extraction measures how well a record type $\overline{T}$ minimises the quantity of lost information for a specific oid $o \in Oid(\overline{s})$. In particular, precision is the ratio between the number of edges outgoing o correctly extracted by the algorithm to satisfy the typing for $\overline{T}$, and the number of edges outgoing o that were candidate for that extraction. The overall precision of a record type is calculated in terms of two separate forms of precision, *non-collection field precision* and *collection field precision*:

Non-collection field precision: it is related with the extraction of edges relative to record fields that are not associated to collection types. This is the case for *child* in the first extraction exemplified in Figure 7.4. The structural requirements of a non-collection record field entail the extraction of the first edge satisfying the typing, in the example $< o_2, child, o_4 >$. However, there is another edge labelled as *child*, which is therefore likely to be relevant for the user. The system associates with this field a precision of $\frac{1}{2}$, meaning that, due to structural constraints, the extraction returned only *one* edge out of *two* interesting edges.

The user, by observing that this measure is related with a non-collection record field, may try to improve precision of his extraction by *adding a collection type* to $\overline{T}$, thereby turning it into,

$$[person : coll([name : string, child : coll([name : string])])]$$

which is the type relative to the second extraction illustrated in Figure 7.4.

Collection field precision: it is related with the extraction of edges relative to record fields that are associated to collection types. This is the case for *person* in the second extraction exemplified in Figure 7.4. The structural requirements of a collection record field entail the extraction of all edges satisfying the typing, in the example $< o_1, person, o_2 >$. However, there are other edges outgoing o labelled as *person*, which are therefore likely to be relevant for the user. The system associates with this field the precision $\frac{1}{4}$, meaning that, due to structural constraints, the extraction returned only *one* edge out of *four* potentially interesting edges. The user, by observing that this measure is related with a collection record field, may try to improve precision of his extraction by *adding a union type* to $\overline{T}$, thereby turning it into,

$$\begin{array}{l}[person : coll([name : string, child : coll([name : string])] + \\ \qquad\qquad [name : string, son : coll([name : string])]) \\ ]\end{array}$$

which is the type relative to the third extraction illustrated in Figure 7.4.

Given $d \in D$ extracted from $\overline{s} \in S$ according to $\overline{T} \in \mathbf{T}$, we can formally give the following definitions:

Definition 7.4.2 *(Non-collection field precision) Let $T \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n] \sqsubseteq \overline{T}$ and $o \in Oid(d)$. For all field l_i such that $T_i \not\equiv coll(U)$, the* non-collection field precision *in s for l_i and o is:*

$$ncprec_s(o, l_i) = \frac{1}{\mid s_e(o, l_i) \mid}$$

Definition 7.4.3 *(Collection field precision) Let $T \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n] \sqsubseteq \overline{T}$ and $o \in Oid(d)$. For all field l_i such that $T_i \equiv_{\mathbf{T}} coll(U)$, the* collection field precision *in s for l_i and o is:*

$$cprec_{s,d}(o, l_i) = \begin{cases} \frac{|d_e(o,l_i)|}{|s_e(o,l_i)|} & \mid s_e(o, l_i) \mid \neq 0 \\ 1 & \mid s_e(o, l_i) \mid = 0 \end{cases}$$

Observe that the precision of a collection field with respect to an oid o is certainly 1 if its extraction involved an empty collection.

Finally, we are able to provide a measure of the total precision of a record type with respect to a given oid. This measure is obtained in terms of the measures of precision for the individual fields of the record type.

Definition 7.4.4 *(Record precision) Let $T \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n]$ in $\overline{T}$ and o an oid in s' that has been extracted according to T. The record loss for $\overline{T}$ in s with o is:*

$$rprec_{s,d}(o, T) = \frac{1}{n} \cdot \left(\sum_{i=1, T_i \not\equiv_{\mathbf{T}} coll(U)}^{n} ncprec_s(o,\ l_i) \ + \sum_{i=1, T_i \equiv_{\mathbf{T}} coll(U)}^{n} cprec_{s,d}(o,\ l_i) \right)$$

This measure provides minimal information about every single extraction of an oid with respect to a record type. If properly combined, these precisions may provide extremely interesting information to the user.

For example, each record type T appearing in an input type $\overline{T}$ could be associated with a *total precision.* That is the ratio between the sum of the record precisions for oids in d conforming to T, and the number of oids in d conforming to T.

$$totrprec_{s,d}(T) = \frac{\sum_{o \in Oid(d).\ o::T} rprec_{s,d}(o, T)}{\mid \{o \in Oid(d) \mid o :: T\} \mid}$$

However, different measures of precision could be conceived, such as providing weights for different fields, or simply returning the total sum of the individual losses for each record, or calculating the loss of an extraction with respect to a type in terms of the loss of the related subextractions.

It is interesting to note that in practice relevance will be available to programs. Hence, its first use may be by the programs themselves, for example providing some thresholds.

With reference to *rprec*, after a successful extraction of d from $\overline{s}$ and $\overline{T}$, the user is presented the list of the pairs (o, T) where $T \sqsubseteq \overline{T}$ is a record type. Each record type comes along with its precision, as well as the precision of the single fields. These measures draw a picture of the extraction process that may help the user to improve its extraction.

For instance, the first extraction shown above produces the following measures of precision:

Record type	*Oid*	*Fields Precision*	*Total Precision*
$[person : coll(T_1)]$	o_1	$(person, \frac{1}{4})$	$\frac{1}{4}$
$[name : string, child : T_2]$	o_2	$(name, 1), (child, \frac{1}{2})$	$\frac{3}{4}$
$[name : string]$	o_4	$(name, 1)$	1

The relatively low total precision of the first two record types suggests to explore the precision of the related fields. Note that, as a general rule, a low non-collection field precision suggests the addition of a collection type, while a low collection field precision suggests the introduction of a union type. Following this reasoning, we reach the third extraction exemplified in Figure 7.4. The measures of precision for that extraction are:

Record type	*Oid*	*Fields Precision*	*Total Precision*
$[person : coll(T_1)]$	o_1	$(person, \frac{1}{2})$	$\frac{1}{2}$
$[name : string, child : T_2]$	o_2	$(name, 1), (child, 1)$	1
$[name : string, son : T_2]$	o_3	$(name, 1), (son, 1)$	1
$[name : string]$	o_4	$(name, 1)$	1
$[name : string]$	o_5	$(name, 1)$	1

The developers or the programs may be satisfied with an extraction even if it does not report a precision 1.

7.5 Cost

An execution of `Extraction` may potentially require the repeated traversal of a whole SSDB. This undesirable feature is quite typical in semistructured data research, where applications are often faced with the problem of the traversal of a tree or a graph. Generally, however, these applications show extremely bad performances for worst-case scenarios and are well-behaved in real scenarios. In other words, an application's worst case complexity has not the same relevance as the experimental results.

In this Section, we show the potentially exponential nature of `Extraction`, in order to provide the reader with a better understanding of the algorithm's behaviour. We shall measure cost in terms of the size of the SSDB $\overline{s}$ to be traversed by `Extract`, i.e. in relation to the number of edges in $\overline{s}$. We shall see that the algorithm entails exponential execution costs when applied to specific, simple types and *tree-structured SSDBs*. However, we shall also debate the general practical value of these kind of calculations and claim that the cost for a real application extraction is averagely linear on the number of edges.

Due to the structural mismatch between record fields and edges emanating from oids, the average execution of `Extract` may not traverse arbitrarily large subsets of the tree-structured $\overline{s}$. Accordingly, if no call of `Extract` fails, the worst case is when the input $\overline{s}$ and $\overline{T}$ are such that $\overline{s}$ is in a one-to-one mapping with a value d of type $\overline{T}$. The cost of this extraction is that of a depth-first visit of the SSDB graph, namely $O(m)$ where $m = \mid \overline{s}_e \mid$ is the number of edges in $\overline{s}$.

Due to the failure of specific call paths, however, the same edges may be visited more than once. By observing `Extract`, this may happen when a call issued by a union type case `Extract`(A, E, o, $T_1 + T_2$) fails. Indeed, the failure of the call `Extract`(A, E, o, T_1) causes the invocation of the call `Extract`(A, E, o, T_2), which may traverse again the edges outgoing o and the edges reachable from o. Note that the same problem may arise when selecting candidate edges for a record field.

In the following, we show the exponential behaviour of the algorithm when applied to a union type and a specific SSDB, which maximises the number of failures and visit of edges. In particular, we apply `Extract` to the SSDB $\overline{s}$ with n levels and fan-out $f = 1$ illustrated in Figure 7.5, and the μ-type:

$$T \equiv_{\mathrm{T}} \mu X.\, U,$$

where $U = T_1 + \ldots + T_k$ and $T_i = [a : X]$ for $i : 1, \ldots, k$. In its execution, `Extract` visits all oids in $o \in \mathit{Oid}(\overline{s})$ with a union type $T_1 + \ldots + T_k$, and fails after having tried the extraction of o according to all members T_i. In particular, each record case call with o and $[a : T]$, invoked by a union case, causes the visit of the whole path from o to *ciao* in $\overline{s}_e$ and fails only when visiting that string. This failure propagates back again to the union case call, which invokes an identical record case call relative to the next member of the union. Intuitively, the same path may be thus visited an exponential number of times. We give a formal proof of this in the following proposition.

Proposition 7.5.1 *The execution cost of* `Extraction`$(\overline{s}, U)$ *is* $O(k^m)$*.*

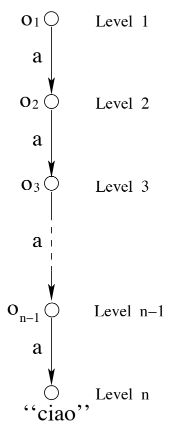


Figure 7.5: Worst case SSDB and type

Proof. First, note that if we consider the root of $\overline{s}$ at level 1, we have that

$$m = \mid \overline{s}_e \mid = n - 1$$

As the cost of `Extraction` is the cost of `Extract`, we prove the statement for the latter, by showing that the number of edges visited by the call `Extract`(A, $\overline{s}_e$, $\overline{s}_r$, T) is k^n, i.e. k^m.

This statement can be proven by induction on the number i of levels of $\overline{s}$ to be traversed by the generic union case call `Extract`(A, E, o_{n-i}, $U\left[{}^{T}/_{X}\right]$), which we shall denote as `Extract`$_{n-i}$.

$\boxed{i = 1}$ `Extract`$_{n-1}$ issues a call `Extract`(A, E, o, $[a : T]$) in order to extract from o_{n-1} according to the first member of the union type. The only candidate edge for the label a is $< o_{n-1}, a, \text{“}ciao\text{''} >$. Accordingly, the call will invoke a further μ-type case call with “*ciao*'' and fail, because “*ciao*'' is not an oid as required by μ-types.

This failure propagates back to `Extract`$_{n-1}$, which will try the extraction of o_{n-1} with the second member of the union. As all members of the union will fail, this process is repeated k times, then the union case call fails. Each time the union case call tries to extract according to one of its members, the edge $< o_{n-1}, a, \text{“}ciao\text{''} >$ is visited once. Hence, the call visits the edge a total of $k^1 = k$ times.

$\boxed{i > 1}$ By IH we know that all calls `Extract`$_{n-j}$, with $j : 1, \ldots, i-1$, fail after visiting k^j edges.

The call $\texttt{Extract}_{n-i}$ issues a call $\texttt{Extract}(A,\ E,\ o_{n-i},\ [a:T])$ in order to extract from o_{n-i} according to the first member of the union type. Such a call identifies the only candidate edge outgoing o_{n-i}, namely $< o_{n-i}, a, o_{n-i+1} >$, and issues the call $\texttt{Extract}(A,\ E,\ o_{n-i+1},\ T) = \texttt{Extract}_{n-i+1}$. The latter is a union case call at level $i-1$. By IH we know that this call traverses k^{i-1} edges before failing. This failure propagates back to $\texttt{Extract}_{n-i}$, which attempts the extraction of $< o_{n-i}, a, o_{n-i+1} >$ with the second member of the union type. This process is repeated k times, then $\texttt{Extract}_{n-i}$ fails. The candidate edge $< o_{n-1}, a, o_{n-i+1} >$ is visited k times. In addition, each of this visits caused the visit of k^{i-1} edges, for a total of k^i edges visited.

In conclusion, extracting an SSDB as simple as $\overline{s}$ with a type as simple as T, entails the visit of k^n edges. Since $m = n-1$ we can state that the execution cost for `Extraction` is $O(k^m)$.

■

Note that the scenario we constructed represents a particularly extreme situation. For $k = 2$ and $m = 30$, for example, the total of edges visited by the algorithm would be close to a billion.

In real applications, we expect the user to extract by means of a reasonable number of union types, not to fall in the category of non optimal types. Besides, only a portion of the oids in the collection would be extracted according to them, and we do not expect all edges reachable from such oids to satisfy all members of a union type. Furthermore, due to the mismatch between the labels of the record fields and the labels of the edges, portions of the SSDB may not be traversed at all.

Consider the extraction of a generic SSDB with m edges with the type:

$$[\mathit{Dept} : [\mathit{Emp} : \mathit{coll}([\mathit{Name} : \mathit{string} + [\mathit{FirstName}, \mathit{SecondName} : \mathit{string}])]].$$

This type cannot possibly give rise to an explosion of complexity. The algorithm will not visit the same edges twice, as the members of the union type are different and there are no objects in the SSDB that may match both. For this quite common scenario, the algorithm may traverse at most all m edges.

Chapter 8

Extraction Algorithm Correctness

In this Chapter we prove correctness of the algorithm `Extraction` with respect to extractability of L as defined in Chapter 7. In particular, according to the definition of correctness in Chapter 4, we shall prove:

Soundness: every successful execution `Extraction`$(\overline{s}, \overline{T})$ $= d$ is such that d is *extractable* from $\overline{s}$ according to $\overline{T}$;

Completeness: if the set of values $D_{\overline{s},\overline{T}} \subseteq D$ which are *extractable* from $\overline{s}$ according to T is not empty, the call `Extraction`$(\overline{s}, \overline{T})$ is successful and returns $d \in D_{\overline{s},\overline{T}}$.

We shall show that `Extraction` is sound with respect to extractability, but not generally complete. Indeed, the algorithm is complete whenever its application is restricted to tree-structured SSDBs. We shall also observe, however, that incompleteness is typically due to particularly critical scenarios, whose exceptionality does not generally compromise the usability of `Extraction` to all SSDBs. Furthermore, on the basis of the examples of incompleteness, we shall suggest how `Extraction` could be modified in order to decrease its degree of incompleteness.

8.1 Soundness

In this Section we prove the following soundness theorem.

Theorem 8.1.1 *(Soundness of* `Extraction`*) Let $\overline{s} \in S$ and $\overline{T} \in \mathbf{T}$. Every successful execution of the algorithm* `Extraction`$(s, \overline{T})$ `=` *d is such that $ssd(d) < \overline{s}$ and $d : \overline{T}$.*

To this aim we require the notion of *success tree* of `Extract`, which is the tree obtained by the call tree eliminating the nodes relative to failing calls and all nodes which are reachable from these. Formally,

Definition 8.1.2 *(Failing node) A* failing node *of a call tree is a node corresponding to a failing call, i.e. a call of* `Extract` *that returns a pair* $(\emptyset, fail)$.

Definition 8.1.3 *(Success tree of* `Extract`*) Let* $\overline{s} \in S$, $\overline{T} \in \mathbf{T}$, `Extraction`$(\overline{s}, \overline{T})$ $= d$ *be a successful execution of the algorithm, and* $CT($`Extract`$(\emptyset,\ \overline{s}_r,\ \overline{s}_e,\ \overline{T}))$ *be the call tree corresponding to such execution.*

The success tree $ST($`Extract`$(\emptyset,\ \overline{s}_r,\ \overline{s}_e,\ \overline{T}))$ *of this execution is the tree obtained from the call tree by dropping all failing nodes and all nodes reachable with a call path originating from a failing node. We denote as* ST_N *the nodes in the success tree* ST.

Furthermore, in the following we shall denote as $\overline{ME}$ the set of marked edges generated by a record case call.

Definition 8.1.4 *(Marked edges generated by a record case call) Let* ST *be a success tree,* $s \in S$, $o \in Oid(s)$, A *an assumption set,* $T \equiv [l_1 : T_1, \ldots, l_n : T_n] \in \mathbf{T}$. *The node* `Extract`$(A,\ s_e,\ o,\ T) \in ST_N$ *has a finite set of children,*

$$Sub = \{\texttt{Extract}(A_1,\ s_e,\ o_1,\ T_1')\ldots, \texttt{Extract}(A_k,\ s_e,\ o_k,\ T_k')\}$$

such that $k \geq 0$. *By definition of* `Extract`, *we know that each of these calls was invoked after identifying a corresponding candidate edge* $< o, l, o_i >$. *The* set of marked edges generated by `Extract`$(A,\ E,\ o,\ T)$ *is the set*

$$\overline{ME} = (\bigcup_{i=1}^{k}\{< o, l, m, o_i >\}) \cup (\bigcup_{T_i \equiv coll(U) \wedge s(o,l_i)=\emptyset}\{< o, l_i, c, \emptyset_U >\})$$

obtained by appropriately decorating the corresponding selected edges and adding the empty collection edges where necessary. `Extract`$(A,\ E,\ o,\ T)$ *will return the set of marked edges*

$$ME = \overline{ME} \cup \bigcup_{i=1}^{k} ME_i$$

where ME_i *is the set of marked edges returned by* `Extract`$(A_i,\ E,\ o_i,\ T_i')$.

8.1.1 Soundness of inclusion

Lemma 8.1.5 *Let* A *be an assumption set,* E *a set of edges,* $o \in Obj$, $T \in \mathbf{T}$, *and* ST *the success tree of a call of* `Extract`. *If,*

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

and,

$$\texttt{Extract}(A,\ E,\ o,\ T) \in ST_N$$

then,

$$ssd((o,\ ME)) < (o,\ E)$$

Proof. We prove this statement by induction on the finite depth i of the success tree $ST(\texttt{Extract}(A,\ E,\ o,\ T))$ and adopting the identity id as the morphism required to prove lt-inclusion.

$\boxed{i = 0}$

Case $\texttt{Extract}(A \cup \{(o,T)\},\ E,\ o,\ T)$**:** the call corresponding to this node returns the pair $(\emptyset,\ \emptyset)$; in general for all set of edges E, $(o,\ \emptyset) < (o,\ E)$ as $ssd((o,\ \emptyset)) = (o,\ \emptyset)$ and $id(o) = o$;

Case $\texttt{Extract}(A,\ E,\ o,\ \mathit{int})$**:** the same as for the case above;

Case $\texttt{Extract}(A,\ E,\ o,\ \mathit{string})$**:** the same as for the case above.

Case $\texttt{Extract}(A,\ E,\ o,\ [l_1 : T_1, \dots , l_n : T_n]) = (\Delta A,\ ME)$**:** where

$$\Delta A = \{(o, [l_1 : T_1, \dots , l_n : T_n])\} \text{ and } ME = \bigcup_{i=1}^{n}\{< o, l_i, c, \emptyset_{U_i} >\};$$

this case occurs when

$$\forall i : 1 \dots n. (T_i \equiv_{\mathrm{T}} \mathit{coll}(U_i)) \ \wedge\ (\mid E(l_i) \mid = 0).$$

Note that $Erase(\bigcup_{i=1}^{n}\{< o, l_i, c, \emptyset_{U_i} >\} = \emptyset$, hence the proof is trivial as above.

$\boxed{i > 0}$ We assume that the thesis holds for all nodes of ST with depth less than i. Since $i > 0$, by definition of ST, for all $\texttt{Extract}(A,\ E,\ o,\ T) \in ST_N$ there exists a finite set of children,

$$Sub = \{\texttt{Extract}(A_1,\ E,\ o_1,\ T_1) \dots , \texttt{Extract}(A_k,\ E,\ o_k,\ T_k)\}$$

such that $k \geq 1$ and their depth is $i - 1$. Furthermore, the call corresponding to the node $\texttt{Extract}(A,\ E,\ o,\ T)$ returns a pair $(\Delta A,\ ME)$ and the calls corresponding to the set of children above return pairs $(\Delta A_j,\ ME_j)$ with $1 \leq j \leq k$. By observing the algorithm we notice that for all $j : 1, \dots , k$, $ME_j \subseteq ME$.

When $T \equiv_{\mathbf{T}} T_1 + T_2$ or $T \equiv_{\mathbf{T}} \mu X.T$, then $k = 1$ and $ME_1 = ME$. As in both situations it also holds that $o = o_1$, the thesis can be inferred directly from the IH $ssd((o_1,\ ME_1)) < (o_1,\ E)$.

When $T \equiv_{\mathbf{T}} [l_1 : T'_1, \ldots, l_n : T'_n]$, by observing the algorithm, we find that $ME = \overline{ME} \cup \bigcup_{j=1}^{k} ME_j$. By IH we know that

$$\forall j : 1, \ldots, k.\, ssd(o_j,\ ME_j)) = (o_j,\ Erase(ME_j)) < (o_j,\ E)$$

hence, in order to prove our thesis, we remain to prove that,

$$\begin{array}{l} \forall e \in Erase(\overline{ME}). \\ \quad \vec{e} \in Oid \;\Rightarrow\; \exists e' \in E.\, (id(\overleftarrow{e}) = \overleftarrow{e}' \;\wedge id(\vec{e}) = \vec{e}' \;\wedge label(e) = label(e')) \\ \quad \vec{e} \in Atomic \;\Rightarrow\; \exists e' \in E.\, (id(\overleftarrow{e}) = \overleftarrow{e}' \;\wedge\; \vec{e} = \vec{e}' \;\wedge label(e) = label(e')) \end{array}$$

Applying identity id as the morphism to prove inclusion, this becomes,

$$\forall e \in Erase(\overline{ME}).\, \exists e' \in E.\, (\overleftarrow{e} = \overleftarrow{e}' \;\wedge\; \vec{e} = \vec{e}' \;\wedge label(e) = label(e'))$$

that is,

$$\forall e \in Erase(\overline{ME}).\, e \in E$$

This can be trivially proven by simply observing the algorithm. Indeed, $\forall me \in \overline{ME}$ with $\vec{me} \in Obj$ ($\vec{me} \notin EC$, i.e. is not an empty collection value), there exists $\texttt{Extract}(A_j,\ E,\ o_j,\ T_j)$, with $j : 1, \ldots, k$ such that

$$\begin{array}{l} \exists i : 1, \ldots, n. \\ \quad (< o, l_i, o_j > \in E(o, l_i) \;\wedge\; \overleftarrow{me} = o \;\wedge\; label(me) = l_i \;\wedge\; \vec{me} = o_j) \end{array}$$

From these conditions we can derive that

$$\forall me \in \overline{ME}.\, (\vec{me} \in Obj \Rightarrow\; \exists e \in E.\, erase(me) = e)$$

which implies, by definition of $Erase$,

$$\forall e \in Erase(\overline{ME}).\, e \in E$$

Therefore,

$$ssd((o,\ ME)) = (o,\ Erase(\overline{ME}) \cup \bigcup_{j=1}^{k} Erase(ME_j))) < (o,\ E)$$

■

Theorem 8.1.6 *(Soundness of inclusion) Let* $\overline{s} \in S$, $\overline{T} \in \mathbf{T}$, $\texttt{Extraction}(\overline{s},\ \overline{T}) = d$. *Then,* $ssd(d) < \overline{s}$.

Proof. We know that

$$\texttt{Extraction}(\overline{s},\ \overline{T}) = d = (\overline{s}_r,\ ME)$$

where

$$\texttt{Extract}(A,\ \overline{s}_e,\ \overline{s}_r,\ \overline{T}) = (\Delta A,\ ME).$$

By Lemma 8.1.5 we obtain

$$ssd((\overline{s}_r,\ ME)) < (\overline{s}_r,\ \overline{s}_e) = \overline{s}.$$

■

8.1.2 Soundness of typing

The following judgement is an extension of the typing judgement $A;\ ME \vdash o :: T$ to the typing of a set A of pairs $(o,\ T)$:

Definition 8.1.7 *($A_1;\ ME \vdash^{\cdot} A_2$) Given two assumption sets A_1 and A_2, a set of marked edges ME,*

$$A_1;\ ME \vdash^{\cdot} A_2 \ \Leftrightarrow\ \forall (o,\ T) \in A_2 .\, A_1;\ ME \vdash o :: T$$

We prove soundness of typing as a direct consequence of the following theorem, which states a strong invariant for the algorithm `Extract`.

Theorem *(Invariant for `Extract`) Let A be an assumption set, E a set of edges, $o \in Obj$, $T \in \mathbf{T}$. Then,*

$$\texttt{Extract}(A,\ E,\ o,\ T) = (\Delta A,\ ME)\ \wedge\ ME \neq fail\ \Rightarrow\ A;\ ME \vdash^{\cdot} \Delta A.$$

We prove this statement by induction on the finite depth i of the success tree associated with the generic call `Extract`$(A,\ E,\ o,\ T)$, by proving the invariant for each case of `Extract`. The hardest part of the proof is the one involving the record type case call,

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

where $T \equiv_{\mathbf{T}} [l_1 : T_1 :, \ldots, l_n : T_n]$. Such call may recursively issue $1 \leq j \leq k$ calls,

$$\texttt{Extract}(A_j,\ E,\ o_j,\ T_j) = (\Delta A_j,\ ME_j)$$

which are all located at depth $i-1$ in the success tree. The results $(\Delta A_j,\ ME_j)$ are combined together to yield $(\Delta A,\ ME)$ in the following way:

$$\Delta A = (o,T) \cup \bigcup_{j=1}^{k} \Delta A_j \qquad ME = \overline{ME} \cup \bigcup_{j=1}^{k} ME_j$$

Accordingly, in order to prove that $A;\ ME \vdash^{\cdot} \Delta A$ is an invariant for the record type case call above, we should prove that,

$$A;\ \overline{ME} \cup \bigcup_{j=1}^{k} ME_j \vdash^{\cdot} (o,T) \cup \bigcup_{j=1}^{k} \Delta A_j$$

is a valid judgement. By induction hypothesis, we can assume that the $k \vdash^{\cdot}$-judgements

$$A_j;\ ME_j \vdash^{\cdot} \Delta A_j$$

are valid; as by observing the algorithm we can infer,

$$A_j = A \cup (o,T) \cup \bigcup_{t=1}^{j-1} \Delta A_t$$

this equates to say that,

$$(A \cup (o,T) \cup \bigcup_{t=1}^{j-1} \Delta A_t);\ ME_j \vdash^{\cdot} \Delta A_j.$$

are valid $\vdash^{\cdot}$-judgements. Consider the $\vdash^{\cdot}$-judgement for $j = k$,

$$(A \cup (o,T) \cup \bigcup_{t=1}^{k-1} \Delta A_t);\ ME_k \vdash^{\cdot} \Delta A_k.$$

From it we can directly produce another valid $\vdash^{\cdot}$-judgement, which almost proves our final statement for the record type case:

$$(A \cup (o,T) \cup \bigcup_{t=1}^{k-1} \Delta A_t);\ ME_k \vdash^{\cdot} (o,T) \cup \bigcup_{t=1}^{k} \Delta A_t.$$

Indeed, to prove our thesis we show that there exists a *dependency* between the ΔA_j's and the ME_j's. Informally, this dependency states that if a pair $(\overline{o}, \overline{T})$ can be proven correct with a judgement of the form $\Delta A_j \cup A;\ ME \vdash \overline{o} :: \overline{T}$ than it can be proven correct also with a judgement of the form $A;\ ME \cup ME_j \vdash \overline{o} :: \overline{T}$. Intuitively, this relation allows to replace the ΔA_j's with the ME_j's in the judgement above, thereby leading to our final proof of statement.

Before we proceed with the proof, we introduce the following formal tools.

Definition 8.1.8 *(Converging and diverging assumption sets) Given an assumption set $A \in \mathcal{P}_{fin}(Oid \times \mathbf{T})$, and $o \in Oid$, A* converges *on o ($A(o) \downarrow$) if and only if there exists a type $T \in \mathbf{T}$ such that $(o,T) \in A$. Vice versa A* diverges *on o ($A(o) \uparrow$) if and only if there is no type $T \in \mathbf{T}$ such that $(o,T) \in A$.*

Definition 8.1.9 *(Independent assumption sets) Two assumption sets* A_1, $A_2 \in \mathcal{P}_{fin}(Oid \times \mathbf{T})$ *are* independent *($A_1 \asymp A_2$) if and only if:*

- $\forall o \in Oid. (A_1(o) \downarrow \Rightarrow A_2(o) \uparrow)$*;*
- $\forall o \in Oid. (A_2(o) \downarrow \Rightarrow A_1(o) \uparrow)$.

Definition 8.1.10 *(Compatible sets of marked edges) Two sets of marked edges* ME_1, $ME_2 \in \mathcal{P}_{fin}(Oid \times Label \times M \times Obj^+)$ *are* compatible *($ME_1 \dot{\asymp} ME_2$) if and only if:*

$$\forall o \in \overleftarrow{ME}_1 \cap \overleftarrow{ME}_2 \,.\, ME_1(o) = ME_2(o)$$

A first interesting property of a successful call

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

is that the sets ΔA and A are independent.

Lemma 8.1.11 *Let E be a set of edges, $T \in \mathbf{T}$, $A \in \mathcal{P}_{fin}(Oid \times \mathbf{T})$, $o \in Obj$, and ST the success tree of a call of* `Extract`*. If,*

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

and,

$$\texttt{Extract}(A,\ E,\ o,\ T) \in ST_N$$

then,

$$A \asymp \Delta A$$

Proof. This statement can be proved by simply observing that ΔA contains the pairs $(\bar{o}, \overline{T})$ relative to calls

$$\texttt{Extract}(\overline{A},\ E,\ \bar{o},\ \overline{T}) \in ST_N$$

in the call chain originating from $\texttt{Extract}(A,\ E,\ o,\ T)$. As $(\bar{o}, \overline{T}) \in \Delta A$ these calls effectively generated marked edges from $E(\bar{o})$. Accordingly, $A(\bar{o}) \uparrow$ otherwise, as $A \subseteq \overline{A}$ the termination test would have prevented these extractions.

On the other hand, if $A(\bar{o}) \downarrow$ all calls

$$\texttt{Extract}(\overline{A},\ E,\ \bar{o},\ \overline{T}) \in ST_N$$

in the call chain originating from $\texttt{Extract}(A,\ E,\ o,\ T)$ would be such that $A \subseteq \overline{A}$, thus $\overline{A}(\bar{o}) \downarrow$. Accordingly, none of these calls could have extracted edges from $E(\bar{o})$, from which $\Delta A(\bar{o}) \uparrow$.

■

Note that, given a successful call `Extract`$(A,\ E,\ o,\ T) = (\Delta A,\ ME)$, the oids of the edges extracted by the call, namely the set $\overleftrightarrow{ME}$, are a subset of the oids in $O_{A,o,E}$. Hence, intuitively, A converges only on the oids $\bar{o} \in \overleftrightarrow{ME}$ such that $\mid\ ME(\bar{o})\ \mid = 0$, while ΔA converges on the remainder. Indeed, $\mid\ ME(\bar{o})\ \mid = 0$ only if the extraction with respect to $\bar{o}$ could not be performed due to $A(\bar{o}) \downarrow$, which means that an earlier call in the call chain had successfully performed the extraction of the edges outgoing $\bar{o}$. To our purposes we require related but simpler results, which we prove in the following Lemma.

Lemma 8.1.12 *Let A be an assumption set, E a set of edges, $o \in Obj$, $T \in \mathbf{T}$, and ST the success tree of a call of* `Extract`*. If,*

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

and,

$$\texttt{Extract}(A,\ E,\ o,\ T) \in ST_N$$

then,

$$\forall \bar{o} \in \overleftarrow{ME}\ .\ \Delta A(\bar{o}) \downarrow$$

Proof. We prove this statement by induction on the finite depth i of the success tree ST.

$\boxed{i = 0}$

Case `Extract`$(A \cup \{(o, T)\},\ E,\ o,\ T) = (\emptyset,\ \emptyset)$**:** trivial;

Case `Extract`$(A,\ E,\ o,\ \mathit{int}) = (\emptyset,\ \emptyset)$**:** trivial;

Case `Extract`$(A,\ E,\ o,\ \mathit{string}) = (\emptyset,\ \emptyset)$**:** trivial;

Case `Extract`$(A,\ E,\ o,\ [l_1 : T_1, \dots, l_n : T_n]) = (\Delta A,\ ME)$**:** where,

$$\Delta A = \{(o, [l_1 : T_1, \dots, l_n : T_n])\} \text{ and } ME = \bigcup_{i=1}^{n} \{< o, l_i, c, \emptyset_{U_i} >\};$$

this case occurs when

$$\forall i : 1 \dots n.\ (T_i \equiv_{\mathrm{T}} \mathit{coll}(U_i))\ \wedge\ (\mid\ E(l_i)\ \mid = 0).$$

the conclusion is trivial.

$\boxed{i > 0}$ We assume that the thesis holds for all nodes of ST with depth less than i. Since $i > 0$, by definition of ST, for all $\texttt{Extract}(A,\, E,\, o,\, T) \in ST_N$ there exists a finite set of children,

$$Sub = \{\texttt{Extract}(A_1,\, E,\, o_1,\, T_1) \ldots ,\texttt{Extract}(A_k,\, E,\, o_k,\, T_k)\}$$

such that $k \geq 1$ and their depth is $i - 1$. Furthermore, the call corresponding to the node $\texttt{Extract}(A,\, E,\, o,\, T)$ returns a pair $(\Delta A,\, ME)$ and the calls corresponding to the set of children above return pairs $(\Delta A_j,\, ME_j)$ with $1 \leq j \leq k$.

When $T \equiv_{\mathbf{T}} T_1 + T_2$ or $T \equiv_{\mathbf{T}} \mu X.T$, then $k = 1$, $A_1 = A$, and $\Delta A_1 = \Delta A$. Therefore, the thesis can be inferred directly from the IH for (i) and (ii).

When $T \equiv_{\mathbf{T}} [l_1 : T'_1, \ldots , l_n : T'_n]$, by observing the algorithm, we note that,

$$\Delta A = A \cup \bigcup_{t=0}^{k} \Delta A_t$$

where $\Delta A_0 = \{(o, [l_1 : T_1, \ldots , l_n : T_n])\}$. By IH, we know that,

$$\forall j : 1, \ldots k. \forall \bar{o} \in \overleftarrow{ME_j} \,. \Delta A_j(\bar{o}).$$

Given $\bar{o} \in \overleftarrow{ME}$, either,

- $\exists j : 1, \ldots , k. \bar{o} \in \overleftarrow{ME_j}$: by IH we know that $\Delta A_j(\bar{o}) \downarrow$, from which, as $\Delta A_j \subseteq \Delta A$, we can conclude

$$\Delta A(\bar{o}) \downarrow;$$

- $\bar{o} \in \overleftarrow{ME}$: then $\bar{o} = o$, and, as $(o,\, [l_1 : T_1, \ldots , l_n : T_n]) \in \Delta A$, we can trivially conclude

$$\Delta A(o) \downarrow .$$

■

The following Lemma describes under which conditions a set of marked edges can be added to a judgement without compromising its validity.

Lemma 8.1.13 *Given two sets of marked edges ME_1 and ME_2, an assumption set A, $o \in Oid$, and $T \in \mathbf{T}$,*

$$(ME_1 \doteqdot ME_2 \;\wedge\; (A;\, ME_1 \vdash o :: T) \;\Rightarrow\; A;\, ME_1 \cup ME_2 \vdash o :: T$$

Proof. The judgement $A;\, ME_1 \vdash o :: T$ may be valid because $A(o)\downarrow$, $o \in \mathit{Atomic}$ and o is a value of T, or because $ME(o)$ satisfies T. In the first case, any set of edges could be added on to ME_1, as rule (HYP) has precedence over the other rules. Similarly, in the second case, the addition of edges to ME_1 would not compromise the typing of o with the atomic type T. In the third case, the addition of edges could compromise the typing. Indeed, the addition of outgoing edges to an oid may no longer satisfy the requirements of T. The proof follows directly from observing that the hypothesis of compatibility between ME_1 and ME_2 ensures that the set of edges outgoing all oids in $\overleftarrow{ME_1}$ cannot be altered by the union with ME_2. ∎

We show that the conditions above apply to the sets of marked edges ME_j resulting from the subcalls `Extract`$(A_j,\, E,\, o_j,\, T_j)$ issued by a record type case call of `Extract`.

Lemma 8.1.14 *Let ST be a success tree for* `Extract`*, with* `Extract`$(A,\, E,\, o,\, T) \in ST_N$*, with* $T \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n]$ *and*

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

Let

$$\texttt{Extract}(A_j,\ E,\ o_j,\ T_j') = (\Delta A_j,\ ME_j)$$

with $1 \leq j \leq k$ *be the children of this node in* ST_N*, then,*

$$\forall 1 \leq j_1 < j_2 \leq k.\, ME_{j_1} \doteqdot ME_{j_2}.$$

Proof. We prove this statement by contradiction, assuming that there are $1 \leq j_1 < j_2 \leq k$ and $o \in \overleftarrow{ME_{j_1}} \cap \overleftarrow{ME_{j_2}}$ such that $ME_{j_1}(o) \neq ME_{j_2}(o)$. According to Lemma 8.1.12, $\Delta A_{j_1}(o)\downarrow$ and $\Delta A_{j_2}(o)\downarrow$; but this is absurd, as we know that $\Delta A_{j_1} \subseteq A_{j_2}$ and, by Lemma 8.1.11 $A_{j_2}(o)\downarrow \Rightarrow\ A_{j_2}(o)\uparrow$.

∎

Notice that a valid judgement $A_1;\, ME_1 \vdash^{\cdot} A_2$ indirectly establishes a sort of dependency between the edges ME_1 and the assumption set A_1 on one side, and the assumption set A_2 on the other side: the pairs $(o,\, T)$ in A_2 are all verified for typing by the edges in ME_1 and the pairs in A_1. Accordingly, we expect to be able to:

1. *replace* the assumption set A_2 with the edges ME_1 in any $\vdash^{\cdot}$-judgement $A_1 \cup A_2;\, ME_2 \vdash^{\cdot} A_3$, without compromising its validity: $A_1;\, ME_2 \cup ME_1 \vdash^{\cdot} A_3$ is still valid;

2. *add* the assumption set A_2 to the right side of the judgement: indeed, as $A_1;\, ME_1 \vdash^{\cdot} A_2$ and $A_1;\, ME_2 \cup ME_1 \vdash^{\cdot} A_3$ are valid, the judgement $A_1;\, ME_2 \cup ME_1 \vdash^{\cdot} A_3 \cup A_2$ is also valid.

The following Lemmas prove that these intuitions are true, but only under particular conditions.

Lemma 8.1.15 *Let A be an assumption set and $A_1;\, ME \vdash^{\cdot} A_2$ be a valid judgement. Then,*

$$A_1 \cup A;\, ME \vdash^{\cdot} A_2$$

is a valid judgement.

Proof. The proof comes directly from the type rules in Definition 6.6.1. An assumption (o, T) is extracted with (HYP) from an assumption set A only to terminate valid branches of potentially infinite proofs. Accordingly, if the judgement $A_1;\, ME \vdash^{\cdot} A_2$ is valid, the addition of new assumptions cannot compromise its validity, but only potentially shorten the corresponding proof of validity. ■

Lemma 8.1.16 *(Replacement) Given the assumption sets A_1, A_2, and A_3, the sets of marked edges ME_1 and ME_2, $o \in Obj$, and $T \in \mathbf{T}$, if*

1. $ME_1 \dot{\asymp} ME_2$,
2. $A_1;\, ME_1 \vdash^{\cdot} A_2$, *and*
3. $A_1 \cup A_2;\, ME_2 \vdash^{\cdot} A_3$*;*

then

$$A_1;\, ME_1 \cup ME_2 \vdash^{\cdot} A_3 \cup A_2;$$

we can replace A_2 *with* ME_1 *and add it to the right side without compromising the validity of the judgement.*

Proof. We first prove that $A_1;\, ME_1 \cup ME_2 \vdash^{\cdot} A_3$ is a valid judgement, by induction on the finite depth of the derivation tree of the valid judgement $A_1 \cup A_2;\, ME_2 \vdash o :: T$ where $(o,\, T) \in A_3$. Afterward, as $A_1;\, ME_1 \vdash^{\cdot} A_2$ and $ME_1 \dot{\asymp} ME_2$, we can trivially claim that $A_1;\, ME_1 \cup ME_2 \vdash^{\cdot} A_3 \cup A_2$ is a valid judgement.

$\boxed{i = 0}$

Case (HYP)**:** that is $(o,\, T) \in A_1 \cup A_2$. It can be either,

- $(o,\, T) \in A_1$: thus $A_1;\, ME_2 \vdash o :: T$ is a valid judgement; by hypothesis 1 and Lemma 8.1.13 we derive $A_1;\, ME_1 \cup ME_2 \vdash o :: T$;
- $(o,\, T) \in A_2$: by hypothesis 2 we know that $A_1;\, ME_1 \vdash o :: T$; by hypothesis 1 and Lemma 8.1.13 we derive $A_1;\, ME_1 \cup ME_2 \vdash o :: T$;

Case (INT)**:** trivial, as assumption sets and marked edges are not required to verify the premises of the rule;

Case $(STRING)$**:** trivial, as assumption sets and marked edges are not required to verify the premises of the rule;

Case $(EMPTY-COLL)$**:** trivial, as assumption sets and marked edges are not required to verify the premises of the rule.

$\boxed{i > 0}$ by induction assuming that the thesis holds for the premises of the rule at hand.

Case (REC)**:** that is $A_1 \cup A_2;\ ME_2 \vdash o :: \mu X.T$; the premise of this rule is the judgement $A_1 \cup A_2;\ ME_2 \vdash o :: T\left[{}^{\mu X.T}/_{X}\right]$. By IH we know that $A_1;\ ME_1 \cup ME_2 \vdash o :: T\left[{}^{\mu X.T}/_{X}\right]$. Applying (REC) we derive $A_1;\ ME_1 \cup ME_2 \vdash o :: \mu X.T$ as expected.

Case $(UNION-L)$**:** that is $A_1 \cup A_2;\ ME_2 \vdash o :: T_1 + T_2$; the premise of this rule is the judgement $A_1 \cup A_2;\ ME_2 \vdash o :: T_1$. By IH we know that $A_1;\ ME_1 \cup ME_2 \vdash o :: T_1$. Applying $(UNION-L)$ we derive $A_1;\ ME_1 \cup ME_2 \vdash T_1 + T_2$ as expected.

Case $(UNION-R)$**:** as for case $(UNION-L)$;

Case $(RECORD-COLL)$**:** that is $A_1 \cup A_2;\ ME_2 \vdash o :: [l_1 : T_1, \ldots, l_n : T_n]$. Note that, the premises of this judgement are $1 \leq j \leq k$ judgements

$$A_1 \cup A_2 \cup \{(o,\ [l_1 : T_1, \ldots, l_n : T_n])\};\ ME_2 \vdash o_j :: T'_j.$$

By Lemma 8.1.15, the judgement

$$A_1 \cup \{(o,\ [l_1 : T_1, \ldots, l_n : T_n])\};\ ME_1 \vdash^{\cdot} A_2$$

is valid. Therefore, by IH we can now state that $\forall j : 1, \ldots, k$:

$$A_1 \cup \{(o,\ [l_1 : T_1, \ldots, l_n : T_n])\};\ ME_1 \cup ME_2 \vdash o_j :: T'_j$$

Since by hypothesis 1 we know that $ME_2(o) = (ME_1 \cup ME_2)(o)$, (REC) can be applied again so as to achieve:

$$A_1;\ ME_1 \cup ME_2 \vdash o :: [l_1 : T_1, \ldots, l_n : T_n]$$

∎

So far, we have formally shown that the subcalls

$$\texttt{Extract}(A_j,\ E,\ o_j,\ T_j) = (\Delta A_j,\ ME_j),$$

with $1 \leq j \leq k$, of the record type case of the algorithm do verify the condition

$$\forall 1 \leq j_1 < j_2 \leq k.\, ME_{j_1} \dot{\asymp} ME_{j_2}.$$

The next Lemma applies lemma $\vdash^{\cdot}$-replacement, on the base of these conditions, so as to to prove the validity of a judgement that *almost* proves the *invariant* for the record type case call.

Lemma 8.1.17 *Let ST be a success tree for* `Extract`*, with* $\texttt{Extract}(A,\ E,\ o,\ T) \in ST_N$*, with* $T \equiv_{\mathrm{T}} [l_1 : T_1, \ldots, l_n : T_n]$ *and*

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

Let

$$\texttt{Extract}(A_j,\ E,\ o_j,\ T_j') = (\Delta A_j,\ ME_j)$$

with $1 \leq j \leq k$ be the children of this node in ST, then $\forall t : 1, \ldots k$:

$$A_t;\ ME_t \vdash^{\cdot} \Delta A_t \ \Rightarrow\ A \cup \{(o,\ T)\};\ \bigcup_{j=1}^{t} ME_j \vdash^{\cdot} \bigcup_{j=1}^{t} \Delta A_j.$$

Proof. We prove this statement by induction on the finite number k of subcalls invoked by the record type case call.

$\boxed{k = 1}$ by observing the algorithm, we know that $A_1 = A \cup \{(o,\ T)\}$; by hypothesis we know that $A_1;\ ME_1 \vdash^{\cdot} \Delta A_1$, which corresponds to what we need to prove, namely $A \cup \{(o,\ T)\};\ ME_1 \vdash^{\cdot} \Delta A_1$.

$\boxed{k > 0}$ if we consider that:

1. $(\bigcup_{j=1}^{k-1} ME_j) \dot{\asymp} ME_k$: direct consequence of Lemma 8.1.14;
2. by IH,

$$A \cup \{(o,\ T)\};\ \bigcup_{j=1}^{k-1} ME_j \vdash^{\cdot} \bigcup_{j=1}^{k-1} \Delta A_j$$

   and,

3. by hypothesis, A_k; $ME_k \vdash^{\cdot} \Delta A_k$, which is

$$A \cup \{(o,\, T)\} \cup \bigcup_{j=1}^{k-1} \Delta A_j;\; ME_k \vdash^{\cdot} \Delta A_k$$

we can apply Lemma 8.1.16, obtaining

$$A \cup \{(o,\, T)\};\; \bigcup_{j=1}^{k} ME_j \vdash^{\cdot} \bigcup_{j=1}^{k} \Delta A_j$$

■

As mentioned above, the result of the previous Lemma *almost* works out the invariant for the record type case of `Extract`. The following Theorem, on the basis of such result, provides a complete proof for our invariant.

Theorem 8.1.18 *(Invariant for* `Extract`*) Let A be an assumption set, E a set of edges, $o \in Obj$, $T \equiv_{\mathbf{T}} [l_1 : T_1, \dots, l_n : T_n] \in \mathbf{T}$, and ST the success tree of a call of* `Extract`*. If,*

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

and,

$$\texttt{Extract}(A,\, E,\, o,\, T) \in ST_N$$

then,

$$A,\, ME \vdash^{\cdot} \Delta A$$

Proof. We prove this statement by induction on the finite number i of recursive calls issued by `Extract`$(A,\, E,\, o,\, T)$ to terminate its execution.

$\boxed{i = 0}$

Case `Extract`$(A \cup \{(o,T)\},\, E,\, o, T) = (\emptyset,\; \emptyset)$**:** $A,\, \emptyset \vdash^{\cdot} \emptyset$ is trivially true;

Case `Extract`$(A,\, E,\, o,\, int) = (\emptyset,\, \emptyset)$**:** as for the case above;

Case `Extract`$(A,\, E,\, o,\, string) = (\emptyset,\, \emptyset)$**:** as for the case above;

Case `Extract`$(A,\, E,\, o,\, [l_1 : T_1, \dots, l_n : T_n]) = (\Delta A,\, ME)$**:** where

$$\Delta A = \{(o, [l_1 : T_1, \dots, l_n : T_n])\} \text{ and } ME = \bigcup_{i=1}^{n} \{< o, l_i, c, \emptyset_{U_i} >\};$$

this case occurs when

$$\forall i : 1 \ldots n. (T_i \equiv_{\mathrm{T}} \mathit{coll}(U_i)) \;\wedge\; (\mid E(l_i) \mid = 0).$$

The conclusion comes directly from the definition of `Extract`, which ensures that $A;\, ME \vdash o :: [l_1 : T_1, \ldots, l_n : T_n]$, from which $A;\, ME \vdash^{\cdot} \Delta A$.

$\boxed{i > 0}$

Case $\texttt{Extract}(A,\, E,\, o,\, \mu X.T) = (\Delta A, ME)$**:** the $i-1$'th step of execution is then

$$\texttt{Extract}(A,\, E,\, o,\, T\left[{}^{\mu X.T}/_{X}\right]) = (\Delta A, ME).$$

Thus, the thesis follows directly from the IH;

Case $\texttt{Extract}(A,\, E,\, o,\, T_1 + T_2) = (\Delta A, ME)$**:** the $i-1$'th step is then either

$$\texttt{Extract}(A,\, E,\, o,\, T_1) = (\Delta A, ME)$$

or

$$\texttt{Extract}(A,\, E,\, o,\, T_2) = (\Delta A, ME).$$

In either case, the thesis follows directly from the IH;

Case $\texttt{Extract}(A,\, E,\, o,\, [l_1 : T_1, \ldots, l_n : T_n]) = (\Delta A, ME)$**:** this call invokes $1 \leq j \leq k$:

$$\texttt{Extract}(A_j,\, E,\, o_j,\, T_j') = (\Delta A_j,\, ME_j)$$

calls, with $ME_j \neq \mathit{fail}$. All these calls require less than i calls to terminate. By IH we can assume that $\forall 1 \leq j \leq k. (A_j,\, ME_j \vdash^{\cdot} \Delta A_j)$ and, by the application of Lemma 8.1.17, obtain the following valid judgement:

$$A \cup \{(o,\, [l_1 : T_1, \ldots, l_n : T_n])\};\, \bigcup_{j=1}^{k} ME_j \vdash^{\cdot} \bigcup_{j=1}^{k} \Delta A_j$$

Since $\overleftarrow{\overline{ME}} = \{o\}$ and $o \notin \overleftarrow{\bigcup_{j=1}^{k} ME_j}$ (because $A_j(o) \downarrow$ for all $j : 1, \ldots, k$), we can deduce that $\overline{ME} \doteqdot (\bigcup_{j=1}^{k} ME_j)$ and apply Lemma 8.1.13 to achieve the valid $\vdash^{\cdot}$-judgement

$$A \cup \{(o,\, [l_1 : T_1, \ldots, l_n : T_n])\};\, ME \vdash^{\cdot} \bigcup_{j=1}^{k} \Delta A_j \qquad (8.1.1)$$

where $ME = \overline{ME} \cup \bigcup_{j=1}^{k} ME_j$.

Note that we can state that,

$$A \cup \{(o, [l_1 : T_1, \dots, l_n : T_n])\};\ ME \vdash o_j :: T'_j; \tag{8.1.2}$$

for all $j : 1, \dots, k$. The proof of this is quite intuitive, but we give it below for completeness of presentation. Consider the generic subcall

$$\texttt{Extract}(A_j, E, o_j, T'_j) = (\Delta A_j, ME_j);$$

by definition of `Extract` it can be that:

- $(o_j, T'_j) \in A_j$: o_j has been already extracted by an earlier call; since

  $$A_j = A \cup \{(o, [l_1 : T_1, \dots, l_n : T_n])\} \cup \bigcup_{t=1}^{j-1} \Delta A_t$$

  if $(o_j, T'_j) \in A \cup \{(o, [l_1 : T_1, \dots, l_n : T_n])\}$ the conclusion derives from the application of rule (HYP) to the $\vdash$-judgement 8.1.2; if $(o_j, T'_j) \in \bigcup_{t=1}^{j-1} \Delta A_t$, the conclusion comes directly from the $\vdash^{\cdot}$-judgement 8.1.1;
- $T'_j \in \{int, string\}$ and $o_j \in Atomic$: in this case the conclusion is trivial, as both rule (INT) or rule $(STRING)$ apply to the $\vdash$-judgement 8.1.2; in particular, as by hypothesis $ME_j \neq fail$, we know that o_j satisfies the atomic type T'_j;
- $(o_j, T'_j) \in \Delta A_j$: if none of the two cases above apply, since $ME_j \neq fail$, then o_j must have been extracted by the subcall; therefore, by definition of `Extract`, $(o_j, T'_j) \in \Delta A_j$; the conclusion derives directly from the $\vdash^{\cdot}$-judgement 8.1.2.

Therefore, as by definition of the algorithm the marked edges $\overline{ME}$ verify the requirements of $[l_1 : T_1, \dots, l_n : T_n]$, rule $(RECORD-COLL)$ can be applied to get the valid judgement:

$$A;\ ME \vdash o :: [l_1 : T_1, \dots, l_n : T_n]$$

which in turn clearly implies that

$$A;\ ME \vdash^{\cdot} \{(o, [l_1 : T_1, \dots, l_n : T_n])\}.$$

Considering that,

1. $\bigcup_{j=1}^{k} ME_j \dot{\asymp} ME$: trivial, as
$$ME = \overline{ME} \cup \bigcup_{j=1}^{k} ME_j$$
and
$$\overline{ME} \dot{\asymp} \bigcup_{j=1}^{k} ME_j.$$
2. $A;\ ME \vdash^{\cdot} \{(o,\ [l_1 : T_1, \dots , l_n : T_n])\}$; and
3. $A \cup \{(o,\ [l_1 : T_1, \dots , l_n : T_n])\};\ \bigcup_{j=1}^{k} ME_j \vdash^{\cdot} \bigcup_{j=1}^{k} \Delta A_j$.

by Lemma 8.1.16, we can state that
$$A;\ ME \vdash^{\cdot} \Delta A$$
where $\Delta A = \{(o,\ [l_1 : T_1, \dots , l_n : T_n])\} \cup \bigcup_{j=1}^{k} \Delta A_j$, is a valid judgement.

■

As we shall see in the following Theorem, soundness of typing is a trivial consequence of the invariant proven above.

Theorem 8.1.19 *(Soundness of typing) Let $s \in S$, $T \in \mathbf{T}$, $\texttt{Extraction}(s,\ T) = d$. Then, $d : T$.*

Proof. By definition of extraction,
$$\texttt{Extraction}(s,\ T) = d = (s_r,\ ME)$$
where $\texttt{Extract}(\emptyset,\ s_e,\ s_r,\ T) = (\Delta A,\ ME)$ and $ME \neq fail$. By Lemma 8.1.18 we know that $\emptyset;\ ME \vdash^{\cdot} \Delta A$. By definition of Extract it can be that:

1. $(s_r,\ T) \notin \Delta A$: this is when $T \in \{int, string\}$ and $s_r \in Atomic$; as $ME \neq fail$, we know that s_r satisfies the atomic type T; hence
$$\emptyset;\ \emptyset \vdash s_r :: T$$
is a valid judgement, from which
$$d = (s_r, \emptyset) : T$$
2. $(s_r,\ T) \in \Delta A$: by definition of $\vdash^{\cdot}$-judgement,
$$\emptyset;\ ME \vdash s_r :: T$$
is a valid judgement, from which
$$d = (s_r, ME) : T$$

■

8.2 Completeness

As shown in Chapter 4, completeness of an extraction algorithm with respect to extractability depends on the definition of the set of *values extractable from an SSDB with respect to a given type.* In the case of L this set is defined as follows.

Definition 8.2.1 *(Extractable values) Let $\overline{s} \in S$, $\overline{T} \in \mathbf{T}$. The set of values extractable from $\overline{s}$ according to $\overline{T}$ is defined as,*

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

The algorithm `Extraction` is not complete with respect to extractability. In other words, given $\overline{s} \in S$ and $\overline{T} \in \mathbf{T}$ such that $D_{\overline{s},\overline{T}}$ is not empty, the algorithm does not guarantee a successful termination.

In the following we provide examples of incompleteness and discuss their impact in the practical usage of our extraction system. Afterward, we show that `Extraction` is complete whenever its application is restricted to the domain of tree-structured SSDBs.

8.2.1 Cases of Incompleteness

We shall find that incompleteness is due to the presence of shared oids in the SSDB to be traversed by the algorithm, in combination with one of the following:

- the determinism introduced in the union type case of `Extract`;
- the determinism introduced in the extraction of one edge out of multiple candidate edges in the record type case of `Extract`;

or to the over-restrictive termination test in the record type case of `Extract`.

Union and record types determinism

Consider the example in Figure 8.1, where the SSDB $\overline{s}$ is extracted according to the type $\overline{T}$,

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

`Extract` begins its execution visiting o_1 with the record type $\overline{T}$. Since o_1 verifies the required properties, the algorithm invokes two recursive calls,

$$\texttt{Extract}(A_1, \overline{s}_e, o_2, [children : [age : int] + [child : string]])$$

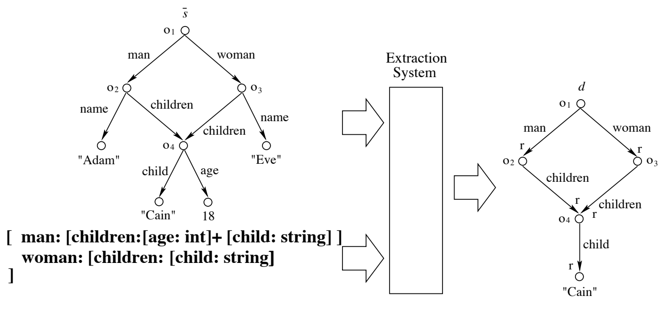


Figure 8.1: Example of incompleteness due to union types

where $A_1 = \{(o, \overline{T})\}$ and,

$$\texttt{Extract}(A_1 \cup \Delta A_1,\ \overline{s}_e,\ o_3,\ [\mathit{children} : [\mathit{child} : \mathit{string}]])$$

where ΔA_1 represents the result of the extraction of the previous call.

The first recursive call recursively issues the union type case of `Extract` with o_4 and $[\mathit{age} : \mathit{int}] + [\mathit{child} : \mathit{string}]$. Thus, the algorithm invokes a call of `Extract` with o_4 and the *first* type in the union, namely $[\mathit{age} : \mathit{int}]$, which will be successfully terminated, extracting the edge labelled as *age*. In summary, the call path of the first recursive call returns the pair $(\Delta A_1, ME_1)$ such that,

$$\Delta A_1 = \{(o_2,\ [\mathit{children} : [\mathit{age} : \mathit{int}] + [\mathit{child} : \mathit{string}]]),\\ (o_4,\ [\mathit{age} : \mathit{int}])\}$$

$$ME = \{< o_2, \mathit{children}, r, o_4 >, < o_4, \mathit{age}, r, 18 >\})$$

The second recursive call recursively visits o_3 with $[\mathit{children} : [\mathit{child} : \mathit{string}]]$, and then o_4 with $[\mathit{child} : \mathit{string}]$, under the assumptions A_2. The latter call will fail, because $(o_4, [\mathit{age} : \mathit{int}]) \in A_2$ and $[\mathit{age} : \mathit{int}] \not\equiv_{\mathrm{T}} [\mathit{child} : \mathit{string}]$.

Nevertheless, as shown in Figure 8.1, the algorithm could have extracted the value $d \in D_{\overline{s},\overline{T}}$. The failure is due to the fact that the algorithm does not support the non-determinism encountered when extracting according to the members of a union type. To do so, rather than fail and terminate, the algorithm should roll back to the first visit of o_4 with a union type and try the next member of the union.

As an example of the second kind of non-determinism, consider the scenario illustrated in Figure 8.2, where the SSDB $\overline{s}$ is extracted according to the type $\overline{T}$,

$$\begin{array}{l} [\\ \quad \mathit{child} : [\mathit{name} : \mathit{string}], \\ \quad \mathit{favouritechild} : [\mathit{age} : \mathit{string}]] \\ ]. \end{array}$$

`Extract` begins its execution extracting o_1 with the record type $\overline{T}$, which requires the existence of an edge labelled as *favouritechild* and of an edge labelled as *child*. Note that, o_1 features two edges that, being labelled as *child*, are candidate for the extraction; `Extraction` fails if the edge to be selected first is $< o_1, \mathit{child}, o_3 >$. Indeed, the extraction according to this edge would succeed and eventually lead to the extraction of o_3 with respect to $[\mathit{name} : \mathit{string}]$. Afterward, the call relative to the label *favouritechild* will then try to extract o_3 with respect to the type $[\mathit{age} : \mathit{int}]$, thereby causing the failure of `Extraction`.

However, observe that had `Extract` first selected the edge $< o_1, \mathit{child}, o_2 >$, `Extraction` would have successfully returned the value d as shown in the picture. As in the case of union types, the algorithm does not support the non-determinism brought into play by the edges candidate for an extraction. In the example above, the algorithm should have rolled back to the extraction of o_1 with the record type and try with another candidate edge labelled as *child*.

Non-determinism could be enforced by endowing the algorithm with backtracking techniques. The successful extraction of o with a record type T, which is a subterm of a union type U, should be memorised in a separate environment B as a tuple (o, T, o_U, U, A_U), where o_U is the oid that was extracted according to U, through which o was extracted, and A_U is the assumption set passed to the call with o_U and U. Whenever a subsequent record type case call fails on o because of a mismatch with T, the algorithm should roll-back to the extraction environment relative to the extraction with U and try the extraction of o with the next member of the union.

The same technique could be applied to the edges with the same label of a record type call. If the call fails, as exemplified above, and there are other non-tried candidate edges available, the algorithm should roll back and re-apply the extraction.

Termination test

Incompleteness is also due to the termination test in the record type case of `Extract`, which is too strong with respect to L's relation of typing. For example, the SSDB $\overline{s}$ in Figure 8.3 could be extracted according to the type,

$$\begin{array}{l} [\\ \quad \mathit{man} : [\mathit{children} : [\mathit{child} : [\mathit{name} : \mathit{string}] + T]], \\ \quad \mathit{woman} : [\mathit{children} : [\mathit{child} : [\mathit{name} : \mathit{string}]]] \\ ]. \end{array}$$

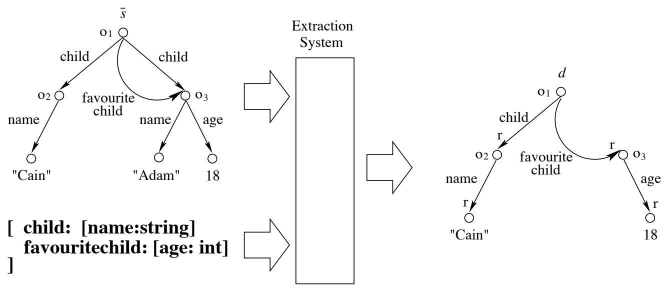


Figure 8.2: Example of incompleteness due to edges with the same label

returning the value d shown in the picture. Indeed, the oid o_5 could be extracted both with $[name : string] + T$, for all $T \in \mathbf{T}$, and with $[name : string]$.[1]

Unfortunately, the algorithm's termination test is so restrictive that it forbids this extraction before it can take place. When visiting for the second time the shared node o_4 with the type $[child : [name : string]]$ the termination test fails for the oid had been previously extracted according to the type $[child : [name : string] + T]$ and,

$$[child : [name : string]] \neq_{\mathbf{T}} [child : [name : string] + T].$$

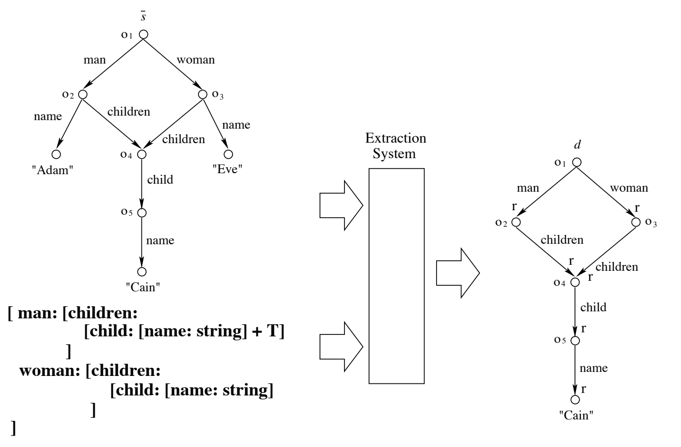
Observe that this happens whenever a generic oid (o_4 in the example above) is extracted according to record types differing for the union types they feature as subterms.

A partial solution to this problem consist in defining a relation of *subtyping* based on the rule $T_1 <_{\mathbf{T}} T_1 + T_2$. By means of this relation the termination test could be refined to check whether an oid falls in the category exemplified above and can therefore be extracted. For instance, the algorithm would successfully visit o_4 as $[child : [name : string]] <_{\mathbf{T}} [child : [name : string] + T]$.

Note that, to be effective, the test should be combined with the backtracking techniques mentioned above. Indeed, the the algorithm may first successfully visit and extract o_4 with $[child : [name : string] + T]$. Accordingly, the second visit with $[child : [name : string]]$ would not satisfy the subtyping check. Backtracking techniques over union types and equally labelled edges would make sure that the algorithm will try the other way around, thereby returning the correct extraction.

Enriched with subtyping test and backtracking techniques, `Extraction` would capture a larger set of extractable values, but would not yet be complete. There are

[1]Note that, with reference to the union type case anomaly discussed above, this extraction may be performed because, luckily, $[name : string]$ is the first matching member of the union type.



```
[ man: [children:
          [child: [name: string] + T]
       ]
  woman: [children:
            [child: [name: string]
         ]
]
```

Figure 8.3: Example of incompleteness due to over-restrictive termination test

other two main problems, whose solution would be too expensive to be implemented: the order of the labels of a record type, which imposes an order of visit of the edges emanating from an oid, and the presence of particular forms of shared oids. These oids are reached by different call paths of the algorithm with record types $T_1, \ldots, T_n$ that are not in subtyping relation and have a non-empty intersection, that is there exists T such that $\forall i : 1, \ldots, n : . T <_{\mathrm{T}} T_i$. In particular, these oids can be extracted according to T, and should therefore be extractable according to all T_i's, but the algorithm fails, as the types are not in subtyping relation. As an example of this scenario, consider again o_4 in the example above, and the two types $[\mathit{child} : [\mathit{name} : \mathit{string}] + T_1]$ and $[\mathit{child} : [\mathit{name} : \mathit{string}] + T_2]$, where $T_1 \not<_{\mathrm{T}} T_2$ and $T_2 \not<_{\mathrm{T}} T_1$. Clearly, d is extractable according to both types, but the algorithm fails.

Observations

The anomalies presented above prove that our algorithm is not complete with respect to extractability. However, a more complex algorithm could be developed, which, heavily exploiting backtracking techniques and a refined termination test, could provide the level of non-determinism required to capture almost all extractable values for any input $\overline{s}$ and $\overline{T}$.

Nonetheless, the anomalies shown above are quite peculiar and may occur only in presence of shared oids and cycles, which are in turn quite rare in SSDBs. In addition, their manifestation does not violate the semantic of the programming language L, but only hamper the extraction of an extractable value. Therefore, we believe the system can be successfully used also in presence of shared oids and cycles.

Finally, as we shall discuss in the next section, our algorithm can be proven complete with respect to tree-structured SSDBs. This restriction does not compromise the importance of our approach as most of the scenarios in the real world do not necessarily feature shared oids and cycles and fall in this category. For instance, the majority of semistructured databases available are represented as XML documents, which are based on tree-structured models.

8.2.2 Case of completeness

`Extraction` is not complete with respect to the definition of extractability, due to the extraction of typed values from SSDBs containing shared entities and cycles. However, if we restrict our attention to a smaller set of SSDBs, completeness can be proven.

Definition 8.2.2 *(Non-shared oids) Let $\overline{s} \in S$. $o \in Oid(\overline{s})$ is* non-shared *if there exists only one path from $\overline{s}_r$ to o.*

Definition 8.2.3 *(Tree-structured SSDBs) The set of* Tree-structured SSDBs *is defined as,*

$$S_t = \{s \in S \,|\, \forall o \in Oid(s).\ o \textit{ is non-shared}\}$$

Restricting to the set S_t ensures that the algorithm `Extraction` cannot successfully visit the same oid twice in the same call chain. Accordingly, if an oid is successfully extracted according to a record type, no termination test can cancel such extraction and the anomalies described earlier cannot take place. As the ΔA's and the termination test are no longer necessary, the algorithm `Extraction` in Figure 7.1 can be significantly simplified, to become the algorithm `Extraction`$_t$ shown in Figure 8.4.

Consider $d \in D_{\overline{s},\overline{T}}$. As d has no shared oids, the proof of conformity of d_r to $\overline{T}$ requires to check conformity of each $o \in Oid(d)$ with a record type $[l_1 : T_1, \dots, l_n : T_n]_o$ *only once*. Accordingly, as $ssd(d) < \overline{s}$, we also know that each oid $o \in Oid(ssd(d))$ *can* be reached by `Extract` with the record type $[l_1 : T_1, \dots, l_n : T_n]_o$. In particular, as o conforms to $[l_1 : T_1, \dots, l_n : T_n]_o$ and $Erase(d(o)) \subseteq \overline{s}(o)$, we know that this call can be performed successfully. Moreover, once extracted, o cannot be visited again in the call chain of `Extract`. Thus, the successful extraction of o cannot affect other extractions in the call chain.

In conclusion, `Extract`$_t(\overline{s}_e, \overline{s}_r, \overline{T})$ cannot fail and `Extraction`$_t$ is complete.

```
Extraction_t:  $S_t \times \mathbf{T} \rightarrow D \cup \{fail\}$
Extraction_t($\overline{s}$, $\overline{T}$) =
  { $ME$:= Extract_t($\overline{s}_e, \overline{s}_r$, $\overline{T}$)
    if $ME = fail$  then return $fail$
                    else return ($\overline{s}_r$, $ME$) }


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

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

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

case Extract_t($E$, $o$, $T_1 + T_2$)
  { $ME$ := Extract_t ($E$, $o$, $T_1$)
    if $ME = fail$  then return Extract_t($E$, $o$, $T_2$))
                    else return $ME$

case Extract_t($E$, $o$, $\overline{T}$), where $\overline{T} \equiv_{\mathbf{T}} [l_1 : T_1, \ldots, l_n : T_n]$
  { $ME := \emptyset$
    for $i := 1$ to $n$ do
    {    $FAILED := true$
         if $T_i \equiv_{\mathbf{T}} coll(U)$
                then if $E(o,l_i) = \emptyset$
                        then { $ME := ME \cup \{<\overleftarrow{e}, l_i, c, \emptyset_U>\}$  <introduces an empty collection edge>
                               $FAILED := false$ }
                        else for all $e \in E(o,l_i)$ do
                              { $ME_t :=$ Extract_t($E$, $\overrightarrow{e}$, $U$)
                                if $ME_t \neq fail$ then
                                { $ME := ME \cup ME_t \cup$ { $<\overleftarrow{e}, l_i, c, \overrightarrow{e}>$}
                                  $FAILED := false$ } }
                else for all $e \in E(o,l_i)$ do
                     { $ME_t :=$ Extract_t($E$, $\overrightarrow{e}$, $T_i$)
                       if $ME_t \neq fail$ then
                          {   $ME := ME \cup ME_t \cup$ { $<\overleftarrow{e}, l_i, r, \overrightarrow{e}>$}
                              $FAILED := false$
                              exitfor; } }
         if $FAILED$ then  <one record's label could not be satisfied by $E(o)$>
                { $ME := fail$
                  $exitfor$ } } }
    return $ME$ }

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

Figure 8.4: Extraction algorithm for tree-structured SSDBs

Chapter 9

SNAQue

In this Chapter we describe a novel and complementary approach to static typing approaches for querying XML, based on the extraction mechanisms presented in Chapter 4.

We present a prototype of the system *SNAQue* – the *Strathclyde Novel Architecture for Querying Extensible markup language* (cf. [46, 47]) – currently under development at Strathclyde University, Glasgow (UK). SNAQue will enable programming languages interlaced with CORBA to be used in querying the information represented in XML format.

SNAQue is based on an extraction algorithm $\texttt{EXTR}_{IDL}$ for the *Interface Definition Language* (*IDL*) of CORBA (cf. [79]), which extracts Java objects corresponding to regular subsets of XML SSDBs. Given an IDL definition and an XML SSDB, our system returns a CORBA object that serves the extracted value across the ORB framework.

In describing the system, we assume the reader has a some understanding of concepts such as *skeletons* and *stubs* in CORBA.

9.1 The prototype

CORBA IDL is an interface definition language based on the syntax of the type language of *C*. IDL describes the most common type abstractions of programming languages, plus some forms of behaviour, such as functions and procedures. In particular, specific mappings from IDL definitions to the types of programming languages (Java, C, and many others) are available, which implicitly define a typing relation between the values of the language and IDL definitions.

The definition of extractability underlying the system SNAQue is based on the mapping `JIDL` from IDL to Java classes, according to which Java objects are in a typing relation with the IDL definitions that map onto their corresponding classes. Hence, a Java object is extractable from an XML SSDB according to an IDL definition if:

- it *conforms* to the IDL definition: the object is of the class to which maps the IDL definition;
- the object *maps* onto an SSDB *included* into the XML SSDB.

Due to the quite straightforward mappings:

- `IDLtoT`: from a subset of IDL, that does not include interfaces, onto the type system $\mathbf{T}$;
- `XMLtoS`: from XML SSDBs to tree-structured SSDB S_t;
- `JAVAtoD`: from Java objects to the values D;

we formally defined extractability based on the definition of extractability for L.

Definition 9.1.1 *(Extractability for IDL and Java) A Java object j is extractable from an XML SSDB xml according to an IDL definition idl if there exists $d \in D$ such that* `JAVAtoD`$(j) = d$*, and d is extractable from* `XMLtoS`(xml) *according to* `IDLtoT`(idl)*.*

Similarly, the algorithm `EXTR`$_{IDL}$, sound with respect to this definition, exploits the algorithm `Extraction`$_t$ defined in Chapter 7.[1] Specifically, given the input *xml* and *idl*, `EXTR`$_{IDL}$ performs the following steps:

1. relying on parsers `IDLtoT` and `XMLtoS` that translate from CORBA IDL to $\mathbf{T}$ and from XML documents to S_t, calculates a type $T =$ `IDLtoT`(idl) and an SSDB $s =$ `XMLtoS`(xml);
2. executes the call `Extraction`$_t(s, T)$, which here we assume to be successful; the call yields a value d such that $d : T$; due to the mapping `IDLtoT`, *d structurally and statically conforms* to *idl* too;
3. generates a Java object j corresponding to d such that j conforms to *idl*.

Note that the equivalence between values of different languages identified by CORBA is similar in principle to the one identified with a mapping between the values of a language and SSDBs. Hence, given the mapping $ssd($`JAVAtoD`$(x))$ from Java objects to SSDBs, any CORBA-compliant language may transparently access the extracted value via the given IDL definition. SNAQue (see Figure 9.1) makes this possible by serving the Java object resulting from `EXTR`$_{IDL}$ across the ORB. Specifically, the system,

[1]Recall that `Extraction`$_t$ takes as input a tree-structured SSDB s and a type T from the type system $\mathbf{T}$ of the language L; the result of the extraction is a value d of type T, this value corresponding to a subset s' of s.

1. generates the CORBA interface I_{idl} corresponding to the simple CORBA definition *idl*: I_{idl} describes a CORBA object that has a method `getData` returning an object of type *idl*;
2. defines the CORBA object, by providing a Java implementation for I_{idl}, that is a Java class whose method `getData` returns the Java object j.

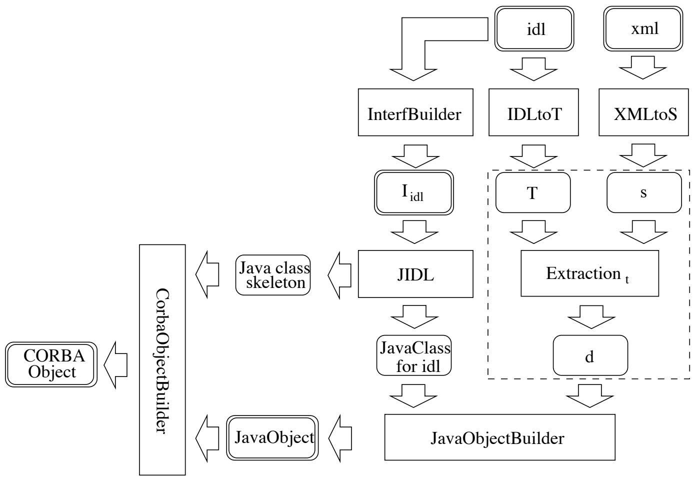


Figure 9.1: SNAQue: extraction of CORBA objects from XML SSDBs.

As described in Chapter 4, the approach consists of two phases. First, a programmer attempts to project an arbitrary XML document onto a given IDL type with SNAQue. If the extraction can be performed, the system creates a CORBA object whose unique method `getData` returns the representation of the extracted value. Secondly, programs in a general-purpose CORBA compliant language can be written with respect to this object, and refer to the object extracted value with a call to the method `getData`.

In the following sections, we formally prove the correctness of SNAQue by providing a mapping from XML documents onto the set S_t and a mapping from IDL definitions onto the types in $\mathbf{T}$. Moreover, we shall observe how the Java object j conforming to *idl* is generated according to a one-to-one mapping from the extracted value d.

Finally, we shall see how j, i.e. the representation of the extracted value, is served to consuming distributed applications via a CORBA object corresponding to I_{idl}.

9.2 From XML documents to SSDBs

There are strong similarities between the logical part of the XML format and the graph-based data models for SSD. Indeed, as illustrated in Chapter 2 (see Figure 2.3), an XML document can be modelled by an SSDB in which oids correspond to document elements and the edge relation corresponds to the nesting relation for elements. In particular, the prototype of SNAQue focuses on mapping from the core subset of the XML format, namely that relative to the logical structure of documents, onto the set S_t of tree-structured SSDBs. To achieve this, XML documents are not considered under the influence of a schema, thus interpreting attribute values as being of type `STRINGTYPE`. This excludes the interpretation of XML documents as graphs through the attribute types `ID` and `IDREF`. The reason for this is that we aim at giving a proof of soundness of the prototype and not a complete data model for XML.

In the following we shall describe the specific design choices characterising the mapping from XML to SSDBs underlying SNAQue. For simplicity, we shall illustrate this mapping by means of graphic representations of graphs rather than relying on the syntactic representation for SSDBs in S_t.

9.2.1 Ordering

Elements of XML documents are ordered according to their position in the text. Ordering of elements is in contrast to most SSD models where two SSDBs are equal regardless of the order of siblings (see Section 5.2). Although order can be easily captured in SSD data models, it complicates the semantics of the corresponding query languages and reduces their performance [88]. The distinction, however, is of no real importance for our purposes, as we can always replace collection of edges with lists of edges in the definition of S_t, and collection types with list types in the type system.

9.2.2 Attributes

XML makes a clear distinction between elements and attributes, although they are similar forms of labelled representation.

An attribute appears always in relation to an element, and has a unique name in the scope of that element. Attribute values are always enclosed in quotation marks, and have a different meaning depending on the type with which they are declared in the document's schema, if one exists. Specifically, attributes can be of type:

- STRINGTYPE or ENUMERATION: their value is interpreted respectively as one or many tokens of free text;
- ID, IDREF and IDREFS: they are used to enhance the representation of relationships between data beyond simple nesting (e.g. shared and cyclic data); an attribute of type ID specifies a name for the element that is unique across the document and can be referenced by attributes of type IDREF (single reference), or IDREFS (multiple references).

As to the modelling of attributes, most approaches distinguish according to the type. An attribute of type STRINGTYPE is usually associated with the oid that models the associated element; it is then lexically distinguished from an element, in query expressions of related languages. Attributes of type ENUMERATION are usually ignored.

Matching attributes of type ID and IDREF/IDREFS instead, are interpreted together as edges between matching vertices. Such edges originate from the oid associated with the referencing attribute, are labelled with the name of the attribute, and reach the oid associated with the element containing the reference attribute. Of course, the correct interpretation of attributes of type ID, IDREF, and IDREFS depends on the availability of a documents DTD, or the like. Without a schema, ID attributes are usually discarded and IDREF/IDREFS attributes can only be interpreted as strings, as the *literal mode* of [62].

In our prototype, we adopt the latter strategy and avoid reference/referencing attributes. Moreover, we view attributes of type STRINGTYPE and ENUMERATION as separate, childless elements. We do this because we are not constrained by the need of recovering the XML document from the associated SSDB. For an example of the mapping, see Figure 9.2.

```
.....
<COORDINATOR HOMEPAGE="www.cs.strath.ac.uk/~richard">
  <NAME> R.Connor </NAME>
</COORDINATOR>
.....
```

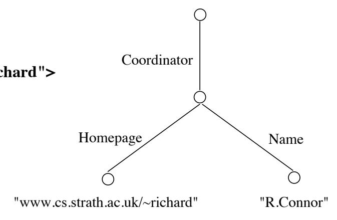


Figure 9.2: Example of mapping for attribute.

9.2.3 Elements with mixed content

An XML element has either an element content, when it contains sub-elements but not free text, or a mixed content, when it contains free text optionally intermixed with other elements.

The mapping of elements with mixed content requires some arbitrary decisions to be made. However, although such elements appear quite often in XML documents, the issue is usually ignored in most approaches. Consider, for example the simple fragment of XML in Figure 9.3.

```
<AUTHOR> Richard Connor
<DEPT> Computer Science </DEPT>
</AUTHOR>
```

Figure 9.3: Example of mixed elements

Intuitively, we would associate the text `Richard Connor` with a childless oid and draw and edge to this oid from the oid associated with the element `AUTHOR`. The problem arises when we consider the label of such an edge, as we do not have one. We solve this problem by labelling these problematic edges with the name of the element associated with their source oid. Such choice can be justified on a semantic ground, as element names are meant to identify the meaning of the content they surround.

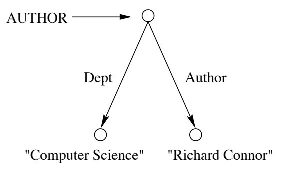


Figure 9.4: Example of graph representation for mixed elements.

For example, the XML fragment in figure 9.3 is modelled by the graph in figure 9.4. Intuitively, in terms of our SSD model, the XML fragment describes an entity corresponding to an author: the entity qualifies itself with name `Richard Connor` and with associated department that of `Computer Science`.

Notice that the same situation occurs when an element has some attributes of type `STRINGTYPE` or `ENUMERATION` as well as a content of sole text (see Figure 9.5).

9.2.4 Empty Elements

XML allows elements that have no content at all, i.e. empty elements. These are denoted by either a succession of start and end tags, such as `<author> </author>`, or by a single empty-element tag, namely `<author/>`. Due to the commitment to flexibility of XML, empty elements have an ambiguous semantics, where possible

```
.....
<NAME OFFICEMATE="ref">
   <FIRSTNAME> Paolo </FIRSTNAME>
   <SECONDNAME> Manghi <SECONDNAME>
</NAME>
.....
<NAME ID="ref"> F.Simeoni </NAME>
.....
```

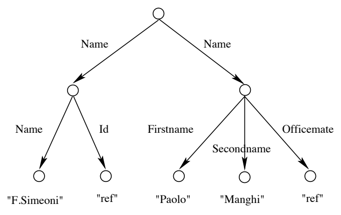


Figure 9.5: Example of mapping for attribute of text elements.

interpretations are that they describe truth or stand as placeholders in partial information. While the first interpretation implies a misuse of modelling concepts, as meta-data would be used as data, the second recalls *null* values in relational databases and is at least questionable in this scenario. Although, we could not represent empty elements altogether, we found more appropriate to our mapping to choose the second interpretation and model empty elements as childless vertices with an associated value of *undefined*. This requires the non-influential addition of this new value to *Atomic* in the definition of S_t. In particular, *undefined* belongs to all types in $\mathbf{T}$. As a result, `<author/>` maps onto the SSDB illustrated in Figure 9.6.

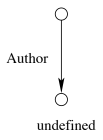


Figure 9.6: Example of graph representation for empty elements

Finally, Figure 9.7 shows the modelling of a fragment of XML that includes all the problematic features discussed above.

9.3 From CORBA IDL to types of L

The prototype of SNAQue performs extraction of Java objects with respect to the subset of IDL that maps onto the types in T.

The mapping onto atomic, record, and collection types is relatively straightforward. Indeed, our type language supports only integers and strings as atomic types, which map from *long* and *string* respectively in IDL. Moreover, record and collection types map from *struct* and *sequence* types respectively:

```
<HIPPO_RESEARCH_GROUP>
  <NAME>
    <FIRSTNAME> Paolo </FIRSTNAME>
    <SECONDNAME> Manghi <SECONDNAME>
    <HOMEPAGE></HOMEPAGE>
  </NAME>
  <NAME> F.Simeoni </NAME>
  <NAME> S.Neely </NAME>
  <COORDINATOR HOMEPAGE="www.cs.strath.ac.uk/~richard">
    <NAME> R.Connor </NAME>
  </COORDINATOR>
</HIPPO_RESEARCH_GROUP>
```

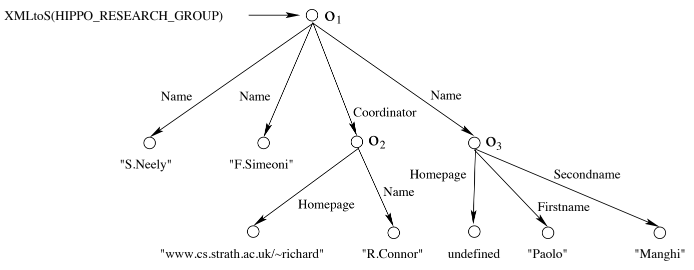


Figure 9.7: An example of our mapping from XML onto graphs.

IDLtoT(`long`) $= Int$
IDLtoT(`string`) $= String$
IDLtoT(`struct l {T`$_1$ `l`$_1$`; ...; T`$_n$ `l`$_n$`}`) $= [l_1 : T_1, \ldots, l_n : T_n]$
IDLtoT(`sequence T l`) $= [l : coll(T)]$

Union types pose a minor problem, as we made the decision to model with untagged unions in **T**, whereas IDL provides tagged unions. This led to a decision to either add tags to our own union types, which would present no special problem, or else to make some arbitrary decisions in the mapping. The latter course was chosen for purely pragmatic reasons.

We do not support mappings from IDL interface definitions. The purpose of such definitions is to model object-oriented classes, incorporating both data and behaviour. We do not, at present, expect behaviour definitions to be included within the source XML, and there is no sense in which they could be mapped to our data-description type language. [2]

[2]Notice however that, in object-oriented host languages, *struct* definitions are translated by IDL compilers into classes. The behaviour associated with these classes is simply to update and retrieve the values of instance variables corresponding to the given *struct* fields.

Finally, we require our IDL definitions to end with a `typedef` declaration, which declares a single type variable, and take this as the *main definition* for our purposes. This definition may rely upon others in the sequence; any in the sequence that are redundant with respect to this purpose are ignored. An example of SNAQue-valid IDL definition is illustrated in Figure 9.8.

```
struct bustype { string coachbuilder; long noOfSeats };
struct cartype { string make; long topSpeed };
typedef union { bustype bus; cartype car } vehicle;
```

Figure 9.8: Example of main definition.

Note that, once a value is extracted according to a main definition, SNAQue needs to define an IDL interface to serve it across the ORB. As earlier mentioned, this interface will simply introduce a method `getData` that returns an object of the type defined in the main definition.

9.3.1 Sequences and Unordered Collections

As mentioned previously, there is an important semantic difference in the treatment of collection types in the core system and both XML and CORBA IDL. The collection type of the core system represents an unordered collection, in keeping with the graph-based semantic model. In XML, the order in which tags appear within the text is significant, giving a repeated tag the semantics of an ordered collection or list. Equally, the sequence type constructor in IDL translates in various languages to ordered bulk constructors, for example to array in Java.

However, we need take no further account of this in the implementation. Although our core system has an unordered semantics, it is implemented on a machine by a representation which has an implicit ordering. So long as our algorithms for parsing and extracting the data work from beginning to end in a consistent manner, then the representation of the extracted subset will preserve the same order as the XML input. When this data is lifted and placed in a Java array behind the generated interface, once again the order is preserved, giving a total preservation of order from the original XML through to the CORBA interface, and then through whatever language is used to access it.

9.3.2 Unions

There are two possible semantic mappings from the labelled unions of IDL to XML data.

The first, which we adopt, assumes that the labels are not significant within the original data, but are instead only used as a programming aid to test for the structure of the underlying data. This interpretation allows a direct mapping from

IDL labelled unions onto the untagged unions of our type system, as the retrieved data perfectly matches the semantics of both types.

The second is that the labels are significant entities within the XML, and the union construct can then be used to abstract over different tags, rather than different structures.

For example, the main definition `vehicle` in Figure 9.8, could be expected to describe either of the XML examples:

```
<VEHICLE>
<COACHBUILDER>Alexander</COACHBUILDER>
<NOOFSEATS>43</NOOFSEATS>
</VEHICLE>

<VEHICLE>
<BUS>
<COACHBUILDER>Alexander</COACHBUILDER>
<NOOFSEATS>43</NOOFSEATS>
</BUS>
</VEHICLE>
```

Our choice, which is purely arbitrary, is the first:

$$\texttt{IDLtoT(union(}T_1\ l_1;\ T_2\ l_2\texttt{))} = T_1 + T_2$$

The identifier `bus` is then significant in the final query code, where it is used as a shorthand for the expected structure.

9.4 CORBA object instantiation

In the event of a successful call to $\texttt{Extraction}_t$, with $\texttt{XMLtoS}(xml)$ and $\texttt{IDLtoT}(idl)$, returning a value d, $\texttt{EXTR}_{IDL}$ must create a Java object j such that $\texttt{JAVAtoD}(j) = d$. Once j is created, SNAQue generates a CORBA object that serves the extracted value across the ORB framework.

Both operations, illustrated in Figure 9.1, depend on the generation of an IDL interface I_{idl} corresponding to *idl*. I_{idl}, which has an automatically generated unique name, defines a single method `getData`, whose return type is *idl*. For example, the IDL definition:

```
struct person {string name};
struct data {long age};
typdef sequence <person> people;
```

extracts a value *d* of the form illustrated in Figure 9.9, and translates in the interface:

```
struct person {string name};
typdef sequence <person> people;
interface Getting_Data28103
{
  people getData()
}
```

where the IDL structure `data` has been removed as it was not reachable from the main definition.

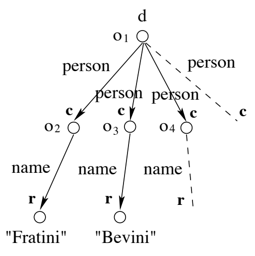


Figure 9.9: Extracted value

Based on I_{idl}, SNAQue instantiates the CORBA object as follows:

1. compiles I_{idl} with *JIDL*, a compiler from IDL to Java, and gets the Java class skeleton for I_{idl} and the Java classes for *idl*;
2. creates a Java object j, an instance of the classes for *idl*, that corresponds to *d*;
3. creates the CORBA object corresponding to I_{idl}: that is a Java object with a unique method `getData` returning the Java object j.

With reference to our example, the interface `Getting_Data28103` defines a sequence `people` of structures `person`, which in turn have one property `name`. When this interface is JIDLed, i.e. processed by a JIDL compiler, three Java source code files are produced along with a number of *helper* classes: the class skeleton `Getting_Data28103`, and the classes `peopleType.java` and `personType.java`.

The class `Getting_Data28103` corresponds to the interface and declares the method `getData()`, returning an object of type `peopleType`. The `peopleType` class contains a constructor and methods to access the elements of the sequence. The `personType` class contains a constructor and one attribute `name` of type string.

This Java source code is generated, manipulated, compiled, and executed by SNAQue to construct first j and then the related CORBA object.

Achieving this requires dynamic class source code creation, method/class name lookup, and compilation. Programs must examine themselves and manipulate internal properties. Programming languages that support *reflection* can achieve this. Reflection is a language mechanism which enables computations to generate language code, compile it and execute it at run-time.

SNAQue uses the Java reflection API, which provides the means for determining the fields, constructors and methods of Java objects and classes, for:

- *creating the Java object*: Java reflection API offers the methods `forName`, `getMethod`, and `getReturnType`, which are used by the system to discover the return type of the `getData` method of the skeleton. In our example this type is the class `peopleType`. The constructors of this class, and arguments to these constructors, are also determined by reflection. The object j is generated by recursively running through the structure of d, in conjunction with the structure of each class definition, and calling the appropriate class constructors as the recursion unwinds.

  This process is based on the static constraint governing the nature of d so that it conforms in structure to *idl*, hence to the Java classes. In our example, the system gets and uses the constructors of the classes `peopleType` and `personType`. A new element of an array of class `peopleType` is created for each oid reachable from the root of d (see Figure 9.9) with a label *person*, i.e. o_2, o_3, and alike. Furthermore, each of these oids must have an outgoing edge labeled as *name*, which corresponds to the instance variable `name` of objects of the class `personType`. Such variable can then be instantiated with the target values of these edges, i.e. *Fratini*, *Bevini* and so on.

- *creating the CORBA object*: the CORBA object is generated by compiling and instantiating the Java class skeleton corresponding to the given IDL interface, where the method `getData` is filled in by SNAQue with Java code that simply returns the newly created object j. In our example, `getData` is completed with code that returns the Java object of class `peopleType` constructed according to the process exemplified above.

Once the CORBA object has been instantiated, any CORBA client can access it through the corresponding stub and operate over the extracted value by invoking the method `getData`.

At the current stage of development, the execution of SNAQue is requested by an invocation to a CORBA object. In particular, both the XML data and the IDL definition reside on the server side of the CORBA gateway. A client program makes a call, through a stub to this object, to a distinguished method at the server. The server executes this method, which performs the projection of the XML data onto the given IDL definition. Future versions of SNAQue, will provide a Web interface through which users can specify an XML URI, an IDL definition, and perform the extraction they need.

9.5 Experience with the prototype

The initial data set was chosen from the numerous biodiversity database projects undertaken by the Palaeobiology Research Group at the University of Glasgow. A test load, documenting the major groups of marine organisms that colonised Tropical America over the last 30 million years, has especially been identified as suitable for experimentation within the approach proposed. This data includes several thousand genera and is insufficiently regular to allow the use of conventional database technology to address the range of queries required. Whilst the bulk of the data is highly structured taxonomic information, there are significant amounts of less structured data that should be included. For example, morphological characteristics, which vary widely in the different groups of organisms, and other crucial data such as population densities, images, life habits and habitats. The data as we received it is in the form of a treatise: it has no explicit structure in terms of tags or labels but it does have a high degree of implicit structure. This implicit structure exists because of a standard format developed by palaeobiologists, over several hundred years, for documenting their findings. Given an understanding of this standard format it was possible to write a parser that converted the plain text into XML. A section of the plain text follows:

> **?Apsilingula** WILLIAMS, 1977, p. 403 [*A. parkesensis; OD]. Elongate oval with subparallel lateral margins; dorsal and ventral pseudointerareas poorly known; both valves strongly thickened posteriorly, with deeply impressed muscle scars; ventral visceral area extending to midvalve; transmedian scars possibly

A portion of the corresponding XML, as produced by the parser, is shown in Figure 9.10. The resulting XML is structured but irregular; for example, not every genus has a *shell*. wInformation of this kind is of vital importance to the study of biodiversity, in terms of analysing extinction patterns, and is still normally gathered by manual inspection of the text of the treatise.

Note, however, that every genus has a *name*, an *author* and a *date*. We use these fields for an example application, which is to print a list of all these attributes for each genus in order of the recorded dates. Operations of higher complexity could be performed over these data, which include the insertion in a DBMS and/or their combination with data stored in other DBMSs or SSDBs.

As a first step, we illustrate the process of extraction of a regular subset of our XML SSDB corresponding to an object with IDL,

```
<FAMILY>
   <GENUS>
      <NAME confirmed="false">Apsilingula</NAME>
      <AUTHOR>WILLIAMS</AUTHOR>
      <DATE>1977</DATE>
      <PAGE>p. 403</PAGE>
      <SYNONYM>[*A. parkesensis; OD]</SYNONYM>
      <DESCRIPTION>
      <SHELL>Elongate oval with subparallel lateral margins</SHELL>
      <VALVES>both valves strongly thickened posteriorly </VALVES>
      ⋮
   </GENUS>
   ⋮
</FAMILY>
```

Figure 9.10: XML source produced by the parser.

```
struct nametype {string confirmed; string name;};
struct genustype {nametype name; string author; long date;};
typedef sequence <genustype> genus;
```

which translates in the type,

```
[ genus:  coll([author:  string;
                name:    [confirmed:  string; name:  string];
                date:    int ])
]
```

Figure 9.11 shows the value d extracted from the SSDB in Figure 9.10.

SNAQue returns a CORBA object, implemented in Java as an object of class Getting_Data10, that serves the value d across the ORB. Client programmers that want to use this CORBA object, must first compile its IDL interface, in order to get a Java class stub to the object. Then, around the stub, they can write and compile the code of consuming applications.

For simplicity, we give an example of a Java client in Figure 9.12.[3] We assume the IDL interface has been compiled, and that *xmlObj* (line 1) is the name reference to the CORBA object across the ORB framework.

The client coerces, i.e. *narrows*, the reference *xmlObj* to a corresponding Java object of type Getting_Data10 (line 1). In particular, *xmlObj* is narrowed to p, which becomes the reference to the CORBA object for the consuming applications.

[3]Note that our client was written in Java just for simplicity, while *xmlObj* could have been coerced into an object of any CORBA-compliant language.

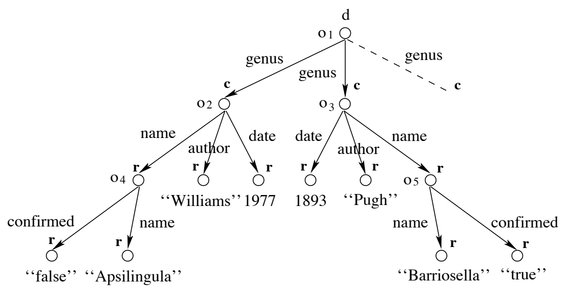


Figure 9.11: Extracted value d

Indeed, calling the `getData` method (line 2) of p returns an array of `genustype` objects as defined by the IDL and in the mapping from CORBA to Java.

The array of `genustype` objects contains Java objects corresponding to the two structs in the IDL. The code outputs all the data in the array (line $3 - 5$) and then uses a quicksort algorithm to sort them by date (line 6). The sorted array is then displayed (lines $7 - 9$).

```
1) Getting_Data10 p = Getting_Data10Helper.narrow(xmlObj);
2) getting10.Getting_Data10Package.genustype[] obj_data = p.getData();
3) System.out.println("Name, Author, Year");
4) for (int i=0; i<obj_data.length; i++)
5)     System.out.println(obj_data[i].name.name + ", " +
                          obj_data[i].author + ", " +
                          obj_data[i].date + ".\n");
6) QuickSort(obj_data, 0, obj_data.length-1);
7) System.out.println("\nSorted data:");
8) for (int i=0; i<obj_data.length; i++)
9)     System.out.println(obj_data[i].name.name + ", " +
                          obj_data[i].author + ", " +
                          obj_data[i].date + ".\n");
```

Figure 9.12: Client query in Java.

SNAQue identifies a novel concept that we believe to be of significant importance within the field of querying XML data. Rather than adopting the more common approach of designing a special-purpose query language for the domain in particular, SNAQue provides a semantic model through which the information encoded in XML can be exported into the domain of traditional programming systems. We do not claim that this approach is in any sense better, but that it is complementary in that it is stronger for certain classes of problem. In particular it seems to be well-suited to querying tasks which require generalised computation, especially over large sets of data.

Efficiency aspects of the approach are also clearly important; we believe in pragmatic terms that the subset creation is quite tractable. However, we are determined to go through a deep analysis of performance as long as SNAQue will reach a usable version, so as to provide precise results and claims about usability and effectiveness of our approach.

There are many future directions for this research topic. CORBA has been chosen as a convenient proof of concept mechanism, but the same approach can be used for any programmable domain that is based around a static typing discipline. For example it seems an interesting idea to use a similar mechanism to translate from XML into a relational database.

Chapter 10

Conclusions and Future Issues

The main contribution of this thesis is to have laid the foundations of mechanisms enabling applications of typed programming languages to indirectly compute over regular subsets of SSDBs. In itself, this result is a step forward towards the integration of SSD with structured data and typed programming languages.

Known approaches for typing SSDBs either statically impose a type and constrain data population or infer a type from the database after data population. The first offer the advantages of static typing at the cost of limiting the number of irregularities in the database. The second, while enabling unconstrained database population, offer only part of these advantages. Our approach is complementary to these, as it fully recovers the benefits of static typing after data population. We believe that in combination with navigational approaches, extraction mechanisms may provide complete functionality for SSD.

In particular, this work provides a specification for the realisation of an extraction mechanism in a typed programming language. Any language can be associated with a notion of extractability, which captures the concept of value extractable from an SSDB according to a given type. Based on extractability, an extraction algorithm can thus be constructed. Moreover, algorithm's correctness can be proven verifying specific properties of soundness and completeness with respect to extractability.

In order to show the feasibility of extraction mechanisms for most programming languages currently in use, we have realised an extraction mechanism for a representative typed language L featuring a set of standard types. First, we have defined extractability of L and then given a corresponding algorithm `Extraction`. Given an SSDB and a type of L, `Extraction` returns an extractable value, along with the measures of precision and data capturing. By interpreting these measures, users may be able to improve the results of their subsequent extractions.

`Extraction` is proven to be sound with respect to extractability for L, but not complete. While soundness is fundamental for extraction algorithms, however, we have shown that in many application contexts the incompleteness of `Extraction` has no relevant impact. This is the case for tree-structured SSDBs, for which we provide a specific complete algorithm $\texttt{Extraction}_t$.

Finally, as a practical example of an extraction mechanism, we presented the system SNAQuE. SNAQuE extracts Java objects from XML SSDBs and is built around the algorithm $\texttt{Extraction}_t$. Indeed, the system focuses on XML documents interpreted as tree-structured SSDBs and a subset of Java types that matches the types of L. A prototype of SNAQuE has been developed and is currently under improvement at the Computer Science Department of Strathclyde University.

We believe that important and immediate enhancements to our work should be the following:

Complete algorithm Compared with others, which typically disregard these issues, the query methodology derived from the usage of `Extraction` enables applications to indirectly operate over SSDBs with shared oids. On the other hand, the presence of shared oids is the principal cause of algorithm's incompleteness. We have seen that incompleteness is due to quite specific combination of types and SSDBs and does not generally represent a problem for users. However, providing a reliable version of the algorithm would certainly be a stronger result. Therefore, an important future issue is that of realising a version of `Extraction`, only suggested in this work, which proves to be complete with respect to extractability.

Formal definition of extractability for a language Extractability is based on a mapping from values of the language to SSDBs and on a definition of inclusion between SSDBs. While the mapping is strictly related with the target language, the inclusion relation is language-independent. Indeed, the behaviour of an extraction mechanism for a specific language varies depending on the form of inclusion adopted.

An algebra for the definition of inclusion relations could be a useful tool for extraction mechanism designers. Such an algebra, for instance, may support operators for the definition of equivalences between different sets of labels or for the construction of particular structural associations between SSDBs. Hence, designers could first define a mapping from SSDBs onto the language values, and then construct a customised definition of inclusion depending on the specific SSD scenario.

Measures When successful, `Extraction` returns also the measures of precision and data capturing. We have seen that their importance is that of disclosing to users the backstage of the extraction process. As a further improvement, other measures may be conceived to provide different forms of feedback to the users. For example, users may be interested in a single quantity qualifying the overall precision, rather than relying on a list of record-specific quantifications.

Experiments Due to the absence of a working implementation, the need for extraction systems stands on a strong claim only. Thus, once completed, the SNAQuE system will be deeply tested so as to provide evidence for the usability and effectiveness of our approach.

In summary, extraction mechanisms bridge the SSD domain to typed programming language domains. Accordingly, they become useful tools in many application contexts, as both extensions to extant programming environments or platforms for the construction of new SSD-based programming systems. In the following we show some ideas which may give rise to novel research avenues.

10.1 Customised extraction mechanisms

The benefits of an extraction mechanism depend on the host language. In this work, for example, we have described the system SNAQuE, which extracts Java objects from SSDBs. This gives users the immediate advantage of writing type-correct Java applications over subsets of SSDBs, as well as specifying computations over high-level types rather than over simple labelled graphs. A further peculiarity of SNAQuE is that Java objects can be served as a CORBA service across the CORBA ORB framework.

Moreover, through the appropriate interfaces with various DBMSs, Java applications may become the means to populate a DBMS with regular subsets of SSDBs. This way, SSDBs could be queried with the benefits of DBMSs. However, extraction mechanisms could be devised to directly inject SSD into relational DBMSs. Here, other issues come into play, such as how to extract relationships, specified via external keys, from SSDBs.

10.2 Persistence and extraction mechanisms

In *Persistent Programming Languages* [13, 17, 75, 48, 16], computations store values together with their types into a *persistent store*. Values thus survive the applications that created them and may be retrieved by any run-time computation by performing an *intern* operation. Such operation takes a reference to the value, i.e. a name, and a type. If the value type matches the given type, the values is safely imported into the interested computation.

Similarly, an extraction mechanism checks for the existence of a regular SSDBs that corresponds to a value of a given type, and possibly returns such value. Accordingly, extraction mechanisms could be naturally hosted in a persistent language, where programmers are used to accessing values through dynamic controls.

A more ambitious project is that of realising a persistent language around a *semistructured persistent store*. Here, a persistent store would become an SSDB whose data can be retrieved by high-level applications exploiting an extraction mechanism. Similarly, applications would persistently store the data they create according to a corresponding SSD representation. The persistent store could thus be queried over by means of SSDQLs. In particular, by adopting an XML representation of the store, persistent data could be also accessed via HTTP and visualised with a browser.

This idea suggests the further realisation of a persistent language integrating under the same binding mechanism, high-level commands with commands for the manipulations of SSDBs in a navigational fashion.

10.3 Databases data-first design

Schema-first techniques for DBMSs design (see Chapter 2) are not well suited to the creation of databases for complex application domains, such as scientific, geographical, financial, and multimedia domains. This is due to the fact that major structural changes to the schema of a populated database may be turn out to be critical.

Extraction mechanisms suggest a novel design technique, which may be apt to these particular scenarios. Data collections are built in an untyped fashion as SSDBs and become constrained by a schema only at a later stage. Precisely, users populate an SSDB with their relevant data and incrementally project relational or object-oriented schemas assumptions onto the database. Such process terminates when and if a satisfactory binding between schema and the SSDB can be reached.

These techniques are currently under investigation at the Department of Computing and Information Science of Strathclyde University.

Bibliography

[1] S. Abiteboul. Querying semi-structured data. *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece*, pages 1–18, January 8-10 1997.

[2] S. Abiteboul, R. Goldman, T. Lahiri, J. McHugh, and J. Widom. Ozone: Integrating structured and semistructured data. Technical report, Stanford University Database Group, 1998.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistuctured Data. *Journal of Digital Libraries, 1(1)*, pages 68–88, April 1997.

[4] S. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[5] P. Aczel. An introduction to inductive definitions. *In Handbook of Mathematical Logic J. Barwise ed.*, pages 739–782, 1977.

[6] P. Aczel. *Non-well-founded sets.* CSLI Lecture Notes. Stanford, CSLI Publications 14, 1988.

[7] B. Adelberg. NoDoSE: A tool for semi-automatically extracting structured and semi-structured data from text documents. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 283–294, New York, June 1–4 1998. ACM Press.

[8] B. Adelberg and M. Denny. Nodose version 2.0. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMod-99)*, volume 28,2 of *SIGMOD Record*, pages 559–561, New York, June 1–3 1999. ACM Press.

[9] V. Aguilera, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying xml documents in xyleme. In *Proceedings of ACM SIGIR 2000 Workshop On XML and Information Retrieval*, Athens, Greece, July 2000.

[10] A. Albano, D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. A type system for querying XML documents. *SIGIR00-XML ACM Workshop held in conjunction with 23rd International SIGIR Conference on Research and Development in Information Retrieval, Athens, Greece*, 2000.

[11] A. Albano, D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. A typed query language for XML documents. Preliminary Draft, 2000.

[12] A. Albano, D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. A text retrieval query language for XML documents. *Special Topic Issue of JASIS on XML and Information Retrieval*, August 2001.

[13] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *Journal of Very Large Data Bases*, 4(3):403–444, 1995.

[14] G. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems(TOPLAS), 15(4)*, pages 575–631, 1993.

[15] Arnaud Sahuguet. Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask. In *WebDB-2000*, 2000.

[16] M. Atkinson and M. Jordan. A review of the rationale and architectures of pjama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform. Technical report, Sun Microsystems Laboratories, CA, USA, Last revised Sept. 2000.

[17] M. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[18] P. Atzeni and G. Mecca. Cut and Paste. In *Sixteenth ACM SIGMOD International Symposium on Principles of Database System (PODS '97)*, 1997.

[19] J. Barwise and J. Etchemendy. *The Liar. An Essay on Truth and Circularity.* Oxford, Oxford University Press, 1987.

[20] J. Barwise and L. Moss. *Vicious Circles. On the Mathematics of Non-Wellfounded Phenomena.* CSLI Lecture Notes Number 60. Stanford, CSLI Publications, 1996.

[21] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes. Technical report, World Wide Web Consortium, Dec. 1999. W3C Working Draft.

[22] M. Brandt. Recursive subtyping: Axiomatisations and computational interpretations. *Master Thesis*, 1997.

[23] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae XX*, pages 1–24, 1998.

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

[25] P. Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 117–121. ACM Press, 1997.

[26] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. *Lecture Notes in Computer Science*, 1186:336–350, 1997.

[27] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimisation techniques for unstructured data. *In Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[28] P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proceedings of 5th International Workshop on Database Programming Languages*, Gubbio, Italy, September 1995.

[29] P. Buneman, M. Fernandez, and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.

[30] P. Buneman and B. Pierce. Union types for semistructured data. In *Internet Programming Languages.* Springer, Sept. 1998. Proceedings of the International Database Programming Languages Workshop. LNCS 1686.

[31] P. Buneman and B. Pierce. Union types for semistructured data. Technical report, University of Pennsylvania, 1999.

[32] J. Campbell, D. Grossman, and A.-M. Popescu. Quilt2Sql - An XML Storage Schema and Query Engine for the Quilt Query Language. 2000.

[33] L. Cardelli and G. Ghelli. A query language for semistructured data based on the ambient logic. Manuscript, April 2000.

[34] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, jun 2001. W3C Working Draft.

[35] D. Chamberlin, D. Florescu, and J. Robie. Quilt: an XML query language for heterogeneous data sources. In *Proceedings of WebDB*, Dallas, TX, May 2000.

[36] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, 1994.

[37] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 313–324. ACM Press, 1994.

[38] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of theACM SIGMOD International Conference on Management of Data*, volume 25, 2 of *ACM SIGMOD Record*, pages 413–422, New York, June 4–6 1996. ACM Press.

[39] V. Christophides, S. Cluet, and J. Siméon. Semistructured and Structured Integration Reconciled: YAT += Efficient Query Processing. Technical report, INRIA, Verso database group, November 1998.

[40] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Technical report, World Wide Web Consortium, Nov. 1999. W3C Recommendation.

[41] S. Cluet. Modeling and querying semi-structured data. *SCIE '97*, 1997.

[42] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In M. Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*, pages 383–392. ACM Press, 1992.

[43] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 177–188, New York, June 1998. ACM Press.

[44] S. Cluet and G. Moerkotte. Nested queries in object bases. In C. Beeri, A. Ohori, and D. Shasha, editors, *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993*, Workshops in Computing, pages 226–242. Springer, 1993.

[45] S. Cluet and J. Siméon. Data integration based on data conversion and restructuring. Technical report, INRIA, Verso database group, October 1997.

[46] R. Connor, D. Lievens, P. Manghi, S. Neely, and F. Simeoni. Extracting typed values from XML databases. *OOPSLA'01 Workshop on Objects, <XML> and Databases, Tampa Bay, Florida, USA*, October 2001.

[47] R. Connor, D. Lievens, P. Manghi, S. Neely, and F. Simeoni. An Approach to High-Level Language Bindings for XML. *Elsevier Journal on Information and*

Software Technology, Special Issue on Object, XML and Databases, To appear on spring 2002.

[48] R. Connor, K. Sibson, and P. Manghi. On the unification of persistent programming and the world wide web. *Workshop on the Web and Data Bases (WebDB98) held in conjunction with EDBT'98 (Springer Verlag, Vol. 1590, LNCS series)*, 1998.

[49] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. Technical report, World Wide Web Consortium, Aug. 1998. Submission to the World Wide Web Consortium.

[50] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMod-99)*, volume 28,2 of *SIGMOD Record*, pages 431–442, New York, June 1–3 1999. ACM Press.

[51] R. Durbin and J. T. Mieg. *ACeDB – A C. elegans Database: Syntactic definitions for the ACeDB data base manager*, 1992. `http://probe.nalusda.gov:8000/acedocs/syntax.html`.

[52] D. C. Fallside. XML Schema Part 0: Primer. Technical report, World Wide Web Consortium, Oct. 2000. W3C Candidate Recommendation.

[53] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. The XML Query Algebra. Technical report, World Wide Web Consortium, Dec. 2000. W3C Working Draft.

[54] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. STRUDEL: A Web site management system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):549–560, 1997.

[55] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a Web-site management system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(3):4–11, September 1997.

[56] M. Fernandez and J. Robie. XML Query Data Model. Technical report, World Wide Web Consortium, May 2000. W3C Working Draft.

[57] M. Fernandez, J. Siméon, and P. Wadler. XML Query Languages: Experiences and Exemplars, December 1999. Manuscript available from `http://www-db.research.bell-labs.com/user/simeon/xquery.ps`.

[58] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas (full version), February 1997. Manuscript available from `http://www.research.att.com/~{mff,suciu}`.

[59] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the Boat with Strudel: Experiences with a Web-Site Management System. In Haas and Tiwary [65], pages 414–425.

[60] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 14–23. IEEE Computer Society, 1998.

[61] D. Florescu and D. Kossmann. Storing and querying XML data using an rdbms. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[62] R. Goldman, S. Chawathe, A. Crespo, and J. McHugh. A Standard Textual Interchange Format for the Object Exchange Model (OEM). Technical report, Stanford University, 1995.

[63] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proceedings of the second International Workshop WebDB '99, Pennsylvania*, June 1999.

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

[65] L. M. Haas and A. Tiwary, editors. *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. ACM Press, 1998.

[66] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report), May 2000. WebDB workshop.

[67] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proceedings of the 16th International Conference on Data Engineering, 28 February - 3 March, 2000, San Diego, California, USA*, 2000.

[68] W. Kim. On optimizing an SQL-like nested query. *TODS*, 7(3):443–469, 1982.

[69] N. Kushmerick, D. S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 729–737, San Francisco, Aug. 23–29 1997. Morgan Kaufmann Publishers.

[70] A. Malhotra, J. Robie, and M. Rys. XML syntax for XQuery 1.0 (XQueryX). Technical report, World Wide Web Consortium, June 2001. W3C Working Draft.

[71] J. McHugh and J. Widom. Compile-time path expansion in Lore. Technical report, Stanford University Database Group, November 1998.

[72] G. Mecca, A. O. Mendelzon, and P. Merialdo. Efficient Queries over Web Views. Technical Report 2, Araneus Project Working Report, AWR-2-99 (version 1.0 - March, 16, 1999.

[73] Milo and Suciu. Index structures for path expressions. In *ICDT: 7th International Conference on Database Theory*, 1999.

[74] F. D. Monte. Meccanismi per l'evoluzione dello schema nel linguaggio Fibonacci,. Master's thesis, Universitá degli Studi di Pisa, 1995.

[75] R. Morrison, R. Connor, G. Kirby, D. Munro, M. Atkinson, Q. Cutts, A. Brown, and A. Dearle. The Napier88 persistent programming language and environment. In *Fully Integrated Data Environments*, pages 98–154. Springer, 1999.

[76] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(4):39–52, 1997.

[77] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In Haas and Tiwary [65], pages 295–306.

[78] S. Nestorov, J. Ullman, J. Weiner, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchical data. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 79–90, Birmingham, UK, Apr. 1997. IEEE.

[79] OMG. CORBA OMG IDL text file - The Object Management Group, 1999. ftp://ftp.omg.org/pub/docs/formal/99-04-01.txt.

[80] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. *Lecture Notes in Computer Science*, 1013:161–180, 1995.

[81] Y. Papakonstantinou, J. Widom, and H. Molina. Object exchange across heterogeneous information sources. *Proceedings of IEEE Int. Conference on Data Engineering, Birmingham, England*, 1996.

[82] D. Quass, A. Rajaraman, Y. Sagiv, and J. Ullman. Querying semistructured heterogeneous information. *Lecture Notes in Computer Science*, 1013:319–331, 1995.

[83] D. Raggett, A. L. Hors, and I. Jacobs. HTML 4.01 specification. Technical report, World Wide Web Consortium, Dec. 1999. W3C Recommendation.

[84] J. Robie, J. Lapp, D. Schach, M. Hyman, and J. Marsh. XML Query Language (XQL). Note to the W3C XSL Working Group, Sept. 1998.

[85] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[86] J. Siméon and S. Cluet. Using yat to build a web server. In *International Workshop on the Web Database (WebDB '98), Valencia, Spain*, March 1998.

[87] L. D. Stein and J. Thierry-Mieg. AceDB: A genome database management system. *Computing in Science and Engineering*, 1(3):44–68, May/June 1999.

[88] D. Suciu. From Semistructured Data to XML, 1999. VLDB Tutorial.

[89] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelson. XML Schema Part 1: Structures. Technical report, World Wide Web Consortium, Dec. 1999. W3C Working Draft.

[90] R. G. J. Widom. Enabling query formulation and optimization in semistructured databases. pages 436–445. VLDB, 1997.

[91] XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`.