Semistructured Computation

Luca Cardelli

Microsoft Research

1 Introduction

This paper is based on the observation that the areas of *semistructured databases* [1] and *mobile computation* [3] have some surprising similarities at the technical level. Both areas are inspired by the need to make better use of the Internet. Despite this common motivation, the technical similarities that arise seem largely accidental, but they should still permit the transfer of some techniques between the two areas. Moreover, if we can take advantage of the similarities and generalize them, we may obtain a broader model of data and computation on the Internet.

The ultimate source of similarities is the fact that both areas have to deal with extreme dynamicity of data and behavior. In semistructured databases, one cannot rely on uniformity of structure because data may come from heterogeneous and uncoordinated sources. Still, it is necessary to perform searches based on whatever uniformity one can find in the data. In mobile computation, one cannot rely on uniformity of structure because agents, devices, and networks can dynamically connect, move around, become inaccessible, or crash. Still, it is necessary to perform computations based on whatever resources and connections one can find on the network.

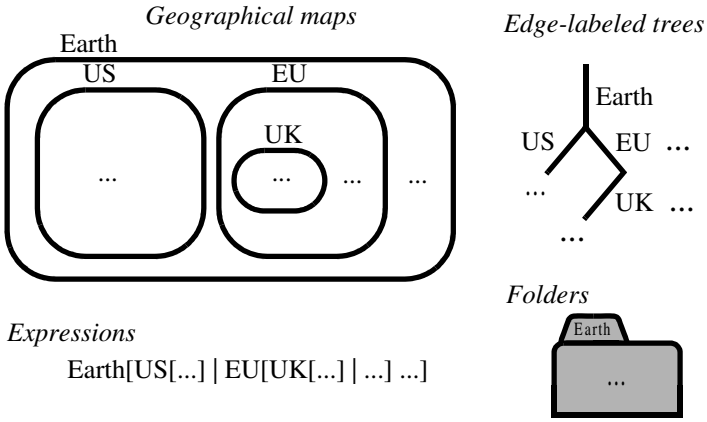
We will develop these similarities throughout the paper. As a sample, consider the following arguments. First, one can regard data structures stored inside network nodes as a natural extension of network structures, since on a large time/space scale both networks and data are semistructured and dynamic. Therefore, one can think of applying the same navigational and code mobility techniques uniformly to networks and data. Second, since networks and their resources are semistructured, one can think of applying semistructured database searches to network structure. This is a well-known major problem in mobile computation, going under the name of resource discovery.

2 Information

2.1 Representing Dynamic Information

In our work on mobility [3, 5] we have been describing mobile structures in a variety of related ways. In all of these, the spatial part of the structure can be represented abstractly as an edge-labeled tree.

For example, the following figure shows at the top left a nested-blob representation of geographical information. At the bottom left we have an equivalent representation in the nested-brackets syntax of the Ambient Calculus [5]. When hierarchical information is used to represent document structures, a more appropriate graphical representation is in terms of nested folders, as shown at the bottom right. Finally, at the top right we have a more schematic representation of hierarchies in terms of edge-labeled trees.



We have studied the Ambient Calculus as a general model of mobile computation. The Ambient Calculus has so far been restricted to edge-labeled trees, but it is not hard to imagine an extension (obtained by adding recursion) that can represent edge-labeled directed graphs. As it happens, edge-labeled directed graphs are also the favorite representation for semistructured data [1]. So, basic data structures used to represent semistructured data and mobile computation, essentially agree. Coincidence?

It should be stressed that edge-labeled trees and graphs are a very rudimentary way of representing information. For example, there is no exact representation of record or variant data structures, which are at the foundations of almost all modern programming languages. Instead, we are thrown back to a crude representation similar to LISP's S-expressions.

The reason for this step backward, as we hinted earlier, is that in semistructured databases one cannot rely on a fixed number of subtrees for a given node (hence no records) and one cannot even rely of a fixed set of possible shapes under a node (hence no variants). Similarly, on a network, one cannot rely on a fixed number of machines being alive at a given node, or resources being available at a given site, nor can one rule out arbitrary network reconfiguration. So, the similarities in data representation arise from similarities of constraints on the data.

In the rest of this section we discuss the representation of mobile and semistructured information. We emphasize the Ambient Calculus view of data representation, mostly because it is less well known. This model arose independently from semistructured data; it can be instructive to see a slightly different solution to what is essentially the same problem of dynamic data representation.

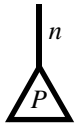
2.2 Information Expressions and Information Trees

We now describe in more detail the syntax of *information expressions*; this is a subset of the Ambient Calculus that concerns data structures. The syntax is interpreted as representing *finite-depth edge-labeled unordered trees*; for short: *information trees*.

The tree that consists just of a root node is written as the expression **0**:

0 represents ▪

A tree with a single edge labeled *n* from the root, leading to a subtree represented by *P*, is written as the expression *n*[*P*]:

n[*P*] represents 

A tree obtained by joining two trees, represented by *P* and *Q*, at the root, is written as the expression *P* | *Q*.

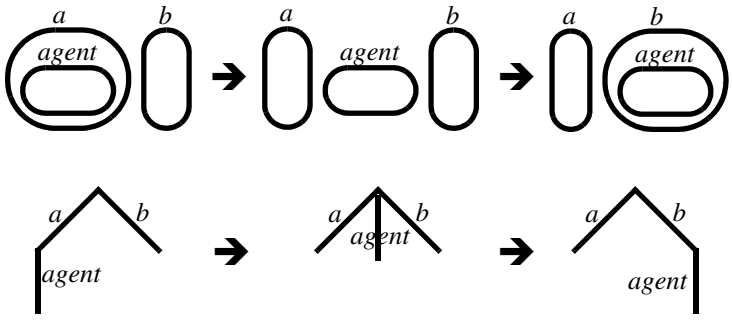
P | *Q* represents

A tree obtained by joining an infinite number of equal trees, represented by *P*, at the root, is written as the expression **!***P*. (This can be used to represent abstractly unbounded resources.)

!*P* represents

The description of trees in this syntax is not unique. For example the expressions *P* | *Q* and *Q* | *P* represent the same (unordered) tree; similarly, the expressions **0** | *P* and *P* represent the same tree. More subtle equivalences govern **!**. We will consider two expression equivalent when they represent the same tree.

The Ambient Calculus uses these tree structures to describe mobile computation, which is seen as the evolution of tree structures over time. The following figure gives, first, a blob representation of an *agent* moving from inside node *a* to inside node *b*, with an intermediate state where the agent is traveling over the network.

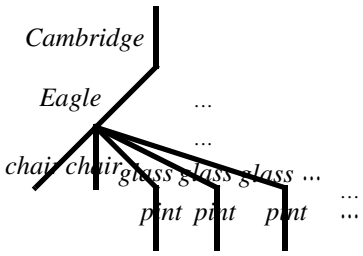


a[*agent*[...]] | *b*[**0**] ➔ *a*[**0**] | *agent*[...] | *b*[**0**] ➔ *a*[**0**] | *b*[*agent*[...]]

Then, the same situation is represented as transformation of information trees, where hierarchy represents containment and the root is the whole network. Finally, the same sit-

uation is represented again as transformation of information expressions. The Ambient Calculus has additional syntax to represent the actions of the agent as it travels from *a* to *b* (indicated here by "..."); we will discuss these actions later.

Note that information trees are not restricted to be finite-branching. For example, the following information tree describes, in part, the city of Cambridge, the Cambridge Eagle pub, and within the pub two empty chairs and an unbounded number of full glasses of beer.



This tree can be represented by the following expression:

$$Cambridge[Eagle[chair[\mathbf{0}] \mid chair[\mathbf{0}] \mid \mathbf{!}glass[pint[\mathbf{0}]]] \mid ...]$$

Here is another example: an expression representing the (invalid!) fact that in Cambridge there is an unlimited number of empty parking spaces:

$$Cambridge[\mathbf{!}ParkingSpace[\mathbf{0}] \mid ...]$$

Equivalence of information trees can be characterized fairly easily, even in presence of infinite branching. Up to the equivalence relation induced by the following set of equations, two information expressions are equivalent if and only if they represent the same information tree [9]. Because of this, we will often confuse expressions with the trees they represent.

$$P \mid Q = Q \mid P \qquad !(P \mid Q) = !P \mid !Q$$
$$(P \mid Q) \mid R = P \mid (Q \mid R) \qquad !\mathbf{0} = \mathbf{0}$$
$$P \mid \mathbf{0} = P \qquad !P = P \mid !P$$
$$!P = !!P$$

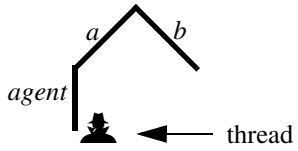
In contrast to our information trees, the standard model of semistructured data consists of *finitely-branching edge-labeled unordered directed graphs*. There there is no notion of unbounded resource there, but there is a notion of node sharing that is not present in the Ambient Calculus. It should be interesting to try and combine the two models; it is not obvious how to do it, particularly in terms of syntactical representation. Moreover, the rules of equivalence of graph structures are more challenging; see Section 6.4 of [1].

2.3 Ambient Operations

The Ambient Calculus provides operations to describe the transformation of data. In the present context, the operations of the Ambient Calculus may look rather peculiar, because they are intended to represent agent mobility rather than data manipulation. We

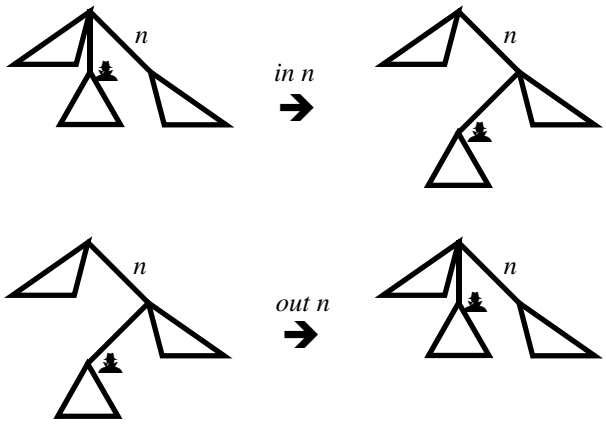
present them here as an example of a set of operations on information trees; other sets of operations are conceivable. In any case, their generalization to directed graphs does not seem entirely obvious.

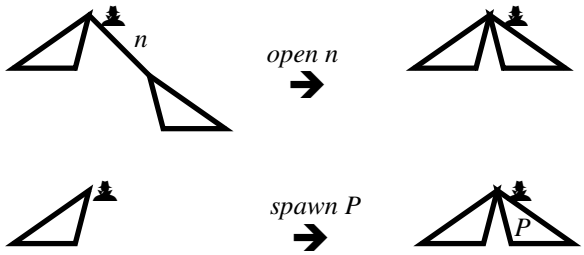
Information expressions and information trees are a special case of *ambient expressions* and *ambient trees*; in the latter we can represent also the dynamic aspects of mobile computation and mutable information. An ambient tree is an information tree where each node in the tree may have an associated collection of concurrent threads that can execute certain operations. The fact that threads are associated to nodes means that the operations are "local": they affect only a small number of nodes near the thread node (typically three nodes). In our example of an *agent* moving from *a* to *b*, there would usually be a thread in the agent node (the node below the *agent* edge) that is the cause of the movement.



Therefore, the full Ambient Calculus has both a spatial and a temporal component. The spatial component consists of information trees, that is, semistructured data. The temporal component includes operations that locally modify the spatial component. Rather than giving the syntax of these operations, we describe them schematically below. The location of the thread performing the operations is indicated by the thread icon.

The operation *in n*, causes an ambient to enter another ambient named *n* (i.e., it causes a subtree to slide down along an *n* edge). The converse operation, *out n*, causes an ambient to exit another ambient named *n* (i.e., it causes a subtree to slide up along an *n* edge). The operation *open n* opens up an ambient named *n* and merges its contents (i.e., it collapses an edge labeled *n*); these contents may include threads and subtrees. Finally, the spawning operation creates a new configuration within the current ambient (i.e., it creates a new tree and merges its root with the current node).





It should be clear that, by strategically placing agents on a tree, we can rearrange, collapse, and expand sections of the tree at will.

2.4 Summary

We have seen that there are some fundamental similarities of data representation in the areas of semistructured data and mobile computation. Moreover, in the case of mobile computation, we have ways of describing the manipulation of data. (In semistructured database, data manipulation is part of the query language, which we discuss later.)

3 Data Structures

We discuss briefly how traditional data structures (records and variants) fit into the semistructured data and ambients data models.

3.1 Records

A *record r* is a structure of the form $\{l_1=v_1, ..., l_n=v_n\}$, where l_i are distinct labels and v_i are the associated values; the pairs l_i,v_i are called record *fields*. Field values can be extracted by a *record selection* operation, $r.l_i$, by indexing on the field labels.

Semistructured data can naturally represent record-like structures: a root node represents the whole record, and for each field $l_i=v_i$, the root has an edge labeled l_i leading to a subtree v_i. Record fields are unordered, just like the edges of our trees. However, semistructured data does not correspond exactly to records: labels in a record are unique, while semistructured data can have any number of edges with the same label under a node. Moreover, records usually have uniform structure throughout a given collection of data, while there is no such uniformity on semistructured data.

It is interesting to compare this with the representation of records in the Ambient Calculus. There, we represent records $\{l_1=v_1, ..., l_n=v_n\}$ as:

$$r[l_1[... v_1 ...] \mid ... \mid l_n[... v_n ...]]$$

where r is the name (address) of the record, which is used to name an ambient $r[...]$ representing the whole record. This ambient contains subambients $l_1[...] ... l_n[...]$ representing labeled fields (unordered because $\mid$ is unordered). The field ambients contain the field values $v_1, ..., v_n$ and some machinery (omitted here) to allow them to be read and rewritten.

However, ambients represent mobile computation. This means that, potentially,

field subambients l_i[...] can take off and leave, and new fields can arrive. Moreover, a new field can arrive that has the some label as an existing field. In both cases, the stable structure of ordinary records is destroyed.

3.2 Variants

A *variant v* is a structure of the form [*l*=*v*], where *l* is a label and *v* is the associated value, and where *l* is restricted to be a member of a finite set of labels l_1 ... l_n. A *case analysis* operation can be used to determine which of these labels is present in the variant, and to extract the associated value.

A variant can be easily represented in semistructured data, as an edge labeled *l* leading to a subtree *v*, with the understanding that *l* is a unique edge of its parent node, and that *l* is a member of a finite collection l_1 ... l_n. But the latter restrictions are not enforced in semistructured data. A node meant to represent a variant could have zero outgoing edges, or two or more edges with different labels, or even two or more edges with the same label, or an edge whose label does not belong to the intended set. In all these situations, the standard case analysis operation becomes meaningless.

A similar situation happens, again, in the case of mobile computation. Even if the constraints of variant structures are respected at a given time, a variant may decide to leave its parent node at some point, or other variants may come to join the parent node.

3.3 Summary

We have seen that fundamental data structures used in programming languages becomes essentially meaningless both in semistructured data and in mobile computation. We have discussed the untyped situation here, but this means in particular that fundamental notions of types in programming languages become inapplicable. We discuss type systems next.

4 Type Systems

4.1 Type Systems for Dynamic Data

Because of the problems discussed in the previous section, it is quite challenging to devise type systems for semistructured data or mobile computation. Type systems track invariants in the data, but most familiar invariants are now violated. Therefore, we need to find weaker invariants and weaker type systems that can track them.

In the area of semistructured data, ordinary database schemas are too rigid, for the same reasons that ordinary type systems are too rigid. New approaches are needed; for example, *union types* have been proposed [2]. Here we give the outline of a different solution devised for mobile computation. Our task is to find a type system for the information trees of Section 2, subject to the constraint that information trees can change dynamically, and that the operations that change them must be typeable too.

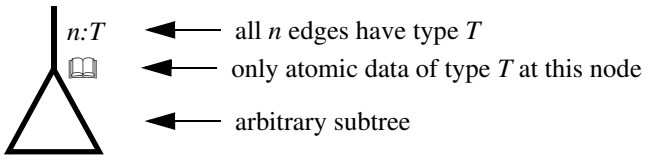
4.2 A Type System for Information Trees

The type system we present here may appear to be very weak, in the sense of imposing very few constraints on information trees. However, this appearance is deceptive: within this type system, when applied to the full Ambient Calculus, we can represent stan-

dard type systems for the λ-calculus and the π-calculus [6]. Moreover, more refined type systems for mobility studied in [4] enforce more constraints by forcing certain substructures to remain "immobile". Here we give only an intuitive sketch of the type system; details can be found in [6].

The task of finding a type systems for information trees is essentially the same as the task of finding a type system for ordinary hierarchical file systems. Imagine a file system with the following constraints. First, each folder has a name. Second, each name has an associated data type (globally). Third, each folder of a given name can contain only data of the type associated with its name. Fourth, if there is a thread operating at a node, it can only read and write data of the correct type at that node. Fifth, any folder can contain any other kind of folder (no restrictions).

In terms of information trees, these rules can be depicted as follows. Here we add the possibility that the nodes of information tree may contain atomic data (although in principle this data can also be represented by trees):



Next, we need to examine the operations described in section 2.3 (or any similar set of operations) to make sure they can be typed. The type system can easily keep track of the global associations of types to names. Moreover, we need to type each thread according to the type of data it can read, write, or merge (by performing *open*) at the current node.

The *in* and *out* operations change the structure of the tree (which is not restricted by the type system) but do not change the relationship between an edge and the contents of the node below it; so no type invariant is violated. The *open* operation, though, merges the contents of two nodes. Here the type system must guarantee that the labels above those two nodes have the same type; this can be done relatively easily, by keeping track of the type of each thread, as sketched above. Finally, the *spawn* operation creates a new subtree, so it must simply enforce the relationship between the edges it creates and the attached data.

This is a sensible type system in the sense that it guarantees well-typed interactions: any process that reads or writes data at a particular node (i.e., inside a particular folder) can rely on the kind of data it will find there. On the other hand, this type system does not constrain the structure of the tree, therefore allowing both heterogeneity (for semi-structured data) and mutability (for mobile computation).

Note also that this type system does not give us anything similar to ordinary record types. Folder types are both weaker than record types, because they do not enforce uniformity of substructures, and stronger, because they enforce global constraints on the typing of edges.

4.3 Summary

Because of the extreme dynamicity present both in semistructured data and in mobile computation, new type systems are needed. We have presented a particular type system as an example of possible technology transfers: we have several ready-made type systems for mobile computation that could be applicable to semistructured data.

5 Queries

Semistructured databases have developed flexible ways of querying data, even though the data is not rigidly structured according to schemas [1]. In relational database theory, query languages are nicely related to query algebras and to query logics. However, query algebras and query logics for semistructured database are not yet well understood.

For reasons unrelated to queries, we have developed a specification logic for the Ambient Calculus [7]. Could this logic, by an accident of fate, lead to a query language for semistructured data?

5.1 Ambient Logic

In classical logic, assertions are simply either *true* or *false*. In *modal logic*, instead, assertions are true or false relative to a *state* (or *world*). For example, in epistemic logic assertions are relative to the knowledge state of an entity. In temporal logic, assertions are relative to the execution state of a program. In our Ambient Logic, which is a modal logic, assertions are relative to the current place and the current time.

As an example, here is a formula in our logic that makes an assertion about the shape of the current location at the current time. It is asserting that right now, right here, there is a location called *Cambridge* that contains at least a location called *Eagle* that contains at least one empty *chair* (the formula **0** matches an empty location; the formula **T** matches anything):

$$Cambridge[Eagle[chair[\mathbf{0}] \mid \mathbf{T}\ ] \mid \mathbf{T}]$$

This assertion happens to be true of the tree shown in Section 2.2. However, the truth of the assertion will in general depend on the current time (is it happy hour, when all chairs are taken?) and the current location (Cambridge England or Cambridge Mass.?).

Formulas of the Ambient Logic

η	a name n or a variable x
$\mathcal{A}, \mathcal{B} : \mathbf{\Phi} ::=$	
$\mathbf{T}$	true
$\neg\mathcal{A}$	negation
$\mathcal{A} \vee \mathcal{B}$	disjunction
$\mathbf{0}$	void
$\eta[\mathcal{A}]$	location
$\mathcal{A} \mid \mathcal{B}$	composition
$\diamond\mathcal{A}$	somewhere modality
$\Diamond\mathcal{A}$	sometime modality

$\mathcal{A}@\eta$	location adjunct
$\mathcal{A}\triangleright\mathcal{B}$	composition adjunct
$\forall x.\mathcal{A}$	universal quantification over names

More generally, our logic includes both assertions about trees, such as the one above, and standard logical connectives for composing assertions. The following table summarizes the formulas of the Ambient Logic. The first three lines give classical propositional logic. The next three lines describe trees. Then we have two modal connective for assertions that are true somewhere or sometime. After the two adjunctions (discussed later) we have quantification over names, giving us a form of predicate logic; the quantified names can appear in the location and location adjunct constructs.

5.2 Satisfaction

The exact meaning of logical formulas is given by a *satisfaction relation* connecting a tree with an formula. The term *satisfaction* comes from logic; for reasons that will become apparent shortly, we will also call this concept *matching*. The basic question we consider: is this formula satisfied by this tree? Or: does this tree match this formula?

The satisfaction relation between a tree P (actually, an expression P representing a tree) and a formula $\mathcal{A}$ is written:

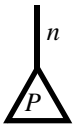
$$P \models \mathcal{A}$$

For the basic assertions on trees, the satisfaction/matching relation can be described as follows; for graphical effect we relate tree shapes to formulas:

- **0**: *here now* there is absolutely nothing:

   matches **0**

- $n[\mathcal{A}]$: *here now* there is one edge called n, whose descendant satisfies the formula $\mathcal{A}$:

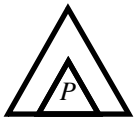
  

  matches $n[\mathcal{A}]$ if P matches $\mathcal{A}$.

- $\mathcal{A} \mid \mathcal{B}$: *here now* there are exactly two things next to each other, one satisfying $\mathcal{A}$ and one satisfying $\mathcal{B}$:

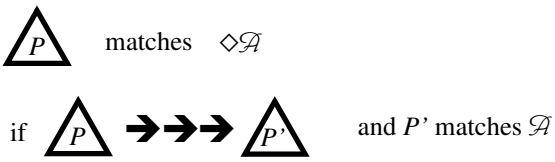
  matches $\mathcal{A} \mid \mathcal{B}$ if P matches $\mathcal{A}$ and Q matches $\mathcal{B}$ (or if P matches $\mathcal{B}$ and Q matches $\mathcal{A}$)

- $\diamond\mathcal{A}$: *somewhere now*, there is a place satisfying $\mathcal{A}$:

  matches $\diamond\mathcal{A}$ if P matches $\mathcal{A}$ (i.e., there must be a subtree P that matches $\mathcal{A}$)

- $\Diamond\mathcal{A}$: *here sometime*, there is a thing satisfying $\mathcal{A}$, after some reductions:



The propositional connectives and the universal quantifier have fairly standard interpretations. A formula $\neg\mathcal{A}$ is satisfied by anything that does not satisfy $\mathcal{A}$. A formula $\mathcal{A} \vee \mathcal{B}$ is satisfied by anything that satisfies either $\mathcal{A}$ or $\mathcal{B}$. Anything satisfies the formula **T**, while nothing satisfies its negation, **F**, defined as $\neg$**T**. A formula $\forall x.\mathcal{A}$ is satisfied by a tree P if for all names n, the tree P satisfies $\mathcal{A}$ where x is replaced by n.

Many useful derived connectives can be defined from the primitive ones. Here is a brief list:

- *Normal Implication:* $\mathcal{A} \Rightarrow \mathcal{B} \triangleq \neg\mathcal{A} \vee \mathcal{B}$. This is the standard definition, but note that in our modal logic this means that P matches $\mathcal{A} \Rightarrow \mathcal{B}$ if whenever P matches $\mathcal{A}$ then *the same P* matches $\mathcal{B}$ at the same time and in the same place. As examples, consider *Borders*[**T**] $\Rightarrow$ *Borders*[*Starbucks*[**T**] | **T**], stating that a *Borders* bookstore contains a *Starbucks* shop, and (*NonSmoker*[**T**] | **T**) $\Rightarrow$ (*NonSmoker*[**T**] | *Smoker*[**T**] | **T**), stating that next to a non-smoker there is a smoker.
- *Everywhere*: $\boxplus\mathcal{A} \triangleq \neg\Diamond\neg\mathcal{A}$. What is true everywhere? Not much, unless we qualify it. We can write $\boxplus(\mathcal{A} \Rightarrow \mathcal{B})$ to mean that everywhere $\mathcal{A}$ is true, $\mathcal{B}$ is true as well. For example, *US*[$\boxplus$(*Borders*[**T**] $\Rightarrow$ *Borders*[*Starbucks*[**T**] | **T**])].
- *Always*: $\Box\mathcal{A} \triangleq \neg\Diamond\neg\mathcal{A}$. This can be used to express temporal invariants, such as: $\Box$*Pisa*[*LeaningTower*[**T**] | **T**].
- *Parallel Implication:* $\mathcal{A} \mid\!\Rightarrow \mathcal{B} \triangleq \neg(\mathcal{A} \mid \neg\mathcal{B})$. This means that it is not possible to split the root of the current tree in such a way that one part satisfies $\mathcal{A}$ and the other does not satisfy $\mathcal{B}$. In other words, every way we split the root of the current tree, if one part satisfies $\mathcal{A}$, then the other part must satisfy $\mathcal{B}$. For example, *Bath*[$\boxplus$(*NonSmoker*[**T**] $\mid\!\Rightarrow$ *Smoker*[**T**] | **T**)] means that at the *Bath* pub, anywhere there is a non-smoker there is, nearby, a smoker. Note that parallel implication makes the definition of this property a bit more compact than in the earlier example about smokers.
- Nested Implication: $n[\Rightarrow\mathcal{A}] \triangleq \neg n[\neg\mathcal{A}]$. This means that it is not possible that the contents of an n location do not satisfy $\mathcal{A}$. In other words, if there is an n location, its contents satisfy $\mathcal{A}$. For example: *US*[$\boxplus$*Borders*[$\Rightarrow$*Starbucks*[**T**] | **T**]]; again, this is a bit more compact than the previous formulation of this example.

5.3 Adjunctions

The adjunction connectives, $\mathcal{A} \triangleright \mathcal{B}$ and $\mathcal{A}@n$, are of special interest; they are the logical inverses, in a certain sense, of $\mathcal{A} \mid \mathcal{B}$ and $n[\mathcal{A}]$ respectively. In ordinary logic, we have a fundamental adjunction between conjunction and implication given by the property: $\mathcal{A} \wedge \mathcal{B}$ entails C iff $\mathcal{A}$ entails $\mathcal{B} \Rightarrow C$. Similarly, in our logic we have that $\mathcal{A} \mid \mathcal{B}$ entails C

iff $\mathcal{A}$ entails $\mathcal{B} \triangleright C$, and that $n[\mathcal{A}]$ entails C iff $\mathcal{A}$ entails $C@n$. We now explore the explicit meaning of these adjunctions.

The formula $\mathcal{A} \triangleright \mathcal{B}$ means that the tree present here and now satisfies the formula $\mathcal{B}$ when it is merged at the root with any tree that satisfies the formula $\mathcal{A}$. We can think of this formula as a requirement/guarantee specification: given any context that satisfies $\mathcal{A}$, the combination of that context with the current tree will satisfy $\mathcal{B}$.

matches $\mathcal{A} \triangleright \mathcal{B}$

if for all

that match $\mathcal{A}$ we have that

matches $\mathcal{B}$

For example, consider a representation of a fish consisting of a certain structure (beginning with *fish*[...]), and a certain behavior. A prudent fish would satisfy the following specification, stating that even in presence of *bait*, the *bait* and the *fish* remain separate:

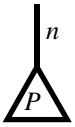
$$fish[...] \quad \models \quad bait[\mathbf{T}] \triangleright \Box(fish[\mathbf{T}] \mid bait[\mathbf{T}])$$

On the other hand, a good *bait* would satisfy the following specification, stating that in presence of a *fish*, it is possible that the *fish* will eventually ingest the bait:

$$bait[...] \quad \models \quad fish[\mathbf{T}] \triangleright \Diamond fish[bait[\mathbf{T}] \mid \mathbf{T}]$$

These two specifications are, of course, incompatible. In fact, it is possible to show within our logic that, independently of any implementation of *fish* and *bait*, the composition of the *fish* spec with the *bait* spec leads to a logical contradiction.

The formula $C@n$ means that the tree present here and now satisfies the formula C when it is placed under an edge named *n*. This is another kind of requirement/guarantee specification, regarding nested contexts instead of parallel contexts: even when "thrown" inside an *n* context, the current tree will manage to satisfy the property C.

matches $C@n$ if 

matches C

For example, an aquarium fish should satisfy the following property, stating that the fish will survive when placed in a (persistent) tank:

$$(\Box tank[fish[\mathbf{T}] \mid \mathbf{T}]) \ @ \ tank$$

5.4 From Satisfaction to Queries

A satisfaction relation, such as the one defined in the previous section, is not always decidable. However, in our case, if we rule out the $!P$ operator on trees, which describes infinite configurations, and also the $\mathcal{A} \triangleright \mathcal{B}$ formulas, which involve a quantification over an infinite set of trees, then the problem of whether $P \models \mathcal{A}$ becomes decidable [7]. A decision procedure for such a problem is also called a *modelchecking* algorithm. Such

an algorithm implements essentially a matching procedure between a tree and a formula, where the result of the match is just success of failure.

For example, the following match succeeds. The formula can be read as stating that there is an empty chair at the *Eagle* pub; the matching process verifies that this fact holds in the current situation:

$$\begin{aligned} &\mathit{Eagle}[\mathit{chair}[\mathit{John}[\mathbf{0}]] \mid \mathit{chair}[\mathit{Mary}[\mathbf{0}]] \mid \mathit{chair}[\mathbf{0}]] \\ &\vDash \mathit{Eagle}[\mathit{chair}[\mathbf{0}] \mid \mathbf{T}] \end{aligned}$$

More generally, we can conceive of collecting information during the matching process about which parts of the tree match which parts of the formula. Further, we can enrich formulas with markers that are meant to be bound to parts of the tree during matching; the result of the matching algorithm is then either failure or an association of formula markers to the trees that matched them.

We thus extend formulas with *matching variables*, $\mathcal{X}$, which are often placed where previously we would have placed a **T**. For example by matching:

$$\begin{aligned} &\mathit{Eagle}[\mathit{chair}[\mathit{John}[\mathbf{0}]] \mid \mathit{chair}[\mathit{Mary}[\mathbf{0}]] \mid \mathit{chair}[\mathbf{0}]] \\ &\vDash \mathit{Eagle}[\mathit{chair}[\mathcal{X}] \mid \mathbf{T}] \end{aligned}$$

we obtain, bound to $\mathcal{X}$, either somebody sitting at the *Eagle*, or the indication that there is an empty chair. Moreover, by matching:

$$\begin{aligned} &\mathit{Eagle}[\mathit{chair}[\mathit{John}[\mathbf{0}]] \mid \mathit{chair}[\mathit{Mary}[\mathbf{0}]] \mid \mathit{chair}[\mathbf{0}]] \\ &\vDash \mathit{Eagle}[\mathit{chair}[(\neg\mathbf{0})\wedge\mathcal{X}] \mid \mathbf{T}] \end{aligned}$$

we obtain, bound to $\mathcal{X}$, somebody (not **0**) sitting at the *Eagle*. Here the answer could be either *John*[**0**] or *Mary*[**0**], since both lead to a successful global match. Moreover, by using the same variable more than once we can express constraints: the formula $\mathit{Eagle}[\mathit{chair}[(\neg\mathbf{0})\wedge\mathcal{X}] \mid \mathit{chair}[\mathcal{X}] \mid \mathbf{T}]$ is successfully matched if there are two people with the same name sitting at the *Eagle*.

These generalized formulas that include matching variables can thus be seen as *queries*. The result of a successful matching can be seen as a possible answer to a query, and the collection of all possible successful matches as the collection of all answers.

For serious semistructured database applications, we need also sophisticated ways of matching names (e.g. with wildcards and lexicographic orders) and of matching paths of names. For the latter, though, we already have considerable flexibility within the existing logic; consider the following examples:

- *Exact path*. The formula $n[m[p[\mathcal{X}]] \mid \mathbf{T}]$ means: match a path consisting of the names n, m, p, and bind $\mathcal{X}$ to what the path leads to. Note that, in this example, other paths may lead out of n, but there must be a unique path out of m and p.
- *Dislocated path*. The formula $n[\diamond(m[\mathcal{X}] \mid \mathbf{T})]$ means: match a path consisting of a name n, followed by an arbitrary path, followed by a name m; bind $\mathcal{X}$ to what the path leads to.
- *Disjunctive path*. The formula $n[p[\mathcal{X}]] \vee m[p[\mathcal{X}]]$ means: bind $\mathcal{X}$ to the result of following either a path $n.p$, or a path $m.p$.

- *Negative path*. The formula $\diamond m[\neg(p[\mathbf{T}] \mid \mathbf{T}) \mid q[\mathcal{X}]]$ means: bind $\mathcal{X}$ to anything found somewhere under m, inside a q but not next to a p.
- *Wildcard and restricted wildcard*. $m[\exists y.y{\neq}n \wedge y[\mathcal{X}]]$ means: match a path consisting of m and any name different from n, and bind $\mathcal{X}$ to what the path leads to. (Inequality of names can be expressed within the logic [7]).

5.5 Adjunctive Queries

Using adjunctions, we can express queries that not only produce matches, but also reconstruct a results.

Consider the query:

$$m[\mathcal{X}@n]$$

This is matched by a tree $m[P]$ if P matches $\mathcal{X}@n$. By definition of P matching $\mathcal{X}@n$, we must verify that $n[P]$ matches $\mathcal{X}$. The latter simply causes the binding of $\mathcal{X}$ to $n[P]$, and we have this association as the result of the query. Note that $n[P]$ is not a subtree of the original tree: it was constructed by the query process. A similar query, $\diamond m[\mathcal{X}@q@n]$, means: if somewhere there is an edge m, wrap its contents P into $q[n[P]]$, and return that as the binding for $\mathcal{X}$.

Consider now the query

$$n[\mathbf{0}]{\triangleright}\mathcal{X}$$

We have that P matches $n[\mathbf{0}]{\triangleright}\mathcal{X}$ if for all Q that match $n[\mathbf{0}]$, $P \mid Q$ matches $\mathcal{X}$. This immediately gives a result binding of $P \mid Q$ for $\mathcal{X}$. But what is Q? Fortunately there is only one Q that matches the formula $n[\mathbf{0}]$, and that is the tree $n[\mathbf{0}]$. So, this query has the following meaning: compose the current tree with $n[\mathbf{0}]$, and give that as the binding of $\mathcal{X}$. Note, again, that this composition is not present in the original tree: it is constructed by the query. In this particular case, the infinite quantification over all Q does not hurt. However, as we mentioned above, we do not have a general matching algorithm for $\triangleright$, so we can at best handle some special cases.

It is not clear yet how much expressive power is induced by adjunctive queries, but the idea of using adjunctions to express query-and-recombination seems interesting, and it comes naturally out of an existing logic. It should be noted that basic questions of expressive power for semistructured database query languages are still open.

In other work [8], we are using a more traditional SQL-style *select* construct for constructing answers to queries. The resulting query language seems to be very similar to XML-QL [1], perhaps indicating a natural convergence of query mechanisms. However, it is also clear that new and potentially useful concepts, such as adjunctive queries, are emerging from the logical point of view.

5.6 Summary

We have seen that what was originally intended as a specification logic for mobile systems can be interpreted (with some extension) as a powerful query language for semistructured data. Conversely, although we have not discussed this, well-known efficient techniques for computing queries in databases can be used for modelchecking certain classes of mobile specifications.

6 Update

Sometimes we wish to change the data. These changes can be expressed by computational processes outside of the domain of databases and query languages. For example, we can use the Ambient Calculus operations described in Section 2.3 to transform trees. In general, if we have a fully worked-out notion of semistructured computation, instead of just semistructured data, then we already have a notion of semistructured update.

In database domains, however, we may want to be able express data transformations more declaratively. For example, transformations systems based on tree grammar transducers have been proposed for XML. It turns out that in our Ambient Logic we also have ways of specifying update operations declaratively, as we now discuss.

6.1 From Satisfiability to Update

In the examples of queries given so far we have considered only a static notion of matching. Remember, though, that we also have a temporal operator in the logic, $\Diamond\mathcal{A}$, that requires matching $\mathcal{A}$ after some evolution of the underlying tree. If we want to talk about update, we need to say that *right now*, we have a certain configuration, and *later*, we achieve another configuration.

To this end, we consider a slightly different view of the satisfaction problem. So far we have considered questions of the form $P \models \mathcal{A}$ when both P and $\mathcal{A}$ are given. Consider now the case where only $\mathcal{A}$ is given, and where we are looking for a tree that satisfies it; we can write this problem as $X \models \mathcal{A}$. In some cases this is easy: any formula constructed only by composing $\mathbf{0}$, $n[\mathcal{A}]$, and $\mathcal{A} \mid \mathcal{B}$ operations is satisfied by a unique tree. If other logical operators are used, the problem becomes harder (possibly undecidable).

Consider, then, the problem $X \models \mathcal{A} \triangleright \Diamond \mathcal{B}$. By definition, we have that X matches $\mathcal{A} \triangleright \Diamond \mathcal{B}$ if when composed with any tree P that matches $\mathcal{A}$, the composition $P \mid X$ can evolve into a tree that satisfies $\mathcal{B}$. Therefore, whatever X is, it must be something that transforms a tree satisfying $\mathcal{A}$ into a tree satisfying $\mathcal{B}$. In other words, X is a *mutator* of arbitrary $\mathcal{A}$ trees into $\mathcal{B}$ trees, and $X \models \mathcal{A} \triangleright \Diamond \mathcal{B}$ is a specification of such a mutator.

So, we can see $X \models \mathcal{A} \triangleright \Diamond \mathcal{B}$ as an inference problem where we are trying to synthesize an appropriate mutator. We believe that this is very much in the database style, where transformations are often specified declaratively, and synthesized by sophisticated optimizers. Of course, this problem can be hard. Alternatively, if we have a proposed mutator P to transform $\mathcal{A}$ trees into $\mathcal{B}$ trees, we can try to verify the property $P \models \mathcal{A} \triangleright \Diamond \mathcal{B}$, to check the correctness of the mutator.

6.2 Summary

We have seen that query languages for semistructured data and specification logics for mobility can be related. In one direction, this can gives us new query languages for semistructured data, or at least a new way of looking at existing query languages. In the other direction, this can gives us modelchecking techniques for mobility specifications.

Conclusions

In conclusion, we have argued that semistructured data and mobile computation are naturally related, because of a hidden similarity in the problems they are trying to solve.

From our point of view, we have discovered that the Ambient Calculus can be seen as a computational model over semistructured data. As a consequence, type systems already developed for the Ambient Calculus can be seen as weak schemas for semistructured data. Moreover, the Ambient Logic, with some modifications, can be seen as a query language for semistructured data.

We have also discovered that it should be interesting to integrate ideas and techniques arising from semistructured databases into the Ambient Calculus, and in mobile computation in general. For example, the generalization of the Ambient Calculus to graph structures, the use of database techniques for modelchecking, and the use of semistructured query languages for network resource discovery.

We hope that, conversely, people in the semistructured database community will find this connection interesting, and will be able to use it for their own purposes. Much, of course, remains to be done.

Acknowledgments

This paper arose from discussions with Giorgio Ghelli about semistructured databases.

References

[1] Abiteboul, S., Buneman, P., Suciu, D.: **Data on the Web**. Morgan Kaufmann Publishers, San Francisco, 2000.

[2] Buneman, P., Pierce, B.: **Union Types for Semistructured Data.** Proceedings of the International Database Programming Languages Workshop, 1999. Also available as University of Pennsylvania Dept. of CIS technical report MS-CIS-99-09.

[3] Cardelli, L.: **Abstractions for Mobile Computation.** Jan Vitek and Christian Jensen, Editors. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. LNCS. 1603, 51-94, Springer, 1999.

[4] Cardelli, L., Ghelli, G., Gordon, A.D.: **Mobility Types for Mobile Ambients.** ICALP'99. LNCS 1644, 230-239, Springer, 1999.

[5] Cardelli, L., Gordon, A.D.: **Mobile Ambients**. FoSSaCS'98, LNCS 1378, 140-155, Springer, 1998.

[6] Cardelli, L., Gordon, A.D.: **Types for Mobile Ambients.** POPL'99, 79-92, 1999.

[7] Cardelli, L., Gordon, A.D.: **Anytime, Anywhere. Modal Logics for Mobile Ambients.** Proceedings POPL'00. (To appear.)

[8] Ghelli, G., Cardelli, L., Gordon, A.D. To appear.

[9] Engelfriet, J.: **A Multiset Semantics for the π-calculus with Replication**. TCS **153**, 65-94, 1996.