Union Types for Semistructured Data

Peter Buneman **Benjamin Pierce**

University of Pennsylvania
Dept. of Computer & Information Science
200 South 33rd Street
Philadelphia, PA 19104-6389, USA
{peter,bcpierce}@cis.upenn.edu

Technical report MS-CIS-99-09

Corrected Version
July 26, 1999

Abstract

Semistructured databases are treated as dynamically typed: they come equipped with no independent schema or type system to constrain the data. Query languages that are designed for semistructured data, even when used with structured data, typically ignore any type information that may be present. The consequences of this are what one would expect from using a dynamic type system with complex data: fewer guarantees on the correctness of applications. For example, a query that would cause a type error in a statically typed query language will return the empty set when applied to a semistructured representation of the same data.

Much semistructured data originates in structured data. A semistructured representation is useful when one wants to add data that does not conform to the original type or when one wants to combine sources of different types. However, the deviations from the prescribed types are often minor, and we believe that a better strategy than throwing away all type information is to preserve as much of it as possible. We describe a system of untagged *union types* that can accommodate variations in structure while still allowing a degree of static type checking.

A novelty of this system is that it involves non-trivial equivalences among types, arising from a law of distributivity for records and unions: a value may be introduced with one type (e.g., a record containing a union) and used at another type (a union of records). We describe programming and query language constructs for dealing with such types, prove the soundness of the type system, and develop algorithms for subtyping and typechecking.

1 Introduction

Although semistructured data has, by definition, no schema, there are many cases in which the data obviously possesses some structure, even if it has mild deviations from that structure. Moreover it typically has this structure because it is derived from sources that have structure. In the process of annotating data or combining data from different sources one needs to accommodate the irregularities that are introduced by these processes. Because there is no way of describing "mildly irregular" structure, current approaches start by ignoring the structure completely, treating the data as some dynamically typed object such as a labelled graph and then, perhaps, attempting to recover some structure by a variety of pattern matching and data mining techniques [NAM97, Ali99]. The purpose of this structure recovery is typically to provide optimization techniques for query evaluation or efficient storage storage structures, and it is partial. It is not intended as a technique for preserving the integrity of data or for any kind of static type-checking of applications.

When data originates from some structured source, it is desirable to preserve that structure if at all possible. The typical cases in which one cannot require rigid conformance to a schema arise when one wants to annotate or modify the database with unanticipated structure or when one merges two databases with slight differences in structure. Rather than forgetting the original type and resorting to a completely dynamically type, we believe a more disciplined approach to maintaining type information is appropriate. We propose here a type system that can "degrade" gracefully if sources are added with variations in structure, while preserving the common structure of the sources where it exists.

The advantages of this approach include:

- The ability to check the correctness of programs and queries on semistructured data. Current semistructured query languages [BDHS96, AQM+96, DFF+] have no way of providing type errors – they typically return the empty answer on data whose type does not conform to the type assumed by the query.
- The ability to create data at one type and query it at another (equivalent) type. This is a natural consequence of using a flexible type system for semistructured data.
- New query language constructs that permit the efficient implementation of "case" expressions and increase the expressive power of a OQL-style query languages.

As an example, biological databases often have a structure that can be expressed naturally using a combination of tuples, records, and collection types. They are typically cast in special-purpose data formats, and there are groups of related databases, each expressed in some format that is a mild variation on some original format. These formats have an intended type, which could be expressed in a number of notations. For example a source ($source_1$) could have type

$$\begin{array}{l}\mathsf{set}[\; id\colon Int,\\ \quad description\colon Str,\\ \quad bibl\colon \mathsf{set}[\; title\colon Str,\; authors\colon \mathsf{list}[name\colon Str,\; address\colon Str],\; year\colon Int\ldots],\\ \quad \ldots]\end{array}$$

A second source ($source_2$) might yield a closely related structure:

$$\begin{array}{l}\mathsf{set}[\; id\colon Int,\\ \quad description\colon Str,\\ \quad bibl\colon \mathsf{set}[\; title\colon Str,\; authors\colon \mathsf{list}[fn\colon Str,\; ln\colon Str,\; address\colon Str],\; year\colon Int\;\ldots],\\ \quad \ldots]\end{array}$$

This differs only in the way in which author names are represented. (This example is fictional, but not far removed from what happens in practice.)

The usual solution to this problem in conventional programming languages is to represent the union of the sources using some form of tagged union type:

$$\mathsf{set}(\langle\, tag_1 : [\, id\colon Int, \ldots\,],\; tag_2 : [\, id\colon Int, \ldots\,]\,\rangle).$$

The difficulty with this solution is that a program such as

$$\mathsf{for\ each}\ x\ \mathsf{in}\ source_1\ \mathsf{do\ print}(x.description) \tag{1}$$

that worked on $source_1$ must now be modified to

$$
\begin{array}{l}
\mathsf{foreach}\ x\ \mathsf{in}\ source_1\ \mathsf{union}\ source_2\ \mathsf{do} \\
\quad \mathsf{case}\ x\ \mathsf{of} \\
\qquad\quad \langle\, tag_1 = y_1 \,\rangle \Rightarrow \mathsf{print}(y_1.description) \\
\qquad | \quad \langle\, tag_2 = y_2 \,\rangle \Rightarrow \mathsf{print}(y_2.description)
\end{array}
$$

in order to work on the union of the sources, even though the two branches of the case statement contain identical code! This is also true for the (few) database query languages that deal with tagged union types [BLS+94].

Contrast this with a typical semi-structured query:

$$
\begin{array}{l}
\mathsf{select}\ [\, description = d,\ title = t \,] \\
\mathsf{where}\ [\, description = d,\ bibl = [\ Title = t \,]\,] \leftarrow source_1
\end{array} \tag{2}
$$

This query works by pattern matching based on the (dynamically determined) structure of the data. Thus the *same* query works equally well against either of the two sources, and hence also against their union[1]. The drawback of this approach, however, is that incorrect queries – for example, queries that use a field that does not exist in *either* source – yield the empty set rather than an error.

In this paper we define a system that combines the advantages of both approaches, based on a system of type-safe *untagged* union types. As a first example, consider the two forms of the *author* field in the types above. We may write the union of these types as:

$$[\, name\colon Str,\ address\colon Str \,] \vee [\, ln\colon Str,\ fn\colon Str,\ address\colon Str \,]$$

It is intuitively obvious that an *address* can always be extracted from a value of such a type. To express this formally, we begin by writing a multi-field record type $[l_1 : T_1, l_2 : T_2, \ldots]$ as a product of single-field record types: $[l_1 : T_1] \times [l_2 : T_2] \times \ldots$. In this more basic form, the union type above is:

$$([\, name\colon Str \,] \times [\, address\colon Str \,]) \vee ([\, ln\colon Str \,] \times [\, fn\colon Str \,] \times [\, address\colon Str \,])$$

We now invoke a *distributivity* law that allows us to treat

$$[a : T_a] \times ([b : T_b] \vee [c : T_c]) \quad \text{and} \quad ([a : T_a] \times [b : T_b]) \vee ([a : T_a] \times [c : T_c])$$

as equivalent types. Using this, the union type above rewrites to:

$$([\, name\colon Str \,] \vee [\, fn\colon Str \times ln\colon Str \,]) \times [\, address\colon Str \,]$$

In this form, it is evident that the the selection of the *address* field is an allowable operation.

Type-equivalences like this distributivity rule allow us to introduce a value at one type and operate on it another type. Under this system both the program (1) and the query (2) above will type-check when extended to the union of the two sources. On the other hand, queries that reference a field that is not in either source will fail to type check.

Some care is needed in designing the operations for manipulating values of union types. Usually, the interrogation operation for records is field selection and the corresponding operation for unions is a case expression. However it is not enough simply to use these two operations. Consider the type $([a_1 : T_1] \vee [b_1 : U_1]) \times \ldots \times ([a_n : T_n] \vee [b_n : U_n])$. The form of this type warrants neither selecting a field nor using a case expression. We can, if we want, use distributivity to rewrite it into a disjunct of products, but the size of this

[1] One could also achieve the same effect through the use of inheritance rather than union types in some object-oriented language. This would involve the introduction of named classes with explicit subclass assertions. As we shall shortly see, the number of possible classes is exponential in the size of the type.

disjunct is exponential in n and so, presumably, would be the corresponding case expression. We propose, instead, an extended pattern matching syntax that allows us to operate on the type in its original, compact, form.

More sophisticated pattern matching operations may be useful additions even to existing semistructured query languages. Consider the problem of writing a query that produces a uniform output from a single source that contains two representations of names:

$$\begin{array}{l}
(\ \mathsf{select}\ [\, \mathit{description} = d, \mathit{name} = n \,] \\
\quad \mathsf{where}\ [\, \mathit{description} = d, \mathit{bibl} = [\, \mathit{author} = [\, \mathit{name} = n \,]\,]\,] \leftarrow \mathit{source}\ ) \\
\mathsf{union} \\
(\ \mathsf{select}\ [\, \mathit{description} = d, \mathit{name} = \mathit{string\text{-}concat}(f, l) \,] \\
\quad \mathsf{where}\ [\, \mathit{description} = d, \mathit{bibl} = [\, \mathit{author} = [\, \mathit{ln} = l, \mathit{fn} = f \,]\,]\,] \leftarrow \mathit{source}\ )
\end{array}$$

This is the only method known to the authors of expressing this query in current semistructured query languages. It suggests an inefficient execution model and may not have the intended semantics when, for example, the source is a list and one wants to preserve the order. Thus some enhancement to the syntax is desirable.

This paper develops a type system based on untagged union types along with operations to construct and deconstruct these types. In particular, we define a syntax of patterns that may be used both for an extended form of case expression and as an extension to existing query languages for semi-structured data. We should remark that we cannot capture all aspects of semistructured query languages. For example, we have nothing that corresponds to "regular path expressions" [BDHS96, AQM$^+$96]. However, we believe that for most examples of "mildly" semistructured data – especially the forms that arise from the integration of typed data sources – a language such as proposed here will be adequate. Our main technical contribution is a proof of the decidabiliity of subtyping for this type system (which is complicated by the non-trivial equivalences involving union and record types).

To our knowledge, untagged union types never been formalized in the context of database programming languages. *Tagged* union types have been suggested in several papers on data models [AH87, CM94] but have had minimal impact on the design of query languages. CPL [BLS$^+$94], for example, can match on only one tag of a tagged union, and this is one of the few languages that makes use of union types. Pattern matching has been recently exploited in languages for semi-structured data and XML [BDHS96, DFF$^+$]. In the programming languages and type theory communities, on the other hand, untagged union types have been studied extensively from a theoretical perspective [Pie91, BDCd95, Hay91, Dam94, DCdP96, etc.], but the interactions of unions with higher-order function types have been shown to lead to significant complexities; the present system provides only a very limited form of function types (like most database query languages), and remains reasonably straightforward. .

Section 2 develops our language for programming with record and union types, including pattern matching primitives that can be used in both case expressions and query languages. Section 3 describes the system formally and demonstrates the decidability of subtyping and type equivalence. Proofs will be provided in the full paper. Section 4 offers concluding remarks.

2 Programming with Union Types

In this section we shall develop a syntax for the new programming constructs that are needed to deal with union types. The presentation is informal for the moment – more precise definitions appear in Section 3. We start with operations on records and extend these to work with unions of records; we then deal with operations on sets. Taken in conjunction with operations on records, these operations are enough to define a simple query language. We also look at operations on more general union types and give examples of a "typecase" operation.

2.1 Record formation

Just as we defined a record type $[\,l_1 : T_1, \ldots, l_n : T_n\,]$ as the product $[\,l_1 : T_1\,] \times \ldots \times [\,l_n : T_n\,]$ of elementary or "singleton" record types, we can define a record value as the disjoint concatenation of singleton records. The operations for creating records are the empty record $[\,]$, the singleton record $[\,l = e\,]$ (where e is an expression), and the disjoint concatenation of records $e\#e$. (Actually, we also allow multi-field record values of the form $[\,l_1 = e_1, \ldots, l_n = e_n\,]$, as this makes the operational semantics easier to state.)

2.2 Case expressions

Records are decomposed through the use of case expressions. These allow us to take alternative actions based on the structure of values. We shall also be able to use components of the syntax of case expressions in the development of matching constructs for query languages. The idea in developing a relatively complex syntax for the body of case expressions is that the structure of the body can be made to match the expected structure of the type of the value on which it is operating. There should be no need to "flatten" the type into disjunctive normal form and write a much larger case expression at that type.

We start with a simple example:

$$\begin{array}{ll} \mathsf{case}\ e\ \mathsf{of} & [\,fn = f{:}Str,\ \ ln = l{:}Str\,] \Rightarrow string\text{-}concat(f, l) \\ \quad\ \ | & [\,name = n{:}Str\,] \Rightarrow n \end{array}$$

This matches the result of evaluating e to one of two record types. If the result is a record with fn and ln fields, the variables f and l are bound and the right-hand side of the first clause is evaluated. If the first pattern does not match, the second clause is tried. This case expression will work provided e has type $[\,fn{:}\ Str, ln{:}\ Str\,] \vee [\,name{:}\ Str\,]$.

We should note that pattern matching introduces identifiers such as l, f, n in this example, and we shall make a short-sighted assumption that identifiers are introduced when they are associated with a type $(x : T)$. This ignores the possibility of type inference. See [BLS+94] for a more sophisticated syntax for introducing identifiers in patterns.

Field selection is given by a one-clause case expression: $\mathsf{case}\ e\ \mathsf{of}\ [\,l = x{:}T\,] \Rightarrow x$.

We shall also allow case expressions to dispatch on the run-time type of an argument:

$$\begin{array}{ll} \mathsf{case}\ e\ \mathsf{of} & x{:}Int \Rightarrow x \\ \quad\ \ | & y{:}\mathsf{set}(Int) \Rightarrow sum(y) \end{array}$$

This will typecheck when $e : Int \vee \mathsf{set}(Int)$

The clauses of a case expression have the form $p \Rightarrow e$, where p is a *pattern* that introduces (binds) identifiers which may occur free in the expression e. Thus each clause defines a *function*. Two or more functions can be combined by writing $p_1 \Rightarrow e_1 \ \mid\ p_2 \Rightarrow e_2 \mid \ldots$ to form another function. The effect of the case expression $\mathsf{case}\ e\ \mathsf{of}\ f$ is to apply this function to the result of evaluating e.

Now suppose we want to extract information from a value of type

$$([\,name{:}\ Str\,] \vee [\,ln{:}\ Str, fn{:}\ Str\,]) \times [\,age{:}\ Int\,] \tag{2}$$

The age field may be extracted using field selection using a one-clause case expression as described above. However information from the left-hand component cannot be extracted by extending this case expression. What we need is need something that will turn a multi-clause function back into a pattern that binds a new identifier. We propose the syntax x as f, in which f is a multi-clause function. In the evaluation of x as f, f is applied to the appropriate structure and x is bound to the result.

$$
\begin{array}{l}
\mathsf{case}\ \ e\ \mathsf{of} \\
\quad x\ as\ ([\,fn = f{:}Str,\ \ ln = l{:}Str\,] \Rightarrow string\text{-}concat(f, l)\ \mid\ [\,name = n{:}Str\,] \Rightarrow n) \\
\quad \#\ [\,age = a{:}Int\,] \\
\quad \Rightarrow [\,name = x, age = a + 1\,]
\end{array}
$$

could be applied to an expression e of type (2) above. Note the use of # to combine two patterns so that they match on a product type. This symbol is used to concatenate patterns in the same way that it is used to concatenate record values.

There are some useful extensions to case expressions and pattern matching that we shall briefly mention here but omit in the formal development (they are essentially syntactic sugar). The first is the addition of a "fall-through" or *else* branch of a case expression. The pattern else matches any value that has not been matched in a previous clause. Most programming languages have an analogous construct.

Such branches are particularly useful if we allow constants in patterns. For example

$$
\begin{array}{l}
\mathsf{case}\ \ e\ \mathsf{of}\ [\,name = n{:}Str, age = 21\,] \Rightarrow e \\
\quad \mid \qquad \mathsf{else} \Rightarrow \ldots
\end{array}
$$

Here only tuples with a specific value for *age* are matched. Tuples with a different value will be matched in the *else* clause. Note that patterns bind variables, and that if one allows constants in patterns, one wants to discriminate between those variables that are used as constants and those that are bound in the pattern. CPL [BLS+94] uses a special marker to flag bound variables. In that language $[\,name = n, age = \backslash a\,]$ is a pattern in which a is bound and n is treated as a constant – it is bound in some outer scope. This extended syntax of patterns is especially convenient when used in query languages for sets.

2.3 Sets

We shall follow the approach to collection types given in [BNTW95]. It is known that both relational and complex-object languages can be expressed using this formalism. The operations for forming sets are $\{e\}$ (singleton set) and e union e (set union).[2] For "iterating" over a set we use the form

$$\mathsf{collect}\ e\ \mathsf{where}\ p\ \leftarrow\ e'.$$

Here, e and e' are both expressions of set type, and p is a pattern as described above. The meaning of this is (informally) $\bigcup\{\sigma(e)\ \mid\ \sigma(p) \in e'\}$, in which σ is a substitution that binds the variables of p to match an element of e'.

These operations, taken in conjunction with the record operations described above and an equality operation, may be used as the basis of practical query languages. Conditionals and booleans may be added, but they can also be simulated with case expressions and some appropriately chosen constants.

Unlike typed systems with tagged unions, in our system there is no formation operation directly associated with the union type. However we may want to introduce operators such as "relaxed set-union," which takes two sets of type $\mathsf{set}(t_1)$ and $\mathsf{set}(t_2)$ and returns a set of type $\mathsf{set}(t1 \vee t2)$.

2.4 Examples

We conclude this section with some remarks on high-level query languages. A typical form of a query that makes use of pattern matching is:

[2] The present system does not include {} (empty set). It can be added, at the cost of a slight extension to the type system; see Section 4.

$$
\begin{array}{l}
\textsf{select}\ \ \mathrm{e} \\
\textsf{where}\ p_1 \leftarrow e_1, \\
\qquad\quad p_2 \leftarrow e_2, \\
\qquad\quad \ldots \\
\qquad\quad \mathit{condition}
\end{array}
$$

Here the p_i are patterns and the expressions $e_1, \ldots, e_i$ have set types. Variables introduced in pattern p_i may be used in expression e_j and (as constants) in pattern p_j where $j > i$. They may also be used in the expression e and the *condition*, which is simply a boolean expression. This query form can be implemented using the operations described in the previous section.

As an example, here is a query based on the example types in the introduction. We make use of the syntax of patterns as developed for case expressions, but here we are using them to match on elements of one or more input sets.

$$
\begin{array}{ll}
\textsf{select} & [\,\mathit{description} = d,\ \mathit{authName} = a,\ \mathit{year} = y\,] \\
\textsf{where} & [\,\mathit{description} = d{:}\mathit{Str},\ \mathit{bibl} = b{:}\mathit{BT}\,] \leftarrow \mathit{source}_1\ \textsf{union}\ \mathit{source}_2, \\
 & [\,\mathit{authors} = \mathit{aa}{:}\mathit{AT},\ \mathit{year} = y{:}\mathit{Int}\,] \leftarrow b, \\
 & a\ \textsf{as}\ ([\,\mathit{fn} = f{:}\mathit{Str},\ \mathit{ln} = l{:}\mathit{Str}\,] \Rightarrow \mathit{string\text{-}concat}(f,\ l)\ \mid\ [\,\mathit{name} = n{:}\mathit{Str}\,] \Rightarrow n) \leftarrow \mathit{aa}, \\
 & y > 1991
\end{array}
$$

Note that we have assumed a “relaxed” union to combine the two sources. In the interests of consistency with the formal development, we have also inserted all types for identifiers, so AT and BT are names for the appropriate fragments of the expected source type. In many cases such types can be inferred.

Here are two examples that show the use of paterns in matching on types rather than record structures. Examples of this kind are commomly used to illustrate the need for semistructured data.

$$
\begin{array}{l}
\textsf{select}\ \ x \\
\textsf{where}\ x\ \textsf{as}\ (s : \textsf{set}(\mathit{Num}) \Rightarrow \mathit{average}(s)\ \mid\ r : \mathit{Num} \Rightarrow r)\ \leftarrow\ \mathit{source}
\end{array}
$$

$$
\begin{array}{l}
\textsf{select}\ \ s \\
\textsf{where}\ s\ \textsf{as}\ (n : \mathit{Str} \Rightarrow\ n\ \mid\ [\,\mathit{fn} = f{:}\mathit{Str}, \mathit{ln} = l{:}\mathit{Str}\,] \Rightarrow \mathit{string\text{-}concat}(f, l)) \leftarrow \mathit{source}'
\end{array}
$$

In the first case we have a set *source* that may contain both numbers and sets of numbers. In the second case we have a set that may contain both base types and record types. Both of these can be statically type-checked. If, for example, in the first query, s has type $\textsf{set}(\mathit{Str})$, the query would not type-check.

To demonstrate the proposed syntax for the use of functions in patterns, here is one last (slightly contrived) example. We want to calculate the mass of a solid object that is either rectangular or a sphere. Each measure of length can be either integer or real. The type is

$$
\begin{array}{l}
[\,\mathit{density}{:}\ \mathit{Real}\,] \\
\times \\
(\quad [\,\mathit{intRadius}{:}\ \mathit{Int}\,] \vee [\,\mathit{realRadius}{:}\ \mathit{Real}\,] \\
\quad\ \vee \\
\quad\ (\quad ([\,\mathit{intHeight}{:}\ \mathit{Int}\,] \vee [\,\mathit{realHeight}{:}\ \mathit{Real}\,]) \\
\qquad\quad \times \\
\qquad\quad ([\,\mathit{intWidth}{:}\ \mathit{Int}\,] \vee [\,\mathit{realWidth}{:}\ \mathit{Real}\,]) \\
\qquad\quad \times \\
\qquad\quad ([\,\mathit{intDepth}{:}\ \mathit{Int}\,] \vee [\,\mathit{realDepth}{:}\ \mathit{Real}\,])\ )\ )
\end{array}
$$

The following case expression makes use of matching based on both unions and products of record structures. Note that the structure of the expression follows that of the type. It would be possible to write an equivalent case expression for the disjunctive normal form for the type and avoid the use of the form x as$f$, but such an expression would be much larger than the one given here.

$$
\begin{array}{l}
\mathsf{case}\ e\ \mathsf{of} \\
\quad [\,\mathit{density} = d{:}\mathit{Real}\,] \\
\quad \# \\
\quad v\ \mathsf{as} \\
\qquad (\quad r\ \mathsf{as}\ ([\,\mathit{intRadius} = \mathit{ir}{:}\mathit{Int}\,] \Rightarrow \mathit{float}(\mathit{ir}) \mid [\,\mathit{realRadius} = \mathit{rr}{:}\mathit{Real}\,] \Rightarrow \mathit{rr}) \\
\qquad\qquad \Rightarrow r{*}{*}3) \\
\qquad \mid \\
\qquad (\quad h\ \mathsf{as}\ ([\,\mathit{intHeight} = \mathit{ih}{:}\mathit{Int}\,] \Rightarrow \mathit{float}(\mathit{ih}) \mid [\,\mathit{realHeight} = \mathit{rh}{:}\mathit{Real}\,] \Rightarrow \mathit{rh}) \\
\qquad\quad \# \\
\qquad\quad w\ \mathsf{as}\ ([\,\mathit{intWidth} = \mathit{iw}{:}\mathit{Int}\,] \Rightarrow \mathit{float}(\mathit{iw}) \mid [\,\mathit{realWidth} = \mathit{rw}{:}\mathit{Real}\,] \Rightarrow \mathit{rw}) \\
\qquad\quad \# \\
\qquad\quad d\ \mathsf{as}\ ([\,\mathit{intDepth} = \mathit{id}{:}\mathit{Int}\,] \Rightarrow \mathit{float}(\mathit{id}) \mid [\,\mathit{realDepth} = \mathit{rd}{:}\mathit{Real}\,] \Rightarrow \mathit{rd}) \\
\qquad\qquad \Rightarrow h * w * d) \\
\quad \Rightarrow d * v
\end{array}
$$

3 Formal Development

With the foregoing intuitions and examples in mind, we now proceed to the formal definition of our language, its type system, and its operational semantics. Along the way, we establish fundamental properties such as run-time safety and the decidability of subtyping and type-checking.

3.1 Types

We develop a type system that is based on conventional complex object types, those that are constructed from the base types with record (tuple) and set constructors. As described in the introduction, the record constructors are $[\,]$, the empty record type, $[l\colon t]$, the singleton record type, and $R \times R$, the *disjoint* concatenation of two records types. (By disjoint we mean that the two record types have no field names in common.) Thus a conventional record type $[\,l_1 : T_1, \ldots, l_n : T_n\,]$ is shorthand for $[\,l_1 : Y_1\,] \times \ldots \times [\,l_n : T_n\,]$. To this we add an untagged union type $T \vee T$. We also assume a single base type B and a set type $\mathsf{set}(T)$. Other collection types such as lists and multisets would behave similarly,

The syntax of types is described by the following grammar:

T	$::=$	B	base type
		$[\,]$	empty record type
		$[\,l : T\,]$	labeling (single-field record type)
		$T_1 \times T_2$	record type concatenation
		$T_1 \vee T_2$	union type
		$\mathsf{set}(T)$	set type

3.2 Kinding

We have already noted that certain operations on types are restricted. For example, we cannot take the product of two record types with a common field name. This in turn means that any operation on records whose typing rules make improper use of a type constructor is also illegal. In order to control the formation of types we introduce a system of *kinds*. This consists of the kind of all types, Type, and a subkind $\mathsf{Rcd}(L)$, which is the kind of all record types whose labels are included in the label set L.

K	$::=$	Type	kind of all types
		$\mathsf{Rcd}(L)$	kind of record types with (at most) labels L

The kinding relation is defined as follows:

$$\mathsf{B} \in \mathsf{Type} \tag{K-Base}$$

$$[\,] \in \mathsf{Rcd}(\{\}) \tag{K-Empty}$$

$$\frac{T \in \mathsf{Type}}{[\, l : T \,] \in \mathsf{Rcd}(\{l\})} \tag{K-Field}$$

$$\frac{S \in \mathsf{Rcd}(L_1) \qquad T \in \mathsf{Rcd}(L_2) \qquad L_1 \cap L_2 = \emptyset}{S \times T \in \mathsf{Rcd}(L_1 \cup L_2)} \tag{K-Rcd}$$

$$\frac{S \in K \qquad T \in K}{S \vee T \in K} \tag{K-Union}$$

$$\frac{T \in \mathsf{Type}}{\mathsf{set}(T) \in \mathsf{Type}} \tag{K-Set}$$

$$\frac{T \in \mathsf{Rcd}(L_1)}{T \in \mathsf{Rcd}(L_1 \cup L_2)} \tag{K-Subsumption-1}$$

$$\frac{T \in \mathsf{Rcd}(L)}{T \in \mathsf{Type}} \tag{K-Subsumption-2}$$

There are two important consequences of these rules. First, record kinds extend to the union type. For example, $([\, A : t \,] \times [\, B : t \,]) \times ([\, C : t \,] \vee [\, D : t \,])$ has kind $\mathsf{Rcd}(\{A, B, C, D\})$. Second, the kinding rules require the labels in a concatenation of two record types to be disjoint. (However the union type constructor is not limited in the same way; $Int \vee Str$ and $Int \vee [a : Str]$ are well-kinded types.)

3.3 Subtyping

As usual, the subtype relation written $S <: T$ captures a principle of "safe substitutibility": any element of S may safely be used in a context expecting an element of T.

For sets and records, the subtyping rules are the standard ones: $\mathsf{set}(S) <: \mathsf{set}(T)$ if $S <: T$ (e.g., a set of employees can be used as a set of people), and a record type S is a subtype of a record type T if S has more fields than T and the types of the common fields in S are subtypes of the corresponding fields in T. This effect is actually achieved by the combination of several rules below. This "exploded presentation" of record subtyping corresponds to our presentation of record types in terms of separate empty set, singleton, and concatenation constructors.

For union types, the subtyping rules are a little more interesting. First, we axiomatize the fact that $S \vee T$ is the least upper bound of S and T – that is, $S \vee T$ is above both S and T, and everything that is above both S and T is also above their union (rules S-Union-UB and S-Union-L below). We then have two rules (S-Dist-Rcd and S-Dist-Field) showing how union distributes over records.

Formally, the subtype relation is the least relation on well-kinded types closed under the following rules.

$$T <: T \tag{S-Refl}$$

$$\frac{R <: S \qquad S <: T}{R <: T} \tag{S-Trans}$$

$$[\, l : T \,] <: [\,] \tag{S-Rcd-FE}$$

$$S \times T <: S \tag{S-Rcd-RE}$$

$$S \times T <: T \times S \tag{S-Rcd-Comm}$$

$$S \times (T \times U) <: (S \times T) \times U \tag{S-Rcd-Assoc}$$

$$S <: S \times [\,] \tag{S-Rcd-Ident}$$

$$\frac{S <: T}{[\, l : S \,] <: [\, l : T \,]} \tag{S-Rcd-DF}$$

$$\frac{S_1 <: T_1 \qquad S_2 <: T_2}{S_1 \times S_2 <: T_1 \times T_2} \tag{S-Rcd-DR}$$

$$\frac{S <: T}{\mathsf{set}(S) <: \mathsf{set}(T)} \tag{S-Set}$$

$$\frac{R <: T \qquad S <: T}{R \vee S <: T} \tag{S-Union-L}$$

$$S_i <: S_1 \vee S_2 \tag{S-Union-UB}$$

$$R \times (S \vee T) <: (R \times S) \vee (R \times T) \tag{S-Dist-Rcd}$$

$$[\, l : S \vee T \,] <: [\, l : S \,] \vee [\, l : T \,] \tag{S-Dist-Field}$$

Note that we restrict the subtype relation to *well-kinded* types: S is never a subtype of T if either S or T is ill-kinded. (The typing rules will be careful only to "call" the subtype relation on types that are already known to be well kinded.)

If both $S <: T$ and $T <: S$, we say that S and T are *equivalent* and write $S \sim T$. Note, for example, that the distributive laws S-Dist-Rcd and S-Dist-Field are actually equivalences: the other directions follow from the laws for union (plus transitivity). Also, note the absence of the "other" distributivity law for unions and records: $P \vee (Q \times R) \sim (P \vee Q) \times (P \vee R)$. This law doesn't make sense here, because it violates the kinding constraint that products of record types can only be formed if the two types have disjoint label sets.

The subtype relation includes explicit rules for associativity and commutativity of the operator $\times$. Also, it is easy to check that the associativity, commutativity and idempotence of $\vee$ follow directly from the rules given. We shall take advantage of this fluidity in the following by writing both records and unions in a compound, n-ary form:

$$\begin{array}{lcl} [\, l_1 : T_1, \ldots, l_n : T_n \,] & \stackrel{def}{=} & [\, l_1 : T_1 \,] \times \cdots \times [\, l_n : T_n \,] \\ \bigvee(T_1, \ldots, T_n) & \stackrel{def}{=} & T_1 \vee \cdots \vee T_n \end{array}$$

(In the first line, n may be 0—we allow empty records—but in the second, it must be positive—for brevity, we do not allow "empty unions" in the present system. See Section 4.)

We often write compound unions using a simple comprehension notation. For example,

$$\bigvee(A{\times}B \ \mid\ A \in A_1{\vee}A_2{\vee}\ldots{\vee}A_m \text{ and } B \in B_1{\vee}B_2{\vee}\ldots{\vee}B_n)$$

denotes

$$A_1{\times}B_1 \vee A_1{\times}B_2 \vee \ldots \vee A_1{\times}B_n \vee A_2{\times}B_1 \vee \ldots \vee A_m{\times}B_n.$$

3.4 Properties of Subtyping

For proving properties of the subtype relation, it is convenient to work with types in a more constrained syntactic form:

3.4.1 Definition: The sets of *normal* (N) and *simple* (A) types are defined as follows:

$$
\begin{array}{lll}
N & ::= & \bigvee(A_1,\ldots,A_n) \\[1em]
A & ::= & \mathsf{B} \\
 & & [\, l_1 : A_1, \ldots, l_n : A_n \,] \\
 & & \mathsf{set}(N)
\end{array}
$$

Intuitively, a simple type is one in which unions only appear (immediately) inside of the set constructor; a normal type is a union of simple types. Note that every simple type is also normal. □

The restricted form of normal and simple types can be exploited to give a much simpler subtyping relation, written $S \underline{<:} T$, in terms of the following "macro rules":

$$\mathsf{B} \underline{<:} \mathsf{B} \qquad \text{(SA-Base)}$$

$$\frac{N \underline{<:} M}{\mathsf{set}(N) \underline{<:} \mathsf{set}(M)} \qquad \text{(SA-Set)}$$

$$\frac{\{k_1,\ldots,k_m\} \subseteq \{l_1,\ldots,l_n\} \qquad \text{for all } k_i \in \{k_1,\ldots,k_m\},\ \ A_{k_i} \underline{<:} B_{k_i}}{[\, l_1 : A_{l_1}, \ldots l_m : A_{l_m} \,] \underline{<:} [\, k_1 : B_{k_1}, \ldots, k_m : B_{k_m} \,]} \qquad \text{(SA-Rcd)}$$

$$\frac{\forall i \leq m.\ \exists j \leq n.\ A_i \underline{<:} B_j}{\bigvee(A_1,\ldots,A_m) \underline{<:} \bigvee(B_1,\ldots,B_n)} \qquad \text{(SN-Union)}$$

3.4.2 Fact: $N \underline{<:} M$ is decidable. □

Proof: The macro rules can be read as a pair of algorithms, one for subtyping between simple types and one for subtyping between normal types. Both of these algorithms are syntax directed and obviously terminate on all inputs (all recursive calls reduce the size of the inputs). □

3.4.3 Lemma: $N \underline{<:} N$, for all N. □

3.4.4 Lemma: If $N \underline{<:} M$ and $M \underline{<:} L$ then $N \underline{<:} L$. □

Proof: By induction on the total size of L, M, N. First suppose that all of L, M, N are simple. The induction hypothesis is immediately satisfied for SA-Base and SA-Set. For SA-Rcd use the transitivity of set inclusion and induction on the appropriate subterms.

If at least one of L, M, N is (non-trivially) normal, use the transitivity of the functional relationship expressed by the SN-Union rule and induction on the appropriate subterms. □

To transfer the property of decidability from $\underline{<:}$ to $<:$, we first show how any type may be converted to an equivalent type in *disjunctive normal form*.

3.4.5 Definition: The disjunctive normal form (dnf) of a type T is defined as follows:

$$
\begin{array}{lcll}
\mathit{dnf}(\mathsf{B}) & = & \mathsf{B} & \\
\mathit{dnf}([\,]) & = & [\,] & \\
\mathit{dnf}(P \times Q) & = & \bigvee(A_i \times B_j \mid A_i \in \mathit{dnf}(P),\ B_j \in \mathit{dnf}(Q)) & \text{(a)} \\
\mathit{dnf}([\,l : P\,]) & = & \bigvee([\,l : A_i\,] \mid A_i \in \mathit{dnf}(P)) & \text{(b)} \\
\mathit{dnf}(P \vee Q) & = & \mathit{dnf}(P) \vee \mathit{dnf}(Q) & \text{(c)} \\
\mathit{dnf}(\mathsf{set}(P)) & = & \mathsf{set}(\mathit{dnf}(P)) & \text{(d)}
\end{array}
$$

□

3.4.6 Fact: $\mathit{dnf}(P) \sim P$. □

3.4.7 Fact: $\mathit{dnf}(P)$ is a normal type, for every type P. □

3.4.8 Fact: $N \underline{<:} M$ implies $N <: M$ □

3.4.9 Lemma: $S <: T$ iff $\mathit{dnf}(S) \underline{<:} \mathit{dnf}(T)$ □

Proof: ($\Leftarrow$) By 3.4.6 we have derivations of $S <: \mathit{dnf}(S)$ and $\mathit{dnf}(T) <: T$, and by 3.4.8 we have a derivation of $\mathit{dnf}(S) <: \mathit{dnf}(T)$. Use transitivity to build a derivation of $S <: T$.

($\Rightarrow$) By induction on the height of the derivation of $S <: T$. We consider the final rule in the derivation. By induction we assume we can build a derivation of the normal forms for the antecedents, and now we consider all possible final rules.

We start with the axioms.

(S-Refl) By reflexivity of $\underline{<:}$ (3.4.3).

(S-Rcd-FE) $\mathit{dnf}([\,l : T\,]) = [\,l : \mathit{dnf}(T)\,]$, and $[\,l : \mathit{dnf}(T)\,] \underline{<:} [\,]$ by SA-Rcd.

(S-Rcd-RE) $\mathit{dnf}(S \times T) = \bigvee(S_i \times T_j \mid S_i \in \mathit{dnf}(S),\ T_j \in \mathit{dnf}(T))$. Now $\mathit{dnf}(S_i) \times \mathit{dnf}(T_j) \underline{<:} \mathit{dnf}(S_i)$ by SA-Rcd, and the result follows from SN-Union.

(S-Rcd-Comm) If $\mathit{dnf}(S)$ and $\mathit{dnf}(T)$ are simple then $\mathit{dnf}(S) \times \mathit{dnf}(T) \underline{<:} \mathit{dnf}(T) \times \mathit{dnf}(S)$ by SA-Rcd. If not, use SN-Union first.

(S-Rcd-Assoc) As for S-Rcd-Comm.

(S-Rcd-Ident) As for S-Rcd-Comm.

(S-Union-UB) By SN-Union.

(S-Dist-Rcd)
$$
\begin{array}{rcl}
\mathit{dnf}(R \times (S \vee T)) & = & \bigvee(R_i \times U_j \mid R_i \in \mathit{dnf}(R),\ U_j \in \mathit{dnf}(S \vee T)) \\
& = & \bigvee(R_i \times S_j \mid R_i \in \mathit{dnf}(R),\ U_j \in \mathit{dnf}(S)) \vee \\
& & \bigvee(R_i \times T_k \mid R_i \in \mathit{dnf}(R),\ T_k \in \mathit{dnf}(T)) \\
& = & \mathit{dnf}((R \times S) \vee (R \times T))
\end{array}
$$

$$
\begin{array}{llll}
(\textsc{S-Dist-Field}) & \mathit{dnf}([\,l : S \vee T\,]) & = & \bigvee([\,l : U_i\,] \mid U_i \in \mathit{dnf}(S \vee T)) \\
& & = & \bigvee([\,l : S_i\,] \mid S_i \in \mathit{dnf}(S)) \vee \bigvee([\,l : T_i\,] \mid T_i \in \mathit{dnf}(T)) \\
& & = & \mathit{dnf}([\,l : S\,]) \vee \mathit{dnf}([\,l : T\,]) \\
& & = & \mathit{dnf}([\,l : S\,] \vee [\,l : T\,])
\end{array}
$$

Now for the inference rules. The premises for all the rules are of the form $S <: T$ and our inductive hypothesis is that for the premises of the final rule we have obtained a derivation using SA-* and SN-Union rules of the corresponding $\mathit{dnf}(S) \underline{<:} \mathit{dnf}(T)$ Without loss of generality we may assume that the final rule in the derivation of each such premise is SN-Union. We examine the remaining inference rules.

(S-Trans) By Lemma 3.4.4.

(S-Rcd-DF) Since $\mathit{dnf}(S) \underline{<:} \mathit{dnf}(T)$ was derived by SN-Union we know that for each $A_i \in \mathit{dnf}(S)$ there is a $B_j \in \mathit{dnf}(T)$ such that $A_i \underline{<:} B_j$. Therefore, for each such A_i, we may use SA-Rcd to derive $[\,l : A_i\,] \underline{<:} [\,l : B_j\,]$. These derivations may be combined using SN-Union to obtain a derivation of $\mathit{dnf}([\,l : S\,]) \underline{<:} \mathit{dnf}([\,l : T\,])$.

(S-Rcd-DR) For each $A^1_{i_1} \in \mathit{dnf}(S_1)$ and each $A^2_{i_2} \in \mathit{dnf}(S_2)$ there exist $B^1_{j_1} \in \mathit{dnf}(T_1)$ and $B^2_{j_2} \in \mathit{dnf}(T_2)$ such that we have a derivations of $A^1_{i_1} \underline{<:} B^1_{j_1}$ and $A^2_{i_2} \underline{<:} B^2_{j_2}$. For each such pair we can therefore use SA-Rcd to derive $A^1_{i_1} \times A^2_{i_2} \underline{<:} B^1_{j_1} \times B^2_{j_2}$ and then use SN-Union to derive $\mathit{dnf}(S_1 \times S_2) \underline{<:} \mathit{dnf}(T_1 \times T_2)$.

(S-Set) Immediate, by SA-Set.

(S-Union-L) For each $A_i \in \mathit{dnf}(R)$ there is a $C_j \in \mathit{dnf}(T)$ such that $A_i \underline{<:} C_j$ and for each $B_k \in \mathit{dnf}(S)$ there is a $C_l \in \mathit{dnf}(T)$ such that $B_k \underline{<:} C_l$. From these $\mathit{dnf}(R \vee S) \underline{<:} \mathit{dnf}(T)$ can be derived directly using SN-Union. □

3.4.10 Theorem: The subtype relation is decidable. □

Proof: Immediate from Lemmas 3.4.9 and 3.4.2. □

We do not yet have any results on the *complexity* of checking subtyping (or equivalence). (The proof strategy we have adopted here leads to an algorithm with running time exponential in the size of its inputs.)

The structured form of the macro rules can be used to derive several *inversion properties*, which will be useful later in reasoning about the typing relation.

3.4.11 Corollary: If $S <: \mathsf{set}(T_1)$, then $S = \mathsf{set}(S_1)$, with $S_1 <: T_1$. □

3.4.12 Corollary: If $W <: U$, with

$$
\begin{array}{l}
W = [\,l_1 : W_1\,] \times \ldots \times [\,l_m : W_m\,] \times \ldots \times [\,l_n : W_n\,] \\
U = [\,l_1 : U_1\,] \times \ldots \times [\,l_m : U_m\,],
\end{array}
$$

then $W_k <: U_k$ for each $k \leq m$. □

Proof: From the definition of disjunctive normal forms, we know that $\mathit{dnf}(W) = \bigvee([\,l_1 : W_{i1}\,] \times \ldots \times [\,l_m : W_{im}\,] \times \ldots \times [\,l_n : W_{in}\,] \mid W_{i1} \ldots W_{im} \ldots W_{in} \in \mathit{dnf}(W_1) \ldots \mathit{dnf}(W_m) \ldots \mathit{dnf}(W_n))$ and $\mathit{dnf}(U) = \bigvee([\,l_1 : U_{j1}\,] \times \ldots \times [\,l_m : U_{jm}\,] \mid U_{j1} \ldots U_{jm} \in \mathit{dnf}(U_1) \ldots \mathit{dnf}(U_m))$. By SN-Union,

for each $A_i = [\,l_1 : W_{i1}\,] \times \ldots \times [\,l_m : W_{im}\,] \times \ldots \times [\,l_n : W_{in}\,] \in \mathit{dnf}(W)$
  there is some $B_j = [\,l_1 : U_{j1}\,] \times \ldots \times [\,l_m : U_{jm}\,] \in \mathit{dnf}(U)$
    with $A_i \underline{<:} B_j$.

This derivation must be an instance of SA-RCD, with $W_{ik} \trianglelefteq U_{jk}$. In other words, for each $W_{ik} \in \mathit{dnf}(W_k)$ there is some $Ujk \in \mathit{dnf}(U_k)$ with $W_{ik} \trianglelefteq U_{jk}$. By SN-UNION, $\mathit{dnf}(W_k) \trianglelefteq \mathit{dnf}(U_k)$. The desired result, $W_k <: U_k$ now follows by Lemma 3.4.9. □

3.4.13 Corollary: If S is a simple type and $S <: T_1 \vee T_2$, then either $S <: T_1$ or else $S <: T_2$. □

3.5 Terms

The sets of programs, functions, and patterns are described by the following grammar:

$$
\begin{array}{lcll}
e & ::= & \mathsf{b} & \text{base value} \\
 & & x & \text{variable} \\
 & & [\, l_1 = e_1, \ldots, l_n = e_n \,] & \text{record construction} \\
 & & e_1 \mathbin{\#} e_2 & \text{record concatenation} \\
 & & \mathsf{case}\ e\ \mathsf{of}\ f & \text{pattern matching} \\
 & & \{\, e_1, \ldots, e_n \,\} & \text{set} \\
 & & e_1\ \mathsf{union}\ e_2 & \text{union of sets} \\
 & & \mathsf{collect}\ e_1\ \mathsf{where}\ p \leftarrow e_2 & \text{set comprehension} \\
 & & & \\
p & ::= & x : T & \text{variable pattern (typecase)} \\
 & & [\, l_1 = p_1, \ldots, l_n = p_n \,] & \text{record pattern} \\
 & & p_1 \mathbin{\#} p_2 & \text{pattern concatenation} \\
 & & x\ \mathsf{as}\ f & \text{function nested in pattern} \\
 & & & \\
f & ::= & p \Rightarrow e & \text{base function} \\
 & & f_1 \mid f_2 & \text{compound function}
\end{array}
$$

3.6 Typing

The typing rules are quite standard.

Expressions $(\Gamma \vdash e \in T)$

$$\Gamma \vdash \mathsf{b} \in \mathsf{B} \qquad \text{(T-BASE)}$$

$$\Gamma \vdash x \in \Gamma(x) \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_i \in T_i \qquad \text{all the } l_i \text{ are distinct}}{\Gamma \vdash [\, l_1 = e_1, \ldots, l_n = e_n \,] \in [\, l_1 : T_1 \,] \times \cdots \times [\, l_n : T_n \,]} \qquad \text{(T-RCD)}$$

$$\frac{\Gamma \vdash e_1 \in T_1 \qquad \Gamma \vdash e_2 \in T_2 \qquad T_1 \times T_2 \in K}{\Gamma \vdash e_1 \mathbin{\#} e_2 \in T_1 \times T_2} \qquad \text{(T-CONCAT)}$$

$$\frac{\Gamma \vdash f \in S \to T \qquad \Gamma \vdash e \in R \qquad R <: S}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ f \in T} \qquad \text{(T-CASE)}$$

$$\frac{\Gamma \vdash e_i \in T_i \quad \text{for each } i \qquad n \geq 1}{\Gamma \vdash \{\, e_1, \ldots, e_n \,\} \in \mathsf{set}(T_1 \vee \cdots \vee T_n)} \qquad \text{(T-SET)}$$

$$\frac{\Gamma \vdash e_1 \in \mathsf{set}(T_1) \qquad \Gamma \vdash e_2 \in \mathsf{set}(T_2)}{\Gamma \vdash e_1\ \mathsf{union}\ e_2 \in \mathsf{set}(T_1 \vee T_2)} \qquad \text{(T-UNION)}$$

$$\frac{\Gamma \vdash e_2 \in \mathsf{set}(S) \qquad \Gamma \vdash p \in U \Rightarrow \Gamma' \qquad S <: U \qquad \Gamma, \Gamma' \vdash e_1 \in \mathsf{set}(T)}{\Gamma \vdash \mathsf{collect}\ e_1\ \mathsf{where}\ p \leftarrow e_2 \in \mathsf{set}(T)} \tag{T-Collect}$$

Functions $(\Gamma \vdash f \in S \to T)$

$$\frac{\Gamma \vdash p \in S \Rightarrow \Gamma' \qquad \Gamma, \Gamma' \vdash e \in T}{\Gamma \vdash p \Rightarrow e \in S \to T} \tag{TF-Pat}$$

$$\frac{\Gamma \vdash f_1 \in S_1 \to T_1 \qquad \Gamma \vdash f_2 \in S_2 \to T_2}{\Gamma \vdash f_1 \mid f_2 \in S_1 \vee S_2 \to T_1 \vee T_2} \tag{TF-Alt}$$

Patterns $(\Gamma \vdash p \in T \Rightarrow \Gamma')$

$$\frac{T \in K}{\Gamma \vdash x : T \in T \Rightarrow x : T} \tag{TP-Var}$$

$$\frac{\Gamma \vdash p_i \in T_i \Rightarrow \Gamma'_i \qquad \text{the } \Gamma'_i \text{ all have disjoint domains}}{\Gamma \vdash [\, l_1 = p_1, \ldots, l_n = p_n \,] \in [\, l_1 : T_1 \,] \times \cdots \times [\, l_n : T_n \,] \Rightarrow \Gamma'_1, \ldots, \Gamma'_n} \tag{TP-Rcd}$$

$$\frac{\begin{array}{c}\Gamma \vdash p_1 \in [\, k_1 : S_1, \ldots, k_m : S_m \,] \Rightarrow \Gamma'_1 \qquad \Gamma \vdash p_2 \in [\, l_1 : T_1, \ldots, l_n : T_n \,] \Rightarrow \Gamma'_2 \\ \{k_1, \ldots, k_m\} \cap \{l_1, \ldots, l_n\} = \emptyset \qquad \Gamma'_1 \text{ and } \Gamma'_2 \text{ have disjoint domains}\end{array}}{\Gamma \vdash p_1 \mathbin{\#} p_2 \in [\, k_1 : S_1, \ldots, k_m : S_m, l_1 : T_1, \ldots, l_n : T_n \,] \Rightarrow \Gamma'_1, \Gamma'_2} \tag{TP-Concat}$$

$$\frac{\Gamma \vdash f \in S \to T}{\Gamma \vdash x\ \mathsf{as}\ f \in S \Rightarrow x : T} \tag{TP-As}$$

3.7 Properties of Typing

3.7.1 Proposition: The typing relation is decidable. □

Proof: Immediate from the decidability of subtyping and the syntax-directedness of the typing rules. □

3.7.2 Definition: A substitution σ is a finite function from variables to terms. We say that a substitution σ *satisfies* a context Σ, written $\sigma \models \Sigma$, if they have the same domain and, for each x in their common domain, we have $\vdash \sigma(x) \in S_x$ for some S_x with $S_x <: \Sigma(x)$. □

3.7.3 Definition: We say that a typing context Γ *refines* another context Γ', written $\Gamma <: \Gamma'$, if their domains are the same and, for each $x \in \mathit{dom}(\Gamma)$, we have $\Gamma(x) <: \Gamma'(x)$. □

3.7.4 Fact [Narrowing]: If $\Gamma \vdash e \in T$ and $\Gamma <: \Gamma'$, then $\Gamma' \vdash e \in T$. □

3.7.5 Lemma [Substitution preserves typing]:

1. If $\Sigma \models \sigma$ and $\Sigma, \Delta \vdash e \in Q$ then $\Delta \vdash \sigma(e) \in P$, for some $P <: Q$.
2. If $\Sigma \models \sigma$ and $\Sigma, \Delta \vdash f \in S \to Q$ then $\Delta \vdash \sigma(f) \in S \to P$, for some $P <: Q$.
3. If $\Sigma \models \sigma$ and $\Sigma, \Delta \vdash p \in U \Rightarrow \Delta'$ then $\Delta \vdash \sigma(p) \in U \Rightarrow \Delta''$, for some $\Delta'' <: \Delta$. □

Proof: By simultaneous induction on derivations. The arguments are all straightforward, using previously established facts. (For the second property, note that substitution into a pattern only affects functions that may be embedded in the pattern, since all other variables mentioned in the pattern are binding occurrences. Moreover, by our conventions about names of bound variables, we must assume that the variables bound in an expression, function, or pattern are distinct from those defined by σ.) □

3.8 Evaluation

The operational semantics of our language is again quite standard: we define a relation $e \Downarrow v$, read "(closed) expression e evaluates to result v," by a collection of syntax-directed rules embodying a simple abstract machine.

3.8.1 Definition: We will use the metavariables v and w to range over *values* – closed expressions not involving case, union, concatenation, or $\mathsf{collect}$.

$$
\begin{array}{rcl}
v & ::= & \mathsf{b} \\
 & & [\,l_1 = v_1, \ldots, l_n = v_n\,] \\
 & & \{\,v_1, \ldots, v_n\,\}
\end{array}
$$

We write $\bar{v}$ as shorthand for a set of values $v_1, ..., v_n$. □

3.8.2 Definition: A *substitution* σ is a finite function from variables to values. When σ_1 and σ_2 have disjoint domains, we write $\sigma_1 + \sigma_2$ for their combination. □

Reduction ($e \Downarrow v$, for closed terms e)

$$\mathsf{b} \Downarrow \mathsf{b} \qquad \text{(E-Base)}$$

$$\frac{e_i \Downarrow v_i \quad \text{for each } i}{[\,l_1 = e_1, \ldots, l_n = e_n\,] \Downarrow [\,l_1 = v_1, \ldots, l_n = v_n\,]} \qquad \text{(E-Rcd)}$$

$$\frac{\begin{array}{c} e_1 \Downarrow [\,l_1 = v_1, \ldots, l_m = v_m\,] \qquad e_2 \Downarrow [\,j_1 = w_1, \ldots, j_n = w_n\,] \\ \{l_1, \ldots, l_m\} \cap \{j_1, \ldots, j_n\} = \emptyset \end{array}}{e_1 \mathbin{\#} e_2 \Downarrow [\,l_1 = v_1, \ldots, l_m = v_m, j_1 = w_1, \ldots, j_n = w_n\,]} \qquad \text{(E-Concat)}$$

$$\frac{e \Downarrow v \qquad \mathit{match}(v,\, f) \Rightarrow v'}{\mathsf{case}\ f\ \mathsf{of}\ e \Downarrow v'} \qquad \text{(E-Case)}$$

$$\frac{e_i \Downarrow v_i \quad \text{for each } i}{\{\,e_1, \ldots, e_n\,\} \Downarrow \{\,v_1, \ldots, v_n\,\}} \qquad \text{(E-Set)}$$

$$\frac{e_1 \Downarrow \{\,\bar{v_1}\,\} \qquad e_2 \Downarrow \{\,\bar{v_2}\,\}}{e_1\ \mathsf{union}\ e_2 \Downarrow \{\,\bar{v_1} \cup \bar{v_2}\,\}} \qquad \text{(E-Union)}$$

$$\frac{\begin{array}{c} e_2 \Downarrow \{\,v_1, \ldots, v_n\,\} \\ \text{for each } i, \quad \mathit{match}(v_i,\, p) \Rightarrow \sigma_i \text{ and } \sigma_i(e_1) \Downarrow \{\,\bar{w_i}\,\} \end{array}}{\mathsf{collect}\ e_1\ \mathsf{where}\ p \leftarrow e_2 \Downarrow \{\,\bar{w_1} \cup \cdots \cup \bar{w_n}\,\}} \qquad \text{(E-Collect)}$$

Function matching ($\mathit{match}(v,\, f) \Rightarrow v'$)

$$\frac{match(v,\,p) \Rightarrow \sigma \qquad \sigma(e) \Downarrow v'}{match(v,\,p \Rightarrow e) \Rightarrow v'} \tag{EF-Pat}$$

$$\frac{match(v,\,f_1) \Rightarrow v'}{match(v,\,f_1 \mid f_2) \Rightarrow v'} \tag{EF-Alt1}$$

$$\frac{\neg(match(v,\,f_1)) \qquad match(v,\,f_2) \Rightarrow v'}{match(v,\,f_1 \mid f_2) \Rightarrow v'} \tag{EF-Alt2}$$

Matching ($match(v,\,p) \Rightarrow \Gamma$)

$$\frac{\vdash v \in S \qquad S <: T}{match(v,\,x : T) \Rightarrow x = v} \tag{EP-Var}$$

$$\frac{match(v_i,\,p_i) \Rightarrow \sigma_i \qquad \text{the } \sigma_i \text{ have disjoint domains}}{\begin{array}{c} match([\,l_1 = v_1,\,\ldots,\,l_m = v_m,\,\ldots,\,l_n = v_n\,],\,[\,l_1 = p_1,\,\ldots,\,l_m = p_m\,]) \\ \Rightarrow \sigma_1 + \ldots + \sigma_m \end{array}} \tag{EP-Rcd}$$

$$\frac{match(v,\,p_1) \Rightarrow \sigma_1 \qquad match(v,\,p_2) \Rightarrow \sigma_2 \qquad \sigma_1 \text{ and } \sigma_2 \text{ have disjoint domains}}{match(v,\,p_1 \,\#\, p_2) \Rightarrow \sigma_1 + \sigma_2} \tag{EP-Concat}$$

$$\frac{match(v,\,f) \Rightarrow v'}{match(v,\,x \;\mathsf{as}\; f) \Rightarrow x = v'} \tag{EP-As}$$

3.9 Properties of Evaluation

3.9.1 Fact: If v is a value and $\vdash v \in V$, then V is a simple type. □

3.9.2 Theorem [Subject reduction]:

1. If $e \Downarrow v$
   $\vdash e \in Q$,
   then $\vdash v \in V$
   $V <: Q$.
2. If $match(v,\,f) \Rightarrow v'$
   $\vdash f \in U \to V$
   $\vdash v \in W$
   $W <: U$,
   then $\vdash v' \in X$
   $X <: V$.
3. If $match(v,\,p) \Rightarrow \sigma$
   $\vdash v \in W$
   $\vdash p \in U \Rightarrow \Sigma$
   $W <: U$,
   then $\Sigma \models \sigma$. □

Proof: By simultaneous induction on evaluation derivations.

1. Straightforward, using part (2) of the induction hypothesis for the interesting case (E-Case).
2. Consider the final rule in the given derivation.

Case EF-PAT: $f = p \Rightarrow e$
$match(v,\ p) \Rightarrow \sigma$
$\sigma(e) \Downarrow v'$

From $\vdash f \in U{\to}V$, we know $\vdash p \in U \Rightarrow \Sigma$ and $\Sigma \vdash e \in V$. By part (3) of the induction hypothesis, $\Sigma \models \sigma$. By Lemma 3.7.5, $\vdash \sigma(e) \in V$. Now, by the induction hypothesis, $\vdash v' \in X$ and $X <: V$, as required.

Case EF-ALT1: $f = f_1 \ |\ f_2$
$match(v,\ f_1) \Rightarrow v'$

From rule TF-ALT, we see that $\vdash f_1 \in U_1{\to}V_1$ and $\vdash f_2 \in U_2{\to}V_2$, with $U = U_1 \vee U_2$ and $V = V_1 \vee V_2$. The induction hypothesis yields $\vdash v' \in X$ with $X <: V_1$, from which the result follows immediately by S-UNION.

Case EF-ALT2: $f = f_1 \ |\ f_2$
$\neg(match(v,\ f_1))$
$match(v,\ f_2) \Rightarrow v'$

Similar.

3. Consider the final rule in the given derivation.

Case EP-VAR: $p = (x : T)$
$\vdash v \in S$
$S <: T$
$\sigma = (x = v)$
$\Sigma = (x : T)$

Immediate.

Case EP-RCD: $v = [\,l_1 = v_1,\ \ldots,\ l_m = v_m,\ \ldots,\ l_n = v_n\,]$
$p = [\,l_1 = p_1,\ \ldots,\ l_m = p_m\,]$
$match(v_i,\ p_i) \Rightarrow \sigma_i$
$\sigma = \sigma_1 + \ldots + \sigma + m$

From T-RCD, we have $W = [\,l_1 : W_1\,] \times \ldots \times [\,l_n : W_n\,]$ and $\vdash v_i \in W_i$ for each i. Similarly, by TP-RCD, we have $U = [\,l_1 : U_1\,] \times \ldots \times [\,l_m : W_m\,]$ with $\vdash p_i \in U_i \Rightarrow \Sigma_i$. Finally, by Corollary 3.4.12, we see that $W_i <: U_i$. Now, by the induction hypothesis, $\Sigma_i \models \sigma_i$ for each i. But this means that $\Sigma \models \sigma$, as required.

Case EP-CONCAT: $p = p_1 \# p_2$
$match(v,\ p_1) \Rightarrow \sigma_1 \qquad match(v,\ p_2) \Rightarrow \sigma_2$
σ_1 and σ_2 have disjoint domains
$\sigma = \sigma_1 + \sigma + 2$

By TP-CONCAT, we have $U = [\,k_1 : S_1, \ldots, k_m : S_m, l_1 : T_1, \ldots, l_n : T_n\,]$ with $\vdash p_1 \in [\,k_1 : S_1, \ldots, k_m : S_m\,] \Rightarrow \Sigma_1$ and $\vdash p_2 \in [\,l_1 : T_1, \ldots, l_n : T_n\,] \Rightarrow \Sigma_2$. Since $U <: [\,k_1 : S_1, \ldots, k_m : S_m\,]$ and $U <: [\,l_1 : T_1, \ldots, l_n : T_n\,]$ by the subtyping laws, transitivity of subtyping gives us $W <: [\,k_1 : S_1, \ldots, k_m : S_m\,]$ and $W <: [\,l_1 : T_1, \ldots, l_n : T_n\,]$. Now, by the induction hypothesis, $\Sigma_1 \models \sigma_1$ and $\Sigma_2 \models \sigma_2$. But this means that $\Sigma \models \sigma$, as required.

Case EP-AS: $p = x \Rightarrow f$
$match(v,\ f) \Rightarrow v'$
$\sigma = (x = v')$

By TP-AS, $\vdash f \in U{\to}V$ and $\Sigma = (x : V)$. By part (2) of the induction hypothesis, $\vdash v' \in X$ for some $X <: V$. So $(x = v') \models (x : V)$ by the definition of satisfaction. □

3.9.3 Theorem [Safety]:

1. If $\vdash e \in T$, then $e \Downarrow v$ for some v. (That is, the evaluation of a closed, well-typed expression cannot lead to a match-failure or otherwise "get stuck.")

2. If $\vdash f \in S \rightarrow T$ and $\vdash v \in R <: S$, then $match(v, f) \Rightarrow v'$ with $\vdash v' \in T' <: T$.

3. If $\vdash p \in U \Rightarrow \Gamma'$ and $\vdash v \in S <: U$, then $match(v, p) \Rightarrow \sigma$ with $\Gamma' \models \sigma$. □

Proof: Straightforward induction on derivations. □

4 Conclusions

We have described a type system that may be of use in checking programs or queries that apply to semistructured data. Unlike other approaches to the problem, it is a "relaxed" version of a conventional system that can handle the kinds of irregular types that occur in semistructurd data.

Although we have established the basic properties of the type system, a good deal of work remains to be done. First, there are some extensions that we do not see as problematic. These include:

- Both strict and relaxed set-union operations. (In the former case the two types are constrained to be equivalent.) Similarly, one can imagine strict and relaxed case expressions.
- Equality. Both "absolute" equality and "equality at type T" fit with this scheme.
- A $\perp$ ("bottom") type – the null-ary case of union types. An immediate application is in the typing rule T-SET for set formation, where we can remove the side condition $n \geq 1$ to allow formation of the empty set: $\{\emptyset\} \in \perp$.
- Additional base types such as booleans and operations such as set filtering.
- A $\top$ ("top") type. Such a type would be completely dynamic and would be analyzed by typecase expressions. One could also add type inspection primitives along the lines described for Amber [Carrk]
- An "otherwise" or "fall-through" branch in case expressions.

A number of more significant problems also remain to be addressed.

- **Complexity**. The obvious method of checking whether two types are equivalent or whether one is a subtype of the other involves first reducing both to disjunctive normal form. As we have observed, this process may be exponential in the size of the two type expressions. We conjecture that equivalence (and subtyping) can be checked faster, but we have not been able to show this.

  Even if these problems turn out to be intractable in general, it does not necessarily mean that this approach to typing semistructured data is pointless. Type inference in ML, for example, is known to be exponential [KTU94], yet the forms of ML programs that are the cause of this complexity never occur in practice. Here, it may be the case that types that types that only have "small" differences will not give rise to expensive transformations.
- **Recursive types**. The proof of the decidability of subtyping (3.4.10) works by induction on the derivation tree of a type, which is closely related to the structure of the type. We do not know whether the same result holds in the presence of recursive types.
- **Relationship with other typing schemes.** There may be some relationship between the typing scheme proposed here and those mentioned earlier [NAM97, Ali99] that work by inferring structure from semi-structured data. Simulation, for example, gives rise to something like a subtyping relationship [BDFS97]; but it is not clear what would give rise to union types.

- **Applications**. Finally, we would like to think that a system like this could be of practical benefit. We mentioned that there is a group of biological data formats that are all derived from a common basic format. We should also mention that the pattern matching constructs introduced in section 2.2, independently of any typing issues, might be used to augment other query languages such as XML-QL [DFF+] that exploit pattern matching.

5 Acknowledgements

The idea of using untagged unions to type semistructured data started when Peter Buneman was on a visiting research fellowship provided by the Japanese Society for the Promotion of Science. He is grateful to Atsushi Ohori for stimulating discussions. Benjamin Pierce is supported by the National Science Foundation under CAREER grant CCR-9701826.

An implementation project and meticulous reading by Davor Obradovic helped us correct several flaws in an earlier version of this paper.

References

[AH87] Serge Abiteboul and Richard Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[Ali99] Alin Deutsch and Mary Fernandez and Dan Suciu. Storing semistructured data with STORED. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1999.

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *Journal on Digital Libraries*, 1(1), 1996.

[BDCd95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, June 1995.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proc. ICDT*, 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *ACM-SIGMOD*, pages 505–516, 1996.

[BLS+94] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, September 1995.

[Carrk] L. Cardelli. Amber. In B. Robinet G. Cousineau, P.L. Curien, editor, *Combinators and Functional programming languages*, page 1986. Springer-Verlag, New-York.

[CM94] M. Consens and T. Milo. Optimizing queries on files. In *Proc. ACM Sigmod, Minneapolis*, 1994.

[Dam94] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 687–706. Springer-Verlag, April 1994.

[DCdP96] M. Dezani-Ciancaglini, U. de'Liguoro, and A. Piperno. Filter models for conjunctive-disjunctive λ-calculi. *Theoretical Computer Science*, 170(1-2):83–128, December 1996.

[DFF+] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Xml-ql: A query language for xml. `http://www.w3.org/TR/NOTE-xml-ql`.

[Hay91] Susumu Hayashi. Singleton, union and intersection types for program extraction. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 701–730. Springer-Verlag, September 1991. Full version in *Information and Computation*, 109(1/2):174-210, 1994.

[KTU94] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, March 1994.

[NAM97] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. In *Proceedings of the Workshop on Management of Semi-structured Data*, 1997. Available from `http://www.research.att.com/~suciu/workshop-papers.html`.

[Pie91] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.