Regular Expression Pattern Matching for XML

Haruo Hosoya Benjamin Pierce

Department of Computer and Information Science
University of Pennsylvania
{hahosoya,bcpierce}@cis.upenn.edu

Abstract

We propose *regular expression pattern matching* as a core feature for programming languages for manipulating XML (and similar tree-structured data formats). We extend conventional pattern-matching facilities with regular expression operators such as repetition (*), alternation (|), etc., that can match arbitrarily long *sequences* of subtrees, allowing a compact pattern to extract data from the middle of a complex sequence. We show how to check standard notions of exhaustiveness and redundancy for these patterns.

Regular expression patterns are intended to be used in languages whose type systems are also based on the *regular expression types*. To avoid excessive type annotations, we develop a type inference scheme that propagates type constraints to pattern variables from the surrounding context. The type inference algorithm translates types and patterns into regular tree automata and then works in terms of standard closure operations (union, intersection, and difference) on tree automata. The main technical challenge is dealing with the interaction of repetition and alternation patterns with the *first-match* policy, which gives rise to subtleties concerning both the termination and the precision of the analysis. We address these issues by introducing a data structure representing closure operations lazily.

1 Introduction

XML [XML] is a simple format for tree-structured data. As its popularity increases, a need is emerging for better programming language support for XML processing—in particular, for (1) static analyses capable of guaranteeing that generated trees conform to an appropriate Document Type Definition (DTD) [XML] or to a schema in a richer language such as XML-Schema [XS00], DSD [KMS], or Relax [Rel]; and (2) convenient programming constructs for tree manipulation.

In previous work [HVP00], we proposed *regular expression types* as a basis for static typechecking in a language for processing XML. Regular expression types capture (and generalize) the regular expression notations commonly found in schema languages for XML, and support a natural "semantic" notion of subtyping. We argued that this flexibility was necessary to support smooth evolution of XML-based systems and showed that subtype checking, though exponential in general (it reduces to checking language inclusion between tree automata), can be computed with acceptable efficiency for a range of practical examples.

In the present paper, we pursue the second question—developing convenient programming constructs for tree manipulation in a statically typed setting. We propose *regular expression pattern matching* for this purpose.

Regular expression pattern matching is similar in spirit to the pattern matching facilities found in languages of the ML family [BMS80, MTH90, LVD+96, etc.]. Its extra expressiveness comes from the use of regular expression types to dynamically match values. We illustrate this by an example.

The following declarations introduce a collection of regular expression types describing records in a simple address database.

```
type Person = person[Name,Email*,Tel?]
type Name   = name[String]
type Email  = email[String]
type Tel    = tel[String]
```

Type constructors of the form `label[...]` classify tree nodes with the label `label` (i.e., XML structures of the form `<label>...</label>`). Thus, the inhabitants of the types `Name`, `Email`, and `Tel` are all strings with an appropriate identifying label. Type constructors of the form `T*` denote a sequence of arbitrarily many `T`s, while `T?` denotes an optional `T`. Thus, the inhabitants of the type `Person` are nodes labeled `person` whose content is a sequence consisting of a `name`, zero or more `email` addresses, and an optional telephone number.

Using these types, we can write a regular expression pattern match that, given a value `p` of type `Person`, checks whether `p` contains a `tel` field, and if so, extracts the contents of `name` and `tel`.

```
match p with
   person[name[n], Email*, tel[t]]
     → (* do some stuff involving n and t *)
 | person[p]
     → (* do other stuff *)
```

The first case of the `match` expression matches a node labeled `person` whose content is a sequence of a `name`, zero or more `email`s, and a `tel`. In this case, we bind the variable

POPL '01 1/01 London, UK

n to the name's content and t to the tel's content. The second case matches a label person with any content and binds p to the content. The second case is invoked only when the first case fails, i.e., when there is no Tel component. Note how the first pattern uses the regular expression type Email* to "jump over" an arbitrary-length sequence and extract the tel node following it. This style of matching (which goes beyond ML's capabilities) is often useful in XML processing, since XML data structures often contain sequences where repetitive, optional, and fixed parts are mixed together; regular expression pattern matching allows direct access to the parts of such sequences.

We concentrate in this paper on pattern matching with a "single-match" semantics, which yields just one binding for a given pattern match. We also follow ML in adopting a "first-match" policy, which allows ambiguous patterns and gives higher priority to patterns appearing earlier. A different alternative that is arguably more natural in the setting of query languages and document processing languages [DFF+, AQM+97, CS98, CG00a, NS00, NS98, Mur97] is an "all-matches" style, where each pattern match yields a *set* of bindings. We will compare the two styles at several points in what follows.

To support regular expression pattern matching in a statically typed programming language, it is important that the compiler be able to infer the types of most variable bindings in patterns (otherwise, the type annotations tend to become quite heavy). We propose a type inference scheme that automatically computes types for pattern variables. The type inference scheme is "local" in the sense that it focuses only on pattern matches; it takes a pattern match and a type for the values being matched against, and propagates the type constraints through the patterns to the pattern variables. For example, in the pattern match above, given the input type Person, type inference computes the type String for the variables n and t and the type (Name,Email*) for the variable p. The intuition behind the type for p is that, since all persons with tel are captured by the first pattern, only persons with no tel can be matched by the second pattern.

Our type inference algorithm represents both types and patterns in the form of regular tree automata and propagates type information through patterns in a top-down manner (i.e., it starts with a given type and pattern, calculates types for the immediate substructures of the pattern, and repeats this recursively). The technical difficulties in the development of the algorithm arise from the interaction between the first-match policy and the repetition operator. The first-match policy implies that, in order to maintain the precision of our analysis, we need to be able to reason about the types of values that did *not* match preceding patterns. To this end, we exploit the closure properties of tree automata—in particular, the *(language-)difference* operation. However, since repetition patterns are translated to tree automata whose state transition functions contain loops, the algorithm requires some care to ensure termination of the algorithm. A naively constructed algorithm might infinitely traverse the patterns; more seriously, if the algorithm uses the closure operations each time it encounters the same state, an unbounded number of types may be propagated to the same state (this is discussed further in Section 4.2). We address this problem by introducing a data structure representing closure operations lazily. As a result, we achieve *exact* type inference: it predicts a value for a bound variable if and only if the variable can actually be bound to this value as a result of a successful match of a value from the input type. Previous papers on type inference for pattern matching have considered either recursion [MS99, PV00, Mur97] or the first-matching policy [WC94, PS90], but, as far as we know, no papers have treated both.

In summary, the main contributions of this work are: (1) the motivation and design of regular expression pattern matching; and (2) the algorithm for propagating types to pattern variables from the surrounding context.

The rest of the paper is organized as follows. In the following section, we illustrate regular expression pattern matching by several examples. Section 3 gives basic definitions of types and patterns and sketches the translation from the user-level external syntax to the tree-automata-based internal representation. Section 4 develops the type inference algorithm and proves its correctness. Section 5 discusses the relationship of our work with other work. Section 6 concludes and suggests some possible directions for future research. Appendix A gives some technical details omitted from the earlier discussion of the closure operations. For brevity, proofs are omitted in this summary. They can be found in an expanded version, available electronically [HP00a].

We have used regular expression pattern matching (and regular expression types) in the design of a statically typed XML processing language called XDuce ("transduce") [HP00b]. Interested readers are invited to visit the XDuce home page

```
http://www.cis.upenn.edu/~hahosoya/xduce
```

for more information on the language as a whole.

2 Examples

We give a series of examples motivating our design of pattern matching and illustrating the associated algorithmic problems.

2.1 Regular Expression Types

Each type in our language denotes a set of sequences. Types like String and tel[String] denote singleton sequences; the type Tel* denotes sequences formed by repeating the singleton sequence Tel any finite number of times. So each element of the type person[Tel*] is a singleton sequence labeled with person, containing an arbitrary-length sequence of Tels. If S and T are types, then the type S,T denotes all the sequences formed by concatenating a sequence from S and a sequence from T. The comma operator is associative: the types (Name,Tel*),Addr and Name,(Tel*,Addr) have exactly the same set of elements. (Comma is *not* commutative, however: we consider only ordered sequences.) As the unit element for the comma operator, we have the *empty* sequence type, written (). Thus, Name,() and (),Name are equivalent to Name.

The *subtype* relation between two types is simply inclusion between the sets of sequences that they denote. (See Section 3.3 for a formal presentation of this definition.) For example, (Name*,Tel*) is a subtype of (Name|Tel)* since the first one is more restrictive than the second. That is, Names must appear before any Tel in the first type, while Names and Tels can appear in any order in the second type.

2.2 Regular Expression Pattern Matching

As in ML, a regular expression pattern match consists of one or more clauses, each of which is a pair of a pattern and a body. The pattern describes the shape of input values that we want to identify, and may contain bound variables for extracting subcomponents of the input value. The body is an expression in some term language (whose details we do not need to be precise about, for purposes of this paper) that is executed when a match against the pattern succeeds.

To introduce the notation, consider the following simple pattern match expression, which analyzes a value of type `Person`.

```
match p with
    person[name[n], tel[t]]
      → ...
  | person[name[n], rest]
      → ...
```

The first case matches a label `person` whose content is a sequence of a `name` node and a `tel` node. It binds the variable `n` to the `name`'s content and `t` to the `name`'s content while evaluating the body. The second case is similar except that it binds the variable `rest` to the (possibly empty) sequence follows the `name` node.

Patterns can contain regular expression types. For example, the following pattern match contains the type `Email*`.

```
match p with
    person[name[n], e as Email*, tel[t]]
      → ...
  | person[name[n], e as Email*]
      → ...
```

This example is similar to the previous one except that the variable `e` is bound to the intermediate sequence of zero or more `emails` between `name` and `tel`. (In general, an "as" pattern "`x as P`" performs matching with `P` as well as binding `x` to the whole sequence that matches `P`. Notice also that we treat types in the same category as patterns.) The use of the repetition operator `*` yields an iterative behavior during pattern matching. That is, when the pattern matcher looks at the pattern `(e as Email*)`, no hint is available about how many `emails` there are. Therefore the matcher must walk through the input value until it finds the end of the chain of `emails`. This matching of *arbitrary* length sequences is beyond ML pattern matching, and is often quite useful in programming with XML. For example, the pattern above is substantially more compact than explicitly writing a recursive function that traverses the sequence, as we would need to do if only ML-style matching of fixed-length sequences were supported.

The usefulness of matching against regular expression types is yet more evident in the following complex pattern, which extracts the subcomponents of an HTML table.

```
match t with
    table[cap as Caption?,
          col as (Col*|Colgroup*),
          hd as Thead,
          ft as Tfoot?,
          bd as (Tbody+|Tr+)]
      → ...
```

An HTML table consists of several optional fields (`Caption?` and `Tfoot?`) and repetitive fields (`Col*`, `Colgroup*`, `Tbody+`, and `Tr+`). (We assume the types `Caption`, `Col`, etc., to be defined elsewhere.) Again, by matching against regular expression types, we can directly pick out each subcomponent, whose position in the input sequence is statically unknown. Imagine equivalent code written only with simpler ML-like pattern matching.

2.3 Ambiguous Patterns

Regular expression pattern matches can have two kinds of ambiguity.

The first kind of ambiguity occurs when multiple patterns match the same input value. For example, the patterns in the first example above are ambiguous, since any value that matches the first pattern also matches the second pattern. In such a case, we simply take the first matching pattern ("first-match policy"). The reason why we take this policy rather than simply disallowing ambiguity is the same as in ML: it makes it easy to write a "default case" at the end of a pattern match, whereas restricting to non-ambiguous sets of patterns would force us to write a cumbersome final pattern explicitly matching the "negation" of the other cases.

The second form of ambiguity occurs when a single pattern can match a given value in different ways, giving rise to different bindings for the pattern variables. This possibility is intrinsic to regular expression pattern matching. For example, in the pattern

```
match e with
    e1 as Email*, e2 as Email*
      → ...
```

which splits a sequence of `emails` into two, it is ambiguous how many `emails` the variable `e1` should take. We resolve this ambiguity by adopting a "longest match" policy where patterns appearing earlier have higher priority (as in most implementations of regular expression pattern matching on strings). In the example, `e1` is bound to the whole input sequence, `e2` to the empty sequence.

Again, an alternative design choice would be to disallow such ambiguity. However, the longest-match policy can make patterns more concise. Consider the contents of an HTML `description`, which is a sequence of type `(Dt|Dd)*`, where `Dt` (term) and `Dd` (description) are defined as `dt[...]` and `dd[...]`, respectively (the content types ... are not important here). Suppose we want to format this sequence in such a way that each term is associated with all the following descriptions before the next term (if any). We may write an iteration for scanning the sequence where, at each step, the following pattern match analyzes cases on the current sequence.

```
match l with
    dt[t], d as Dd*, rest
      → (* display term t with d, and do rest *)
  | ()
      → (* finish *)
```

Here, the first case matches a sequence beginning with `dt`, where we extract the content of the `dt` and take the following `dd`s as many as possible, using the longest match. Note that, without the longest match, it is ambiguous how many `dd`s are taken by each of the consecutive patterns `(d as Dd*)` and `rest`. If we rewrite this pattern to an unambiguous one, the variable `rest` must be restricted not to match a sequence that begins with `dd`, resulting in a somewhat more cumbersome pattern:

```
dt[t], d as Dd*, rest as ((Dt,(Dd|Dt)*) | ())
```

The longest-match and first-match policies turn out to fit cleanly together in the same framework, as we shall see in Section 3.1.

2.4 Exhaustiveness and Redundancy Checks

We support the usual checks for exhaustiveness and redundancy of pattern matches. For these checks, we assume that the "domain" type (i.e., the type of the input values) is known from the context. A pattern match is then exhaustive iff every value from the domain type can be matched by at least one of the patterns. Likewise, a clause in a pattern match is redundant iff all the input values that can be matched by the pattern are covered by the preceding patterns.

Although these definitions themselves are the same as usual (cf., for example, [MTH90, page 30]), checking them is somewhat more demanding. Consider the following pattern match, which, given a sequence of persons, finds the first person node with a tel field and extracts the name and tel fields from this person.

```
match p with
    person[Name, Email*]*,
        person[name[n], Email*, tel[t]], rest
      → ...
  | person[Name, Email*]*
      → ...
```

This pattern match is "obviously" exhaustive—the first clause captures the sequences containing *at least one* person with tel and the second captures the sequences containing *no* such person. But how can a machine figure this out? Section 3 describes our approach, which is based on language inclusion between regular tree automata.

2.5 Type Inference

Since we intend regular expression pattern matching to be used in a typed language, we need a mechanism for inferring types for variables in patterns, to avoid excessive type annotations.

The type inference algorithm assumes that a domain type T for the pattern match is given by the context. It then infers an *exact* type U for each pattern variable x. That is, the type U contains *all* and *only* the values v such that x can be bound to v as a result of successful match of the whole pattern against a value from T.

Since the semantics of pattern matching uses a first-match policy, obtaining this degree of precision requires some care. For example, consider the following pattern match, where the domain type is Person = person[Name,Email*,Tel?].

```
match p with
    person[name[n], tel[t]]
      → ...
  | person[name[n], rest]
      → ...
```

We can easily see that n and t should be given String. But what type should be given to the variable rest? At first glance, the answer may appear to be (Email*,Tel?), because the content type of person is (Name,Email*,Tel?), according to the definition of the type Person. But in fact, the precise type for rest is

```
(Email+,Tel?)  | ().
```

To see why, recall that the second case matches values that are *not* matched by the first case. This means that, if a value fails in the first case, the name in the value is not immediately followed by a tel. Therefore what follows after the name should be either one or more emails or nothing at all.

How do we calculate this type? The trick is to calculate a set-difference between types. In the above example, the type of the values that are not matched by the first case is computed by the difference between person[Name,Email*,Tel?] and person[Name,Tel], which is person[Name,((Email+,Tel?)|())]. The computation of difference is feasible because types are equivalent to tree automata and tree automata are closed under difference (Section 3.4). The type person[Name,((Email+,Tel?)|())] is then propagated to the rest variable by simply checking the matching of the label person of the type and the pattern, and similarly for the label name. However, we have to carefully propagate types through repetition patterns (*) so that the algorithm terminates. Further, the combination of repetition patterns and choice patterns with the first-match policy requires a delicate construction of the inference algorithm, as we will explain in Section 4.

So far, we have seen inference for "bare" variable patterns (which match any values). The type inference can also compute a type for an "as-ed" variable of the form (x as P). The inferred type can be more refined than the type that can be formed from the associated pattern P. For example, consider the following pattern match (where the domain type is Person):

```
match p with
    person[Name, x as (Email|Tel)+]
      → ...
  | ...
```

Here, the pattern (Email|Tel)+ imposes the restriction that x can be bound to sequences of length one or more. However, we know from the domain type that at most one tel may follow emails. Thus, type inference computes a more precise type:

```
(Email+,Tel?)  | Tel
```

This refinement is useful since the body of the pattern match may actually depend on the fact that there is at most one tel and type inference alleviates the burden of writing the more verbose annotation.

Our type inference method works only for variables that appear in the tail position in a sequence (we call such variables "tail variables."), for a technical reason explained in Section 3.2. We require each non-tail pattern variable to be supplied with an as pattern, so that we can construct a type for the variable from the supplied pattern in a straightforward way. Fortunately, this limitation turns out not to be too annoying in practice: in our experience, the most common uses of pattern variables are (1) binding the whole contents of a label (as in the examples in Section 2.2), and (2) binding the "rest" of a sequence during iteration over a repetitive sequence (as in the example in Section 2.4). Both of these uses occur in tail positions.

3 Syntax and Semantics

For purposes of formalization (and implementation), it is useful to distinguish two forms of types—*external* and *internal*—and two corresponding forms of patterns. The external form is the one that the user actually reads and writes; all the examples in the previous sections are in this form. Internally, however, the type inference algorithm uses a simpler representation to streamline both the implementation and its accompanying correctness proofs. Below, we give the syntax of each form and the semantics of the internal form, and sketch the translation from external to internal form. Then we define inclusion relations and closure operations on the internal form, and give simple methods for checking exhaustiveness and redundancy of patterns.

3.1 External Form

For brevity, we omit base values (like strings) and the corresponding types and patterns from our formalization.

We assume a countably infinite set of labels, ranged over by `l`, and a countably infinite set of type names, ranged over by `X`. Type expressions are then defined as follows.

`T`	::=	`()`	empty sequence
		`X`	type name
		`l[T]`	label
		`T,T`	concatenation
		`T|T`	union

The bindings of type names are given by a single, global set E of type definitions of the following form.

```
type X = T
```

The body of each definition may mention any of the defined type names (in particular, definitions may be recursive). We regard E as a mapping from type names to their bodies.

We represent the Kleene closure `T*` of a type `T` by a type `X` that is recursively defined as follows.

```
type X = T,X | ()
```

The other regular expression constructors are defined as follows.

```
T+  ≡  T , T*
T?  ≡  T | ()
```

As we have defined them so far, types correspond to arbitrary context-free grammars. Since we instead want types to correspond to regular tree languages, we impose a syntactic restriction, called *well-formedness*, on types. (The reason why we want to restrict attention to regular tree languages is that the inclusion problem for context-free grammars is undecidable [HU79].) Intuitively, well-formedness requires unguarded (i.e., not enclosed by a label) recursive uses of type names to occur only in tail positions. See [HVP00] for the formal definition.

We assume a countably infinite set of pattern names, ranged over by `Y`, and a countably infinite set of variables, ranged over by `x`. Pattern expressions are then defined as follows.

`P`	::=	`x`	bare variable
		`x as P`	as-ed variable
		`()`	empty sequence
		`Y`	pattern name
		`l[P]`	label
		`P,P`	concatenation
		`P|P`	choice

(Notice that the syntax of pattern expressions differs from that of type expressions only in variable patterns.) The bindings of pattern names are given by a single, global, mutually recursive set F of pattern definitions of the following form.

```
pat Y = P
```

For convenience, we assume that F includes all the type definitions in E where the type expressions appearing in E are considered as pattern expressions in the evident way. Pattern expressions must obey the same well-formedness restriction as types. In writing pattern expressions, we use the same abbreviations for regular expression operators (`*`, `+`, and `?`). We write $BV(\mathtt{P})$ for bound variables appearing in `P` and $FN(\mathtt{P})$ for free pattern names appearing `P`.

The longest-match policy mentioned in Section 2.3 actually arises from these abbreviations and the first-match policy. That is, `Email*` is defined as a variable `Y` that is recursively defined as

```
pat Y = Email,Y | ()
```

and, with the first-match policy, the first branch `(Email,Y)` is taken as often as possible, which accounts for the longest-match policy. The same argument is applied to the other operator `+` and `?`. Notice that the order of union clauses in the definitions of the abbreviations matters for the semantics of pattern matching.

We impose an additional syntactic restriction *linearity* on patterns in order to make sure that pattern matching always yields environments with no missing bindings and no multiple bindings for the same variable. For simple ML-style patterns, linearity is just a check that each variable appears in a pattern just once. In the present setting, we need to extend this notion to patterns with choices and recursion. Intuitively, a linear variable must occur exactly once in *each branch* of a choice, and must be unreachable from itself. The formal definition is given in the full version of the paper [HP00a].

Notice that, in the above definition of patterns, nothing prevents us from writing a single pattern that traverses a tree to an arbitrary depth. For example, consider the following recursively defined type for binary trees, with two forms of leaves, `b[]` and `c[]`, and internal nodes labeled `a`,

```
type T = a[T],T | b[] | c[]
```

and the match expression

```
match t with
  P  →  ...
```

where `P` is recursively defined as follows:

```
pat P = a[P],T | a[T],P | x as b[]
```

The pattern `P` matches a tree that has at least one `b[]`, and yields exactly one binding of the variable `x`. Since `P` has the choice of patterns `a[P],T` and `a[T],P` in this order, the first-match policy ensures that the variable `x` is bound to the first `b[]` in depth-first order. Although this "deep" matching is somewhat attractive, we are not sure about its usefulness, because, after obtaining the first `b[]` as above, it is not clear what to do to get the *next* one, or more generally to iterate through all the `b[]`s in the input tree. (By contrast, this sort of deep matching would be more clearly useful if we had chosen the "all-matches" semantics instead.)

3.2 Internal Form

Values, types, and patterns in the external form are labeled trees of arbitrary arity (i.e., any node can have an arbitrary number of children). In the internal form, we consider only binary trees.

The labels l in the internal form are the same as labels in the external form. Internal (binary) *tree values* are defined by the following syntax.

$$\begin{array}{rcll} t & ::= & \epsilon & \text{leaf} \\ & & l(t,t) & \text{label} \end{array}$$

There is an isomorphism between binary trees and sequences of arbitrary-arity trees. That is, ϵ corresponds to the empty sequence, while $l(t,t')$ corresponds to a sequence whose head is a label `l` where t corresponds to the content of `l` and t' corresponds to the remainder of the sequence. For example, from the arbitrary-arity tree

```
person[name[], email[]]
```

we can read off the binary tree

$$\texttt{person}(\texttt{name}(\epsilon, \texttt{email}(\epsilon,\epsilon)),\epsilon),$$

and vice versa.

For types, we begin as before by assuming a countably infinite set of (internal) *type states*, ranged over by X. A binary *tree automaton* M is a finite mapping from states to (internal) *type expressions*, where type expressions T are defined as follows:

$$\begin{array}{rcll} T & ::= & \emptyset & \text{empty set} \\ & & \epsilon & \text{leaf} \\ & & T \mid T & \text{union} \\ & & l(X,X) & \text{label} \end{array}$$

There is an one-to-one correspondence between external and internal types, following the same intuition as for values. For example, the external type `person[name[],email[]*]` corresponds to the internal type $\texttt{person}(X_1, X_0)$ where the states X_1 and X_0 are defined by the corresponding automaton M as follows.

$$\begin{array}{l} M(X_0) = \epsilon \\ M(X_1) = \texttt{name}(X_0, X_2) \\ M(X_2) = \texttt{email}(X_0, X_2) \mid \epsilon \end{array}$$

The formalization of the translation from external types to internal types can be found in [HVP00].

We use the metavariable A to range over both type states and type expressions—jointly called *types*—since it is often convenient to treat them uniformly. The free states $FS(T)$ of a type expression T are the states appearing in T. This is extended to the free states of an automaton M by $FS(M) = \bigcup \{FS(M(X)) \mid X \in dom(M)\}$. We assume that every automaton M satisfies $FS(M) \subseteq dom(M)$.

The semantics of types is given by the acceptance relation $t \in A$ (relative to some tree automaton M), which is read "tree t has type A" or "t is accepted by A." (We usually elide M, to lighten the notation.) The rules for the acceptance relation are as follows.

$$\frac{t \in M(X)}{t \in X} \quad \text{(Acc-St)}$$

$$\epsilon \in \epsilon \quad \text{(Acc-Eps)}$$

$$\frac{t \in T_1}{t \in T_1 \mid T_2} \quad \text{(Acc-Or1)}$$

$$\frac{t \in T_2}{t \in T_1 \mid T_2} \quad \text{(Acc-Or2)}$$

$$\frac{t_1 \in X_1 \qquad t_2 \in X_2}{l(t_1,t_2) \in l(X_1,X_2)} \quad \text{(Acc-Lab)}$$

The definition of patterns is similar to that of types. We assume a countably infinite set of *pattern states*, ranged over by Y. Pattern variables x are the same as in the external form. A *pattern automaton* is a finite mapping from states to (internal) *pattern expressions*, which are defined as follows.

$$\begin{array}{rcll} P & ::= & x : P & \text{variable} \\ & & \emptyset & \text{failure} \\ & & \mathcal{T} & \text{wild-card} \\ & & \epsilon & \text{leaf} \\ & & P \mid P & \text{choice} \\ & & l(Y,Y) & \text{label} \end{array}$$

Note that, in the internal form, we drop bare variable patterns, but introduce the wild-card pattern $\mathcal{T}$. A bare external variable pattern x is encoded as an internal pattern $x : \mathcal{T}$. We use the metavariable D to range over both pattern states and pattern expressions, jointly called *patterns*. We write $BV(P)$ for the variables occurring in P.

The semantics of patterns is given by the matching relation $t \in D \Rightarrow V$ (relative to a pattern automaton N, which we normally elide), where an *environment* V is a finite mapping from variables to trees. This relation is read "tree t is matched by pattern D, yielding environment V." The rules for the matching relation are as follows.

$$\frac{t \in N(Y) \Rightarrow V}{t \in Y \Rightarrow V} \quad \text{(Mat-St)}$$

$$\frac{t \in P \Rightarrow V}{t \in x : P \Rightarrow V \cup \{(x \mapsto t)\}} \quad \text{(Mat-Bind)}$$

$$t \in \mathcal{T} \Rightarrow \emptyset \quad \text{(Mat-Any)}$$

$$\epsilon \in \epsilon \Rightarrow \emptyset \quad \text{(Mat-Eps)}$$

$$\frac{t \in P_1 \Rightarrow V}{t \in P_1 \mid P_2 \Rightarrow V} \quad \text{(Mat-Or1)}$$

$$\frac{t \notin P_1 \qquad t \in P_2 \Rightarrow V}{P_1 \mid P_2 \Rightarrow V} \quad \text{(Mat-Or2)}$$

$$\frac{t_1 \in Y_1 \Rightarrow V_1 \qquad t_2 \in Y_2 \Rightarrow V_2}{l(t_1,t_2) \in l(Y_1,Y_2) \Rightarrow V_1 \cup V_2} \quad \text{(Mat-Lab)}$$

We write $t \in D$ to mean $t \in D \Rightarrow V$ for some V. Also, we write $t \in D \Rightarrow (x \mapsto u)$ when $t \in D \Rightarrow V$ and $V(x) = u$ for some V.

Note that the matching relation is based on a "first-match" policy, as in ML: when a tree matches both branches of a choice pattern, we take the first one. This follows from the fact that the rule MAT-OR2 is applicable only when MAT-OR1 is not.

The correspondence between external patterns and internal patterns is similar to what we have seen for types, except for the treatment of variable patterns. External patterns can contain variable patterns that are not in tail positions. For example, the following pattern contains a non-tail variable pattern labeled with the variable x:

```
(x as (name[],email[])),tel[]
```

Such a pattern cannot be directly translated to an internal pattern because a variable pattern in the internal form can only be bound to a whole subtree, which, in the external form, corresponds to a sub-sequence from some point to the tail. To deal with this discrepancy, we transform each non-tail variable pattern labeled with a variable x to a *pair* of tail variable patterns labeled with new variables $\mathtt{x}_b$ and $\mathtt{x}_e$. The scope of the variable pattern labeled $\mathtt{x}_b$ opens at the beginning of the original x pattern and closes at the tail; the scope of the variable $\mathtt{x}_e$ opens at right after the end of the original x pattern and closed at the tail. Thus, we transform the above pattern to

$$(\mathtt{x}_b \texttt{ as } (\texttt{name[],email[],(}\mathtt{x}_e \texttt{ as tel[]}))).$$

Now, since the newly introduced variable patterns both extend all the way to the end of the sequence, we can translate the whole pattern to the internal pattern

$$\mathtt{x}_b \ : \ \texttt{name}(X_0, X_1)$$

where X_0 and X_1 are defined by the automaton N as follows:

$$\begin{aligned} N(X_0) &= \epsilon \\ N(X_1) &= \texttt{email}(X_0, X_2) \\ N(X_2) &= \mathtt{x}_e : \texttt{tel}(X_0, X_0) \end{aligned}$$

Finally, since the body of the pattern match actually wants to use the original variable x instead of the new variables $\mathtt{x}_b$ and $\mathtt{x}_e$, we insert a bit of extra code, at the beginning of the body, that recovers the original behavior. This extra code "trims off" the sequence assigned to $\mathtt{x}_e$ from the sequence assigned to $\mathtt{x}_b$ (note that the former is a suffix of the latter), and binds the original variable x to the result. The formalization of the translation of patterns can be found in [HP00a].

As we mentioned in Section 2.5, our type inference method cannot compute exact types for non-tail variables. To see why, consider the following pattern with the domain type (T,T?).

```
(x as T), T?
```

This pattern is encoded as

$$\mathtt{x}_b \texttt{ as (T, (}\mathtt{x}_e \texttt{ as T?))}$$

by the translation described above. From the type inference algorithm described later (in Section 4), we will obtain the type (T,T?) for $\mathtt{x}_b$ and T? for $\mathtt{x}_e$. But it is not immediately clear how to obtain the desired type T for x from these two. Naively, it seems we want to compute a type such that each inhabitant t is obtained by taking some tree t_b from (T,T?) and some tree t_e from T? and then cutting off the suffix t_e from t_b. But the type we get by this calculation is

```
(T,T | T | ()),
```

which is bigger than we want. How to infer exact types for non-tail variables is still an open question.

In what follows, all the definitions are implicitly parameterized on the tree automaton and the pattern automaton that define the types and patterns appearing there. In places where we are talking about only a single tree automaton and a single pattern automaton, we simply assume a "global" tree automaton M and a global pattern automaton N. In a few cases, where we are dealing with operations that create *new* types, we will need to talk explicitly about the tree automaton before the creation and the one after.

Finally, whenever we talk about a type A and a pattern D at the same time, we assume either that they are both states or that they are a type expression and a pattern expression.

3.3 Inclusion

We define subtyping as inclusion between the sets of trees in the two given types. Since types are represented as tree automata, subtyping can be decided by an algorithm for checking inclusion of regular tree languages [Sei90]. (The complexity of this decision problem is exponential in the worse case, but algorithms are known that appear to behave well on practical examples [HVP00].) For what follows, we must also define an inclusion relation between types and patterns.

3.3.1 Definition [Subtyping and Inclusion]: A type A is a *subtype* of a type B, written $A <: B$, if $t \in A$ implies $t \in B$ for all t. A type A is *included* in a pattern D, written $A <: D$, if $t \in A$ implies $t \in D$ for all t.

Using the inclusion relation between types and patterns, exhaustiveness of pattern matches can be defined as follows:

3.3.2 Definition [Exhaustiveness]: A pattern match $P_1 \to e_1 \mid \ldots \mid P_n \to e_n$ is *exhaustive* with respect to a type T if

$$T <: P_1 \mid \ldots \mid P_n.$$

3.4 Closure Operations

The check for redundancy of pattern matches uses an intersection operation that takes a type and a pattern as inputs and returns a type representing their intersection:

3.4.1 Definition [Intersection]: A type B is an intersection of a type A and a pattern D, written $A \cap D \Rightarrow B$, if $t \in B$ iff $t \in A$ and $t \in D$.

That is, an intersection of A and D represents the set of trees that are in the type A and also match the pattern D. The redundancy condition can now be expressed as follows:

3.4.2 Definition [Redundancy]: In a pattern match $P_1 \to e_1 \mid \ldots \mid P_n \to e_n$, a pattern P_i is *redundant* with respect to a type T if, for some U,

$$T \cap P_i \Rightarrow U \quad \wedge \quad U <: P_1 \mid \ldots \mid P_{i-1}.$$

That is, a pattern is redundant if it can match only trees already matched by the preceding patterns.

3.4.3 Proposition: For all types A defined under a tree automaton M and patterns D defined under a pattern automaton N, we can effectively calculate a type B defined under a tree automaton $M' \supseteq M$ such that $A \cap D \Rightarrow B$.

The actual algorithm for the intersection operation can be found in Appendix A.

Our type inference algorithm needs to calculate not only intersections of types and patterns, but also differences between types and patterns. If T is the type of the input trees to a pattern $P_1 \mid P_2$, we can infer that the type for the trees that can be matched by P_1 is the intersection of T and P_1, while the type for the trees matched by P_2 is the difference between T and P_1 (i.e., those trees *not* matched by the preceding pattern).

One might now expect to define a difference operation just as we did for the intersection operation and to use these operations in building the type inference algorithm. However, this approach turns out to be problematic. To see why, notice that the above proposition tells that the types returned by the intersection operation may in general contain freshly generated states that were not in the input types or patterns (and the same holds for the difference operation). This becomes an issue in our type inference algorithm. If one step of the algorithm uses the results of the operations in calculating partial results that become inputs to the next step, then it becomes difficult to place a bound on the number of "distinct states" encountered by the algorithm, making it difficult to guarantee termination. We will come back to this point in Section 4.2, where we define a data structure for *representing* intersections and differences of types and patterns without actually calculating them.

4 Type Inference for Pattern Matching

We now consider the problem of inferring types for the variables bound by a pattern, given the domain type of the whole pattern.

4.1 Specification

We assume that a domain type T and a pattern P are given. We want to know the "range" of each variable x—the set of *all* and *only* the trees that x can be bound to as a result of a successful match of a tree from T against P.

4.1.1 Definition [Range]: The *range* of P with respect to T, written $\rho^{T,P}$, is the function mapping each variable x that is reachable from P to the set $\{u \mid \exists t.\ t \in T \wedge t \in P \Rightarrow (x \mapsto u)\}$. A type environment Γ (mapping variables to types) *represents* $\rho^{T,P}$ if $u \in \Gamma(x)$ implies $u \in \rho^{T,P}(x)$, and vice versa, for all x.

Given a type T and a pattern P, the result of type inference should be a type environment Γ representing the range of P with respect to T.

4.2 Highlights of the Algorithm

Given a pattern P and its domain type T, the goal of our type inference algorithm is to obtain a domain type T' for each subpattern P' of P, where T' represents the set of trees that are matched by P' as a result of a successful match of a tree from T against P. Having computed domain types for all subpatterns, the range of P can be obtained as a mapping from each variable x to the union of the domain types for all the variable patterns binding x.

The algorithm proceeds by a top-down propagation of type information through subpatterns. We begin with the domain type T for the whole pattern P. For each immediate subpattern P' of P, we compute its domain type T' from T and P, and recursively apply the same operation to all the subpatterns. For example, consider the labeled type $T = l(X_1, X_2)$ where the global tree automaton M defines

$$M(X_1) = \epsilon \mid l(X_2, X_2)$$
$$M(X_2) = \epsilon$$

and the labeled pattern $P = l(Y_1, Y_2)$ where the pattern automaton N defines

$$N(Y_1) = y_1 : \mathcal{T}$$
$$N(Y_2) = y_2 : \mathcal{T}.$$

We compute a domain type for each subcomponent of P by taking the corresponding subcomponent of T. For the first subcomponent Y_1 of P (which expands to $y_1 : \mathcal{T}$), we obtain the domain type $\epsilon \mid l(X_2, X_2)$ from the first subcomponent of T; similarly, for the second subcomponent Y_1 of P (which expands to $y_2 : \mathcal{T}$), we obtain the domain type ϵ from the second subcomponent of T. From these domain types, we can calculate the type environment $\{y_1 : (\epsilon \mid l(X_2, X_2)), y_2 : \epsilon\}$ as the result of the whole type inference.

Choice patterns need careful treatment because their first-match policy gives rise to complex control flows. Suppose T is a domain type for the choice pattern $P_1 \mid P_2$. We want to obtain a domain type for each of the subpatterns P_1 and P_2. Since the domain type T_1 for P_1 should denote the set of trees from T that are matched by P_1, the type T_1 can be characterized by the intersection of T and P_1. On the other hand, since the domain type T_2 for P_2 should denote the set of trees from T that are *not* matched by the first pattern, the type T_2 can be characterized by the difference between T and P_1.

Since patterns can be recursive, we need to do some extra work to make sure that the propagation described above will always terminate. We apply a standard technique used in many type-related analyses, keeping track of all the inputs to recursive calls to the algorithm and immediately returning when the same input appears for the second time (the intuition being that processing the same input again will not change the final result). The termination of the algorithm then follows from the fact that there are only finitely many possible inputs. Typical uses of this technique can be found in recursive subtyping algorithms [GLP00, HVP00]. In the present setting, since each input to the algorithm is a pair of a type and a pattern, we keep track of such pairs. (It is not sufficient to keep to track of only the *patterns* we have already seen. Suppose that we have already seen a pattern P with a domain type T, but encounter the same pattern P with a different domain type T', in particular, larger than T. Since the pattern P may match more trees than those from T, we need to go through P again with the new domain type T'.)

We need one additional trick, however, to ensure termination. In the propagation of types for choice patterns, if we simply compute the intersection of T and P_1 and the difference between T and P_1, we can create "new" states in

the resulting types (cf. Proposition 3.4.3). This means that we cannot guarantee that there are only finitely many types encountered by the algorithm, which makes it difficult to ensure termination. Instead, our algorithm delays actually calculating intersections and differences by explicitly manipulating expressions containing what we call "compound states," which are a form composed of intersections, differences, and the states appearing in the input type and pattern. Because there are only a finite number of such states, only finitely many compound states can be generated, ensuring termination.

4.3 Preliminaries

A *compound state* $\overline{X}$ consists of a single type state, a set of "intersecting" pattern states, and a set of "subtracting" pattern states. Intuitively, $\overline{X}$ denotes the set of trees that are in the type state and also in each intersecting pattern state, but not in any subtracting pattern state. Formally, a compound state $\overline{X}$ has the form $X \cap \{Y_1 \ldots Y_m\} \backslash \{Z_1 \ldots Z_n\}$, where X is a state and all the Ys and Zs are pattern states. We write $\overline{X} \cap W$ for the compound state $X \cap \{Y_1 \ldots Y_m, W\} \backslash \{Z_1 \ldots Z_n\}$ and $\overline{X} \backslash W$ for $X \cap \{Y_1 \ldots Y_m\} \backslash \{Z_1 \ldots Z_n, W\}$.

Further, we adapt several definitions on types given in Section 3.2 to handle compound states. *Compound type expressions* $\overline{T}$ are just like type expressions except that they contain compound states instead of type states:

$$\begin{array}{rcl} \overline{T} & ::= & \emptyset \\ & & \epsilon \\ & & \overline{T} \mid \overline{T} \\ & & l(\overline{X}, \overline{X}) \end{array}$$

We use the metavariable $\overline{A}$ to range over both compound states and compound type expressions, jointly known as *compound types*. The acceptance relation $t \in \overline{A}$ is defined for compound types just as it is for types, plus the following cases:

$$\frac{t \in \overline{X} \qquad t \in Y}{t \in \overline{X} \cap Y} \qquad \text{(DAcc-Isect)}$$

$$\frac{t \in \overline{X} \qquad t \notin Y}{t \in \overline{X} \backslash Y} \qquad \text{(DAcc-Diff)}$$

Inclusion $\overline{A} <: D$ means that $t \in \overline{A}$ implies $t \in D$ for all t.

Using compound types, we can now define intersection and difference operations that do not introduce new states (unlike the intersection operation defined previously). These operations take a compound type expression and a pattern expression and returns a compound type representing their intersection or difference. The "compound" intersection operation isect is defined as follows.

$$\begin{array}{lcll} \overline{T} \text{ isect } \emptyset & = & \emptyset & \\ \emptyset \text{ isect } P & = & \emptyset & \\ \overline{T} \text{ isect } x : P & = & \overline{T} \text{ isect } P & \\ \overline{T} \text{ isect } \mathcal{T} & = & \overline{T} & \\ \epsilon \text{ isect } \epsilon & = & \epsilon & \\ \epsilon \text{ isect } l(Y_1, Y_2) & = & \emptyset & \\ (\overline{T}_1 \mid \overline{T}_2) \text{ isect } P & = & (\overline{T}_1 \text{ isect } P) \mid (\overline{T}_2 \text{ isect } P) & \\ \overline{T} \text{ isect } (P_1 \mid P_2) & = & (\overline{T} \text{ isect } P_1) \mid (\overline{T} \text{ isect } P_2) & \\ l(\overline{X}_1, \overline{X}_2) \text{ isect } \epsilon & = & \emptyset & \\ l(\overline{X}_1, \overline{X}_2) \text{ isect } l'(Y_1, Y_2) & = & \emptyset & l \neq l' \\ l(\overline{X}_1, \overline{X}_2) \text{ isect } l(Y_1, Y_2) & = & l(\overline{X}_1 \cap Y_1, \overline{X}_2 \cap Y_2) & \end{array}$$

Similarly, the following defines the "compound" difference operation diff.

$$\begin{array}{lcll} \overline{T} \text{ diff } \emptyset & = & \overline{T} & \\ \overline{T} \text{ diff } x : P & = & \overline{T} \text{ diff } P & \\ \emptyset \text{ diff } P & = & \emptyset & \\ \overline{T} \text{ diff } \mathcal{T} & = & \emptyset & \\ \epsilon \text{ diff } \epsilon & = & \emptyset & \\ \epsilon \text{ diff } l(Y_1, Y_2) & = & \epsilon & \\ (\overline{T}_1 \mid \overline{T}_2) \text{ diff } P & = & (\overline{T}_1 \text{ diff } P) \mid (\overline{T}_2 \text{ diff } P) & \\ \overline{T} \text{ diff } (P_1 \mid P_2) & = & (\overline{T} \text{ diff } P_1) \text{ diff } P_2 & \\ l(\overline{X}_1, \overline{X}_2) \text{ diff } \epsilon & = & l(\overline{X}_1, \overline{X}_2) & \\ l(\overline{X}_1, \overline{X}_2) \text{ diff } l'(Y_1, Y_2) & = & l(\overline{X}_1, \overline{X}_2) & l \neq l' \\ l(\overline{X}_1, \overline{X}_2) \text{ diff } l(Y_1, Y_2) & = & l(\overline{X}_1 \backslash Y_1, \overline{X}_2) \mid l(\overline{X}_1, \overline{X}_2 \backslash Y_2) & \end{array}$$

The last case means that if a tree $l(t_1, t_2)$ is in $l(\overline{X}_1, \overline{X}_2)$ but not in $l(Y_1, Y_2)$, then either t_1 is not in Y_1 or t_2 is not in Y_2. Note that the above operations never unfold a state. When the type inference algorithm needs to proceed to the "unfolding" of a compound state, we use the following *unf* function:

$$\begin{array}{lcl} \mathit{unf}(X) & = & M(X) \\ \mathit{unf}(\overline{X} \cap Y) & = & \mathit{unf}(\overline{X}) \text{ isect } N(Y) \\ \mathit{unf}(\overline{X} \backslash Y) & = & \mathit{unf}(\overline{X}) \text{ diff } N(Y) \end{array}$$

The type inference algorithm mainly manipulates compound types, but, for calculating the final results, it uses a "conversion" operation from compound types to their equivalent non-compound types. Formally, a compound type $\overline{A}$ is *convertible* to B, written $\overline{A} \Rightarrow B$, if $t \in \overline{A}$ iff $t \in B$, for all t. The actual algorithm for the conversion operation can be found in Section A.

Finally, we need several definitions on type environments. We write $\{x : T\}$ for the type environment that maps x to T and any other variables to the empty-set type $\emptyset$; the empty environment $\emptyset$ maps all variables to the empty set type $\emptyset$. We define $\Gamma <: \Gamma'$ as $\Gamma(x) <: \Gamma'(x)$ for all variables x, and define $\Gamma \mid \Gamma'$ as $(\Gamma \mid \Gamma')(x) = \Gamma(x) \mid \Gamma'(x)$. We can easily see that $u \in (\Gamma_1 \mid \Gamma_2)(x)$ iff $u \in \Gamma_1(x)$ or $u \in \Gamma_2(x)$.

4.4 Inference Algorithm

The type inference algorithm is presented as a set of syntax-directed rules defining a relation of the form $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Pi'; \Gamma$, where Π ranges over sets of pairs of a compound state and a pattern state, written in the form $(\overline{X} \triangleright Y)$. The algorithm computes, from a (compound) domain type $\overline{A}$ for a pattern D, a type environment Γ that represents the range

of D with respect to $\overline{A}$. To detect termination, the algorithm takes as input the set Π of already-encountered pairs of compound states and pattern states, and returns as output a set Π' containing all the pairs in the input set Π plus the additional pairs encountered during the processing of $\overline{A}$ and D. This output set becomes the input to the next step in the algorithm.

The whole type inference takes as inputs a domain type T and a target pattern P. We assume that T is included in P. (In general, T may not be included in P. But because only trees matched by P contribute to the ranges, we can take the intersection $T \cap P$ for the the starting type, which is automatically included in P.) We start the type inference by calling the general inference relation with $\emptyset \vdash T \triangleright P \Rightarrow \Pi'; \Gamma$. The output Γ is the final result of type inference. (The other output Π' is thrown away.)

We now give the rules for the type inference relation $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Pi'; \Gamma$. For a variable pattern, we add the domain type $\overline{T}$ to the range of x.

$$\frac{\Pi \vdash \overline{T} \triangleright P \Rightarrow \Pi'; \Gamma \qquad \overline{T} \Rightarrow T}{\Pi \vdash \overline{T} \triangleright x : P \Rightarrow \Pi'; (\Gamma \mid \{x : T\})} \quad \text{(INFA-BIND)}$$

The second premise converts the compound type $\overline{T}$ to a non-compound type T so that it can be added to the output type environment.

If the type $\overline{T}$ is less than the empty type (and therefore contains no trees), we return the empty type environment since no successful matches are possible. If the pattern is either a leaf or a wild-card, we return the empty type environment since matching against the pattern will yield no bindings.

$$\frac{\overline{T} <: \emptyset}{\Pi \vdash \overline{T} \triangleright P \Rightarrow \Pi; \emptyset} \quad \text{(INFA-EMP)}$$

$$\Pi \vdash \epsilon \triangleright \epsilon \Rightarrow \Pi; \emptyset \quad \text{(INFA-EPS)}$$

$$\Pi \vdash \overline{T} \triangleright \mathcal{T} \Rightarrow \Pi; \emptyset \quad \text{(INFA-ANY)}$$

For a choice pattern, we compute a domain type for each choice by the compound intersection operation isect and the compound difference operation diff.

$$\frac{\Pi \vdash (\overline{T} \text{ isect } P_1) \triangleright P_1 \Rightarrow \Pi_1; \Gamma_1 \qquad \Pi_1 \vdash (\overline{T} \text{ diff } P_1) \triangleright P_2 \Rightarrow \Pi_2; \Gamma_2}{\Pi \vdash \overline{T} \triangleright P_1 \mid P_2 \Rightarrow \Pi_2; (\Gamma_1 \mid \Gamma_2)} \quad \text{(INFA-OR1)}$$

If the type is a union, we simply generate a subgoal for each component.

$$\frac{\Pi \vdash \overline{T}_1 \triangleright P \Rightarrow \Pi_1; \Gamma_1 \qquad \Pi_1 \vdash \overline{T}_2 \triangleright P \Rightarrow \Pi_2; \Gamma_2}{\Pi \vdash \overline{T}_1 \mid \overline{T}_2 \triangleright P \Rightarrow \Pi_2; (\Gamma_1 \mid \Gamma_2)} \quad \text{(INFA-OR2)}$$

If the type and the pattern are both labels, we propagate each component of the type to the corresponding component of the pattern.

$$\frac{l(X, X') \not<: \emptyset \qquad \Pi \vdash \overline{X} \triangleright Y \Rightarrow \Pi_1; \Gamma_1 \qquad \Pi_1 \vdash \overline{X'} \triangleright Y' \Rightarrow \Pi_2; \Gamma_2}{\Pi \vdash l(\overline{X}, \overline{X'}) \triangleright l(Y, Y') \Rightarrow \Pi_2; (\Gamma_1 \mid \Gamma_2)} \quad \text{(INFA-LAB)}$$

The side-condition $l(\overline{X}, \overline{X'}) \not<: \emptyset$ is necessary for the precision of the type inference. Suppose that $l(\overline{X}, \overline{X'}) <: \emptyset$. Then this means that one of $\overline{X}$ and $\overline{X'}$ is empty, but the other may not necessarily be empty. If such a non-empty type is propagated to the corresponding component of the pattern $l(Y, Y')$, this may augment the range of the pattern. But this augmentation is unnecessary because the type $l(\overline{X}, \overline{X'})$ contains no trees and there can therefore be no successful matches against the pattern.

Finally, we have two rules for type and pattern states.

$$\frac{(\overline{X} \triangleright Y) \in \Pi}{\Pi \vdash \overline{X} \triangleright Y \Rightarrow \Pi; \emptyset} \quad \text{(INFA-ST)}$$

$$\frac{(\overline{X} \triangleright Y) \notin \Pi \qquad \Pi \cup \{(\overline{X} \triangleright Y)\} \vdash unf(\overline{X}) \triangleright N(Y) \Rightarrow \Pi'; \Gamma}{\Pi \vdash \overline{X} \triangleright Y \Rightarrow \Pi'; \Gamma} \quad \text{(INFA-UNF)}$$

That is, if we have already seen the pair $(\overline{X} \triangleright Y)$, we simply return the empty type environment since proceeding to the unfoldings of $\overline{X}$ and Y again will not add anything to the final type environment. If we have not seen the pair, we add it to Π (so that we will be able to tell if we encounter it again) and proceed with the unfoldings.

The worst-case complexity of this algorithm is double-exponential. The rule INFA-UNF may be applied at most as many times as the number of possibilities for the form $(\overline{X} \triangleright Y)$, which is exponential in the size of the input types and patterns. In addition, each time the rule is applied, we may convert compound types to non-compound types the same number of times as variable patterns appear, which is linear. The conversion takes exponential time in the worst case (cf. Appendix A). However, despite these frightening possibilities, in our experience using type inference with several small applications in XDuce, the performance of the algorithm is quite acceptable. The reason is that the patterns used in these applications are "almost" non-recursive (in the case of completely non-recursive patterns, the rule INFA-UNF is applied only a linear number of times in the size of the pattern), and that the optimization techniques used in our implementation (cf., Appendix A) make the conversion operation quick for these examples.

The algorithm is sound, that is, all trees in the predicted range of x are also in the actual range of x.

4.4.1 Theorem [Soundness]: Suppose $\emptyset \vdash A \triangleright D \Rightarrow \Pi; \Gamma$ and $A <: D$. Then, $u \in \Gamma(y)$ implies $u \in \rho^{A,D}(y)$.

The (routine) proof can be found in the full version [HP00a].

Conversely, all trees in the actual range of x are also in the predicted range of x.

4.4.2 Theorem [Completeness]: Suppose $\emptyset \vdash A \triangleright D \Rightarrow \Pi; \Gamma$. Then, $u \in \rho^{A,D}(y)$ implies $u \in \Gamma(y)$.

The key to the proof of completeness lies in characterizing the partial results Π' and Γ in an intermediate state of the algorithm expressed by the form $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Pi'; \Gamma$. That is, when the algorithm is given Π, $\overline{A}$, and D, what will it return in Π' and Γ? To see the intuition, first observe that the algorithm will behave as follows. (1) The algorithm performs type propagation from the pair of the compound type $\overline{A}$ and pattern D. (2) When the algorithm sees a pair of a compound state and a pattern state that is not in Π, it will proceed to their unfoldings and record the pair in Π'. (3) When the algorithm sees a pair that is already in Π, then it will skip this pair. From these, we can expect that Γ contains partial results collected by type propagation from the pair of $\overline{A}$ and D and from the unfoldings of each pair in $\Pi' \setminus \Pi$.

To capture the above intuition precisely, we introduce a *partial validation* relation. Partial validation can be seen as a "checking" version of the type inference algorithm. That is, it performs type propagation similarly to the inference algorithm but, rather than computing a type environment, it checks whether a given type environment is big enough. We check "big enough" because our purpose here is to show completeness (i.e., the predicted range Γ is bigger than the actual range). In addition, partial validation checks the type environment with types and patterns only "shallowly," without unfolding any definitions. This is because we want to characterize each individual pair that the algorithm went through (and avoid wrongly including the pairs that were skipped).

Formally, we first define the relation $\Pi \vdash A \triangleright D \Rightarrow \Gamma$, which is read "the type environment Γ is partially valid under Π w.r.t $\overline{A}$ and D." This relation is defined by the following set of rules:

$$\frac{(\overline{X} \triangleright Y) \in \Pi}{\Pi \vdash \overline{X} \triangleright Y \Rightarrow \Gamma} \quad \text{(Inf-St)}$$

$$\frac{\Pi \vdash \overline{T} \triangleright P \Rightarrow \Gamma \qquad \overline{T} \Rightarrow T \qquad \{x : T\} <: \Gamma}{\Pi \vdash \overline{T} \triangleright x : P \Rightarrow \Gamma} \quad \text{(Inf-Bind)}$$

$$\frac{\overline{T} <: \emptyset}{\Pi \vdash \overline{T} \triangleright P \Rightarrow \Gamma} \quad \text{(Inf-Emp)}$$

$$\Pi \vdash \epsilon \triangleright \epsilon \Rightarrow \Gamma \quad \text{(Inf-Eps)}$$

$$\Pi \vdash \overline{T} \triangleright \mathcal{T} \Rightarrow \Gamma \quad \text{(Inf-Any)}$$

$$\frac{\Pi \vdash (\overline{T} \text{ isect } P_1) \triangleright P_1 \Rightarrow \Gamma \qquad \Pi \vdash (\overline{T} \text{ diff } P_1) \triangleright P_2 \Rightarrow \Gamma}{\Pi \vdash \overline{T} \triangleright P_1 \mid P_2 \Rightarrow \Gamma} \quad \text{(Inf-Or1)}$$

$$\frac{\Pi \vdash \overline{T}_1 \triangleright P \Rightarrow \Gamma \qquad \Pi \vdash \overline{T}_2 \triangleright P \Rightarrow \Gamma}{\Pi \vdash \overline{T}_1 \mid \overline{T}_2 \triangleright P \Rightarrow \Gamma} \quad \text{(Inf-Or2)}$$

$$\frac{l(\overline{X}, \overline{X'}) \neq \emptyset \qquad \Pi \vdash \overline{X} \triangleright Y \Rightarrow \Gamma \qquad \Pi \vdash \overline{X'} \triangleright Y' \Rightarrow \Gamma}{\Pi \vdash l(\overline{X}, \overline{X'}) \triangleright s(Y, Y') \Rightarrow \Gamma} \quad \text{(Inf-Lab)}$$

Each rule is similar to one of the algorithmic rules, with the following differences. First, the validation rules do not return an output Π'. Second, the input Π from the conclusion is directly passed to each premise. Third, the type environment Γ is passed through all of the rules, and, each time we reach a variable pattern, we check that the passed type environment contains sufficient type information for the range at the variable. And fourth, the validation relation has no rule corresponding to Inf-Unf: validation stops at states. We additionally define the relation $\Pi \vdash \Pi' \Rightarrow \Gamma$ (which is read "Γ is partially valid under Π w.r.t Π'") as follows:

$$\frac{\forall (\overline{X} \triangleright Y) \in \Pi'.\ \Pi \vdash \mathit{unf}(\overline{X}) \triangleright N(Y) \Rightarrow \Gamma}{\Pi \vdash \Pi' \Rightarrow \Gamma} \quad \text{(Inf-Cons)}$$

That is, it checks if, for each $(\overline{X} \triangleright Y)$ in Π', the type environment Γ is partially valid under Π w.r.t. the unfoldings of $\overline{X}$ and Y. Finally, Γ is *fully valid* w.r.t $\overline{A}$ and D, written $\overline{A} \triangleright D \Rightarrow \Gamma$, iff both $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Gamma$ and $\Pi \vdash \Pi \Rightarrow \Gamma$ hold for some Π.

The completeness of type inference is now proved in two steps. First, we show that the final result Γ of the algorithm is fully valid w.r.t. A and D (Lemma 4.4.3). Then we show that a type environment Γ that is fully valid w.r.t. A and D is big enough for the actual range of D w.r.t A (Lemma 4.4.4).

4.4.3 Lemma: If $\emptyset \vdash A \triangleright D \Rightarrow \Pi; \Gamma$, then $A \triangleright D \Rightarrow \Gamma$.

4.4.4 Lemma: Suppose $A \triangleright D \Rightarrow \Gamma$. Then, $t \in \rho^{A,D}(y)$ implies $u \in \Gamma(y)$.

In the proof of Lemma 4.4.3, we use the above intuition for the characterization of partial results by partial validation.

Finally, the type inference algorithm constructed as above is guaranteed to terminate.

4.4.5 Theorem [Termination]: For all types A defined under a tree automaton M and patterns D defined under a pattern automaton N, we can effectively calculate a type environment Γ defined under a tree automaton $M' \supseteq M$ such that $\emptyset \vdash A \triangleright D \Rightarrow \Pi; \Gamma$ for some Π.

5 Related Work

Pattern matching is found in a wide variety of languages, and in a variety of styles. One axis for categorization that we have discussed already is how many bindings a pattern match yields. In *all-matches* style, a pattern match yields a set of bindings corresponding to all possible matches. This style is often used in query languages [DFF+, AQM+97, CS98, CG00a, NS00] and document processing languages [NS98, Mur97]. In the *single-match* style, a successful match yields just one binding. This style is usually taken in programming languages [MTH90, LVD+96, JHH+93]. In particular, most functional programming languages allow ambiguous patterns with a first-match policy. Our design follows this tradition.

Another axis is the expressiveness of the underlying "pattern logic." Some query languages and document processing languages use pattern matching mechanisms based on tree automata [NS98, Mur97] or monadic second-order logic (which is equivalent to tree automata) [NS00], and therefore they have a similar expressiveness to our pattern matching. TQL [CG00a] is based on Ambient Logic [CG00b], which appears to be at least as expressive as tree automata. On the

other hand, pattern matching based on *regular path expressions*, popular in query languages for semistructured data [DFF+, AQM+97, CS98], is less expressive than tree automata. In particular, these patterns usually cannot express patterns like "subtrees that contain *exactly* these labels." Both tree automata and regular path expressions can express extraction of data from an arbitrarily nested tree structure (although, with the single-match style, the usefulness of such deep matching is questionable, as we discussed in Section 3.1).

Type inference with tree-automata-based types has been studied both in query languages for semistructured data [MS99, PV00] and in the setting of a document transformation framework [Mur97]. The target languages in these studies have both matching of inputs and reconstruction of outputs (while we consider only matching here). Their pattern matches choose the all-matches style—in particular, an input tree is matched symmetrically against *all* the patterns in a choice pattern. Consequently, these inference algorithms do not involve a difference operation.

Milo, Suciu, and Vianu have studied a typechecking problem for the general framework of *k-pebble tree transducers*, which can capture a wide range of query languages for XML [MSV00]. They use types based on tree automata and build an *inverse type inference* to compute the type for *inputs* from a given type for *outputs* (which is the opposite direction to ours).

Another area related to our type inference method is *set-constraint solving* [AW92] (also known as tree set automata [GTT96]). This framework takes a system of inclusion constraints among types with free variables and checks the satisfiability of the constraints [AW92] or finds a least solution if it exists [GTT96]. Since they allow intersection and difference operations on types, it seems possible to encode our problem into their framework and obtain the solutions by their algorithm. If we used this encoding, we would need to do some work (similar to what we have done here) to prove the existence of least solutions for the sets of constraints we generate, because least solutions do not exist in general in their setting.

Wright and Cartwright incorporate in their *soft type system* a type inference technique for pattern matching [WC94]. Their type system uses a restricted form of union types and their patterns do not involve recursion. (A more precise comparison with our scheme is difficult, since the details of their handling of pattern matching are not presented in their paper.)

Puel and Suárez [PS90] develop a technique for pattern match compilation using what they call *term decomposition*. Although their goal is different from ours, the technique itself resembles our type propagation scheme. Their term decomposition calculates a precise representation of the set of input values that match each pattern, and their calculation of the "values not covered by the preceding patterns" is similar to our use of difference operations. They do not treat recursive patterns.

6 Future Work

Some important extensions are left as future work. The most important is that we would like to support an `Any` type, denoting all sequences of trees, as well as patterns including the `Any` type. `Any` is useful for encoding object-style "extension subtyping" [HVP00], and also for writing patterns that extract parts of sequences (we can use `Any` to match the parts we do not care about). We have not included `Any` in the present treatment, because adding it in a naive way destroys the property of closure under difference (see Appendix A for a related discussion), which makes exact type inference impossible. Another extension is the inference of types for pattern variables in non-tail positions. We have some preliminary ideas for addressing these issues.

Acknowledgments

Our main collaborator in the XDuce project, Jérôme Vouillon, contributed a number of ideas, both in the techniques presented here and in their implementation. We are also grateful to the other XDuce team members (Peter Buneman and Phil Wadler) and to Sanjeev Khanna for productive discussions, to Xavier Leroy and David MacQueen for help with references to related work, to the anonymous POPL'01 referees for comments and suggestions that substantially improved the paper, and to the database group and the programming language club at Penn and the members of Prof. Yonezawa's group at Tokyo for a great working environment.

This work was supported by the Japan Society for the Promotion of Science and the National Science Foundation under NSF Career grant CCR-9701826 and IIS-9977408.

References

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[AW92] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints (extended abstract). In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[BMS80] R. Burstall, David MacQueen, and Donald Sannella. HOPE: an experimental applicative language. In *Proceedings of the 1980 LISP Conference*, pages 136–143, Stanford, California, 1980. Stanford University.

[CG00a] Luca Cardelli and Giorgio Ghelli. A query language for semistructured data based on the ambient logic. Manuscript, April 2000.

[CG00b] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere. Modal logics for mobile ambients. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 365–377, 2000.

[CS98] Sophie Cluet and Jérôme Siméon. Using YAT to build a web server. In *Intl. Workshop on the Web and Databases (WebDB)*, 1998.

[DFF+] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML. `http://www.w3.org/TR/NOTE-xml-ql`.

[GLP00] Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 221–232, 2000.

[GTT96] R. Gilleron, S. Tison, and M. Tommasi. Set constraints and automata. Technical Report it-292, Laboratoire d'Informatique fondamentale de Lille, UniversitLille 1, 1996.

[HP00a] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for XML. Available through `http://www.cis.upenn.edu/~hahosoya/papers/tapat-full.ps`, November 2000.

[HP00b] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, May 2000.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22, September 2000.

[JHH+93] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 93.

[KMS] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. DSD: A schema language for XML. `http://www.brics.dk/DSD/`.

[LVD+96] Xavier Leroy, Jérôme Vouillon, Damien Doligez, et al. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[MS99] Tova Milo and Dan Suciu. Type inference for queries on semistructured data. In *Proceedings of Symposium on Principles of Database Systems*, pages 215–226, Philadelphia, May 1999.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Type-checking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, May 2000.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[Mur97] Makoto Murata. Transformation of documents and schemas by patterns and contextual conditions. In *Principles of Document Processing '96*, volume 1293 of *Lecture Notes in Computer Science*, pages 153–169. Springer-Verlag, 1997.

[NS98] Andreas Neumann and Helmut Seidl. Locating matches of tree patterns in forests. In *18th FSTTCS*, volume 1530 of *LNCS*, pages 134–145, 1998.

[NS00] Frank Neven and Thomas Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 145–156. ACM, 2000.

[PS90] Laurence Puel and Ascánder Suárez. Compiling pattern matching by term decomposition. In *1990 ACM Conference on Lisp and Functional Programming*, pages 272–281, June 1990.

[PV00] Yannis Papakonstantinou and Victor Vianu. DTD Inference for Views of XML Data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 35–46, Dallas, Texas, May 2000.

[Rel] RELAX (REgular LAnguage description for XML). `http://www.xml.gr.jp/relax/`.

[Sei90] Hermut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, June 1990.

[WC94] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. In *In Proceedings of ACM Conference on Lisp and Functional Programming*, pages 250–262, 1994.

[XML] Extensible markup language (XML™). `http://www.w3.org/XML/`.

[XS00] XML Schema Part 0: Primer, W3C Working Draft. `http://www.w3.org/TR/xmlschema-0/`, 2000.

A Closure Algorithms

This appendix defines an algorithm for the conversion operation $\overline{A} \Rightarrow B$ introduced in Section 4.3, which computes a non-compound type equivalent to a given compound type. From this, we can derive, as a special case, an algorithm for the intersection operation introduced in Section 3.4.

The formalization here depends on the definitions given in Section 4.3. We use the definitions of compound type expressions $\overline{T}$, compound states $\overline{X}$, compound types $\overline{A}$, their acceptance relations, the intersection operation isect and difference operation diff for compound types, and the unfolding function *unf* for compound states.

We first give a characterization of the conversion operation $\overline{A} \Rightarrow B$. We define two relations $\Pi \vdash \overline{A} \Rightarrow B$ and $\vdash \Pi$, where Π maps compound states to type states, and we claim that a compound type $\overline{A}$ is convertible to a type B iff both $\Pi \vdash \overline{A} \Rightarrow B$ and $\vdash \Pi$ hold, for some Π. Intuitively, $\Pi \vdash \overline{A} \Rightarrow B$ means that $\overline{A}$ is "immediately" (without unfolding) convertible to B, assuming that the $\overline{X}$ is convertible to Z for each $\overline{X} \mapsto Z$ in Π. Similarly, $\vdash \Pi$ means that the assumptions in Π are consistent—that is, that, for each $\overline{X} \mapsto Z$ in Π, the unfolding of $\overline{X}$ is indeed convertible to the unfolding of Z. The following rules define these relations.

$$\frac{\overline{X} \mapsto Z \in \Pi}{\Pi \vdash \overline{X} \Rightarrow Z} \quad \text{(E-St)}$$

$$\Pi \vdash \emptyset \Rightarrow \emptyset \quad \text{(E-Emp)}$$

$$\Pi \vdash \epsilon \Rightarrow \epsilon \quad \text{(E-Eps)}$$

$$\frac{\Pi \vdash \overline{T}_1 \Rightarrow U_1 \qquad \Pi \vdash \overline{T}_2 \Rightarrow U_2}{\Pi \vdash \overline{T}_1 \mid \overline{T}_2 \Rightarrow U_1 \mid U_2} \quad \text{(E-Or)}$$

$$\frac{\Pi \vdash \overline{X}_1 \Rightarrow Z_1 \qquad \Pi \vdash \overline{X}_2 \Rightarrow Z_2}{\Pi \vdash l(\overline{X}_1, \overline{X}_2) \Rightarrow l(Z_1, Z_2)} \quad \text{(E-Lab)}$$

$$\frac{\forall \overline{X} \mapsto Z \in \Pi.\ \ \Pi \vdash \mathit{unf}(\overline{X}) \Rightarrow M(Z)}{\vdash \Pi} \quad \text{(E-Cons)}$$

The only essential work is done in the rule E-Cons, which computes the unfolding of $\overline{X}$ (which involves consecutive applications of isect and diff operations), and confirms that the result is convertible to the unfolding of Z. The other rules simply replace each compound state with the corresponding state according to the mapping Π.

A.1 Lemma: $\Pi \vdash \overline{A} \Rightarrow B$ and $\vdash \Pi$ iff $\overline{A} \Rightarrow B$.

From the above characterization, we can read off an actual algorithm for the conversion as follows. We start by setting Π to the empty mapping and apply the rules to the given type and pattern in a goal-directed manner. When we reach the rule E-ST, we may not find the compound state $\overline{X}$ in the domain of Π. In this case, we generate a fresh state Z and add a mapping $\overline{X} \mapsto Z$ to Π. We then proceed to convert the unfolding of the compound state $\overline{X}$ to a non-compound type and "back-patch" the result as the unfolding of Z in the tree automaton. This algorithm eventually terminates because only a finite number of compound states can be constructed from the states in the given type and pattern. Also, notice that newly created states appear only in the output type and never in the input type, so there is no danger of trying to unfold one of them before it has been back-patched with its definition.

A.2 Lemma: For all compound types $\overline{A}$ defined with respect to a tree automaton M and pattern automaton N, we can effectively calculate a type B defined under a tree automaton $M' \supseteq M$ such that $\overline{A} \Rightarrow B$.

The algorithm for the conversion operation takes exponential time in the worst case because an exponential number of compound states can be generated from the states in the given type and pattern. However, the algorithm has several opportunities for optimization. Suppose that a compound state has the form $X \cap \{Y_1 \ldots Y_m\} \setminus \{Z_1 \ldots Z_n\}$. We can remove Y_i from the compound state if $X <: Y_i$. Likewise, when $X <: Z_i$, we can replace the whole compound state by a state associated with the empty set type $\emptyset$. Furthermore, when X and Z_i denote disjoint sets, we can remove Z_i from the compound state. (The disjointness can be checked by first calculating the intersection of X and Z_i and then testing the emptiness of the result. Note that if we simply use the conversion operation for calculating the intersection, this introduces circularity. But we can avoid it by specializing the conversion operation for intersection where no subtracting states appear in compound states. See the next paragraph.) Although the inclusion tests in these optimizations are themselves potentially expensive (exponential in the worst-case), these checks turn out usually to be relatively cheap, in our experience [HVP00].

The intersection of a (non-compound) type and a pattern is a special case of the above operation. To compute an intersection of T and P, we can first calculate (T isect P) and then convert the resulting compound type to a non-compound type. Proposition 3.4.3 can be derived as a corollary of Lemma A.2. The worst-case complexity of the intersection operation is quadratic. To see why, observe that, from the definition of isect, the compound types obtained by (A isect D) contain only compound states of the form $X \cap \{Y\}$. Moreover, the unfolding of the compound state $X \cap \{Y\}$ is also a compound type that contains only compound states of this form. Since only a quadratic number of such compound states can be generated from the states in the given type and pattern, the intersection operation completes in quadratic time.

Although the operations we have defined are all we need in our framework, one may wonder which others can be defined. Indeed, it is possible to compute an intersection of two patterns (since types can be treated as a special case of patterns, intersections on other combinations are also possible). On the other hand, we cannot compute differences between patterns and patterns in general. For example, to compute the difference $\mathcal{T} \setminus l(X, X)$, we would need to enumerate all the labels *except* l, an infinite set. For the same reason, neither types nor patterns are closed under negation.