Regular Expression Pattern Matching for XML

HARUO HOSOYA
Research Institute for Mathematical Sciences, Kyoto University
Sakyo-ku, Kyoto 606-8502, Japan
hahosoya@kurims.kyoto-u.ac.jp

BENJAMIN C. PIERCE
Department of Computer and Information Science, University of Pennsylvania
200 S. 33rd Street, Philadelphia, PA 19104-6389, USA
bcpierce@cis.upenn.edu

Abstract

We propose *regular expression pattern matching* as a core feature of programming languages for manipulating XML. We extend conventional pattern-matching facilities (as in ML) 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 with type systems based on *regular expression types*. To avoid excessive type annotations, we develop a type inference scheme that propagates type constraints to pattern variables from the type of input values. 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 these closure operations lazily.

1 Introduction

XML (Bray *et al.*, 2000) 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) (Bray *et al.*, 2000) or to a schema in a richer language such as XML-Schema (Fallside, 2001), DSD (Klarlund *et al.*, 2000), or RELAX NG (Clark & Murata, 2001); and (2) convenient programming constructs for tree manipulation.

In previous work (Hosoya *et al.*, 2000), we proposed *regular expression types* as a basis for static typechecking in a language for processing XML. Regular expression types capture 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 the worst case (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 (Burstall *et al.*, 1980; Milner *et al.*, 1990; Leroy *et al.*, 1996). 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[c]
     →  (* 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 `emails`, and a `tel`. In this case, we bind the variable `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 `c` 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 (Deutsch *et al.*, 1998; Abiteboul *et al.*, 1997; Cluet & Siméon, 1998; Cardelli & Ghelli, 2000; Neven & Schwentick, 2000; Neumann & Seidl, 1998; Murata, 1997) is an "all-matches" style, where each pattern match yields a *set* of bindings. We compare the two styles at several points in what follows.

Regular expression pattern matching by itself is not new. As we will see, the essence of this mechanism is ML pattern matching extended with recursion—an idea that has been proposed in the past (Fähndrich & Boyland, 1997; Queinnec, 1990). The main novelty of our work is the type inference techniques outlined below.

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 `c`. The intuition behind the type for `c` is that, since all `person`s with `tel` are captured by the first pattern, only `person`s 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, their closure under *(language-)difference*. However, since repetition patterns are translated to tree automata whose state transition functions contain loops, it is not so easy to ensure the algorithm to terminate. As we will argue in Section 4.2, a naively constructed algorithm might use the closure operations each time it encounters the same state and, since the state can loop to itself, an unbounded number of types can be propagated to the same state. 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 (Milo & Suciu, 1999; Papakonstantinou & Vianu, 2000; Murata, 1997) or the first-matching policy (Wright & Cartwright, 1994; Puel & Suárez, 1990), but, as far as we know, no papers have treated both.

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. Appendices A, B, and C give some technical details omitted from the earlier discussion.

We have used regular expression pattern matching (and regular expression types) in the design of a statically typed XML processing language called XDuce ("transduce") (Hosoya & Pierce, 2000). Interested readers are invited to visit the XDuce home page

`http://xduce.sourceforge.net`

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

Values in our type system are sequences of labeled values (or base values) and thus representing fragments of XML structures. An example of values is

```
name["Hosoya"],email["hahosoya"],tel["123-456"].
```

Note that a single label containing some other sequence such as

```
person[name["Pierce"],email["bcpierce"]]
```

is also a value.

Each type 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 `Tel`s. 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.[1]) As the unit element for the comma operator, we have the *empty* sequence type, written `()`. Thus, `Name,()`

[1] Several existing schema languages allow commutative operators to describe unordered sequences, e.g., "&"-operator in SGML DTD (Sperberg-McQueen & Burnard, n.d.), "all"-operator in XML Schema (Fallside, 2001), and "interleave"-operator in RELAX NG (Clark & Murata, 2001).

and (),Name are equivalent to Name. A union type S|T denotes the union of the values denoted by S and those denoted by T.

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 (for the purposes of this paper, we do not need to be precise about the term language) 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 tel's content and evaluates the body. The second case is similar except that it binds the variable rest to the (possibly empty) sequence following 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, that is, types can appear anywhere patterns can appear. Though, patterns like name[n]* are not allowed since we also have a usual "linearity" requirement to ensure patterns to yield exactly one binding for each variable. We will discuss the linearity requirement in details in Section 3.1.) 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 iterative behavior enables matching of *arbitrary* length sequences, which is beyond ML pattern matching and often quite useful in programming with XML.

The usefulness of matching against regular expression types is 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.

2.3 Ambiguous Patterns

Although regular expression pattern matching yields just a single binding, we allow ambiguous patterns, which may yield multiple possible bindings. We choose one of those bindings by a priority rule called "first-match" policy. The reason we take this semantics rather than requiring unambiguity is that patterns become more concise, as we will see below.

Regular expression pattern matches can have two kinds of ambiguity.

The first kind of ambiguity occurs when multiple patterns can match the same input value. For example, the patterns in the first example in Section 2.2 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"). We take this policy rather than simply disallowing ambiguity for the same reason as in ML: this 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 "complement" 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. 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 `dl` (description list), 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 as many as possible of the following `dd`s, 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)*) | ())
```

Although the longest-match and first-match policies might look quite different at first glance, they turn out to fit cleanly together in the same framework, as we shall see in Section 3.1. (Specifically, the longest match policy can be derived from the first match policy.)

2.4 Exhaustiveness and Redundancy Checks

We support the usual checks for exhaustiveness and redundancy of pattern matches. For these checks, we assume that a type for input values to the given pattern match is known from the context. A pattern match is then exhaustive if and only if every value from the input 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, page 30 of (Milner *et al.*, 1990)), 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 this be automated? Section 3.3 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 in order to avoid excessive type annotations.

The type inference algorithm is *local*: it assumes that a type `T` for input values to the pattern match is given by the context, and determines types for the pattern variables only from the type `T` and the pattern match itself. The type inference algorithm infers a *locally precise* type `U` for each pattern variable `x`. That is, assuming that all values from `T` may be matched against the pattern match, the type `U` contains *all* and *only* the values that `x` may be bound to.

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 input type is `Person` (which is defined to be `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 type `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 `email`s 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 result type `(Email+,Tel?)|`

`()` is obtained by matching the labels `person` and `name` in the type `person[Name, ((Email+,Tel?)|())]` and the pattern `person[name[n],rest]`. In this particular example, not much difficulty arises. However, in general, we have to propagate types carefully through repetition patterns (`*`) so that the algorithm terminates. Furthermore, the combination of repetition patterns and choice patterns with the first-match policy requires a delicate construction of the inference algorithm. We will explain these issues in detail in Section 4.2.

So far, we have seen inference for "bare" variable patterns (which match any values). Type inference can also compute a type for a variable `x` in a pattern 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 input 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 input type that at most one `tel` may follow `emails`. Thus, an exact type for the variable `x` (which type inference computes) is smaller:

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

This refinement can be useful in alleviating the burden on the user. That is, the body of the pattern match may actually depend on the fact that there is at most one `tel` and, in order to typecheck such a body, the variable `x` must be given the above exact type. Without type inference capable of such refinement, the user would have to explicitly write the above type in the pattern, which would be quite tedious.

Our type inference method works only for variables that appear in the tail position in a sequence, for a technical reason explained in Section 3.2. We require each pattern variable in a non-tail position 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. For example, in the pattern

```
(x as a[y as b[]]), ...
```

we can construct the type `a[b[]]` for the variable `x` from the pattern `a[y as b[]]`. Fortunately, this limitation turns out not to be too annoying in practice: in our experience (of programming in XDuce), 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 the 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.

In this section, we first give the syntax of each form and the semantics of the internal form, and sketch the translation from the external form to the internal form. (Supplementary definitions for the external form are given in Appendix A and a formalization of the translation is in Appendix B.) Then we define inclusion relations and closure operations and give simple methods to check exhaustiveness and redundancy of patterns by using these relations and operations.

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`. *Values* are defined as follows.

$$\texttt{v} \quad ::= \quad \texttt{l}_1\texttt{[v]},\ldots,\texttt{l}_n\texttt{[v]} \quad \text{sequence } (n \geq 0)$$

We write `()` for the empty sequence and write `v,w` for the concatenation of sequences `v` and `w`.

We assume 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, finite 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 (Hopcroft & Ullman, 1979).) Intuitively, well-formedness requires unguarded (i.e., not enclosed by a label) recursive uses of type names to occur only in tail positions. For example, the following is prohibited.

```
type X = a[],X,b[] | ()
```

The formal definition is given in Appendix A.1.

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 pattern
		`()`	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- and as-patterns. Notice also that the `P` in `(x as P)` can have other as-patterns, though the previous examples did not use it.) The bindings of pattern names are given by a single, global, finite, 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 obvious way. Pattern definitions must obey the same well-formedness restriction as type definitions. In writing pattern expressions, we use the same abbreviations for regular expression operators (`*`, `+`, and `?`). We write BV(`P`) for the variables appearing in `P` and FN(`P`) for the pattern names appearing `P`.

The longest-match policy mentioned in Section 2.3 actually arises from these abbreviations and the first-match policy. For example, `Email*` is defined as a pattern name `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 operators `+` and `?`. Notice that the order of union clauses in the definitions of the abbreviations matters for the semantics of pattern matching.

Only with the definition of patterns given so far, the first match may not be definable for some patterns. For example, suppose that we want to match the value `a[]` against the pattern `Y` where:

```
pat Y = Y | a[]
```

Clearly, the second clause `a[]` will match the value. But since the first clause `Y` has higher priority, we should examine this first. Unfolding this pattern name, we see the pattern `(Y|a[])` again and therefore the same argument can be repeated. Thus, for any match of the value against this pattern, we can find another match with higher priority; therefore there is no first match. Notice that this anomaly arises because the unguarded recursive use of `Y` has no pattern in front of it and therefore it recurs without decreasing the size of the input value. In order to ensure that each pattern has the first match for all input values, we impose a syntactic restriction of *no head-recursion*, where any unguarded recursive use of pattern names must be preceded by a pattern that does not match the empty sequence.

Furthermore, we impose an additional syntactic restriction called "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 only once. In the present setting, we need to extend this notion to patterns with choices, as-patterns, and recursion. Informally, our linearity requires that (1) the branches of a concatenation pattern must contain disjoint sets of variables, (2) each branch of a choice pattern must contain exactly the same set of variables, and (3) the inner pattern of an as-pattern must not contain the same variable. For example, the following patterns are illegal.

```
name[x]|email[y]    (x as name[x])    name[x]*
```

(For the last one, it expands to a variable `X` that is defined as `name[x],X|()`, where the pattern `name[x]` is concatenated with `X` whose definition contains `x` again.) The formal definition is given in Appendix A.3.

From now on, we assume that the set F of pattern definitions that we talk about is always well-formed, and contains no head recursion, and is linear in each reachable variable.

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` to one of the `b[]`s. 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

In the external form, values are arbitrary-arity trees (i.e., any node can have an arbitrary number of children), whereas, 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.

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

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. The height h of a tree t is defined as follows.

$$\begin{array}{lcl} h(\epsilon) & = & 1 \\ h(l(t_1,t_2)) & = & max(h(t_1),h(t_2))+1 \end{array}$$

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 type states to (internal) *type expressions*, where type expressions T are defined as follows:

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

Note that this syntax ensures that type names can be used only through labels and labels cannot be nested (as opposed to the external form that allows arbitrary nesting). This restriction is convenient in simplifying the formalization below by obviating cases where both type states and type expressions appear (i.e., comparing a type state and a type expression).

There is a 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 automaton M as follows.

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

The formalization of the translation from external types to internal types can be found in (Hosoya *et al.*, 2000). (Or see Appendix B for the translation of patterns, which is very similar to the translation of types.)

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. We write $\Big|_{i=1,\ldots,n} T_i$ for $T_1 \mid \ldots \mid T_n$. We write $FS(T)$ for the set of states appearing in T. This is extended to the states appearing in 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.

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

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

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

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

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

The definition of patterns is similar to that of types. We assume a countably infinite set of *pattern states*, ranged over by Y, Z, and W. 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.

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

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 (three-place) *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.

$$\frac{t \in N(Y) \Rightarrow V}{t \in Y \Rightarrow V} \qquad \text{(MAT-ST)}$$

$$\frac{t \in P \Rightarrow V \qquad x \notin dom(V)}{t \in x : P \Rightarrow V \cup \{(x \mapsto t)\}} \qquad \text{(MAT-BIND)}$$

$$t \in \mathcal{T} \Rightarrow \emptyset \qquad \text{(MAT-ANY)}$$

$$\epsilon \in \epsilon \Rightarrow \emptyset \qquad \text{(MAT-EPS)}$$

$$\frac{t \in P_1 \Rightarrow V}{t \in P_1 \mid P_2 \Rightarrow V} \qquad \text{(MAT-OR1)}$$

$$\frac{t \notin P_1 \qquad t \in P_2 \Rightarrow V}{t \in P_1 \mid P_2 \Rightarrow V} \qquad \text{(MAT-OR2)}$$

$$\frac{\begin{array}{c} t_1 \in Y_1 \Rightarrow V_1 \qquad t_2 \in Y_2 \Rightarrow V_2 \\ dom(V_1) \cap dom(V_2) = \emptyset \end{array}}{l(t_1, t_2) \in l(Y_1, Y_2) \Rightarrow V_1 \cup V_2} \qquad \text{(MAT-LAB)}$$

We write $t \in D$ to mean $t \in D \Rightarrow V$ 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. [2]

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

[2] Efficient evaluation for pattern matching is a big issue by itself and should be discussed separately from this paper. Though, one may easily image the most naive "backtracking" algorithm (which is actually used in the current XDuce implementation). That is, starting with a given tree and a given pattern, we traverse these two in a top-down way, applying the matching rules from the conclusion to the premises. Note that the rules are deterministic except for a choice pattern (MAT-OR1 and MAT-OR2). We keep encountered choice points in the stack. When we reach the point where no rule can be applied, we backtrack to the previous choice point and try the remaining choices.

patterns can contain variable patterns that are not in tail positions. For example, in the following

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

the pattern `(x as ...)` is not in a tail position. Such a non-tail variable pattern can be bound to a sub-sequence of the input sequence from some point to another point that is not necessarily the tail, whereas 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*. Therefore a non-tail variable pattern cannot be directly translated to an internal pattern. To deal with this discrepancy, we transform each non-tail variable pattern that binds a variable `x` to a *pair* of tail variable patterns that bind new variables $\mathtt{x}_b$ and $\mathtt{x}_e$. The scope of the variable $\mathtt{x}_b$ opens at the same position as the original `x` pattern opens, and closes at the tail; the scope of the variable $\mathtt{x}_e$ opens at right after the original `x` pattern opens, and closes at the tail. Thus, we transform the above pattern to

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

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

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

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

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

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 is given in Appendix B.

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 input type `(T,T?)`.

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

This pattern is encoded as

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

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 in which 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 left for future work.

In what follows, all the definitions are implicitly parameterized on the tree automaton and the pattern automaton that define the types and patterns appearing in the definitions. 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 (Seidl, 1990). (The complexity of this decision problem is exponential in the worse case, but algorithms are known that appear to behave well on practical examples (Hosoya *et al.*, 2000).) 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 by the (three-place) relation $A \cap D \Rightarrow B$, 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 any type A defined w.r.t. a tree automaton M and pattern D defined w.r.t. a pattern automaton N, we can effectively calculate a type B defined w.r.t. 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 C.

Our type inference algorithm needs to calculate not only intersections of types and patterns, but also differences between types and patterns, that is, a type that denotes the set of trees of a given type A that are not matched by a pattern D. We do not define here a difference operation just as we did for the intersection operation. Instead, we will introduce a more complex treatment for these operations ("lazy representation" of closure operations), which is needed to guarantee termination of the type inference algorithm.

4 Type Inference for Pattern Matching

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

4.1 Specification

Suppose that a pattern match

$$\texttt{match v with } P_1 \to \texttt{e}_1 \mid \ldots \mid P_n \to \texttt{e}_n$$

and a type for the input `v` are given. (In this section, we consistently use the internal form of types and patterns.) Let us compose the "target pattern" $P = P_1 \mid \ldots \mid P_n$ from the given pattern match[3] Our interest is then to obtain, from the input type T, the "range" of the pattern P at each variable x (reachable from P)—that is, assuming that all trees from T may be matched against P, the range of x is the set of all and only the trees that x may be bound to. The job of type inference is to obtain a type that represents this range.

[3] Here, each of $P_1, \ldots, P_n$ is linear as specified in the previous section. However, we do not assume the composed pattern $P_1 \mid \ldots \mid P_n$ to be linear—the inference algorithm does not require linearity.

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

$$\{u \mid \text{ there is } t, V \text{ such that } t \in T \text{ and } t \in P \Rightarrow V \text{ with } V(x) = u\}.$$

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.

4.2 Highlights of the Algorithm

The core part of our type inference algorithm is to compute a type T' that represents the "domain" of P' for each subpattern P' (reachable from P)—that is, assuming that all trees from the input type T may be matched against the target pattern P, the domain of P' is the set of all and only the trees that are matched by P'. The algorithm proceeds by a top-down traversal of the target pattern, during which we propagate the type information from the input type and compute a domain type for each subpattern.

As an example, let us consider the following. The input type T is a labeled type $l(X_1, X_2)$ where the global tree automaton M defines

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

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

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

In the beginning, we take the given input type T as a domain type for the target pattern P, since exhaustiveness ensures that all trees from T can be matched by P. Then we compute a domain type for each component of P by taking the corresponding component of T. For the first component 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 component of T; similarly, for the second component Y_1 of P (which expands to $y_2 : \mathcal{T}$), we obtain the domain type ϵ from the second component of T. From these domain types, we obtain the type environment for the result of the type inference: $\{y_1 : (\epsilon \mid l(X_2, X_2)), y_2 : \epsilon\}$. (In general, there may be multiple patterns binding the same variable, e.g., $((x : \epsilon) \mid (x : l(Y, Y)))$, in which case we take the union of the domain types for all the patterns binding the variable.)

Choice patterns need careful treatment because their first-match policy gives rise to complex control flow. Suppose T is a domain type for the choice pattern $P_1 \mid P_2$. We want to obtain domain types T_1 and T_2 for the subpatterns P_1 and P_2, respectively. Since the 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 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. (Note here that since all trees from T can be matched by $P_1 \mid P_2$, those trees not matched by P_1 can be matched by P_2.)

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, which keeps track of all the inputs to recursive calls to the algorithm and immediately returns when the same input appears for the second time (where the intuition is that processing the same input again will not change the final result). The termination of the algorithm then follows from the finiteness of the set of possible inputs. Typical uses of this technique can be found in recursive subtyping algorithms (Gapeyev *et al.*, 2000; Hosoya *et al.*, 2000). 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 may 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 of and differences among the states appearing in the original input type and target 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* $\overline{X}$ is a triple of the form $X \cap \{Y_1 \dots Y_m\} \backslash \{Z_1 \dots Z_n\}$, where X is a type state and all the Ys and Zs are pattern states. Intuitively, $\overline{X}$ denotes the set of trees that are in the type state X and also in each pattern state Y_i, but not in any pattern state Z_j. We write $\overline{X} {\cap} W$ for the compound state $X \cap \{Y_1 \dots Y_m, W\} \backslash \{Z_1 \dots Z_n\}$ and $\overline{X} \backslash W$ for $X \cap \{Y_1 \dots Y_m\} \backslash \{Z_1 \dots Z_n, W\}$.[4]

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

[4] Readers familiar with automata theory might find compound states similar to alternating tree automata (Slutzki, 1985).

type expressions, jointly known as *compound types.* We write $FS(\overline{T})$ for the set of compound states appearing in $\overline{T}$. The acceptance relation $t \in \overline{A}$ is defined for compound types just as it is for types (except that the name of each rule begins with CAcc), plus the following cases:

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

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

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 in Section 3.4). These operations take a compound type expression and a pattern expression and returns a compound type expression that represents their intersection or difference. The "compound" intersection operation isect is defined as follows.

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

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

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

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:

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

The following desirable properties for the isect, the diff, and the *unf* operations can be proved by straightforward induction.

4.3.1 Lemma: For all trees t,

1. $t \in (\overline{T}\ \textsf{isect}\ P)$ iff $t \in \overline{T}$ and $t \in P$;
2. $t \in (\overline{T}\ \textsf{diff}\ P)$ iff $t \in \overline{T}$ and $t \notin P$;
3. $t \in unf(\overline{X})$ iff $t \in \overline{X}$.

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 $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Pi'; \Gamma$. The algorithm computes, from a pattern D and a compound type $\overline{A}$ (which represents the domain of D), a type environment Γ that represents the range of D with respect to $\overline{A}$. Here, Π ranges over sets of pairs of a compound state and a pattern state, written in the form $(\overline{X}\triangleright Y)$. 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 procedure takes as inputs a domain type T and a target pattern P where T is included in P, and starts the main type inference algorithm by calling the general inference relation $\Pi \vdash T \triangleright P \Rightarrow \Pi'; \Gamma$ with $\Pi = \emptyset$. 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 $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Pi'; \Gamma$. We first show the cases where $\overline{A}$ and D are a compound type expression $\overline{T}$ and a pattern expression P. If the target pattern is a variable pattern $x : P$, we add the domain of the pattern P, which is represented by the type $\overline{T}$, to the range of the target pattern at x.

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

The relation $\overline{T} \Rightarrow T$ (defined in Appendix C) converts a compound type $\overline{T}$ to an equivalent non-compound type T. The second premise above uses this relation in order to put the calculated compound type $\overline{T}$ to the output type environment (which maps variables to non-compound types).

If the type $\overline{T}$ is less than the empty set type (and therefore contains no trees), we return the empty type environment since no successful matches are possible. Also, 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.

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

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

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

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

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

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

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

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

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

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 states and pattern states.

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

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

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 C). 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 C) make the conversion operation quick for these examples.

4.5 Correctness of the Algorithm

We now prove the soundness, completeness, and termination properties of our type inference algorithm.

The algorithm is sound, that is, all trees in the predicted range of x are also in the actual range of x. (The definition of "range" is given in Section 4.1.)

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

Proof

We prove the following stronger statement:

> Suppose $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Pi'; \Gamma$ and $\overline{A} <: D$. Then, if $u \in \Gamma(y)$, then $t \in \overline{A}$ and $t \in D \Rightarrow V$ with $V(y) = u$ for some V, t.

The proof proceeds by induction on the derivation of $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Pi'; \Gamma$.

Case InfA-St/InfA-Emp/InfA-Eps**:**

Trivial since $\Gamma = \emptyset$.

Case InfA-Unf**:** $\dfrac{(\overline{X}\triangleright Y) \notin \Pi \qquad \Pi \cup \{(\overline{X}\triangleright Y)\} \vdash \mathit{unf}(\overline{X}) \triangleright N(Y) \Rightarrow \Pi'; \Gamma}{\Pi \vdash \overline{X} \triangleright Y \Rightarrow \Pi'; \Gamma}$

The result follows from the induction hypothesis with Lemma 4.3.1 and Mat-St.

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

From $u \in (\{x : T\} \mid \Gamma)(y)$, either $u \in \Gamma(y)$ or $u \in \{x : T\}(y)$. In the first case, we obtain $u \in P$ by the induction hypothesis. The result follows from MAT-BIND. In the second case, we have $y = x$ and $u \in T$. From the definition of the conversion relation, we obtain $u \in \overline{T}$. From the assumption $\overline{T} <: (x : P)$, we have $u \in x : P \Rightarrow V$ and $V(y) = u$ for some V, as desired.

Case INFA-OR1: $\dfrac{\Pi \vdash (\overline{T} \; \mathsf{isect} \; P_1) \triangleright P_1 \Rightarrow \Pi_1; \Gamma_1 \qquad \Pi_1 \vdash (\overline{T} \; \mathsf{diff} \; P_1) \triangleright P_2 \Rightarrow \Pi_2; \Gamma_2}{\Pi \vdash \overline{T} \triangleright P_1 \mid P_2 \Rightarrow \Pi_2; (\Gamma_1 \mid \Gamma_2)}$

From $u \in (\Gamma_1 \mid \Gamma_2)(y)$, we have two cases.

- $u \in \Gamma_1(y)$. Lemma 4.3.1 implies $(\overline{T} \; \mathsf{isect} \; P_1) <: P_1$, which allows us to use the induction hypothesis; we obtain $t \in (\overline{T} \; \mathsf{isect} \; P_1)$ and $t \in P_1 \Rightarrow V$ with $V(y) = u$ for some V, t. The result follows from Lemma 4.3.1 and MAT-OR1.
- $u \in \Gamma_2(y)$. By Lemma 4.3.1, the assumption $\overline{T} <: P_1 \mid P_2$ implies $(\overline{T} \; \mathsf{diff} \; P_1) <: P_2$. By the induction hypothesis, we obtain $t \in (\overline{T} \; \mathsf{diff} \; P_1)$ and $t \in P_2 V$ with $V(y) = u$ for some V, t. From Lemma 4.3.1, we have $t \notin P_1$. The result follows from MAT-OR2.

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

Similar to the above cases.

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

From the side condition $l(\overline{X}, \overline{X}') \not<: \emptyset$, we have $\overline{X} \not<: \emptyset$ and $\overline{X}' \not<: \emptyset$. To enable the use of the induction hypothesis below, we show that the assumption $l(\overline{X}, \overline{X}') <: l(Y, Y')$ implies $\overline{X} <: Y$ and $\overline{X}' <: Y'$. Take any $v \in \overline{X}$. Since $\overline{X}' \not<: \emptyset$, we can find some $v' \in \overline{X}'$. The assumption implies $l(v, v') \in l(Y, Y')$, and therefore $v \in Y$. We can show $\overline{X}' <: Y'$ similarly.

From $u \in (\Gamma_1 \mid \Gamma_2)(y)$, either $u \in \Gamma_1(y)$ or $u \in \Gamma_2(y)$. In the first case, by the induction hypothesis, $t \in \overline{X}$ and $t \in Y \Rightarrow V$ with $V(y) = u$ for some V, t. Since $\overline{X}' \not<: \emptyset$ and $\overline{X}' <: Y'$, there is t' such that $t' \in \overline{X}'$ and $t' \in Y' \Rightarrow W$. The result follows from CACC-LAB and MAT-LAB. The second case is similar. □

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

4.5.2 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 how to characterize an "intermediate state" expressed by the judgment $\Pi \vdash \overline{T} \triangleright P \Rightarrow \Pi'; \Gamma$. That is, what is the relationship between the intermediate inputs Π, $\overline{T}$, and P and results Π' and Γ? To see the intuition, first observe how the algorithm behaves after receiving the inputs Π, $\overline{T}$, and P: (1) The algorithm first processes the pair of the compound type $\overline{T}$ and pattern P; (2) When the algorithm sees a pair of a compound state and a pattern state that is not in Π, it processes their unfoldings and record the pair in Π'; (3) When the algorithm sees a pair that is already in Π, then it skips the unfoldings of this pair. Therefore, when the algorithm starts with Π, $\overline{T}$, and P and returns with Π' and Γ, it indicates that the algorithm has already processed the pair of $\overline{T}$ and P and the unfoldings of each pair that is in Π' but not in Π, and has collected type information obtained from them in Γ.

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" (rather than "exact") because our purpose here is to show completeness (i.e., the predicted range Γ is at least as big as 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 processed (and avoid wrongly including the pairs that were skipped).

Formally, we first define the relation $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Gamma$, 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:

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

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

$$\frac{\overline{T} <: \emptyset}{\Pi \vdash \overline{T} \triangleright P \Rightarrow \Gamma} \tag{Inf-Emp}$$

$$\Pi \vdash \epsilon \triangleright \epsilon \Rightarrow \Gamma \tag{Inf-Eps}$$

$$\Pi \vdash \overline{T} \triangleright \mathcal{T} \Rightarrow \Gamma \tag{Inf-Any}$$

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

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

$$\frac{\begin{array}{c} l(\overline{X}, \overline{X}') \not<: \emptyset \\ \Pi \vdash \overline{X} \triangleright Y \Rightarrow \Gamma \qquad \Pi \vdash \overline{X}' \triangleright Y' \Rightarrow \Gamma \end{array}}{\Pi \vdash l(\overline{X}, \overline{X}') \triangleright l(Y, Y') \Rightarrow \Gamma} \tag{Inf-Lab}$$

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 INFA-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:

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

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 4.5.3). 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 4.5.4).

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

Proof
Based on the above intuition of partial validation to characterize partial results, we prove the result by showing the following stronger statement:

If $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Pi'; \Gamma$, then $\Pi' \vdash \Pi' \setminus \Pi \Rightarrow \Gamma$ and $\Pi' \vdash \overline{A} \triangleright D \Rightarrow \Gamma$.

The proof follows by straightforward induction on the derivation of $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Pi'; \Gamma$ with the following weakening property: if $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Gamma$ where $\Pi \subseteq \Pi'$ and $\Gamma <: \Gamma'$, then $\Pi' \vdash \overline{A} \triangleright D \Rightarrow \Gamma'$. $\square$

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

Proof
We prove the following stronger statement:

Suppose $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Gamma$ and $\Pi \vdash \Pi \Rightarrow \Gamma$. Then, $t \in \overline{A}$ and $t \in D \Rightarrow V$ with $V(y) = u$ imply $u \in \Gamma(y)$.

The proof proceeds by induction on the lexicographic order on the pair of $h(t) + \mathit{isst}(D)$ (where $\mathit{isst}(T) = 0$ and $\mathit{isst}(X) = 1$) and the derivation of $\Pi \vdash \overline{A} \triangleright D$, with the case analysis on the last rule applied to derive $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Gamma$.

Case INF-ST: $\dfrac{(\overline{X} \triangleright Y) \in \Pi}{\Pi \vdash \overline{X} \triangleright Y \Rightarrow \Gamma}$

We have, by assumption, $\Pi \vdash \Pi \Rightarrow \Gamma$ and $(\overline{X} \triangleright Y) \in \Pi$. So $\Pi \vdash \mathit{unf}(\overline{X}) \triangleright N(Y) \Rightarrow \Gamma$. Furthermore, from Lemma 4.3.1 and MAT-ST, we have $t \in \mathit{unf}(\overline{X})$ and $t \in N(Y) \Rightarrow V$. Note that $h(t) + \mathit{isst}(\overline{X}) > h(t) + \mathit{isst}(\mathit{unf}(\overline{X}))$. This allows us to use the induction hypothesis, from which the result follows.

Case Inf-Bind: $\dfrac{\Pi \vdash \overline{T} \triangleright P \Rightarrow \Gamma \qquad \overline{T} \Rightarrow T \qquad \{x : T\} <: \Gamma}{\Pi \vdash \overline{T} \triangleright x : P \Rightarrow \Gamma}$

We have two cases on y and u.

- When $x = y$, we have $u = t$ from Mat-Bind. Thus, by assumption, we have $u \in \overline{T}$. From the definition of the conversion relation, $u \in T$. The result follows from the side condition $\{x : T\} <: \Gamma$.
- Otherwise, we have $t \in P \Rightarrow V'$ with $V = V' \cup \{x : t\}$ from Mat-Bind. The result follows from the induction hypothesis.

Case Inf-Emp: $\dfrac{\overline{T} <: \emptyset}{\Pi \vdash \overline{T} \triangleright P \Rightarrow \Gamma}$

The result immediately follows since there is no $t \in \overline{T}$.

Case Inf-Eps: $\Pi \vdash \epsilon \triangleright \epsilon \Rightarrow \Gamma$

The result immediately follows since $V = \emptyset$ by Mat-Eps.

Case Inf-Or1: $\dfrac{\Pi \vdash (\overline{T} \; \mathsf{isect} \; P_1) \triangleright P_1 \Rightarrow \Gamma \qquad \Pi \vdash (\overline{T} \; \mathsf{diff} \; P_1) \triangleright P_2 \Rightarrow \Gamma}{\Pi \vdash \overline{T} \triangleright P_1 \mid P_2 \Rightarrow \Gamma}$

We have two cases on $t \in P_1 \mid P_2 \Rightarrow V$.

- $t \in P_1 \Rightarrow V$ by Mat-Or1. By using Lemma 4.3.1 with $t \in \overline{T}$, we have $t \in (\overline{T} \; \mathsf{isect} \; P_1)$. The result follows from the induction hypothesis.
- $t \notin P_1$ and $t \in P_2 \Rightarrow V$ by Mat-Or2. By using Lemma 4.3.1 with $t \in \overline{T}$, we have $t \in (\overline{T} \; \mathsf{diff} \; P_1)$. The result follows from the induction hypothesis.

Case Inf-Or2: $\dfrac{\Pi \vdash \overline{T}_1 \triangleright P \Rightarrow \Gamma \qquad \Pi \vdash \overline{T}_2 \triangleright P \Rightarrow \Gamma}{\Pi \vdash \overline{T}_1 \mid \overline{T}_2 \triangleright P \Rightarrow \Gamma}$

From $t \in \overline{T}_1 \mid \overline{T}_2$, either $t \in \overline{T}_1$ by CAcc-Or1 or $t \in \overline{T}_2$ by CAcc-Or2. In either case, the result follows from the induction hypothesis.

Case Inf-Lab: $\dfrac{l(\overline{X}, \overline{X}') \not<: \emptyset \qquad \Pi \vdash \overline{X} \triangleright Y \Rightarrow \Gamma \qquad \Pi \vdash \overline{X}' \triangleright Y' \Rightarrow \Gamma}{\Pi \vdash l(\overline{X}, \overline{X}') \triangleright l(Y, Y') \Rightarrow \Gamma}$

From $t \in l(\overline{X}, \overline{X}')$ and from CAcc-Lab, we have $t = l(v, v')$ where $v \in \overline{X}$ and $v' \in \overline{X}'$. From $t \in l(Y, Y') \Rightarrow (y \mapsto u)$, we have either $v \in Y \Rightarrow (y \mapsto u)$ or $v' \in Y' \Rightarrow (y \mapsto u)$ by Mat-Lab. In either case, the result follows from the induction hypothesis. □

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

4.5.5 Theorem [Termination]: For all types A defined under a tree automaton M and patterns D defined under a pattern automaton N, the type inference algorithm either fails or effectively calculates 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.

Proof

Let $\mathcal{X}$ be $dom(M) \times \mathcal{P}(dom(N)) \times \mathcal{P}(dom(N))$ (where $\mathcal{P}(S)$ is the powerset of S). We prove a stronger statement:

Suppose $\Pi \subseteq \mathcal{X} \times dom(N)$ with $FS(\overline{A}) \subseteq \mathcal{X}$ and $FS(D) \subseteq dom(N)$. Then the algorithm either fails or effectively calculates a type environment Γ defined under a tree automaton $M' \supseteq M$ such that $\Pi \vdash \overline{A} \triangleright D \Rightarrow \Pi'; \Gamma$ for some Π'.

The proof proceeds by induction on the lexicographic order on $(|(\mathcal{X} \times dom(N)) \setminus \Pi|, \mathsf{size}(D), \mathsf{size}(\overline{A}))$, with the case analysis on the applicable rule (if not applicable, then the result immediately follows).

Case INFA-EMP/INFA-EPS/INFA-ANY/INFA-ST**:**

The result immediately follows.

Case INFA-BIND**:**

The rule decreases $\mathsf{size}(D)$. The result follows from the induction hypothesis and Lemma C.2.

Case INFA-OR1**:**

By inspecting the definition of isect and diff, we can see that $FS(\overline{T}\ \mathsf{isect}\ P_1) \subseteq \mathcal{X}$ and $FS(\overline{T}\ \mathsf{diff}\ P_1) \subseteq \mathcal{X}$. Also, the rule decreases $\mathsf{size}(D)$. The result follows from the induction hypothesis.

Case INFA-OR2/INFA-LAB**:**

The rules decrease $\mathsf{size}(D)$ or else $\mathsf{size}(\overline{A})$. The result follows from the induction hypothesis.

Case INFA-UNF**:**

We have, by the assumption, $\overline{X} \in \mathcal{X}$ and $Y \in dom(N)$. Therefore the rule decreases $|(\mathcal{X} \times dom(N)) \setminus \Pi|$. The result follows by the induction hypothesis. □

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 (Deutsch *et al.*, 1998; Abiteboul *et al.*, 1997; Cluet & Siméon, 1998; Cardelli & Ghelli, 2000; Neven & Schwentick, 2000) and document processing languages (Neumann & Seidl, 1998; Murata, 1997). In the *single-match* style, a successful match yields just one binding. This style is usually taken in programming languages (Milner *et al.*, 1990; Leroy *et al.*, 1996; Peyton Jones *et al.*, 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 "logic." Several papers have proposed extensions of ML-like pattern matching with recursion (Fähndrich & Boyland, 1997; Queinnec, 1990), which have essentially the same expressiveness as ours. Some query languages and document processing languages use pattern matching mechanisms based on tree automata (Neumann & Seidl, 1998; Murata,

1997) or monadic second-order logic (which is equivalent to tree automata) (Neven & Schwentick, 2000), and therefore they have a similar expressiveness to our pattern matching. TQL (Cardelli & Ghelli, 2000) is based on Ambient Logic (Cardelli & Gordon, 2000), 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 (Deutsch *et al.*, 1998; Abiteboul *et al.*, 1997; Cluet & Siméon, 1998), 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 (Milo & Suciu, 1999; Papakonstantinou & Vianu, 2000) and in the setting of a document transformation framework (Murata, 1997). 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 (Milo *et al.*, 2000). 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).

Connections of our work to various standards for XML may be worth discussing. The schema languages DTD (Bray *et al.*, 2000), DSD (Klarlund *et al.*, 2000), and RELAX NG (Clark & Murata, 2001) are based on formal language theory and have strong similarities to our regular expression types. In particular, the core part of RELAX NG is based on tree automata and has essentially the same expressiveness. Another schema language XML-Schema (Fallside, 2001) is based on a mixture of formal languages and object-orientation. A further discussion of these can be found in (Hosoya *et al.*, 2000). The type system of XQuery (Fankhauser *et al.*, 2001) is based on tree automata, and hence is very similar to our type system (in fact, their type system was originally inspired by ours). XQuery also uses case expressions that are similar to but weaker than our pattern matching (variables can appear only at the top level). XSLT (Clark, 1999) has a pattern matching construct XPath (Clark & DeRose, 1999). XPath is quite similar to regular path expressions mentioned above (therefore it has a similar weakness mentioned there) but has an ability to traverse ancestor nodes, which we do not provide.

Another area related to our type inference method is *set-constraint solving* (Aiken & Wimmers, 1992) (also known as tree set automata (Gilleron *et al.*, 1999)). This framework takes a system of inclusion constraints among types with free variables and checks the satisfiability of the constraints (Aiken & Wimmers, 1992) or finds a least solution if it exists (Gilleron *et al.*, 1999). 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 (Wright & Cartwright, 1994). 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 (Puel & Suárez, 1990) 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" (Hosoya *et al.*, 2000) and is also quite handy in writing patterns with "don't care" parts. 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 C 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 and JFP 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

Abiteboul, Serge, Quass, Dallan, McHugh, Jason, Widom, Jennifer, & Wiener, Janet L. (1997). The Lorel query language for semistructured data. *International journal on digital libraries*, **1**(1), 68–88.

Aiken, Alexander, & Wimmers, Edward L. 1992 (June). Solving systems of set constraints (extended abstract). *Pages 329–340 of: Proceedings, seventh annual IEEE symposium on logic in computer science.*

Bray, Tim, Paoli, Jean, Sperberg-McQueen, C. M., & Maler, Eve. (2000). *Extensible markup language (XML^{TM}).* `http://www.w3.org/XML/`.

Burstall, R., MacQueen, David, & Sannella, Donald. (1980). HOPE: an experimental applicative language. *Pages 136–143 of: Proceedings of the 1980 LISP conference.* Stanford, California: Stanford University.

Cardelli, Luca, & Ghelli, Giorgio. 2000 (April). *A query language for semistructured data based on the Ambient Logic.* Manuscript.

Cardelli, Luca, & Gordon, Andrew D. (2000). Anytime, anywhere. Modal logics for mobile ambients. *Pages 365–377 of: Proceedings of the 27th ACM symposium on principles of programming languages.*

Clark, James. (1999). *XSL Transformations (XSLT).* `http://www.w3.org/TR/xslt`.

Clark, James, & DeRose, Steve. (1999). *XML path language (XPath).* `http://www.w3.org/TR/xpath`.

Clark, James, & Murata, Makoto. (2001). *RELAX NG.* `http://www.relaxng.org`.

Cluet, Sophie, & Siméon, Jérôme. (1998). Using YAT to build a web server. *Intl. workshop on the web and databases (WebDB).*

Deutsch, Alin, Fernandez, Mary, Florescu, Daniela, Levy, Alon, & Suciu, Dan. (1998). *XML-QL: A Query Language for XML.* `http://www.w3.org/TR/NOTE-xml-ql`.

Fähndrich, Manuel, & Boyland, John. (1997). Statically checkable pattern abstractions. *Pages 75–84 of: Proceedings of the international conference on functional programming (ICFP).*

Fallside, David C. (2001). *XML Schema Part 0: Primer, W3C Recommendation.* `http://www.w3.org/TR/xmlschema-0/`.

Fankhauser, Peter, Fernández, Mary, Malhotra, Ashok, Rys, Michael, Siméon, Jérôme, & Wadler, Philip. (2001). *XQuery 1.0 Formal Semantics.* `http://www.w3.org/TR/query-semantics/`.

Gapeyev, Vladimir, Levin, Michael, & Pierce, Benjamin. (2000). Recursive subtyping revealed. *Pages 221–232 of: Proceedings of the international conference on functional programming (ICFP).*

Gilleron, R., Tison, S., & Tommasi, M. (1999). Set constraints and automata. *Information and computation*, **149**(1), 1–41.

Hopcroft, John E., & Ullman, Jeffrey D. (1979). *Introduction to automata theory, languages, and computation.* Addison-Wesley.

Hosoya, Haruo, & Pierce, Benjamin C. 2000 (May). XDuce: A typed XML processing language. *Pages 226–244 of: Proceedings of third international workshop on the web and databases (WebDB2000).* Lecture Notes in Computer Science, vol. 1997.

Hosoya, Haruo, Vouillon, Jérôme, & Pierce, Benjamin C. 2000 (Sept.). Regular expression types for XML. *Pages 11–22 of: Proceedings of the international conference on functional programming (ICFP).* Full version under submission to TOPLAS.

Klarlund, Nils, Møller, Anders, & Schwartzbach, Michael I. (2000). *DSD: A schema language for XML.* `http://www.brics.dk/DSD/`.

Leroy, Xavier, Vouillon, Jérôme, Doligez, Damien, *et al.* . (1996). *The Objective Caml system*. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`.

Milner, Robin, Tofte, Mads, & Harper, Robert. (1990). *The definition of Standard ML*. The MIT Press.

Milo, Tova, & Suciu, Dan. 1999 (May). Type inference for queries on semistructured data. *Pages 215–226 of: Proceedings of symposium on principles of database systems.*

Milo, Tova, Suciu, Dan, & Vianu, Victor. (2000). Typechecking for XML transformers. *Pages 11–22 of: Proceedings of the nineteenth acm sigmod-sigact-sigart symposium on principles of database systems.* ACM.

Murata, Makoto. (1997). Transformation of documents and schemas by patterns and contextual conditions. *Pages 153–169 of: Principles of document processing '96*. Lecture Notes in Computer Science, vol. 1293. Springer-Verlag.

Neumann, Andreas, & Seidl, Helmut. (1998). Locating matches of tree patterns in forests. *Pages 134–145 of: 18th FSTTCS*. LNCS, vol. 1530.

Neven, Frank, & Schwentick, Thomas. (2000). Expressive and efficient pattern languages for tree-structured data. *Pages 145–156 of: Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems.* ACM.

Papakonstantinou, Yannis, & Vianu, Victor. 2000 (May). DTD Inference for Views of XML Data. *Pages 35–46 of: Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems.*

Peyton Jones, Simon L., Hall, Cordelia V., Hammond, Kevin, Partain, Will, & Wadler, Philip. 93 (July). The Glasgow Haskell compiler: a technical overview. *Proc. UK joint framework for information technology (JFIT) technical conference.*

Puel, Laurence, & Suárez, Ascánder. 1990 (June). Compiling pattern matching by term decomposition. *Pages 272–281 of: 1990 ACM conference on Lisp and functional programming.*

Queinnec, Christian. (1990). Compilation of non-linear, second order patterns on s-expressions. *Pages 340–357 of: Programming language implementation and logic programming, 2nd international workshop (PLILP'90).* Lecture Notes in Computer Science. Springer-Verlag.

Seidl, Hermut. (1990). Deciding equivalence of finite tree automata. *SIAM journal of computing*, **19**(3), 424–437.

Slutzki, Giora. (1985). Alternating tree automata. *Theoretical computer science*, **41**, 305–318.

Sperberg-McQueen, C. M., & Burnard, Lou. *A gentle introduction to SGML.* `http://www-tei.uic.edu/orgs/tei/sgml/teip3sg`.

Wright, Andrew K., & Cartwright, Robert. (1994). A practical soft type system for scheme. *Pages 250–262 of: In proceedings of ACM conference on Lisp and functional programming.*

A Supplementary Definitions

A.1 Well-formedness of Types

We define well-formedness of types in terms of a set of "non-tail variables" and an auxiliary set of "top-level variables." The set *toplevel*(`T`) of top-level variables of a type `T` is the smallest set satisfying the following equations:

$$\begin{array}{lcll} \mathit{toplevel}(\mathtt{X}) & = & \{\mathtt{X}\} \cup \mathit{toplevel}(E(\mathtt{X})) & \\ \mathit{toplevel}(\mathtt{T}) & = & \emptyset & \text{if } \mathtt{T} = \emptyset,\ \mathtt{()},\ \text{or } \mathtt{l[T']} \\ \mathit{toplevel}(\mathtt{T|U}) & = & \mathit{toplevel}(\mathtt{T}) \cup \mathit{toplevel}(\mathtt{U}) & \\ \mathit{toplevel}(\mathtt{T,U}) & = & \mathit{toplevel}(\mathtt{T}) \cup \mathit{toplevel}(\mathtt{U}) & \end{array}$$

Likewise, the set $\mathit{nontail}(\mathtt{T})$ of non-tail variables of a type `T` is the smallest set satisfying the following equations:

$$\begin{array}{lcll} \mathit{nontail}(\mathtt{X}) & = & \mathit{nontail}(E(\mathtt{X})) & \\ \mathit{nontail}(\mathtt{T}) & = & \emptyset & \text{if } \mathtt{T} = \emptyset,\ \mathtt{()},\ \text{or } \mathtt{l[T']} \\ \mathit{nontail}(\mathtt{T|U}) & = & \mathit{nontail}(\mathtt{T}) \cup \mathit{nontail}(\mathtt{U}) & \\ \mathit{nontail}(\mathtt{T,U}) & = & \mathit{toplevel}(\mathtt{T}) \cup \mathit{nontail}(\mathtt{U}) & \end{array}$$

Now, the set E of type definitions is said to be *well-formed* if

$$\mathtt{X} \notin \mathit{nontail}(E(\mathtt{X})) \text{ for all } \mathtt{X} \in \mathit{dom}(E).$$

A.2 Semantics of Types

The semantics of types is presented by the relation $\mathtt{v} \in \mathtt{T}$, read "`v` has type `T`," which is defined by the following set of rules.

$$\mathtt{()} \in \mathtt{()} \tag{ET-Emp}$$

$$\frac{\begin{array}{c}\mathtt{type\ X{=}T} \in E \\ \mathtt{v} \in \mathtt{T}\end{array}}{\mathtt{v} \in \mathtt{X}} \tag{ET-Var}$$

$$\frac{\mathtt{v} \in \mathtt{T}}{\mathtt{l[v]} \in \mathtt{l[T]}} \tag{ET-Lab}$$

$$\frac{\mathtt{v} \in \mathtt{T} \qquad \mathtt{w} \in \mathtt{U}}{\mathtt{v,w} \in \mathtt{T,U}} \tag{ET-Cat}$$

$$\frac{\mathtt{v} \in \mathtt{T}_1}{\mathtt{v} \in \mathtt{T}_1\mathtt{|T}_2} \tag{ET-Or1}$$

$$\frac{\mathtt{v} \in \mathtt{T}_2}{\mathtt{v} \in \mathtt{T}_1\mathtt{|T}_2} \tag{ET-Or2}$$

A.3 Linearity of Patterns

Given an external pattern `P`, let σ be the set of all pattern names reachable from `P`, that is, the smallest set of pattern names that satisfies

$$\sigma = \mathit{FN}(\mathtt{P}) \cup \bigcup_{\mathtt{Y} \in \sigma} \mathit{FN}(F(\mathtt{Y})).$$

Then, a variable $\mathtt{x}$ is *unreachable* from $\mathtt{P}$, written $\mathtt{P} \downarrow^0_{\mathtt{x}}$ when $\mathtt{x} \notin BV(\mathtt{P}) \cup \bigcup_{\mathtt{Y} \in \sigma} BV(F(\mathtt{Y}))$.

The *linearity* relation $\mathtt{P} \downarrow^1_{\mathtt{x}}$, read "the pattern $\mathtt{P}$ is linear in the variable $\mathtt{x}$," is *co*-inductively defined by the following rules.

$$\frac{\mathtt{P}_1 \downarrow^0_{\mathtt{x}}}{\mathtt{x}\!:\!\mathtt{P}_1 \downarrow^1_{\mathtt{x}}}$$

$$\frac{\mathtt{P}_1 \downarrow^1_{\mathtt{x}} \qquad \mathtt{x} \neq \mathtt{y}}{\mathtt{y}\!:\!\mathtt{P}_1 \downarrow^1_{\mathtt{x}}}$$

$$\frac{F(\mathtt{Y}) \downarrow^1_{\mathtt{x}}}{\mathtt{Y} \downarrow^1_{\mathtt{x}}}$$

$$\frac{\mathtt{P}_1 \downarrow^1_{\mathtt{x}}}{\mathtt{l[P_1]} \downarrow^1_{\mathtt{x}}}$$

$$\frac{\mathtt{P}_1 \downarrow^1_{\mathtt{x}} \qquad \mathtt{P}_2 \downarrow^0_{\mathtt{x}}}{(\mathtt{P}_1,\mathtt{P}_2) \downarrow^1_{\mathtt{x}}}$$

$$\frac{\mathtt{P}_2 \downarrow^1_{\mathtt{x}} \qquad \mathtt{P}_1 \downarrow^0_{\mathtt{x}}}{(\mathtt{P}_1,\mathtt{P}_2) \downarrow^1_{\mathtt{x}}}$$

$$\frac{\mathtt{P}_1 \downarrow^1_{\mathtt{x}} \qquad \mathtt{P}_2 \downarrow^1_{\mathtt{x}}}{(\mathtt{P}_1|\mathtt{P}_2) \downarrow^1_{\mathtt{x}}}$$

(We use here co-induction rather than induction since we need to treat recursion. See (Gapeyev *et al.*, 2000) for a related discussion.)

A.4 Semantics of External Patterns

We define the semantics of patterns in a similar way to types, except for the treatment of the first-match policy. We use a notion of *choice sequence*. During pattern matching, we remember the index of the branch we take at each choice point. A choice sequence is a sequence of such indices, listed according to the order of traversal—from left to right and from outer to inner. Finally, we take the *smallest* choice sequence in the dictionary order.

Formally, we describe the semantics of patterns by first defining the relation $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V} \;/\; \alpha$, read "$\mathtt{v}$ is matched by $\mathtt{P}$, yielding $\mathtt{V}$ and α," where a choice sequence α is a sequence of elements from the ordered set $(\{1,2\}, \leq)$ and an environment $\mathtt{V}$ is a finite mapping from variables to values (written $\mathtt{x}_1\!:\!\mathtt{v}_1,\ldots,\mathtt{x}_n\!:\!\mathtt{v}_n$). Then we define the relation $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}$ that takes the smallest choice sequence w.r.t. the lexicographic order $\sqsubset$ on choice sequences. The following set of rules defines both relations.

$$\mathtt{v} \in \mathtt{x} \Rightarrow \mathtt{x}\!:\!\mathtt{v} \;/\; \cdot \tag{EP-Bare}$$

$$\frac{\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V} \;/\; \alpha \qquad \mathtt{x} \notin dom(\mathtt{V})}{\mathtt{v} \in (\mathtt{x}\ \mathtt{as}\ \mathtt{P}) \Rightarrow \mathtt{x}\!:\!\mathtt{v},\mathtt{V} \;/\; \alpha} \tag{EP-As}$$

$$\texttt{()} \in \texttt{()} \Rightarrow \emptyset \;/\; \cdot \qquad \text{(EP-Emp)}$$

$$\frac{\texttt{pat Y=P} \in F \qquad \texttt{v} \in \texttt{P} \Rightarrow \texttt{V} \;/\; \alpha}{\texttt{v} \in \texttt{Y} \Rightarrow \texttt{V} \;/\; \alpha} \qquad \text{(EP-Var)}$$

$$\frac{\texttt{v} \in \texttt{P} \Rightarrow \texttt{V} \;/\; \alpha}{\texttt{l[v]} \in \texttt{l[P]} \Rightarrow \texttt{V} \;/\; \alpha} \qquad \text{(EP-Lab)}$$

$$\frac{\texttt{v} \in \texttt{P} \Rightarrow \texttt{V} \;/\; \alpha \qquad \texttt{w} \in \texttt{Q} \Rightarrow \texttt{W} \;/\; \beta \qquad \mathit{dom}(\texttt{V}) \cap \mathit{dom}(\texttt{W}) = \emptyset}{\texttt{v,w} \in \texttt{P,Q} \Rightarrow \texttt{V,W} \;/\; \alpha, \beta} \qquad \text{(EP-Cat)}$$

$$\frac{\texttt{v} \in \texttt{P} \Rightarrow \texttt{V} \;/\; \alpha}{\texttt{v} \in \texttt{P|Q} \Rightarrow \texttt{V} \;/\; 1, \alpha} \qquad \text{(EP-Or1)}$$

$$\frac{\texttt{v} \in \texttt{Q} \Rightarrow \texttt{V} \;/\; \alpha}{\texttt{v} \in \texttt{P|Q} \Rightarrow \texttt{V} \;/\; 2, \alpha} \qquad \text{(EP-Or2)}$$

$$\frac{\texttt{v} \in \texttt{P} \Rightarrow \texttt{V} \;/\; \alpha \qquad \forall \beta.(\texttt{v} \in \texttt{P} \Rightarrow \texttt{U} \;/\; \beta \implies \alpha \sqsubset \beta)}{\texttt{v} \in \texttt{P} \Rightarrow \texttt{V}} \qquad \text{(EP)}$$

Notice that in EP-Cat we concatenate the choice sequences left to right, and that in ET-Or1 and ET-Or2 we adjoin the present choice number to the front. These reflect our policy that the priority of choice is from left to right and from outer to inner.

We can prove that if a pattern is linear in a variable, then the variable appears in the domain of the environment that is yielded by any match against the pattern.

A.4.1 Lemma: If $\texttt{P} \downarrow^{1}_{\texttt{x}}$ and $\texttt{v} \in \texttt{P} \Rightarrow \texttt{V} \;/\; \alpha$, then $\texttt{x} \in \mathit{dom}(\texttt{V})$.[5]

Proof
First, we can easily prove that a variable unreachable from a pattern does not appear in the domain of the environment yielded by any match against the pattern:

If $\texttt{P} \downarrow^{0}_{\texttt{x}}$ and $\texttt{v} \in \texttt{P} \Rightarrow \texttt{V} \;/\; \alpha$, then $\texttt{x} \notin \mathit{dom}(\texttt{V})$.

Then, the result can be proved by a straightforward induction on the derivation of $\texttt{v} \in \texttt{P} \Rightarrow \texttt{V} \;/\; \alpha$. □

[5] The converse does not hold. For the pattern `(l[x],y)|(l[x],x)`, every match yields a sensible environment. But this pattern is syntactically not linear.

B Translation from External Patterns to Internal Patterns

This section first shows the translation algorithm, then proves its soundness and completeness. We concentrate on the translation of patterns since the translation of types is exactly the same except that the treatment for pattern variables is dropped. We have not proved the termination of the pattern translation. We believe that we can adapt the proof technique used for the type translation in (Hosoya *et al.*, 2000).[6]

B.1 Algorithm

The translation of patterns consists of (1) the conversion of input values from the external form to the internal form, (2) the translation of pattern expressions from the external form to the internal form, and (3) the conversion of output environments from the internal form back to the external form.

The translation of values is straightforward.

$$\begin{array}{rcl} ts(\texttt{()}) & = & \epsilon \\ ts(\texttt{l[v}_1\texttt{],v}_2) & = & \texttt{l}(ts(\texttt{v}_1), ts(\texttt{v}_2)) \end{array}$$

We now consider our algorithm for the translation of patterns. Let us first illustrate it by an example. Consider the following external pattern:

```
(x as a[]*), d[]
```

By expanding the abbreviation `a[]*`, this pattern is equivalent to `((x as Y), d[])` where:

```
pat Y = (a[], Y) | ()
```

Now, we want to compute the internal pattern corresponding to `((x as Y), d[])`. For this, we transform the above pattern in such a way that all head labels are revealed. We first introduce two new variables to handle the variable pattern `(x as Y)` as explained previously:

$$\texttt{x}_b \texttt{ as (Y, (x}_e \texttt{ as d[]))}$$

Then we expand `Y` to its definition and use distributivity of the union operator, associativity of the concatenation operator, and neutrality of the empty sequence:

$$\begin{array}{ll} \Longrightarrow & \texttt{x}_b \texttt{ as (((a[], Y) | ()), (x}_e \texttt{ as d[]))} \\ \Longrightarrow & \texttt{x}_b \texttt{ as (((a[], Y), (x}_e \texttt{ as d[])) | ((), (x}_e \texttt{ as d[])))} \\ \Longrightarrow & \texttt{x}_b \texttt{ as ((a[], (Y, (x}_e \texttt{ as d[]))) | (x}_e \texttt{ as d[]))} \end{array}$$

Since the head labels `a` and `d` are revealed now, this pattern can be translated to the internal pattern:

$$\texttt{x}_b : (\texttt{a}(Y_0, Y_1) \mid \texttt{x}_e : \texttt{d}(Y_0, Y_0))$$

[6] In Section 3.1, we (informally) mention the restriction "no head-recursion," which requires any unguarded recursive use of pattern names to be preceded by a pattern that does not match the empty sequence. Although we do not make use of this restriction in the formal treatments in this paper, it could be used in the proof of termination of the pattern translation algorithm in the same way as the proof of the type translation presented in (Hosoya *et al.*, 2000).

Since the content of `a` and both the content and the remainder of `d` are all the empty sequence, we can share the internal patterns corresponding to these by the single state Y_0, which is associated with ϵ:

$$N(Y_0) \quad = \quad \epsilon$$

The remainder of `a`, i.e., the pattern $\mathtt{(Y,\ (x}_e\ \mathtt{as\ d[]))}$ is translated in a similar way to the above translation of $\mathtt{x}_b\ \mathtt{as\ (Y,\ (x}_e\ \mathtt{as\ d[]))}$, and the resulting internal pattern is associated with Y_1. (The translation again encounters the same pattern $\mathtt{(Y,\ (x}_e\ \mathtt{as\ d[]))}$, with which we associate the same state Y_1.)

$$N(Y_1) \quad = \quad \mathtt{a}(Y_0, Y_1) \mid \mathtt{x}_e : \mathtt{d}(Y_0, Y_0)$$

We now formalize the translation procedure illustrated above. The translation takes an external pattern $\mathtt{P}_0$ and a set F of pattern definitions, and computes an internal pattern P_0 and a pattern automaton N. The procedure works by double loops where the outer loop constructs the automaton, and the inner loop computes an internal pattern expression, which will be associated with a state in the automaton. Each state corresponds to an external pattern expression from which the internal pattern associated with the state is computed; we therefore index states by external patterns `P`, written $Y_{\mathtt{P}}$.

The translation function ts ("the inner loop") takes an external pattern and returns an internal pattern. The rules below define the function ts.

$$\begin{array}{rcl}
ts(\mathtt{()}) & = & \epsilon \\
ts(\mathtt{l[P]}) & = & \mathtt{l}(Y_{\mathtt{P}}, Y_{\mathtt{()}}) \\
ts(\mathtt{x}) & = & \mathtt{x} : \mathcal{T} \\
ts(\mathtt{x\ as\ P}) & = & \mathtt{x} : ts(\mathtt{P}) \\
ts(\mathtt{P}_1 \mathtt{|} \mathtt{P}_2) & = & ts(\mathtt{P}_1) \mid ts(\mathtt{P}_2) \\
ts(\mathtt{Y}) & = & ts(F(\mathtt{Y}))
\end{array}$$

$$\begin{array}{rcl}
ts(\mathtt{(),P}) & = & ts(\mathtt{P}) \\
ts(\mathtt{l[P}_1\mathtt{],P}_2) & = & \mathtt{l}(Y_{\mathtt{P}_1}, Y_{\mathtt{P}_2}) \\
ts(\mathtt{(x\ as\ P}_1\mathtt{),P}_2) & = & ts(\mathtt{x}_b\ \mathtt{as\ (P}_1\mathtt{,(x}_e\ \mathtt{as\ P}_2\mathtt{))}) \\
ts(\mathtt{(P}_1\mathtt{,P}_2\mathtt{),P}_3) & = & ts(\mathtt{P}_1\mathtt{,(P}_2\mathtt{,P}_3\mathtt{)}) \\
ts(\mathtt{(P}_1\mathtt{|P}_2\mathtt{),P}_3) & = & ts(\mathtt{P}_1\mathtt{,P}_2) \mid ts(\mathtt{P}_1\mathtt{,P}_3) \\
ts(\mathtt{Y,P}) & = & ts(F(\mathtt{Y})\mathtt{,P})
\end{array}$$

These rules simply generalize the operations that we have seen in the example. The rule for $\mathtt{((P}_1\mathtt{,P}_2\mathtt{),P}_3\mathtt{)}$ uses associativity and that for $\mathtt{((P}_1\mathtt{|P}_2\mathtt{),P}_3\mathtt{)}$ uses distributivity. Also, the rules for `l[P]` and `(),P` use the neutrality of `()`. The rule for `x` and `(x as P)` simply construct the corresponding variable patterns; the rule for $\mathtt{((x\ as\ P}_1\mathtt{),P}_2\mathtt{)}$ replaces the variable pattern binding `x` with two tail variable patterns binding $\mathtt{x}_b$ and $\mathtt{x}_e$. The rules for `Y` and `(Y,P)` unfold the pattern name `Y` to their definitions. By using all these rules, the head of a sequence is eventually revealed, and turned into an internal label pattern whose "car" state corresponds to the content pattern of the label and whose "cdr" state corresponds to the remainder of the sequence. For each state $Y_{\mathtt{P}}$ contained in the internal label pattern, the associated external pattern $\mathtt{P}$ may be the target of the translation in the next step. The empty sequence is turned into the leaf transition.

The outer loop proceeds as follows. The translation begins with setting N to the empty automaton and computing P_0 from $\mathtt{P}_0$ by using the above function ts. The resulting internal pattern P_0 may contain a state $Y_{\mathtt{P}}$ corresponding to a subphrase $\mathtt{P}$ of $\mathtt{P}_0$. If the state is not yet in the domain of N, then we calculate $P = ts(\mathtt{P})$ and add the mapping $Y_{\mathtt{P}} \mapsto P$ to N. We repeat this process until the states appearing in P_0 and N are all in the domain of N. Thus, the result of the whole translation satisfies the following.

$$\begin{array}{rcll} P_0 & = & ts(\mathtt{P}_0) & \\ N(Y_{\mathtt{P}}) & = & ts(\mathtt{P}) & \forall Y_{\mathtt{P}} \in dom(N) \\ dom(N) & \supseteq & FS(P_0) \cup FS(N). & \end{array}$$

Finally, we define the conversion of environments from the internal form back to the external form. We write ts^{-1} for the inverse of the translation function ts of values (note that the value translation function is bijective); define $ts^{-1}(V)(\mathtt{x}) = ts^{-1}(V(\mathtt{x}))$. Given an environment $\mathtt{V}$ in the external form, the following *chop* function trims off $\mathtt{x}_e$ from $\mathtt{x}_b$ and assigns the result to the original variable $\mathtt{x}$, for each pair of new variables $\mathtt{x}_b$ and $\mathtt{x}_e$ in the domain of $\mathtt{V}$.

$$chop(\mathtt{V})(\mathtt{x}) = \begin{cases} \mathtt{V}(\mathtt{x}) & \mathtt{x} \in dom(\mathtt{V}) \\ \mathtt{v} & \mathtt{x}_b, \mathtt{x}_e \in dom(\mathtt{V}) \ \wedge \ \mathtt{V}(\mathtt{x}_b) = (\mathtt{v},\mathtt{w}) \ \wedge \ \mathtt{V}(\mathtt{x}_e) = \mathtt{w} \end{cases}$$

Then the back conversion of an internal environment V is defined by $chop(ts^{-1}(V))$.

B.2 Soundness and Completeness

The goal here is to show that matching of a value against an external pattern behaves the same as matching the translated value against the translated internal pattern and converting back the output environment. The semantics of external patterns is described in terms of choice sequences (Section 3.1), whereas that of internal patterns is not. To bridge this gap, we first introduce a matching relation for internal patterns with choice sequences. We then prove the desired property in two steps: (1) matching against an internal pattern without choice sequences is equivalent to matching against the same internal pattern that yields the smallest choice sequence; (2) matching against an external pattern with a choice sequence is equivalent to matching against the translation of the external pattern with the same choice sequence and converting back the output environment.

The (internal) matching relation with choice sequences is defined by the following rules.

$$\frac{t \in N(Y) \Rightarrow V \ / \ \alpha}{t \in Y \Rightarrow V \ / \ \alpha} \qquad \text{(MATS-ST)}$$

$$\frac{t \in P \Rightarrow V \ / \ \alpha \qquad x \notin dom(V)}{t \in x : P \Rightarrow V \cup \{(x \mapsto t)\} \ / \ \alpha} \qquad \text{(MATS-BIND)}$$

$$t \in \mathcal{T} \Rightarrow \emptyset \;/\; \cdot \qquad \text{(MatS-Any)}$$

$$\epsilon \in \epsilon \Rightarrow \emptyset \;/\; \cdot \qquad \text{(MatS-Eps)}$$

$$\frac{t \in P_1 \Rightarrow V \;/\; \alpha}{t \in P_1 \mid P_2 \Rightarrow V \;/\; 1, \alpha} \qquad \text{(MatS-Or1)}$$

$$\frac{t \in P_2 \Rightarrow V \;/\; \alpha}{t \in P_1 \mid P_2 \Rightarrow V \;/\; 2, \alpha} \qquad \text{(MatS-Or2)}$$

$$\frac{t_1 \in Y_1 \Rightarrow V_1 \;/\; \alpha_1 \qquad t_2 \in Y_2 \Rightarrow V_2 \;/\; \alpha_2 \qquad dom(V_1) \cap dom(V_2) = \emptyset}{l(t_1, t_2) \in l(Y_1, Y_2) \Rightarrow V_1 \cup V_2 \;/\; \alpha_1, \alpha_2} \qquad \text{(MatS-Lab)}$$

B.2.1 Theorem [Soundness and Completeness]: Suppose that a given pattern automaton N satisfies $N(Y_{\mathtt{P}}) = ts(\mathtt{P})$ for all $Y_{\mathtt{P}} \in dom(N)$ and $dom(N) \supseteq FS(N)$. Let $ts(\mathtt{v}) = t$ and $ts(\mathtt{P}) = P$. Then, $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}$ iff $t \in P \Rightarrow V$ with $\mathtt{V} = chop(ts^{-1}(V))$.

The theorem follows from the subsequent two lemmas, which correspond to the two steps mentioned above.

B.2.2 Lemma: $t \in D \Rightarrow V$ iff there is α such that (1) $t \in D \Rightarrow V \;/\; \alpha$ and (2) $t \in D \Rightarrow V' \;/\; \beta$ for some V' and β implies $\alpha \sqsubset \beta$.

Proof
Below, we use the fact that, for any t, D, we have that $t \in D \Rightarrow V$ for some V iff $t \in D \Rightarrow V' \;/\; \gamma$ for some V' and γ, which can easily be shown.

We first prove the "only if" part of the lemma. The proof proceeds by induction on the derivation of $t \in D \Rightarrow V$ with the case analysis on the rule used in the last derivation.

Case Mat-Or2**:** $\quad D = P_1 \mid P_2 \qquad t \notin P_1 \qquad t \in P_2 \Rightarrow V$

By the induction hypothesis, $t \in P_2 \Rightarrow V \;/\; \alpha'$. Let $\alpha = 2, \alpha'$. The condition (1) holds by MatS-Or2. We now show the condition (2). Suppose that $t \in D \Rightarrow V' \;/\; \beta$ for some V' and β. We have two cases: $t \in P_1 \Rightarrow V' \;/\; \beta'$ with $\beta = 1, \beta'$ by MatS-Or1, and $t \in P_2 \Rightarrow V \;/\; \beta'$ with $\beta = 2, \beta'$ by MatS-Or2. But, from $t \notin P_1$ and the fact given in the beginning of the proof, the first case never arises. Therefore, by the induction hypothesis, $\alpha' \sqsubset \beta'$, hence $\alpha \sqsubset \beta$.

Case Mat-Lab**:** $\quad t = l(t_1, t_2) \qquad D = l(Y_1, Y_2) \qquad t_1 \in Y_1 \Rightarrow V_1 \qquad t_2 \in Y_2 \Rightarrow V_2$

By the induction hypothesis, $t_1 \in Y_1 \Rightarrow V_1 \;/\; \alpha_1$ and $t_2 \in Y_2 \Rightarrow V_2 \;/\; \alpha_2$ for some α_1 and α_2. Let $\alpha = \alpha_1, \alpha_2$. The condition (1) follows by MatS-Lab. We now show the condition (2). Suppose that $t \in D \Rightarrow V' \;/\; \beta$ for some V' and β. By MatS-Lab, $t_1 \in Y_1 \Rightarrow V_1 \;/\; \beta_1$ and $t_2 \in Y_2 \Rightarrow V_2 \;/\; \beta_2$ with $V = V_1 \cup V_2$ and $\beta = \beta_1, \beta_2$. By the induction hypothesis, $\alpha_1 \sqsubset \beta_1$ and $\alpha_2 \sqsubset \beta_2$, hence $\alpha \sqsubset \beta$.

Other cases:

The result immediately holds or can be shown by a straightforward use of the induction hypothesis.

We next prove the "if" part of the lemma. Similarly to the above, the proof proceeds by induction on the derivation of $t \in D \Rightarrow V \;/\; \alpha$ with the case analysis on the rule used in the last derivation.

Case MatS-Or2**:** $D = P_1 \mid P_2 \qquad t \in P_2 \Rightarrow V \;/\; \alpha' \qquad \alpha = 2, \alpha'$

Suppose that $t \in P_2 \Rightarrow V' \;/\; \beta'$ for some V' and β'. Then, by MatS-Or2, $t \in D \Rightarrow V' \;/\; 2, \beta'$. By the condition (2), $2, \alpha' \sqsubset 2, \beta'$, hence $\alpha' \sqsubset \beta'$. By the induction hypothesis, $t \in P_2 \Rightarrow V$. We have only to show $t \notin P_1$, from which the result follows with Mat-Or2. Suppose that $t \in P_1 \Rightarrow U$ for some U. Then, from the fact given in the beginning of the proof, $t \in P_1 \Rightarrow U \;/\; \gamma$ for some γ. Therefore $t \in D \Rightarrow U \;/\; 1, \gamma$ by MatS-Or1. But we have $\alpha = 2, \alpha' \not\sqsubset 1, \gamma$, which contradicts the condition (2).

Case MatS-Lab**:** $t = l(t_1, t_2) \qquad D = l(Y_1, Y_2)$
$t_1 \in Y_1 \Rightarrow V_1 \;/\; \alpha_1 \qquad t_2 \in Y_2 \Rightarrow V_2 \;/\; \alpha_2$
$V = V_1 \cup V_2 \qquad \alpha = \alpha_1, \alpha_2$

We first establish (2) for the induction hypothesis. Suppose that $t_1 \in Y_1 \Rightarrow V_1' \;/\; \beta_1$ for some V_1' and β_1. Then, by MatS-Lab, $t \in D \Rightarrow V_1' \cup V_2 \;/\; \beta_1, \alpha_2$. By the condition (2), $\alpha_1, \alpha_2 \sqsubset \beta_1, \alpha_2$, hence $\alpha_1 \sqsubset \beta_1$. Similarly, $t_2 \in Y_2 \Rightarrow V_2' \;/\; \beta_2$ implies $\alpha_2 \sqsubset \beta_2$ for any β_2. By the induction hypothesis, $t_1 \in Y_1 \Rightarrow V_1$ and $t_2 \in Y_2 \Rightarrow V_2$. The result follows from by Mat-Lab.

Other cases:

The result immediately holds or can be shown by a straightforward use of the induction hypothesis. □

B.2.3 Lemma: Suppose that a given pattern automaton N satisfies $N(Y_{\mathtt{P}}) = ts(\mathtt{P})$ for all $Y_{\mathtt{P}} \in dom(N)$ and $dom(N) \supseteq FS(N)$. Let $ts(\mathtt{v}) = t$ and $ts(\mathtt{P}) = P$. Then $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V} \;/\; \alpha$ iff $t \in P \Rightarrow V \;/\; \alpha$ with $\mathtt{V} = chop(ts^{-1}(V))$.

Proof
The proof proceeds by induction on the lexicographic order on the pair of $h(t)$ and the derivation of $ts(\mathtt{P}) = P$, with the case analysis on the rule used in the last derivation.

Case: $\mathtt{P} = \mathtt{(x\ as\ P')} \qquad ts(\mathtt{P'}) = P \qquad P = \mathtt{x} : P'$

By EP-As, $\mathtt{v} \in \mathtt{P'} \Rightarrow \mathtt{V'} \;/\; \alpha$ where $\mathtt{V} = \mathtt{V'} \cup \{\mathtt{x} \mapsto \mathtt{v}\}$. By the induction hypothesis, $t \in P' \Rightarrow V' \;/\; \alpha$ with $chop(ts^{-1}(V')) = \mathtt{V'}$. The result follows by Mat-Bind.

Case: $\mathtt{P} = \mathtt{l[P_1],P_2} \qquad P = \mathtt{l}(Y_{\mathtt{P}_1}, Y_{\mathtt{P}_2})$

By EP-Lab and EP-Cat, $\mathtt{v_1} \in \mathtt{P_1} \Rightarrow \mathtt{V_1} \;/\; \alpha_1$ and $\mathtt{v_2} \in \mathtt{P_2} \Rightarrow \mathtt{V_2} \;/\; \alpha_2$ where $\mathtt{v} = \mathtt{l[v_1],v_2}$ and $\mathtt{V} = \mathtt{V_1} \cup \mathtt{V_2}$ and $\alpha = \alpha_1, \alpha_2$. The result follows by using the induction hypothesisand then Mat-St and Mat-Lab.

Case: $\mathtt{P} = \mathtt{(P_1,P_2),P_3} \qquad ts(\mathtt{P_1,(P_2,P_3)}) = P$

The result follows from the fact that $\mathtt{v} \in \mathtt{((P_1,P_2),P_3)} \Rightarrow \mathtt{V} \;/\; \alpha$ iff $\mathtt{v} \in \mathtt{(P_1,(P_2,P_3))} \Rightarrow \mathtt{V} \;/\; \alpha$ (which can easily be shown), and the induction hypothesis.

Case: $\mathtt{P} = (\mathtt{P}_1|\mathtt{P}_2),\mathtt{P}_3 \qquad ts(\mathtt{P}_1,\mathtt{P}_3) \mid ts(\mathtt{P}_2,\mathtt{P}_3) = P$

We show the "only if" part since the converse is similar. From EP-Cat, the matching in the assumption implies $\mathtt{v}_1 \in \mathtt{P}_1|\mathtt{P}_2 \Rightarrow \mathtt{V}_1 \;/\; \alpha_1$ and $\mathtt{v}_2 \in \mathtt{P}_3 \Rightarrow \mathtt{V}_2 \;/\; \alpha_2$ where $\mathtt{v} = \mathtt{v}_1,\mathtt{v}_2$ and $\mathtt{V} = \mathtt{V}_1 \cup \mathtt{V}_2$ and $\alpha = \alpha_1, \alpha_2$. Further, the matching against $\mathtt{P}_1|\mathtt{P}_2$ has two cases: $\mathtt{v}_1 \in \mathtt{P}_1 \Rightarrow \mathtt{V}_1 \;/\; \alpha_1'$ with $\alpha_1 = 1, \alpha_1'$ using EP-Or1, and $\mathtt{v}_1 \in \mathtt{P}_2 \Rightarrow \mathtt{V}_1 \;/\; \alpha_1'$ with $\alpha_1 = 2, \alpha_1'$ using EP-Or2. In the first case, by EP-Cat, $\mathtt{v} \in \mathtt{P}_1,\mathtt{P}_3 \Rightarrow \mathtt{V} \;/\; \alpha_1', \alpha_2$. By the induction hypothesis, $t \in ts(\mathtt{P}_1,\mathtt{P}_3) \Rightarrow V \;/\; \alpha_1', \alpha_2$. The result follows by MatS-Or1. The second case is similar.

Case: $\mathtt{P} = (\mathtt{x}\ \mathtt{as}\ \mathtt{P}_1),\mathtt{P}_2 \qquad ts(\mathtt{x}_b\ \mathtt{as}\ (\mathtt{P}_1,\ (\mathtt{x}_e\ \mathtt{as}\ \mathtt{P}_2))) = P$

We first show that $\mathtt{v} \in (\mathtt{x}\ \mathtt{as}\ \mathtt{P}_1),\mathtt{P}_2 \Rightarrow \mathtt{V} \;/\; \alpha$ iff $\mathtt{v} \in (\mathtt{x}_b\ \mathtt{as}\ (\mathtt{P}_1,\ (\mathtt{x}_e\ \mathtt{as}\ \mathtt{P}_2))) \Rightarrow \mathtt{V}' \;/\; \alpha$ with $chop(\mathtt{V}) = chop(\mathtt{V}')$; thereby, the result follows from the induction hypothesis. We show the "only if" part since the converse is similar. From EP-As and EP-Cat, the former matching implies $\mathtt{v}_1 \in \mathtt{P}_1 \Rightarrow \mathtt{V}_1 \;/\; \alpha_1$ and $\mathtt{v}_2 \in \mathtt{P}_2 \Rightarrow \mathtt{V}_2 \;/\; \alpha_2$ where $\mathtt{v} = \mathtt{v}_1,\mathtt{v}_2$ and $\mathtt{V} = \{\mathtt{x} \mapsto \mathtt{v}_1\} \cup \mathtt{V}_1 \cup \mathtt{V}_2$ and $\alpha = \alpha_1, \alpha_2$. By using EP-As and EP-Cat, we obtain $\mathtt{v} \in (\mathtt{x}_b\ \mathtt{as}\ (\mathtt{P}_1,\ (\mathtt{x}_e\ \mathtt{as}\ \mathtt{P}_2))) \Rightarrow \mathtt{V}' \;/\; \alpha$ where $\mathtt{V}' = \{\mathtt{x}_b \mapsto \mathtt{v}_1, \mathtt{x}_e \mapsto (\mathtt{v}_1,\mathtt{v}_2)\} \cup \mathtt{V}_1 \cup \mathtt{V}_2$. From the definition of *chop*, we have $chop(\mathtt{V}) = chop(\mathtt{V}')$.

Other cases:

The result immediately holds or can be shown by a straightforward use of the induction hypothesis. □

C Closure Algorithms

The type inference algorithm mainly manipulates compound types, but, for calculating the final results, it uses a "conversion" operation that turns compound types into 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. This section defines an algorithm for the conversion operation. From this, we can derive, as a special case, an algorithm for the intersection operation introduced in Section 3.4.

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 W for each $\overline{X} \mapsto W$ in Π. Similarly, $\vdash \Pi$ means that the assumptions in Π are consistent—that is, that, for each $\overline{X} \mapsto W$ in Π, the unfolding of $\overline{X}$ is indeed convertible to the unfolding of W. The following rules define these relations.

$$\frac{\overline{X} \mapsto W \in \Pi}{\Pi \vdash \overline{X} \Rightarrow W} \qquad \text{(CONV-ST)}$$

$$\Pi \vdash \emptyset \Rightarrow \emptyset \qquad \text{(CONV-EMP)}$$

$$\Pi \vdash \epsilon \Rightarrow \epsilon \qquad \text{(CONV-EPS)}$$

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

$$\frac{\Pi \vdash \overline{X}_1 \Rightarrow W_1 \qquad \Pi \vdash \overline{X}_2 \Rightarrow W_2}{\Pi \vdash l(\overline{X}_1, \overline{X}_2) \Rightarrow l(W_1, W_2)} \qquad \text{(CONV-LAB)}$$

$$\frac{\forall \overline{X} \mapsto W \in \Pi.\quad \Pi \vdash \mathit{unf}(\overline{X}) \Rightarrow M(W)}{\vdash \Pi} \qquad \text{(CONV-CONS)}$$

The only essential work is done in the rule CONV-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 W. The other rules simply replace each compound state with the corresponding type state, accordingly to the mapping Π.

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

Proof
We first prove the "only if" part. Let $\Pi = \{\overline{X} \mapsto W \mid \overline{X} \in \mathit{dom}(M) \times \mathcal{P}(N) \times \mathcal{P}(N) \wedge W \in \mathit{dom}(M) \wedge \overline{X} \Rightarrow W\}$. The result follows by induction on the structure of B.

We then prove the "if" part by showing: for all t, if $\vdash \Pi$ and $\Pi \vdash \overline{A} \Rightarrow B$, then $t \in \overline{A}$ iff $t \in B$. This follows by induction on the lexicographic order on the pair of $h(t) + \mathit{isst}(B)$ (where $\mathit{isst}(T) = 0$ and $\mathit{isst}(X) = 1$) and the derivation of $\Pi \vdash \overline{A} \Rightarrow B$.
□

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 CONV-ST, we may not find the compound state $\overline{X}$ in the domain of Π. In this case, we generate a fresh state W and add a mapping $\overline{X} \mapsto W$ 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 W 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. Thus, we obtain the following lemma.

C.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 $X \cap \{Y_1 \ldots Y_m\} \backslash \{W_1 \ldots W_n\}$. We can remove Y_i from the compound state if $X <: Y_i$. Likewise, when $X <: W_i$, we can replace the whole compound state by a state associated with the empty set type $\emptyset$. Furthermore, when X and W_i denote disjoint sets, we can remove W_i from the compound state. (The disjointness can be checked by first calculating the intersection of X and W_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 appear usually to be relatively cheap, in our experience (Hosoya *et al.*, 2000).

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 $(T \ \mathsf{isect}\ P)$ and then convert the resulting compound type to a non-compound type. Proposition 3.4.3 can be derived as a corollary of Lemma C.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 $(T \ \mathsf{isect}\ P)$ contain only compound states of the form $X \cap \{Y\} \backslash \{\}$. Moreover, the unfolding of the compound state $X \cap \{Y\} \backslash \{\}$ 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.