Parametric Polymorphism for XML

HARUO HOSOYA
The University of Tokyo - Japan
and
ALAIN FRISCH
Lexifi - France
and
GIUSEPPE CASTAGNA
PPS (CNRS) - Université Denis Diderot - France

Despite the extensiveness of recent investigations on static typing for XML, parametric polymorphism has rarely been treated. This well-established typing discipline can also be useful in XML processing in particular for programs involving "parametric schemas," i.e., schemas parameterized over other schemas (e.g., SOAP). The difficulty in treating polymorphism for XML lies in how to extend the "semantic" approach used in the mainstream (monomorphic) XML type systems. A naive extension would be "semantic" quantification over all substitutions for type variables. However, this approach reduces to an NEXPTIME-complete problem for which no practical algorithm is known and induces a subtyping relation that may not always match the programmer's intuition. In this paper, we propose a different method that smoothly extends the semantic approach yet is algorithmically easier. The key idea here is to devise a novel and simple *marking* technique, where we interpret a polymorphic type as a set of values with annotations of which subparts are parameterized. We exploit this interpretation in every ingredient of our polymorphic type system such as subtyping, inference of type arguments, etc. As a result, we achieve a sensible system that directly represents a usual expected behavior of polymorphic type systems—"values of abstract types are never reconstructed"—in a reminiscence of Reynold's parametricity theory. Also, we obtain a set of practical algorithms for typechecking by local modifications to existing ones for a monomorphic system.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism; Data types and structure*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

General Terms: Algorithms, Design, Language, Theory

Additional Key Words and Phrases: XML, polymorphism, subtyping, tree automata

1. INTRODUCTION

Recently, static typing for XML processing has actively been investigated in the contexts both of concrete language designs (e.g., XDuce [Hosoya and Pierce 2003], XQuery [Fankhauser et al. 2001], ℂDuce [Benzaken et al. 2003], XMλ [Meijer and Shields 1999], XJ [Harren et al. 2005], XACT [Kirkegaard and Møller 2006], XHaskell [Sulzmann and Lu 2006b]) and of theoretical frameworks [Murata 2001; Milo et al. 2000; Alon et al. 2001; Maneth et al. 2005; Maneth et al. 2007]. All these works lack an important typing facility, namely the parametric polymorphism. This typing discipline—for parameterizing program fragments over types—has been exploited in many programming languages, such as ML [Milner et al. 1990; Leroy

et al. 1996; Appel and MacQueen 1991], Haskell [Peyton Jones et al. 1993], C++ [Stroustrup 2000], and GJ [Bracha et al. 1998], and already established as an important tool for code reusability. Not surprisingly, this usefulness can be extended to XML processing since many types for XML data (usually called *schemas*) are not stand-alone but defined in terms of other schemas given as parameters; it is natural to want to write programs involving such "parametrized schemas" and typecheck them. SOAP [Fallside and Lafon 2004] is a typical example. It is a message format for remote procedure calls providing a generic type for "envelopes" for enclosing transmitted data whose type is user-defined. For instance, consider writing a generic function for wrapping a given data d in a SOAP envelope, the result having the following structure:

```
<envelope>
  <header> example envelope </header>
  <body> d  </body>
</envelope>
```

We do not know the type of d at the moment of writing this function. In other words, the function must work for any type. For expressing this, we may first define a parametrized type SoapEnv(X) for SOAP envelopes, which may be written as

$$\text{SoapEnv}(X) = \texttt{envelope[header[String], body[}X\texttt{]]}$$

using the notation based on regular expression types [Hosoya et al. 2004] (where the form `tag[...]` corresponds to the XML structure `<tag>...</tag>` and the comma is a concatenation operation; we will be more precise on types in Section 2.1). Then, we may give to the wrapper function a polymorphic type, which may be written as:

$$\forall X.\ X \to \text{SoapEnv}(X)$$

There are many other parametrized schemas than SOAP, and they are indeed ubiquitous, though parametrization has been done by using ad-hoc tricks since current XML schema languages do not have an official support. Such schemas implement parametrization by allowing any type of data in particular places (e.g., specified by `Any` in DTD schemas [Bray et al. 2000]) or by using a macro expansion feature (called *parameter entity* in the case of DTD). For example, XHTML 1.1 [Altheim and McCarron 2001] allows user-specific types of data to be contained and a specialization of XHTML 1.1 that can include both mathematical formulas in MathML [Ausbrooks et al. 2003] and vector graphics in SVG [Jackson et al. 2002] is provided [Ishikawa 2002]. More other examples of "customizable" schemas include DocBook [OASIS 2002], ATOM [Nottingham and Sayre 2005], XBEL [Python XML Special Interest Group 1998], and ODF [OASYS 2007].

Thus, the need of polymorphism for XML is clear. However, it has almost never been studied formally. This is probably because most of current research on type systems for XML uses a "semantic" approach and this approach is not trivial to extend with polymorphism. To see the difficulty, let us look more closely at the definition of subtyping, which is a crucial part in XML typechecking. In the monomorphic case, we first give the semantics $[\![T]\!]$ of each type T as a set of documents conforming to the type, and then define the subtype relation by the

subset relation between the semantics of two given types:[1]

$$T \leqslant U \quad \Longleftrightarrow \quad [\![T]\!] \subseteq [\![U]\!]$$

Although the subtype relation, which is equivalent to the tree automata containment problem [Hosoya et al. 2004], has a high worst-case complexity (EXPTIME-complete), several algorithms are known to work well in practice (some based on "top-down, on-the-fly strategies" [Hosoya et al. 2004; Suda and Hosoya 2005; Frisch 2004] and other based on binary decision diagrams [Tozawa and Hagiya 2003]). In the polymorphic case, on the other hand, a naive extension of the semantic approach is to allow type variables to be embedded in types and then quantify the subset relation over all substitutions of types for the type variables:

$$T \leqslant U \quad \Longleftrightarrow \quad \forall S.\, [\![[X \mapsto S]\, T]\!] \subseteq [\![[X \mapsto S]\, U]\!] \qquad (\spadesuit)$$

However, the subtype relation defined in this way is problematic since (1) the algorithmics is substantially more difficult than the monomorphic case and (2) this definition yields unintuitive relations. Specifically, the problem of deciding this subtyping can be reduced to the satisfiability problem for set constraint systems with *negative constraints* [Aiken et al. 1995; Gilleron et al. 1999]; this problem is known to be NEXPTIME-complete [Stefénsson 1994] and, so far, no practical algorithm is known (though it is still an open question whether the above subtype problem could be reduced to an easier problem). In addition, we have noticed some tricky behavior that makes us believe that even if it was possible to solve such a problem, the resulting subtyping relation might be unintuitive from the programmers viewpoint. To show an example of this issue, let `a[]` denote a type representing the singleton set consisting of the value `<a></a>`, the vertical bar | a union operation, and the type A represent the complement of the type `a[]` (which is possible to define since types are essentially regular). Then, with the above definition ($\spadesuit$), the following relation holds:

$$\texttt{l[a[]],}X \quad \leqslant \quad \texttt{l[a[]],}A \quad | \quad \texttt{l[}X\texttt{],a[]}$$

We do not show a concrete proof here (deferred to Appendix B; roughly, it is obtained by a set-theoretical analysis of cases on the type S to be substituted for X). Note, however, that the above example is "strange" since the type variable X occurs in irrelevant positions on both sides, and such a behavior appears to be the hard core in the algorithmics (this is not the only example of strange behaviour; another one based on a "finite" type rather than a "singleton" type is described in Appendix B.). And even if the algorithm were easy, it seems, in any case, quite hard to explain a programmer the intuition of a subtyping relation in which the same type variable may appear in unrelated positions on both sides of the relation.

In this paper, we propose a different method for constructing a polymorphic type system that (1) retains the original spirit of the semantic approach, (2) eliminates tricky cases observed in the above-mentioned naive extension retaining only

[1]The object-oriented language community tends to take the term "polymorphism" as the *subtype polymorphism*. In this paper, however, we follow the tradition of the functional language community, referring by polymorphism to only *parametric polymorphism* and by monomorphism to a type system possibly with subtyping but *without* parametric polymorphism.

relations that should match programmer's intuition, and (3) yields practical typechecking algorithms. The key idea in our approach is to take a type variable not as a "place holder" for which a concrete type is substituted, but as *markings* in a document for indicating their parametrized subparts. For example, we interpret the polymorphic type SoapEnv(X) as a set of documents of the form

```
<envelope>
  <header> ... </header>
  <body> d  </body>
</envelope>
```

where the subpart d (which itself can be any fragment of documents) is marked by X. (We use the term "subpart" for a consecutive sequence of elements appearing in a given document.) Using this interpretation of types, we define the subtype relation essentially by the subset relation, without involving quantification. (We actually need to add a little more flexibility, as we will discuss in Section 2.2.) This indeed removes tricky cases observed above, and thus allows us to reduce the subtyping problem simply to a slight variation of the tree automata containment problem (Section 4.4) and to obtain a reasonably efficient algorithm by incorporating various known algorithmic techniques [Hosoya et al. 2004; Tozawa and Hagiya 2003; Suda and Hosoya 2005; Frisch 2004].

We use the marking technique not only as a simple tweak to make the algorithmics easier, but we push forward this technique to designing a whole type system with a sensible meaning. We will present a minimal XDuce-style calculus with an operational semantics where run-time values carry around explicit markings to indicate parametrized subparts and a type system that captures the flow of such markings. Since we do not allow a new marking to be created during evaluation, any parametrized subpart in a result value of a function must come from some parametrized subpart in the input value. In other words, the type system directly represents a usual expected behavior of a polymorphic type system—"a value of abstract type is never reconstructed"—in a reminiscence of Reynold's parametricity theory [Reynolds 1983]. Note that we would not have such a property if we adopted the quantification-based subtyping (♠) since, by using the example relation given above, a value that has the concrete type `a[]` could be given the abstract type X.

Another by-product from our interpretation of polymorphic types is that the semantics of types can actually be formalized in the same way as to that of *pattern matches*. That is, what both of these do is, given a tree value, first to check conformance and then to return an association of (term or type) variables to subtrees. This coincidence brings two additional benefits. First, we can economize the language specification involving both polymorphic types and pattern matches by sharing many definitions related to these two. Second (and more importantly), we can transfer previously known algorithmic techniques for pattern matches to similar problems related to polymorphic types. Specifically, at applications of polymorphic functions, typechecker needs to infer type arguments to the applications for avoiding verbose and obvious type annotations. And relevantly, we need to perform a form of ambiguity check on formal parameter types for ensuring the existence of a minimum solution when inferring type arguments. Both of these can be obtained by slight modifications to existing algorithms [Hosoya 2003], as shown

in Section 4.5 and 4.6. All these simplifications come at a price: by abandoning a semantic approach to polymorphism to embrace a marking-based syntactic one, it becomes impossible to follow the technique of Frisch et al. [Frisch et al. 2008] to extend the polymorphic subtyping relation to function spaces. The resulting inference for type arguments does not work with function types and thus the present formalization does not include higher-order functions. We do consider it as a significant limitation of our work but we have good hopes to be able to cope with them in future work by implementing, as we are already doing, the ideas described in the Conclusion (Section 7).

The techniques presented in this paper have already been implemented in the most recent version of XDuce (0.5.0) freely available through

`xduce.sourceforge.net`

and the readers are welcome to try it out.

This paper is the full version of our previous publication with the same title at POPL'05 [Hosoya et al. 2005], the main additional materials being the full proofs of theorems and two additional algorithms (linearity in Section 4.2 and substitution in Section 4.3).

The rest of the paper is organized as follows. In the next section, we illustrate basic ideas for constructing our polymorphic type system. Section 3 formalizes the type system and Section 4 describes a set of algorithms needed for typechecking. In Section 5, we explore some possibilities for extensions. Section 6 discusses related work and Section 7 concludes this paper and hints at future work. Appendix A gives some additional algorithms omitted from the main part of the paper. Appendix B discusses in more detail the above-mentioned tricky example that can be built in the "place-holder" subtyping.

2. BASIC IDEAS

In this section, we give an informal, intuitive explanation for our polymorphic type system. In Section 2.1, we review a monomorphic system, in the XDuce syntax for concreteness, and, in Section 2.2, we illustrate our ideas for adding marking-based polymorphism such as subtyping, type inference, and ambiguity check.

2.1 Monomorphic System

We start with considering a minimal, functional language designed for XML processing, in the notation of XDuce [Hosoya and Pierce 2003]. The language provides *values* as fragments of XML documents (they are the only values to be manipulated at run time), *types* based on XML's schemas for describing structures of values, and *pattern matching* as a main programming feature for analyzing and deconstructing values. This paper aims at formalizing the core idea for dealing with polymorphism and concentrates only on the treatment of element structures of XML data. We leave other extensional features, such as XML attributes and higher-order functions, for future work. For example, consider the following XDuce program for searching a database for a data entry that has a specified key.

```
type BibDB = db[BibEntry*]
type BibEntry = entry[key[String], content[Bib]]
```

```
type BibResult = found[Bib] | notfound[]

fun search (String as k)(BibDB as d) : BibResult =
  match d with
    db[Any as l]  ->  iter(k)(l)

fun iter (String as k)(BibEntry* as l) : BibResult =
  match l with
    ()  ->  notfound[]
  | entry[key[String as current_key], content[Any as c]], Any as rest
        ->  if k = current_key then found[c]
            else iter(k)(rest)
```

First, the type `BibDB` is defined to be a type for the values of label `db` that contain a repetition of data of type `BibEntry`, which is in turn defined as a type for the values of label `entry` including a `key` with a string and a `content` with a value of type `Bib`. We assume the type `Bib` to be defined somewhere else. Then, the type `BibResult` is defined to be a type representing either a label `found` with a `Bib` value or a label `notfound` with no content.

The type definitions are followed by two definitions of functions `search` and `iter`. The function `search` takes a string `k` and a value `d` of type `BibDB` as arguments and returns a value of type `BibResult`. The body is a pattern match on `d`. It matches any value of type `db[Any]` (where `Any` matches any value) and binds the variable `l` to the subpart of the input value corresponding to `Any`, i.e., the content of `db`. When the matching succeeds, the function proceeds to evaluate the corresponding body, where it calls the function `iter` with arguments `k` and `l`. The function `iter` takes a string `k` and a value `l` of type `BibEntry*` as arguments and returns a value of type `BibResult`. The body is a pattern match on `l` with two clauses. The first clause matches an empty sequence value `()` and returns a value `notfound[]`. The second clause matches a non-empty sequence that begins with an `entry` containing a `key` and a `content` labels. In the latter clause, the pattern extracts, from a matched value, the content `current_key` of the `key` label, the content `c` of the `content` label, and the remainder sequence `rest` after the first `entry` label. In the corresponding body, if the user-specified `k` and the extracted `current_key` are equal, then the value `c` enclosed by a `found` label is returned; otherwise we continue with the remainder sequence.

In general, XDuce values (written `v`) are sequences of labeled values (`l[v]`), or string values. Types (written `T`) are regular expressions over labeled types (`l[T]`) or `String` type.[2] Thus, types can also be concatenations (`T1,T2`), unions (`T1|T2`), repetitions (`T*`), and the empty sequence type (`()`). We abbreviate `l[()]` by `l[]`. As usual, types can be defined as type names and, in particular, they can be defined recursively for describing arbitrarily nested structures. (For guaranteeing regularity of types, we require recursive occurrences of type names to be enclosed by labels. For

[2]The actual XDuce implementation allows integers, etc., for arithmetic computation, but we omit them here for brevity.

example, a type definition like `T = a[T]|()` is allowed while `T = (a[],T,b[])|()` is not. See [Hosoya et al. 2004; Hosoya 2003] for a precise definition.)

The main part of a program is a set of (recursively defined) functions with an explicit declaration of argument types and a result type. Each defined function contains a body expression, where expressions (written `e`) can be variables (`x`), function calls (`f(e)`), value constructors (labeling `l[e]`, concatenation `e1,e2`, the empty sequence `()`, and string constants), and pattern matches of the form:

```
match e with P1 -> e1 | ... | Pn -> en
```

(We omit here describing other standard expressions such as `if_then_else`.) A pattern match expression first tries matching the input value `e` against the patterns `P1`, ..., `Pn`, and then evaluates the body expression corresponding to one of the matching patterns under the bindings resulted from the matching.

Patterns have exactly the same structure as types except that variable binders of the form `... as x` can be inserted in their subparts. A pattern is matched by values that have the type of the pattern (that is, the type obtained after eliminating all the binders from the pattern). The matching operation returns bindings of each pattern variable to the value's subpart corresponding to the binder of the variable. We restrict patterns by usual "linearity" requirement to ensure them to yield bindings of exactly the same set of variables for any input value. Also, various design choices are possible for the semantics of patterns when they can have more than one possible match for a given input value; since this is a rather orthogonal issue from the subject here, we choose presentationally the simplest design, namely, the nondeterministic semantics, where we take an arbitrary match from all possible ones. (A popular "first-match" semantics can be simulated by the use of "difference" operation on patterns, which we comment on in Section 5.1. Also, detailed discussions on design and implementation techniques for pattern matching can be found in [Hosoya 2003; Hosoya and Pierce 2002; Frisch et al. 2002; Vansummeren 2003].)

Typechecking XDuce programs is mostly straightforward. We basically construct types of expressions, in a bottom-up, syntax-directed way. For example, the expression `()` has type `()`; if `e` has type `T`, then `l[e]` has type `l[T]`; if `e1` and `e2` have type `T1` and `T2` respectively, then `e1,e2` has type `T1,T2`; and so on. Subtype check is performed at critical places to ensure type safety, including function calls (subtype check between the actual argument type and the formal parameter type), function bodies (between the body's type and the declared result type), and pattern matches (between the input's type and the union of the types of the patterns, a.k.a. exhaustiveness check). As mentioned in the introduction, we define subtyping in a semantic way. That is, using the standard "conformance" relation of types (i.e., "a value `v` has type `T`"), we say that a type `S` is a subtype of `T`, written $\mathtt{S} \leqslant \mathtt{T}$, if and only if every value of type `S` is also of type `T`. This way of defining subtyping is quite powerful, and known to be useful in particular for exploiting flexibilities of XML data in processing programs (e.g., one can forget ordering among the input data when unnecessary; also, one can merge data from several different sources and extract parts that are common in these). Moreover, the feasibility is guaranteed by the exact correspondence to finite tree automata, whose containment problem is known to be decidable. (Details on usefulness, correspondence to tree automata, and a containment algorithm can all be found in [Hosoya et al. 2004].)

It remains to explain how we obtain types for the variables bound in patterns. For this, we employ a mechanism for inferring those types from the input type and the patterns. (For example, `iter` function given above has a pattern match and we infer types for the variables `current_key`, `c`, and `rest` using the type of the input `l`.) The inference is guaranteed to have a property called *local precision*. That is, for each bound variable `x` appearing in a pattern `P`, the type inferred for `x` contains *all* and *only* values that may be bound to `x`, with the assumption that all and only values from the input type may be matched against the pattern. As one can see, this typing is not syntactic. And indeed, we use a slightly involved algorithm for constructing types assigned for pattern variables. More discussions on the type inference algorithm can be found in [Hosoya and Pierce 2002; Hosoya 2003; Frisch et al. 2002]. As we will describe in Section 3.4, this inference scheme for patterns turns out to be reusable for the inference of type arguments (types to be passed to polymorphic functions at application).

2.2 Polymorphic System

2.2.1 *Syntax for polymorphism.* In order to add polymorphism to the system described above, we first need to extend the syntax. First, types can contain type variables, and, accordingly, type names can take type parameters. Second, each function definition can declare type parameters, to which the parameter types and the result type can refer. In principle, function applications also take type arguments to instantiate those type variables. However, we automatically infer them, as we will describe later.

As an example, let us write a program that generalizes the example in Section 2.1 so that it now works with any type for data contents.

```
type DB{X} = db[Entry{X}*]
type Entry{X} = entry[key[String], content[X]]

type Result{X} = found[X] | notfound[]

fun search {X}(String as k)(DB{X} as d) : Result{X} =
  match d with
    db[Any as l]  ->  iter{X}(k)(l)

fun iter {X}(String as k)(Entry{X}* as l) : Result{X} =
  match l with
    ()  ->  notfound[]
  | entry[key[String as current_key], content[Any as c]], Any as rest
        ->  if k = current_key then found[c]
            else iter{X}(k)(rest)
```

Changes from the previous program are as follows: what used to be `Bib` is now replaced by the type variable `X`; all the definitions of type names and functions now take the type parameters, written `{X}`; and, finally, all the references to type names are now added type arguments, also written `{X}`. For the sake of explanation, we also show type arguments to polymorphic functions, which actually need not be written explicitly. (There are some restrictions on where type variables can occur;

we will detail them in Section 3.2. Also, our patterns cannot contain type variables so that their behaviors do not depend on type arguments; also see Section 3.2.) As a result of parametrizing the program in this way, we can now search databases with any types of data contents.

```
val contactDB : DB{Contact} = ...
            (* load a contact database file *)
val result1 : Result{Contact} =
            search{Contact}("HosoyaHome")(contactDB)

val bibDB : DB{Bib} = ...
            (* load a bibliography database file *)
val result2 : Result{Bib} =
            search{Bib}("HosoyaPierce00")(bibDB)
```

Note that the types for the results retain the information that their data contents have specific types (`Contact` or `Bib`). If we did not use polymorphism and instead gave the `Any` type for data contents, we would not get such precise type information for the results.

2.2.2 *Intuition behind typing.* Now, the question is: how can we typecheck such a polymorphic program? Or, more fundamentally, what property should we prove about a polymorphic program? For the monomorphic case, given a function of type, for example[3],

$$\texttt{BibEntry*} \rightarrow \texttt{BibResult}$$

the typechecking scheme described in Section 2.1 tries to prove that, whenever the input value has type `BibEntry*`, the result (if any) has type `BibResult`. In the polymorphic case, given a function of type

$$\forall \texttt{X}.\ \texttt{Entry\{X\}*} \rightarrow \texttt{Result\{X\}}$$

we would naturally like to prove that, for any type `S`, whenever the input value has type `Entry{S}*`, the result has type `Result{S}`. Although we certainly want this property to hold, this would bring us to construct a type system that requires overly ambitious algorithmics, e.g., a subtyping relation defined by quantification over all substitutions, as discussed in the Introduction.

Instead, we adopt a stronger condition as an intended property that the type system should try to prove. This consists of two parts:

(1) whenever the input value has type `Entry{Any}*`, the result has type `Result{Any}`, and

(2) *any* subpart corresponding to `X` in the result value (i.e., the content of the `found`) must come from *some* subpart corresponding to `X` in the input value (i.e., the content of some `content` element).

Let us call these conditions "safety property" from now on.

[3]We have only first-order functions, so the above is just an informal notation for functions from `BibEntry*` to `BibResult`.

These conditions will remind many readers of the well-known parametricity property of the polymorphic lambda calculus [Reynolds 1983]. While a usual treatment is to define a calculus and prove this property as a theorem, ours is to formalize a system that directly embodies the property.

Our key idea is to employ a "marking" semantics outlined as follows. First, each value carries a *marking*, that is, some subparts of the value are marked by type variables. Accordingly, we interpret a polymorphic type (which contains type variables) by a set of values whose subparts corresponding to the type variables are marked with those type variables. For example, the input type `Entry{X}*` of the function `iter` represents a set of marked values, each structured as a list of n `entry` elements ($n \geq 0$) in the following way

```
entry[key["..."], content[v1]],
...
entry[key["..."], content[vn]]
```

where each v_i is marked by `X` (for $i = 1, \ldots, n$). Similarly, the result type `Result{X}` denotes a set of marked values that have either the form `found[`v`]` where v is marked by `X` or the form `notfound[]` with no mark.

The operational semantics of programs is defined in such a way that operations in those programs manipulate marked values, preserving all markings from the inputs to the output. For example, a labeling expression `l[e]` adds the label `l` to the marked value resulted from evaluating `e`, preserving the original markings in `e`'s result; likewise, a concatenation expression `e1,e2` combines two marked values resulted from evaluating `e1` and `e2`. A pattern match takes a marked value and extracts its subparts for forming bindings, preserving the markings that were present in those subparts of the input marked value.

Then, the job of the type system is to guarantee that our operational semantics "respects" our interpretation of types. That is, for a function, we verify that, whenever the function body starts with a marked value inhabiting in the input type and performs operations as specified by the body expression, then (if it terminates) it results in a marked value that conforms to the declared result type. Since no operation modifies a subpart marked with `X` if its result preserves this marking, then the whole function body never modifies any subpart whose marking is preserved in the final result. Thus, we attain the above-mentioned safety property.

The construction of our polymorphic type system is mostly similar to the monomorphic one. First, we use exactly the same typing rules for value construction expressions. For example, if `e` has type `T`, then `l[e]` has type `l[T]`. This rule works since all markings in the values of type `T` are also present in the corresponding subparts of the values of type `l[T]`. Also, for pattern matches, we use the same specification as before for the inference of types for pattern variables except that all values mentioned there now may carry markings. Non-trivial changes from the monomorphic system are in subtyping and in checking polymorphic function applications; we describe these below.

It would be ideal if we could define subtyping exactly in the same way as before: `S` is a subtype of `T` if any marked value of type `S` is also of type `T`. However, this definition turns out to be too strong. For example, we often need to give a value of (possibly polymorphic) type `S` to a generic printer function that might expect a

value of type `Any`. In such a situation, a subtype relation like the following would be useful.

$$\texttt{X} \leqslant \texttt{Any}$$

However, this would not be allowed by the above simple definition of subtyping since the right hand side would permit no marking to be present. Nevertheless, this subtype relation is reasonable since it fulfills our safety property. Indeed, in order for the safety property to hold, each subpart of the result (i.e., the right hand side) corresponding to `X` must be identical to some subpart of the input (i.e., the left hand side) corresponding to `X`; but this vacuously holds since such a subpart corresponding to `X` does not exist in the result.

Therefore we need to slightly relax the definition of subtyping. What is observed in the last paragraph is that subtyping transfers a marked value from one context to another, where if the latter context requires fewer markings, the transfer is still safe. From this, we obtain the following new definition:

> `S` is a subtype of `T` if, for any marked value `v` of type `S`, there exists a marked value `w` of type `T` such that `v` and `w` are identical except that some of the marks in `v` can be absent in `w`.

This relaxation of definition is somewhat analogous to the standard "promotion" rule of a type variable to its upper bounds, which often appears in type checking algorithms of systems that combine subtyping and polymorphism (cf. $F_{\leqslant}$ [Cardelli et al. 1994]).

The typing rule for polymorphic function applications is as usual. That is, we substitute type arguments for type variables in the type of the parameter and check that the resulting type is a supertype of the type of the argument. Then, the whole function application is given the declared result type where the type parameters are replaced by the type arguments. For example, consider the definition of `search` function shown in the beginning of this subsection

```
fun search {X} (String as key) (DB{X} as d) : Result{X} = ...
```

and one of its applications:

```
search{Contact}("HosoyaHome")(contactDB)
```

We perform the above-mentioned check for both arguments. In particular, we check that the second argument's type `DB{Contact}` is a subtype of $[\texttt{X} \mapsto \texttt{Contact}]\ \texttt{DB\{X\}} = \texttt{DB\{Contact\}}$, which holds true since they are identical (we use the notation $[\texttt{X} \mapsto \texttt{S}]\texttt{T}$ to denote the substitution of `S` for `X` in `T`). We then give the type $[\texttt{X} \mapsto \texttt{Contact}]\texttt{Result\{X\}} = \texttt{Result\{Contact\}}$ to the result of the application. This standard typing rule fits well our semantic view of polymorphism. Indeed, we can apply our safety property to the `search` function and obtain the same result. First, from the declared parameter type, any input value (we consider only the second argument value) must be structured as a `db` element containing n `entry` elements ($n \geq 0$) in the following way:

```
db[entry[key["..."], content[ v1 ]],
   ...
   entry[key["..."], content[ vn ]]]
```

Also from the declared result type, any result value must have the form `found[`v`]` or `notfound[]`. Note that, when a subpart corresponding to `X` exists in the result value, the subpart is the value v in the `found` element. Since our safety property stipulates that the value v must come from some subpart of the input corresponding to `X`, we conclude that $v = v_i$ for some $1 \leq i \leq n$. As a particular case, when the input value has type `contactDB`, each v_i has type `Contact`. In this case, if the result has the form `found[`v`]`, this must have type `found[Contact]`, or else the result must have type `notfound[]`; thus we conclude that the result has type `Result{Contact}`.

2.2.3 *Type inference.* As in our example, type arguments given to polymorphic function applications are usually obvious and tedious to write. Therefore we provide an automatic scheme that infers type arguments from argument types and parameter types. The inference is specified to compute a minimum type argument that fulfills the above-described requirement between the argument type and the parameter type. Going back to our example, this resorts to compute a minimum type `T` such that

$$\texttt{DB\{Contact\}} \quad \leqslant \quad [\texttt{X} \mapsto \texttt{T}]\ \texttt{DB\{X\}}.$$

Hence, we obtain `T` = `Contact`. (Note that, with no function types, the minimum type argument always minimizes the result type since type parameters occur only in positive positions. We would need to change our inference scheme if we supported higher-order functions.)

A question that may arise here is: does such a minimum type always exist? Within the current setting, the answer is *no*. For example, there does not exist a minimum `T` such that

$$\texttt{a[c[]],b[d[]]} \leqslant [\texttt{X} \mapsto \texttt{T}]\ \texttt{(a[Any],b[X] | a[X],b[Any])}.$$

Indeed, both `c[]` and `d[]` are *minimal* solutions, but neither is smaller than the other. The problem here is that the parameter type `(a[Any],b[X]|a[X],b[Any])` allows two ways of marking `X` on subparts of the value of the argument type `a[c[]],b[d[]]`. Our approach is to reject such an ambiguous parameter type. More precisely, we say that:

> A type `U` is *ambiguous* whenever `U` contains two distinct marked values that are identical if all markings are removed.

If every parameter type is unambiguous in this sense, then we can prove that a minimum type argument specified before always exists (Section 3.4).

2.2.4 *Type constraint.* In our type system, we additionally support type variables with type constraints of the form `T as X` (the constraint `T` itself can contain type variables other than `X`; see Section 3.2). This form of type denotes values of type `T` with a marking of `X` at the top. Thus, a bare type variable `X` may be considered syntactic sugar for `Any as X`. To see the usefulness of type constraints, consider the following example. Suppose that we need to transform a value of type `ol[Li*]` to another of type `ol[Li+]`. A possible semantics is to check whether the content of the input is empty or not, and emit the empty sequence if so, otherwise the input itself (therefore the output type is actually `ol[Li+]?`). Although this

processing is simple to write with a pattern match, it becomes quite tedious when exactly the same processing is required for several other types (e.g., `dl[(Dd|Dt)*]` to `dl[(Dd|Dt)+]?`). While monomorphism would mandate this tediousness, polymorphism obviates it. The following function achieves this (where `~[T]` matches any label with content of type `T` and `AnyElm` abbreviates for `~[Any]`).

```
fun adjust {X} ((~[AnyElm*] as X) as x) : (~[AnyElm+] as X)? =
  match x with
    ~[] -> ()
  | val y as ~[AnyElm+] -> y
```

This function typechecks according to the typing rules described before. The most crucial part is to infer, for the variable `y`, the type `(~[AnyElm+] as X)`. This is deduced from `x`'s type and the `y`'s constraint. Then subtyping is checked between the obtained type for `y` and the result type `(~[AnyElm+] as X)?`. The fact that the function typechecks is reasonable in terms of our safety property since any subpart corresponding to `X` in a result value has the form `l[v]` with `v` non-empty, and it always comes directly from the input, which itself corresponds to `X`. We can invoke this function with different types, as desired. For example, if we apply it to a value `w` of type `ol[Li*]`, then our type inference instantiates the type variable `X` by `ol[Li*]`, which is substituted for `X` in the declared result type. Here, the type `ol[Li*]` is not a subtype of the constraint `~[AnyElm+]` in the declared result type. But, in such a case, our (non-standard) substitution yields a subtype of the constraint type, rather than raising an error; thus, we obtain the result type to be `ol[Li+]?`. (More precisely, our substitution performs intersection between the instance type `ol[Li*]` and the constraint type `~[AnyElm+]`; see Section 3.3. Type constraints have an effect somewhat similar to *bounded quantification* [Cardelli et al. 1994]. Both are, of course, not exactly the same. For example, different occurrences of type variables may have different constraints. Also, constraints are *not* bounds—substitution does not check a type against the constraint, but instead ensures the resulting type to be a subtype of it.)

3. FORMAL SYSTEM

The next two sections formalize the polymorphic type system outlined in the previous section. In this section, we focus on the semantic aspects of the formal system while in the next section we address algorithmic problems.

The surface language that we have seen, which we call *external* language, would be quite complicated to directly deal with. Therefore, in the formalization, we instead treat an *internal* language where two major simplifications are made. First, instead of sequence values and regular-expression-based types, we use a *binary representation* of values and types in the style of [Hosoya et al. 2004; Frisch et al. 2002]. Second, as mentioned before, the behaviors of polymorphic types and patterns are almost the same (while polymorphic types give markings of type variables to subparts of values, patterns yield bindings of term variables to subparts of values). Therefore we share many definitions related to these two.

3.1 Values and Marking

External values are sequences of labeled values or string values. Internally, we represent those values by using only labels and pairs. We assume a set of *labels*, ranged over by a, b, ..., that contains at least a special label ν. *(Internal) values* are then defined by:

$$v ::= a \mid (v, v)$$

We translate external values to internal ones in a way similar to Lisp's encoding of lists by `cons` and `nil`. Each external value `v` that is not an empty sequence is translated to a pair (v_h, v_t) where v_h and v_t are the translation of `v`'s first element and that of the remainder sequence, respectively. If the first element is a labeled value, then v_h is another pair (a, v_c) where a is the label of the element and v_c is the translation of its content. If the first element is a string, then v_h is the label representing the string. An external value that is an empty sequence is translated to ν.

As external ones do, internal values carry marks of variables on their subparts. We formalize a marked value by an (unmarked) value with separate information that indicates which intermediate node is given a mark. We assume a set $\mathcal{X}$ of *variables*, ranged over by x, y, ... Variables are divided into two sets, *term variables* and *type variables*. *Paths* are defined by:

$$\pi ::= \epsilon \mid 1\pi \mid 2\pi$$

We take a path to be a function that maps a value to its subnode locating at the path from the root: inductively, $\epsilon(v) = v$ and $(i\pi)(v_1, v_2) = \pi v_i$ for $i = 1, 2$. A *marking* V is a relation between variables and paths, written $\{(x : \pi), \ldots\}$ and a *marked value* is a pair (v, V) of a value and a marking. We also use metavariables U and W to range over markings. For example, the external value

```
entry[key["abc"], content["ABC"]]
```

where the part `"ABC"` is marked x is translated to the internal value

$$((\texttt{entry}, ((\texttt{key}, (\texttt{abc}, \nu)), ((\texttt{content}, (\texttt{ABC}, \nu)), \nu))), \nu)$$

with the marking $\{(x : 122121)\}$ (pointing to the subpart `ABC`). The marked internal value is depicted in Figure 1 where the left and the right child of each node is drawn by a down and a right arrows, respectively, and a marking is represented by a white arrow.

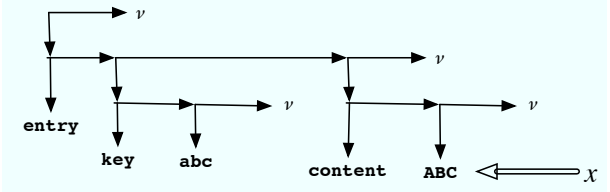

Fig. 1. A marked internal value

We let X range over finite sets of variables. We define the restriction $V|_X$ of V by X to be $\{(x:\pi) \in V \mid x \in X\}$. We write $\mathbf{dom}(V) = \{x \mid (x:\pi) \in V\}$.

When we compose two marked values in a pair, we want the original marks to sink down to deeper places. To express this, we define *push-down* of V by π, written πV, to be $\{(y:\pi\pi') \mid (y:\pi') \in V\}$ (where $\pi\pi'$ is the concatenation of the paths π and π'). Then, the pairing $(v_1, V_1) \otimes (v_2, V_2)$ of two marked values is defined as $((v_1, v_2), (1V_1 \cup 2V_2))$. Likewise, when we extract a subnode from a marked value, we want the original marks to float up to shallower places. For this, we define *pull-up* of V by a path π, written $\pi^{-1}V$, to be $\{(y:\pi') \mid (y:\pi\pi') \in V\}$. Then, the *extraction* $\pi(u, U)$ of a marked value from a path π is defined as $(\pi u, \pi^{-1}U)$. Note that we never lose any marks by push-down, whereas we may by pull-up. Hence, it is always that $\pi^{-1}(\pi V) = V$ and $\pi(\pi^{-1}V) \subseteq V$, but not necessarily $\pi(\pi^{-1}V) = V$.

3.2 Types

External types are regular expressions on labeled values, with recursive top-level type definitions. Internally we represent them,, as we did for values, by labels and pairs. We assume a set of *type names* ranged over by s. A *type definition* Δ is a finite mapping from type names to *(internal) types*, the latter being defined by the following productions:

$$p ::= a \mid (p, p) \mid p \,|\, p \mid x : p \mid \top \mid \bot \mid s$$

In words, a type is either a label, a pair, a union, a variable with a type constraint, a universal type, an empty type, or a type name. We write $\mathbf{var}(p)$ to denote the set of variables occurring in p or in any type associated with a type name reachable from p. We also use metavariables q and r to range over types. We make two restrictions on types. First, any recursive use of a type name must go through a pair. Second, for every occurrence of $x : p$, we require that $x \notin \mathbf{var}(p)$. The first condition is a classic contractivity condition that rules out meaningless terms such as, for instance $x = x \,|\, x$. The second condition ensures that a type never generates a marking where two marks of the same variable occur in the same path, i.e., both $(x:\pi_1)$ and $(x:\pi_1\pi_2)$ are in V for some x, π_1, and π_2. (Allowing the same variable in the same path would correspond to *F-bounded* polymorphism [Canning et al. 1989], but we do not further pursue this direction in this paper.)

Given a type definition Δ, the semantics of types is described by the *matching* relation, written $(v, V) \lhd p$, read "marked value (v, V) matches p" or "value v matches p and yields a marking V." The matching relation is defined by the following set of rules.

$$\textsc{MCon}\ \frac{}{(a, \emptyset) \lhd a} \qquad \textsc{MPair}\ \frac{(v_i, V_i) \lhd p_i \quad \forall\, i = 1, 2}{(v_1, V_1) \otimes (v_2, V_2) \lhd (p_1, p_2)} \qquad \textsc{MAlt}\ \frac{(v, V) \lhd p_i \quad \exists\, i = 1, 2}{(v, V) \lhd p_1 | p_2}$$

$$\textsc{MVar}\ \frac{(v, V) \lhd p}{(v, V \cup \{(x:\epsilon)\}) \lhd x : p} \qquad \textsc{MAll}\ \frac{}{(v, \emptyset) \lhd \top} \qquad \textsc{MName}\ \frac{(v, V) \lhd \Delta(s)}{(v, V) \lhd s}$$

MCon states that a constant matches itself and yields no marking; a pair matches component-wise and yields the pairing of the respective markings (Pair); the non-

deterministic semantics of unions requires that at least one of the two types forming the union must match, and the corresponding marking is yielded (MAlt); a variable adds a mark to the root of the marking yielded by matching the value against the type constraint (MVar); every value matches the top type $\top$ (MAll); finally, a type name yields the same marking as the type associated to it.

Translation from external types to internal types is analogous to the encoding of values. Roughly, each type representing non-empty sequences is translated to a pair type (p_h, p_t), where p_h and p_t are the translation of the type for the first element and that for the remainder sequence, respectively. If the first element is a type labeled by a, then p_h is (a, p_c) where p_c corresponds to the content type. If the first element is a `String` type, then p_h is $\top$. The empty sequence type is translated to ν. The other external constructs are translated in a straightforward manner using the corresponding internal constructs: an alternation translates to an alternation, a repetition to a recursion, a variable to a variable, `Any` type to $\top$, and so on. For example, the external type

```
db[entry[key[String], content[X]]*]
```

can be translated to the internal type

$$((\texttt{db}, s_1), \nu)$$

with definitions

$$\begin{aligned} s_1 &\mapsto \nu \mid ((\texttt{entry}, s_2), s_1) \\ s_2 &\mapsto ((\texttt{key}, s_3), ((\texttt{content}, \texttt{X} : \top), \nu)) \\ s_3 &\mapsto (\top, \nu) \end{aligned}$$

(Note that the bare variable `X` is encoded by $\texttt{X} : \top$ since `X` is an abbreviation for `Any as X`.) A general translation scheme is similar to the monomorphic case and can be found in the literature (e.g., [Hosoya et al. 2004]).

A type containing only term variables is called a *pattern* and a type containing only type variables is called *polymorphic type* or *polytype*. Note that we do not allow patterns to contain any type variables. One would consider that this might be a little inconvenient since, in the monomorphic system, patterns are a superset of types and one can freely embed types inside patterns. However, allowing type variables in patterns adds substantial complexity to the type and run-time system, since in that case the behavior of pattern matches might depend on the types to be passed at run time. Although such a feature would be interesting by itself, we focus, in this paper, on the simplest case, which still is hard enough.

Another difference between patterns and polymorphic types is that it makes sense for polymorphic types to yield a marking with an arbitrary number of marks of the same variable, whereas it does not for patterns. Thus, whenever we use a type p as a pattern, we require p to be *linear*, that is:

> We say that p is *linear* when, for every value v and marking V, if $(v, V) \lhd p$, then $\mathbf{dom}(V) = \mathbf{var}(p)$ and V is a function (i.e., $\pi = \pi'$ for any $(x : \pi), (x : \pi') \in V$).

An algorithm for checking linearity is given in Section 4.2.

We are now able to define the subtyping relation on types in a semantic way. As discussed in Section 2.2, we adopt a "mostly" set-theoretic containment of the

semantics of those types, that is, we require that, for each marked value in the left-hand-side type, there is an marked value in the right-hand-side type that is identical to the former marked value except that some marks can be dropped. Formally,

> We say that a type p is a *subtype* of another type q, written $p \leqslant q$, when, for all v and V, if $(v, V) \lhd p$, then $(v, W) \lhd q$ for some marking $W \subseteq V$.

We present an algorithm for checking subtyping in Section 4.4.

Note that our internal representation allows each mark to be put only on a single node rather than a collection of nodes. This has an implication on which part of an external value can be given a mark. Namely, a mark can only be put on either (1) a *tail-sequence*, i.e., a sequence that ends at the tail of the whole sequence, (2) an *individual* element, or (3) a label. For example, consider the external value

```
a[b[],c[],d[]]
```

and the corresponding internal value

$$((\mathtt{a}, ((\mathtt{b}, \nu), ((\mathtt{c}, \nu), ((\mathtt{d}, \nu), \nu)))), \nu).$$

In the internal value, consider putting a mark on one of the following three subparts (1) $((\mathtt{c}, \nu), ((\mathtt{d}, \nu), \nu))$, (2) $(\mathtt{c}, \nu)$, and (3) $\mathtt{c}$. The above internal value with these three possible markings is depicted in Figure 2. In the external value, each marked subpart corresponds to (1) the tail-sequence `c[],d[]`, (2) the individual element `c[]`, (3) the label `c`. Note that there is no way to put a mark on a non-tail, non-individual sequence like `b[],c[]`. These facts have a further implication on where variables can occur in external types or patterns. Specifically, we can allow (1) a variable at a tail position, e.g., `a[b[],(x as Any)]`, (2) a variable for individual elements, e.g., `(x as a[T])`, and even (3) a variable for labels themselves (which are elided from Section 2), but we cannot allow a variable be concatenated to another type or pattern, e.g., `a[(x as Any), b[]]`. Of course, it would be advisable to relax this restriction, in particular for patterns, where many useful examples rely on extraction of such sequences [Hosoya and Pierce 2002]. However, we found that fully relaxing the restriction both for patterns and polytypes causes a complex interaction between them. Thus, we propose an intermediate solution: we fully eliminate the restriction for patterns but still require polytypes to parametrize only individual elements (only (2) is allowed). The current XDuce implementation (version 0.5.0) adopts this design choice. In Section 5.2, we discuss this issue in more detail.

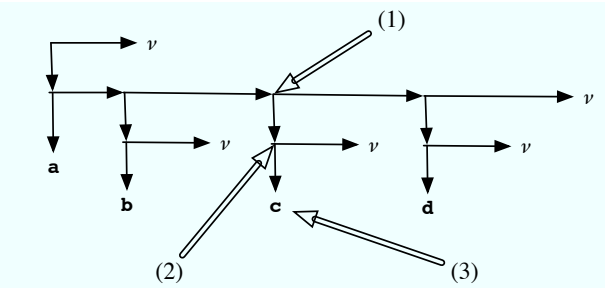

Fig. 2. An internal value with three possible markings

3.3 Type Substitution

As customary, a type substitution is a mapping from a finite set of variables to types. The application of a type substitution to a type produces another type. However, in our system, the definition of the application of a substitution is not so straightforward as usual. In our system, constraints may occur in types and the application of a substitution σ to a type p, noted σp, must take into account both the constraints occurring in p and those occurring in the image of σ.

In order to understand how to define the application of a type substitution let us show an example. Consider the type $p = ((x : \top), \top)$. This is a product of a variable x with the top type. The constraint $x : \top$ indicates that there is no constraint on x, and thus this type denotes all values marked by x. If we apply to this type the substitution $\sigma = \{x \mapsto ((y : \top), a)\}$, it is easy to see that σp is defined as $(((y : \top), a), \top)$ and is obtained by replacing $(x : \top)$ with $((y : \top), a)$. As we see in the case that the variables at issue have no constraint (i.e., they are intersected with the top type), the application of substitution works as usual: the occurrences of an (unconstrained) variable are replaced by its image in σ.

However, in the case that we have a constraint on x as in $p = ((x : (b, \top)), \top)$, this constraint interacts with $((y : \top), a)$. The constraint $(x : (b, \top))$ states that the values in that type are x marked pairs whose left projection is b; thus when we apply the previous substitution to this type, we must compute the intersection between $(b, \top)$ and the type $((y : \top), a)$ and use it for substituting x. This results in $\sigma p = (((y : b), a), \top)$. The situation is even more complicated when the type constraint contains other variables and we need to do a simultaneous substitution for multiple variables, or the image of substitution contains further constraints, or the type p is recursively defined. We see that a syntactic definition of the application of a type substitution becomes soon unfeasible, and that the type resulting from such an application must be defined in a semantic way.

Then, how do we define the application σp of a substitution σ to a type p semantically? Since in our framework types are sets of marked values then this correspond to define which marked values σp denotes. Let us show this point on our running example: $p = ((x : (b, \top)), \top)$ and $\sigma = \{x \mapsto ((y : \top), a)\}$. The set of marked values contained in the type σp should be generated in the following way. Every value in p has the form $((b, v), w)$ where v and w are any values and the subpart (b, v) is marked by x. For each such value, we generate a new marked value $((b, v), w)$ with the mark x replaced with a new marking U, provided that the subpart (b, v) matches $\sigma(x) = ((y : \top), a)$ yielding U. The type σp contains all such marked values and no other. Hence, each marked value in σp has the form $((b, a), w)$ where w is any value and b is marked by y. Syntactically, σp can be written $(((y : b), a), \top)$ as we showed in the previous paragraph.

We can now give the definition of substitution, by generalizing the above argument to the case where the domain substitution contains an arbitrary number of variables and p may contain further variables that are not in the domain of σ.

> An *X-substitution* σ is a mapping from a set X of variables to types. The *application* of X-substitution σ to a type p, written σp, is the type that, for each value v and marking V, satisfies the following property:

$$(v, V) \lhd \sigma p$$
$$\text{iff}$$
$$\begin{array}{lll} \exists W, n, U_1, \ldots, U_n. & (v, W) \lhd p & \\ & V = W|_{\overline{X}} \cup \bigcup_{i=1..n} \pi_i U_i & \\ & \text{where} & \{(x_1 : \pi_1), \ldots, (x_n : \pi_n)\} = W|_X \\ & & (\pi_i(v), U_i) \lhd \sigma(x_i) \quad (\forall\ i = 1, \ldots, n) \end{array}$$

That is, for each marked value (v, W) in p, if each x_i-marked subpart $\pi_i(v)$ (where $x_i \in X$) matches the type $\sigma(x_i)$ with a marking U_i, then σp must contain the marked value (v, V) where V is identical to W except that each x_i-mark is replaced by the new marking U_i pushed down by π_i. The type σp must contain no other marked values. Although the definition does not specify it, such σp always exists and can be computed algorithmically.[4] An algorithm is given in Section 4.3 Also, though the definition does not specify such σp in a syntactically unique way, we assume some strategy that picks up one of types satisfying the above condition (e.g., the algorithm in Section 4.3).

We extend the subtype relation between types to substitutions: $\sigma \leqslant \sigma'$ means $\mathbf{dom}(\sigma) = \mathbf{dom}(\sigma')$ and $\sigma(x) \leqslant \sigma'(x)$ for any $x \in \mathbf{dom}(\sigma)$.

3.4 Type Inference

At each application of a polymorphic function and each pattern match, we perform a form of type inference. Although two different inferences may seem required for these two cases, they can actually be formalized in a uniform way. Both cases involve a "domain" type p and a "target" type q, and a set X of variables. For a function application, the domain p is the actual argument's type, the target q is the formal parameter's type, and X is the set of type parameters that may appear in q. For a pattern match, the domain p is the input value's type, the target q is the pattern, and X is the set of pattern variables that may appear in q. Below, we will describe our inference in two steps: (1) a relationship among the domain, the target, and the inferred types, and (2) two restrictions on the domain and the target. The first part will directly yield the specification of the inference for patterns, whereas combining both parts will yield the specification of the inference for type arguments.

First, let us give the definition of inference.

> An X-*inference* of a (target) type q with respect to a (domain) type p is an X-substitution, written $p \lhd_X q$, satisfying for each value v, marking V, and variable $x \in X$ the following property.
>
> $$(u, U) \lhd (p \lhd_X q)(x)$$
> $$\text{iff}$$
> $$\begin{array}{ll} \exists v, V, W, \pi. & (v, V) \lhd p \\ & (v, W \cup \{(x : \pi)\}) \lhd q \\ & (u, U) = \pi^{-1}(v, V) \end{array}$$

[4] For the sake of precision, the following seemingly equivalent definition does not reflect our intention: $(v, V) \lhd \sigma p$ iff $\exists W.\ (v, W) \lhd p\ \wedge\ V = W|_{\overline{X}} \cup \bigcup \{\pi U \mid (x, \pi) \in W|_X,\ (\pi(v), U) \lhd \sigma(x)\}$. It is because there can be several U yielded from given $\pi(v)$ and $\sigma(x)$, but we intend V to contain exactly one such U (with push-down by π) rather than all of them.

That is, for each marked value (v, V) in p, if v matches q with (at least) a marking of x on a subnode $\pi(v)$, then the inferred type $(p \lhd_X q)(x)$ for x must contain the subnode with all the original marks V pulled up by π (note that we lose marks on the nodes that are not descendants of $\pi(v)$). The inferred type $(p \lhd_X q)(x)$ must contain no other marked value. Section 4.5 proves that such $p \lhd_X q$ always exists and can be found deterministically.

For example, let the domain be $p = (((y : \top), a), \top)$ and the target be $q = ((x : (b, \top)), \top)$, and let us infer a type for x. Each marked value in p has the form $((v, a), w)$ where v and w are any values and v is marked by y. Values of the form $((v, a), w)$ match q when $v = b$ and yield a marking x on the subnode (b, a). Therefore the inferred type for x contains (b, a) where b is marked by y (since so it is marked in the original value). Syntactically, the inferred type can be written as $((y : b), a)$.

As mentioned , we use the above definition directly as the specification of the inference for patterns, but impose additional requirements for the inference of type arguments. To see the reason, first recall that the specification that we want for the latter inference is, as outlined in Section 2.2, the *minimum* X-substitution σ such that $p \leqslant \sigma q$. Let us call an X-substitution σ satisfying $p \leqslant \sigma q$ a *solution*. It would be nice if the inference specification for patterns returned the minimal solution for this problem but, unfortunately, in some cases it does not even yield a solution. For instance, in the previous example, from the domain $p = (((y : \top), a), \top)$ and the target $q = ((x : (b, \top)), \top)$, the inference yielded the $\{x\}$-substitution $\sigma = \{x \mapsto ((y : b), a)\}$, which does not satisfy $p \leqslant \sigma q$. Moreover, even when the inference result *is* a solution, it is not necessarily minimum nor even minimal. For example, consider the domain $p = (a, b)$ and the target $q = ((x : \top), b)|(a, (x : \top))$ (similar to an example used in Section 2.2). The inference yields $\{x \mapsto (a|b)\}$ because there are two possible ways of matching the value in p with q and the type inferred for x captures both values that can be bound to x. This result is not minimal since there are other solutions smaller than this: $\{x \mapsto a\}$ and $\{x \mapsto b\}$. Furthermore, since these two are actually minimal and neither is smaller than the other, there is no minimum substitution. The non-minimality is reasonable in the inference for pattern variables: since we adopt a non-determinstc semantics for union we do not know which case of $(x : \top, b)$ and $(a, x : \top)$ would be taken at run time and therefore the type for x must conservatively include information on both cases. In the inference for type parameters, however, the inferred substitution is more desirable to be *minimal* so as to make the result type of the application of polymorphic function as small as possible and thereby make the remaining program code as typeable as possible. Further, the *minimum* substitution is even more desirable since it makes the result type of the application unique and thereby makes the specification of the inference easy to understand (the existence of several minimal solutions might complicate it even to the point of jeopardizing decidability of type-inference: e.g., [Castagna and Pierce 1995]).

Fortunately, by imposing a couple restrictions on the domain and the target, we can ensure that the inference always computes a minimum solution. The first is for ensuring that the inference result is a solution: we require that p is a subtype of q *ignoring* variables in X, that is, for all v and V, if $(v, V) \lhd p$, then $(v, W) \lhd q$ and $W|_{\overline{X}} \subseteq V$ for some W. We write this requirement $p \leqslant_X q$. The second restriction

is for ensuring that the inference result is equal to or smaller than every solution: we require that q yields a unique marking with respect to X for any value; formally, for all v, V, and V', if $(v, V) \lhd q$ and $(v, V') \lhd q$, then $V|_X = V'|_X$. When this holds, we say that q is X-*unambiguous*. We show an algorithm for ambiguity check in Section 4.6.

Note that the unambiguity requirement is a sufficient condition but not a necessary one: there is an ambiguous parameter type for which a minimum type argument always exists. For example, a parameter type $(x : \top)|\top$ is ambiguous but, for any type argument, the solution $\{x \mapsto \bot\}$ is always minimum. A question may then arise whether our requirement might be too restrictive in practice. Currently, we do not have a clear answer though we believe it to be unlikely. However, even if one actually writes polymorphic functions with ambiguous parameter types, we can still support them by requiring that, in that case, the function must be fed with explicit type arguments. For the sake of the simplicity we decided not to include this possibility in this presentation.

The above claims are summarized by the following proposition.

PROPOSITION 1. *Let $p \leqslant_X q$ and q be X-unambiguous. Then, $(p \lhd_X q)$ is the minimum X-substitution σ satisfying $p \leqslant \sigma q$.*

PROOF: To prove the result, it suffices to show: for any X-substitution σ, (1) $p \leqslant \sigma q$ iff (2) $(p \lhd_X q) \leqslant \sigma$.

We first prove that (1) implies (2). Let $(u, U) \lhd (p \lhd_X q)(x)$ with $x \in X$. Then, by the definition of type inference, there are v, V, W, π such that

$$\begin{array}{l}(v, V) \lhd p \\ (v, W \cup \{(x : \pi)\}) \lhd q \\ (u, U) = \pi(v, V).\end{array}$$

From the assumption $p \leqslant \sigma q$, we have $(v, V') \lhd \sigma q$ for some $V' \subseteq V$. By the definition of substitution, there are V'', n, $U_1, \ldots, U_n$ such that $V' = V''|_{\overline{X}} \cup \bigcup_{i=1,\ldots,n} \pi_i U_i$ where

$$\begin{array}{l}(v, V'') \lhd q \\ \{(x_1 : \pi_1), \ldots, (x_n : \pi_n)\} = V''|_X \\ (\pi_i(v), U_i) \lhd \sigma(x_i) \quad (\forall\ i = 1, \ldots, n).\end{array}$$

Since q is X-unambiguous, we know that $V''|_X = (W \cup \{(x : \pi)\})|_X$ and therefore $(x_i : \pi_i) = (x : \pi)$ for some $i = 1, \ldots, n$. The result follows since $U_i = \pi^{-1}(\pi U_i) \subseteq \pi^{-1} V' \subseteq \pi^{-1} V = U$.

Next, we prove that (2) implies (1). Let $(v, V) \lhd p$. Then, from the assumption $p \leqslant_X q$, we have

$$\begin{array}{l}(v, V') \lhd q \\ V'|_{\overline{X}} \subseteq V\end{array}$$

for some V'. Let $V'|_X = \{(x_1 : \pi_1), \ldots, (x_n : \pi_n)\}$. Then, from the definition of type inference, we have for each $i = 1, \ldots, n$, that

$$(\pi_i(v), \pi_i^{-1} V) \lhd (p \lhd_X q)(x_i).$$

Further, from the assumption $(p \lhd_X q) \leqslant \sigma$, we have that, for each $i = 1, \ldots, n$, there is U_i such that

$$(\pi_i(v), U_i) \lhd \sigma(x_i)$$
$$U_i \subseteq \pi_i^{-1} V.$$

Let $U = V'|_{\overline{X}} \cup \bigcup_i \pi_i U_i$. Then, by the definition of substitution,

$$(v, U) \lhd \sigma q.$$

The remaining is to show $U \subseteq V$, which follows from $V'|_{\overline{X}} \subseteq V$ and $\pi_i U_i \subseteq \pi_i(\pi_i^{-1} V) \subseteq V$ for all $i = 1, \ldots, n$. □

3.5 Type System

With the definitions given above, we can describe our type system in a relatively straightforward way. A *program* consists of a set Φ of (top-level) *functions*, ranged over by ϕ, of the form

$$\mathbf{fun}\ f\{X\}(x : p_1) : p_2 = e$$

(where p_1 and p_2 are polytypes) and an entry-point term e_0, where *terms* are defined by the following syntax.

$$e ::= x \mid a \mid (e, e) \mid f(e) \mid \mathbf{match}\ e\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$$

That is, a term is either a variable, a label, a pair, a function call, or a pattern match (where each p_i is a pattern). Note that the function in a function call may be a polymorphic one. Though, we do not provide a form to explicitly supply type arguments—they are always inferred. In every match expression $\mathbf{match}\ e\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$, we assume that $\mathbf{var}(p_i) \cap \mathbf{var}(p_j) = \emptyset$ for $i \neq j$ (for convenience), and that each pattern p_i is linear (every variable occurs at most once). For a function $\mathbf{fun}\ f\{X\}(x : p_1) : p_2 = e$, we require that $\mathbf{var}(p_1) \cup \mathbf{var}(p_2) \subseteq X$. Henceforward, we consider the set Φ of functions to be given, and require that the names of the functions declared in Φ are all different. Finally, we adopt the usual α-renaming convention of bound variables.

The type system is described by the *typing* relations of the form $\Gamma \vdash e \in p$ ("under type environment Γ, term e has polytype p") and of the form $\vdash \phi$ for a function $\phi = (\mathbf{fun}\ f\{X\}(x : p) : q = e)$ ("function ϕ is well-typed") where a *type environment* Γ is a mapping from term variables to polytypes (hence a substitution of polytypes for term variables). We write $\vdash \Phi$ ("all functions declared in Φ are well-typed") for $\vdash \phi$ for all $\phi \in \Phi$. The typing relations are defined by the following set of rules.

$$\textsc{TVar}\ \frac{}{\Gamma \vdash x \in \Gamma(x)} \qquad \textsc{TCon}\ \frac{}{\Gamma \vdash a \in a} \qquad \textsc{TPair}\ \frac{\Gamma \vdash e_i \in p_i \quad \forall\, i = 1,2}{\Gamma \vdash (e_1, e_2) \in (p_1, p_2)}$$

$$\textsc{TApp}\ \frac{\textbf{fun}\ f\{X\}(x : p) : q = e_2 \in \Phi \qquad \Gamma \vdash e_1 \in r \qquad r \leqslant_X p}{\Gamma \vdash f(e_1) \in (r \lhd_X p)q}$$

$$\textsc{TMatch}\ \frac{\Gamma \vdash e_0 \in p \qquad X = \textbf{var}(p_1 \mid \ldots \mid p_n) \qquad p \leqslant_X p_1 \mid \ldots \mid p_n \qquad \Gamma, (p \lhd_X p_i) \vdash e_i \in r_i \qquad \forall\, i = 1, \ldots, n}{\Gamma \vdash \textbf{match}\ e_0\ \textbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \in r_1 \mid \ldots \mid r_n}$$

$$\textsc{TFun}\ \frac{x : p \vdash e \in r \qquad r \leqslant q \qquad p \text{ is } X\text{-unambiguous}}{\vdash \textbf{fun}\ f\{X\}(x : p) : q = e}$$

These rules are quite standard except for TApp, TMatch, and TFun. In TApp, we first infer the substitution $(r \lhd_X p)$ for the type parameters X from the actual argument type r and the formal parameter type p, as described in Section 3.4. As already stated, in order to ensure that the result of the inference is minimum, we require that the actual argument type r is a subtype of the formal parameter type p, ignoring the type parameters X (checked in TApp), and that the formal parameter type p is X-unambiguous (checked in TFun). In TMatch, we first ensure exhaustiveness of the pattern match with respect to the domain type p. For this, we check that p is a subtype of the union of the patterns $p_1, \ldots, p_n$, ignoring all the term variables appearing in these patterns. Then, for each pattern p_i, we calculate the X-inference $(p \lhd_X p_i)$ of the pattern p_i with respect to the domain type p and, under the current type environment augmented with the obtained substitution $(p \lhd_X p_i)$ (recall that $(p \lhd_X p_i)$ can be seen as a type environment), we typecheck the corresponding body e_i. The type of the whole match expression is the union of the types r_i of all the bodies.

3.6 Evaluation Semantics

The evaluation semantics is fairly straightforward except for its handling of marking. We describe the semantics by the *evaluation* relation $E \vdash e \Downarrow (v, V)$ ("under value environment E, term e evaluates to marked value (v, V)") where a *value environment* E is a mapping from term variables to marked values. The evaluation

relation is defined by the following set of rules.[5]

$$\textsc{SVar}\ \frac{}{E \vdash x \Downarrow E(x)} \qquad \textsc{SCon}\ \frac{}{E \vdash a \Downarrow (a, \emptyset)} \qquad \textsc{SPair}\ \frac{E \vdash e_i \Downarrow (v_i, V_i) \quad \forall\, i = 1, 2}{E \vdash (e_1, e_2) \Downarrow (v_1, V_1) \otimes (v_2, V_2)}$$

$$\textsc{SApp}\ \frac{\textbf{fun}\ f\{X\}(x : p) : q = e_2 \in \Phi \qquad E \vdash e_1 \Downarrow (v, V) \qquad x : (v, V) \vdash e_2 \Downarrow (w, W)}{E \vdash f(e_1) \Downarrow (w, W)}$$

$$\textsc{SMatch}\ \frac{E \vdash e_0 \Downarrow (v, V) \qquad (v, V') \lhd p_i \qquad E, \{(x : \pi(v, V)) \mid x : \pi \in V'\} \vdash e_i \Downarrow (w, W)}{E \vdash \textbf{match}\ e_0\ \textbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \Downarrow (w, W)}$$

In SCon, a label expression a produces a value a with no mark. In SPair, two marked values yielded by e_1 and e_2 are paired by $\otimes$. In SApp, a function call to f simply passes the marked value of the argument expression to the function and yields the marked value returned from the function. In SMatch, after evaluating the input as (v, V), we try matching the "bare" value v against each pattern. If some pattern p_i matches with a marking V', then we form a binding of each pattern variable x to the marked value $\pi(v, V)$ (i.e., the extraction of (v, V) by the path π) where π is the path for the mark x in V'. (Note that, since each pattern is linear, there is always exactly one $(x : \pi)$ in V' for each variable appearing in the pattern.) With the obtained bindings, we evaluate the corresponding body e_i. Note that the semantics of pattern matching chooses an arbitrary clause when multiple clauses can match, unlike the usual semantics choosing the first clause in such a case. This is just for simplifying the formalization, putting aside orthogonal language design issues. Indeed, changing the semantics to the first match can be done by using a standard technique that uses a "difference" operation for excluding, from each pattern, the values matched by the preceding patterns [Hosoya and Pierce 2002; Hosoya 2003; Frisch et al. 2002]. We further discuss the difference operation in Section 5.1.

Finally, note that no construct "creates" marks. Therefore, when we start evaluation with an entry-point expression and an empty environment, we will never see any marking at all during execution. This means that in an actual implementation we do not need either marked values or all run-time machinery related to marking: bare values are enough. Then, why should we nevertheless care about the evaluation semantics with marked values? The answer is: for understanding our polymorphic type system. That is, the intuition behind what the type system is trying to prove—"each parametrized subpart in the result value comes from some parametrized subpart in the input value"—is best explained in terms of our "marked" evaluation semantics.

[5]Our evaluation semantics adopts a so-called big-step semantics, as opposed to a more common small-step semantics. This is because a small-step semantics generally requires values, i.e., results of evaluation, to be syntactically a subset of expressions, which is not the case for us since our semantics "augments" results of expressions with marks.

3.7 Correctness

The goal of this subsection is to prove the correctness of our type system consisting of two main theorems, *type preservation* ("the result of a well-typed expression always has the specified type") and *progress* ("no well-typed expression goes wrong"). The proofs of the main theorems use a number of small lemmas on subtyping, substitution, inference, and the typing relation. The statements in some of these lemmas are apparently obvious, but they need to be proved because we use unusual semantic definitions for subtyping, etc. Fortunately, the proofs are quite terse thanks to the semantic definitions (in particular, Lemma 1 through Lemma 5 do not use structural induction at all).

Subtyping is preserved under pairing (Lemma 1) and under substitution (Lemma 2).

LEMMA 1. *If $p_i \leqslant q_i$ for each $i = 1, 2$, then $(p_1, p_2) \leqslant (q_1, q_2)$.*

PROOF: Let $(v, V) \lhd (p_1, p_2)$. Then, by MPAIR, there are v_1, v_2, V_1, and V_2 with $v = (v_1, v_2)$ and $V = 1V_1 \cup 2V_2$ such that $(v_i, V_i) \lhd p_i$ for each $i = 1, 2$. For each $i = 1, 2$, from the assumption $p_i \leqslant q_i$, we obtain $(v_i, V_i') \lhd q_i$ for some $V_i' \subseteq V_i$. Therefore $(v, 1V_1' \cup 2V_2') \lhd (q_1, q_2)$. The result follows since $1V_1' \cup 2V_2' \subseteq 1V_1 \cup 2V_2$. □

LEMMA 2. *Let $X \cap \mathbf{dom}(\sigma) = \emptyset$. If $p \leqslant_X q$, then $\sigma p \leqslant_X \sigma q$. In particular, if $p \leqslant q$, then $\sigma p \leqslant \sigma q$.*

PROOF: Let $(v, V) \lhd \sigma p$ and $Y = \mathbf{dom}(\sigma)$. Then, by the definition of substitution, there are $V', n, U_1, \ldots, U_n$ such that $V = V'|_{\overline{Y}} \cup \bigcup_{i=1,\ldots,n} \pi_i U_i$ where

$$\begin{array}{l} (v, V') \lhd p \\ \{(x_1 : \pi_1), \ldots, (x_n : \pi_n)\} = V'|_Y \\ (\pi_i(v), U_i) \lhd \sigma(x_i) \quad (\forall i = 1, \ldots, n). \end{array}$$

From the assumption $p \leqslant_X q$, we have $(v, V'') \lhd q$ for some $V''|_{\overline{X}} \subseteq V'$. Since $V''|_Y = (V''|_{\overline{X}})|_Y \subseteq V'|_Y$, let K be the subset of $\{1, \ldots, n\}$ such that $\{(x_k : \pi_k) \mid k \in K\} = V''|_Y$. If we let $W = V''|_{\overline{Y}} \cup \bigcup_{k \in K} \pi_k U_k$, then, by the definition of substitution, we have $(v, W) \lhd \sigma q$. The result follows since:

$$\begin{array}{rcl} W|_{\overline{X}} & = & (V''|_{\overline{Y}})|_{\overline{X}} \cup \bigcup_{k \in K} \pi_k U_k|_{\overline{X}} \\ & = & (V''|_{\overline{X}})|_{\overline{Y}} \cup \bigcup_{k \in K} \pi_k U_k|_{\overline{X}} \\ & \subseteq & V'|_{\overline{Y}} \cup \bigcup_{k \in K} \pi_k U_k \\ & = & V \end{array}$$

□

For a given target, the inference gives a greater substitution from a greater input type.

LEMMA 3. *If $p \leqslant q$, then $(p \lhd_X r) \leqslant (q \lhd_X r)$.*

PROOF: Let $(u, U) \lhd (p \lhd_X r)(x)$ for $x \in X$. Then, by the definition of type inference, there are v, V, W, and π such that

$$\begin{array}{l} (v, V) \lhd p \\ (v, W \cup \{(x : \pi)\}) \lhd r \\ (u, U) = \pi(v, V). \end{array}$$

From the assumption $p \leqslant q$, we have $(v, V') \lhd q$ for some $V' \subseteq V$. Therefore, if we let $U' = \pi^{-1}V'$, then, again by the definition of type inference, $(u, U') \lhd (p \lhd_X r)$. The result follows since $U' \subseteq U$. □

A greater substitution gives a greater type when applied to a given type.

LEMMA 4. *If $\sigma \leqslant \sigma'$, then $\sigma p \leqslant \sigma' p$.*

PROOF: Let $(v, V) \lhd \sigma p$ and $X = \mathbf{dom}(\sigma)$. Then, by the definition of substitution, there are $V', n, U_1, \ldots, U_n$ such that $V = V'|_{\overline{X}} \cup \bigcup_{i=1,\ldots,n} \pi_i U_i$ where

$$\begin{array}{l}(v, V') \lhd p \\ \{(x_1 : \pi_1), \ldots, (x_n : \pi_n)\} = V'|_X \\ (\pi_i(v), U_i) \lhd \sigma(x_i) \quad (\forall i = 1, \ldots, n).\end{array}$$

From the assumption $\sigma \leqslant \sigma'$, we have $(\pi_i(v), U_i') \lhd \sigma'(x_i)$ for some $U_i' \subseteq U_i$. Therefore, if we let $W = V'|_{\overline{X}} \cup \bigcup_i \pi_i U_i'$, then, again by the definition of substitution, $(v, W) \lhd \sigma' p$. The result follows since $W \subseteq V$. □

In an inference, the following result allows us to factor out the substitution applied to the domain type to the result of the inference.

LEMMA 5. *Let $X \cap \mathbf{dom}(\sigma) = \emptyset$. Then, $(\sigma p \lhd_X q) \leqslant \sigma(p \lhd_X q)$.*

PROOF: Let $(u, U) \lhd (\sigma p \lhd_X q)(x)$ where $x \in X$. Then, by the definition of type inference, for some V, W, π,

$$\begin{array}{l}(v, V) \lhd \sigma p \\ (v, W \cup \{(x : \pi)\}) \lhd q \\ (u, U) = \pi(v, V).\end{array}$$

Let $Y = \mathbf{dom}(\sigma)$. By the definition of substitution, for some V', n, $U_1, \ldots, U_n$,

$$\begin{array}{l}(v, V') \lhd p \\ V = V'|_{\overline{Y}} \cup \bigcup_i \pi_i U_i \\ \{(y_1 : \pi_1), \ldots, (y_n : \pi_n)\} = V'|_Y \\ (\pi_i(v), U_i) \lhd \sigma(y_i) \quad (\forall i = 1, \ldots, n).\end{array}$$

If we let $U' = \pi^{-1}V'$, then, again by the definition of type inference,

$$(u, U') \lhd (p \lhd_X q)(x).$$

Note that $U'|_Y = \pi^{-1}(V'|_Y) = \{(y_i : \pi_i') \mid \pi_i = \pi\pi_i'\}$. Also, if $\pi_i = \pi\pi_i'$ for some π_i', then, from $u = \pi(v)$ and $(\pi_i(v), U_i) \lhd \sigma(y_i)$, we have $(\pi_i'(u), U_i) \lhd \sigma(y_i)$. From these, if we let $U'' = U'|_{\overline{Y}} \cup \bigcup\{\pi_i' U_i \mid \pi_i = \pi\pi_i'\}$, then, by the definition of substitution, we obtain

$$(u, U'') \lhd \sigma(p \lhd_X q).$$

The remaining to show is $U'' \subseteq U$. This follows since:

$$\begin{array}{rcl}U &=& \pi^{-1}(V'|_{\overline{Y}} \cup \bigcup_i \pi_i U_i) \\ &=& (\pi^{-1}V')|_{\overline{Y}} \cup \bigcup_i \pi^{-1}(\pi_i U_i) \\ &=& U'|_{\overline{Y}} \cup \bigcup_i \pi^{-1}(\pi_i U_i) \\ &\supseteq& U'|_{\overline{Y}} \cup \bigcup\{\pi_i' U_i \mid \pi_i = \pi\pi_i'\} \\ &=& U''\end{array}$$

□

Note that the other way does not necessarily hold. For example, let $p = (x : (a, a))$, $q = (a, (y : a))$, $X = \{y\}$, and $\sigma = \{x \mapsto \bot\}$. Then, $(\sigma p \lhd_X q) \leqslant \sigma(p \lhd_X q)$ but not $\sigma(p \lhd_X q) \leqslant (\sigma p \lhd_X q)$.

The following lemma ensures that, if we strengthen the type environment, this preserves typability of an expression with its type becoming smaller. Here, we write $\Gamma' \leqslant \Gamma$ when $\Gamma'(x) \leqslant \Gamma(x)$ for every $x \in \mathbf{dom}(\Gamma)$.

LEMMA 6. *If* $\Gamma' \leqslant \Gamma$ *and* $\Gamma \vdash e \in p$, *then* $\Gamma' \vdash e \in p'$ *for some* $p' \leqslant p$.

PROOF: On the derivation of $\Gamma \vdash e \in p$ with case analysis on the last rule applied.

TVar. $\dfrac{}{\Gamma \vdash x \in \Gamma(x)}$ with $e = x$ and $p = \Gamma(x)$.

The result follows from $\Gamma' \leqslant \Gamma$ and TVAR.

TCon. $\dfrac{}{\Gamma \vdash a \in a}$ with $e = a$ and $p = a$.

The result follows immediately by TCON.

TPair. $\dfrac{\Gamma \vdash e_i \in p_i \quad \forall i = 1, 2}{\Gamma \vdash (e_1, e_2) \in (p_1, p_2)}$ with $e = (e_1, e_2)$ and $p = (p_1, p_2)$.

The result follows from the induction hypothesis, Lemma 1, and TPAIR.

TApp. $\dfrac{\mathbf{fun}\ f\{X\}(x : p_2) : p_3 = e_2 \in \Phi \qquad \Gamma \vdash e_1 \in p_1 \qquad p_1 \leqslant_X p_2}{\Gamma \vdash f(e_1) \in (p_1 \lhd_X p_2)p_3}$

with $e = f(e_1)$ and $p = (p_1 \lhd_X p_2)p_3$.

From the induction hypothesis, $\Gamma' \vdash e_1 \in p_1'$ and $p_1' \leqslant p_1$. With $p_1 \leqslant_X p_2$, we have $p_1' \leqslant_X p_2$. Therefore by TAPP, we obtain $\Gamma' \vdash e \in p'$ where $p' = (p_1' \lhd_X p_2)p_3$. The remaining to show is $p' \leqslant p$, which follows by Lemma 3 and Lemma 4.

TMatch. $\dfrac{\begin{array}{c}\Gamma \vdash e_0 \in p_0 \qquad X = \mathbf{var}(p_1 \mid \ldots \mid p_n) \\ p_0 \leqslant_X p_1 \mid \ldots \mid p_n \qquad \Gamma, (p_0 \lhd_X p_i) \vdash e_i \in r_i \qquad \forall i = 1, \ldots, n\end{array}}{\Gamma \vdash \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \in r_1 \mid \ldots \mid r_n}$

with $e = \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$ and $p = r_1 \mid \ldots \mid r_n$.

From the induction hypothesis, $\Gamma' \vdash e_0 \in p_0'$ for some $p_0' \leqslant p_0$. With $p_0 \leqslant_X p_1 \mid \ldots \mid p_n$, we have $p_0' \leqslant_X p_1 \mid \ldots \mid p_n$. From Lemma 3, $(p_0' \lhd_X p_i) \leqslant (p_0 \lhd_X p_i)$ for any $i = 1, \ldots, n$. Therefore by the induction hypothesis, $\Gamma', (p_0' \lhd_X p_i) \vdash e_i \in r_i'$ for some $r_i' \leqslant r_i$. The result follows by TMATCH and $p' = r_1' \mid \ldots \mid r_n' \leqslant p$. □

The following lemma ensures that applying a substitution to the type environment preserves the typability of an expression. The type of the expression becomes smaller than the type after applying the substitution to its original type. Here, we define $(\sigma\Gamma)(x) = \sigma(\Gamma(x))$.

LEMMA 7. *Let* σ *be a substitution for type variables. If* $\Gamma \vdash e \in p$, *then* $\sigma\Gamma \vdash e \in p'$ *for some* $p' \leqslant \sigma p$.

PROOF: On the derivation of $\Gamma \vdash e \in p$ with case analysis on the last rule applied.

TVar. $\dfrac{}{\Gamma \vdash x \in \Gamma(x)}$ with $e = x$ and $p = \Gamma(x)$.

The result follows from TVAR with $p' = \sigma\Gamma(x)$.

TCon. $\dfrac{}{\Gamma \vdash a \in a}$ with $e = a$ and $p = a$.

The result follows immediately by TCon.

TPair. $\dfrac{\Gamma \vdash e_i \in p_i \quad \forall i = 1, 2}{\Gamma \vdash (e_1, e_2) \in (p_1, p_2)}$ with $e = (e_1, e_2)$ and $p = (p_1, p_2)$.

The result follows from the induction hypothesis, Lemma 1, and TPair.

TApp. $\dfrac{\mathbf{fun}\ f\{X\}(x : p_2) : p_3 = e_2 \in \Phi \qquad \Gamma \vdash e_1 \in p_1 \qquad p_1 \leqslant_X p_2}{\Gamma \vdash f(e_1) \in (p_1 \lhd_X p_2)p_3}$

with $e = f(e_1)$ and $p = (p_1 \lhd_X p_2)p_3$.

By α-renaming convention, we can assume $X \cap \mathbf{dom}(\sigma) = \emptyset$. From the induction hypothesis, $\sigma\Gamma \vdash e_1 \in p_1'$ for some $p_1' \leqslant \sigma p_1$. With $p_1 \leqslant_X p_2$ and Lemma 2, we have $p_1' \leqslant_X \sigma p_2$ (note that $p_1' \leqslant \sigma p_1$ obviously implies $p_1' \leqslant_X \sigma p_1$). We have $\sigma p_2 = p_2$ since $\mathbf{var}(p_2) \cap \mathbf{dom}(\sigma) \subseteq X \cap \mathbf{dom}(\sigma) = \emptyset$. Therefore by TApp, we obtain $\sigma\Gamma \vdash e \in p'$ where $p' = (p_1' \lhd_X p_2)p_3$. The remaining to show is $p' \leqslant \sigma p$, which follows from $p' \leqslant (\sigma p_1 \lhd_X p_2)p_3$ by Lemma 3 and Lemma 4, and then $(\sigma p_1 \lhd_X p_2)p_3 \leqslant \sigma(p_1 \lhd_X p_2)p_3$ by Lemma 5 and Lemma 4.

TMatch. $\dfrac{\Gamma \vdash e_0 \in p_0 \qquad X = \mathbf{var}(p_1 \mid \ldots \mid p_n) \qquad p_0 \leqslant_X p_1 \mid \ldots \mid p_n \qquad \Gamma, (p_0 \lhd_X p_i) \vdash e_i \in r_i \qquad \forall i = 1, \ldots, n}{\Gamma \vdash \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \in r_1 \mid \ldots \mid r_n}$

with $e = \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$ and $p = r_1 \mid \ldots \mid r_n$.

Since X has only term variables, $X \cap \mathbf{dom}(\sigma) = \emptyset$. From the induction hypothesis, $\sigma\Gamma \vdash e_0 \in p_0'$ for some $p_0' \leqslant \sigma p_0$. With $p_0 \leqslant_X p_1 \mid \ldots \mid p_n$ and Lemma 2, we have $p_0' \leqslant \sigma p_0 \leqslant_X \sigma(p_1 \mid \ldots \mid p_n) = (p_1 \mid \ldots \mid p_n)$ where the last equality holds since $X \cap \mathbf{dom}(\sigma) = \emptyset$. By the induction hypothesis, $\sigma\Gamma, \sigma(p_0 \lhd_X p_i) \vdash e_i \in r_i'$ for some $r_i' \leqslant \sigma r_i$ for any $i = 1, \ldots, n$. Further, since $(p_0' \lhd_X p_i) \leqslant (\sigma p_0 \lhd_X p_i) \leqslant \sigma(p_0 \lhd_X p_i)$ from Lemma 3 and Lemma 5, we obtain that $\sigma\Gamma, (p_0' \lhd_X p_i) \vdash e_i \in r_i''$ and $r_i'' \leqslant r_i'$ by Lemma 6. The result follows by TMatch with $p' = r_1'' \mid \ldots \mid r_n'' \leqslant r_1' \mid \ldots r_n' \leqslant \sigma p$. □

Type preservation can be proved by standard induction on the derivation of the evaluation relation. We write $\Gamma \vdash E$ when $\mathbf{dom}(\Gamma) = \mathbf{dom}(E)$, and, for each $x \in \mathbf{dom}(E)$, if $(v, V) = E(x)$, then $(v, V') \lhd \Gamma(x)$ for some $V' \subseteq V$. Note that the values in the value environment may contain extra marks not specified by the corresponding types in the type environment.

Theorem (Type preservation). *Let* $\vdash \Phi$. *If* $\Gamma \vdash e \in p$ *and* $E \vdash e \Downarrow (w, W)$ *with* $\Gamma \vdash E$, *then* $(w, W') \lhd p$ *for some* $W' \subseteq W$.

Proof: Induction on the derivation of $E \vdash e \Downarrow (w, W)$.

SVar. $\dfrac{}{E \vdash x \Downarrow E(x)}$ with $e = x$ and $(w, W) = E(x)$.

From TVar, $\Gamma \vdash x \in \Gamma(x)$. The result follows from $\Gamma \vdash E$.

SCon. $\dfrac{}{E \vdash a \Downarrow (a, \emptyset)}$ with $e = a$ and $(w, W) = (a, \emptyset)$.

From TCon, $\Gamma \vdash a \in a$. The result follows from MCon.

SPair. $\dfrac{E \vdash e_i \Downarrow (v_i, V_i) \quad \forall i = 1, 2}{E \vdash (e_1, e_2) \Downarrow (v_1, V_1) \otimes (v_2, V_2)}$ with $e = (e_1, e_2)$ and $w = ((v_1, v_2), (1V_1 \cup 2V_2))$.

By inversion of TPair, there are p_1, p_2 with $p = (p_1, p_2)$ such that $\Gamma \vdash e_i \in p_i$ for each $i = 1, 2$. By the induction hypothesis, $(v_i, V_i') \lhd p_i$ for some $V_i' \subseteq V_i$ for each $i = 1, 2$. The result follows from MPair since $1V_1' \cup 2V_2' \subseteq 1V_1 \cup 2V_2$.

SApp. $$\frac{\textbf{fun}\ f\{X\}(x : p_1) : q = e_2 \in \Phi \quad E \vdash e_1 \Downarrow (v, V) \quad x : (v, V) \vdash e_2 \Downarrow (w, W)}{E \vdash f(e_1) \Downarrow (w, W)}$$

with $e = f(e_1)$.

By inversion of TApp, there is r with $r \leqslant_X p_1$ and $p = (r \lhd_X p_1)q$ such that $\Gamma \vdash e_1 \in r$. By the induction hypothesis, $(v, V') \lhd r$ for some $V' \subseteq V$. From $r \leqslant_X p_1$, we have $r \leqslant (r \lhd_X p_1)p_1$ by Proposition 1. Therefore $(v, V'') \lhd (r \lhd_X p_1)p_1$ for some $V'' \subseteq V'$. Let $\Gamma' = x : (r \lhd_X p_1)p_1$ and $E' = x : (v, V)$. Then, $\Gamma' \vdash E'$. On the other hand, since $\vdash \textbf{fun}\ f\{X\}(x : p_1) : q = e_2$, we have $x : p_1 \vdash e_2 \in p_2$ and $p_2 \leqslant q$. By Lemma 7, $\Gamma' \vdash e_2 \in p_2'$ for some $p_2' \leqslant (r \lhd_X p_1)p_2$. Further, by the induction hypothesis, $(w, W') \lhd p_2'$ for some $W' \subseteq W$. We have $(r \lhd_X p_1)p_2 \leqslant (r \lhd_X p_1)q$ from $p_2 \leqslant q$ and Lemma 2, and therefore, together with $p_2' \leqslant (r \lhd_X p_1)p_2$, we obtain $p_2' \leqslant (r \lhd_X p_1)q$. This implies $(w, W'') \lhd (r \lhd_X p_1)q$ for some $W'' \subseteq W'$. The result follows since $W' \subseteq W$.

SMatch. $$\frac{E \vdash e_0 \Downarrow (v, V) \quad (v, U) \lhd p_i \quad E, \{x : \pi(v, V) \mid x : \pi \in U\} \vdash e_i \Downarrow (w, W)}{E \vdash \textbf{match}\ e_0\ \textbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \Downarrow (w, W)}$$

with $e = \textbf{match}\ e_0\ \textbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$.

Let $X = \textbf{var}(p_1 \mid \ldots \mid p_n)$. By inversion of TMatch, there are $p_0, r_1, \ldots, r_n$ with $p = (r_1 \mid \ldots \mid r_n)$ such that $\Gamma \vdash e_0 \in p_0$ and $\Gamma, (p_0 \lhd_X p_i) \vdash e_j \in r_j$ for each $j = 1, \ldots, n$. By induction hypothesis, $(v, V') \lhd p_0$ for some $V' \subseteq V$. Since $(v, U) \lhd p_i$ for the particular i in the assumption, we have $(\pi(v), \pi^{-1}V') \lhd (p_0 \lhd_X p_i)(x)$ for some $(x : \pi) \in U$ by the definition of type inference. Since $\pi^{-1}V' \subseteq \pi^{-1}V$, $\Gamma, (p_0 \lhd_X p_i) \vdash E, \{x : \pi(v, V) \mid (x : \pi) \in U\}$. From the induction hypothesis, $(w, W') \lhd r_i$ for some $W' \subseteq W$. Since $r_i \leqslant p$, we obtain $(w, W'') \lhd p$ for some $W'' \subseteq W'$. The result follows since $W' \subseteq W$. □

In order to state the progress theorem, we first clarify what we mean by "go wrong." Since just saying $E \not\vdash e \Downarrow (v, V)$ can mean "either e goes wrong or e diverges," we need to define the additional *failure* relation $E \vdash e \Downarrow \mathsf{Err}$ to explicitly state that e encounters a run-time (type) error. Thus, the failure relation is defined by the following set of rules.

NSPair $$\frac{E \vdash e_i \Downarrow \mathsf{Err}}{E \vdash (e_1, e_2) \Downarrow \mathsf{Err}}$$

NSApp1 $$\frac{E \vdash e_1 \Downarrow \mathsf{Err}}{E \vdash f(e_1) \Downarrow \mathsf{Err}}$$

NSApp2 $$\frac{\textbf{fun}\ f\{X\}(x : p) : q = e_2 \in \Phi \qquad E \vdash e_1 \Downarrow (v, V) \qquad x : (v, V) \vdash e_2 \Downarrow \mathsf{Err}}{E \vdash f(e_1) \Downarrow \mathsf{Err}}$$

NSMatch1 $$\frac{E \vdash e_0 \Downarrow \mathsf{Err}}{E \vdash \textbf{match}\ e_0\ \textbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \Downarrow \mathsf{Err}}$$

$$\textsc{NSMatch2}\ \frac{E \vdash e_0 \Downarrow (v, V) \qquad \nexists V'.\ (v, V') \lhd p_i \qquad \forall\ i = 1, \ldots, n}{E \vdash \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \Downarrow \mathsf{Err}}$$

$$\textsc{NSMatch3}\ \frac{E \vdash e_0 \Downarrow (v, V) \qquad (v, V') \lhd p_i \qquad E, \{x : \pi(v, V) \mid (x : \pi) \in V'\} \vdash e_i \Downarrow \mathsf{Err}}{E \vdash \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \Downarrow \mathsf{Err}}$$

Now, we can prove that if an expression is well-typed, then it never fails.

THEOREM (PROGRESS). *Let* $\vdash \Phi$. *If* $\Gamma \vdash E$ *and* $\Gamma \vdash e \in p$, *then* $E \vdash e \Downarrow \mathsf{Err}$ *never holds.*

PROOF: We instead prove that $\Gamma \vdash E$ and $\Gamma \vdash e \in p$ with $E \vdash e \Downarrow \mathsf{Err}$ lead to a contradiction. The proof proceeds by induction on the derivation of $E \vdash e \Downarrow \mathsf{Err}$.

NSPair. $\dfrac{E \vdash e_i \Downarrow \mathsf{Err}}{E \vdash (e_1, e_2) \Downarrow \mathsf{Err}}$ with $e = (e_1, e_2)$.

By inversion of TPAIR, there are p_1, p_2 such that $\Gamma \vdash e_i \in p_i$ for each $i = 1, 2$. The result follows from the induction hypothesis.

NSApp1. $\dfrac{E \vdash e_1 \Downarrow \mathsf{Err}}{E \vdash f(e_1) \Downarrow \mathsf{Err}}$ with $e = f(e_1)$.

The result follows similarly to the case NSPAIR.

NSApp2. $\dfrac{\mathbf{fun}\ f\{X\}(x : p_1) : q = e_2 \in \Phi \qquad E \vdash e_1 \Downarrow (v, V) \qquad x : (v, V) \vdash e_2 \Downarrow \mathsf{Err}}{E \vdash f(e_1) \Downarrow \mathsf{Err}}$

with $e = f(e_1)$.

By inversion TAPP, there is r such that $\Gamma \vdash e_1 \in r$ and $r \leqslant_X p_1$. From Theorem 1, $(v, V) \lhd r$. With $r \leqslant_X p_1$, we have $(v, V') \lhd p_1$ for some $V' \subseteq V$, and therefore $x : p_1 \vdash x : v$. Since $\vdash \mathbf{fun}\ f\{X\}(x : p_1) : q = e_2$, we have $x : p_1 \vdash e_2 \in p_2$. The result follows from the induction hypothesis.

NSMatch1. $\dfrac{E \vdash e_0 \Downarrow \mathsf{Err}}{E \vdash \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \Downarrow \mathsf{Err}}$

with $e = \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$.

The result follows similarly to the case NSPAIR.

NSMatch2. $\dfrac{E \vdash e_0 \Downarrow (v, V) \qquad \nexists V'.\ (v, V') \lhd (p_1 \mid \ldots p_n)}{E \vdash \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \Downarrow \mathsf{Err}}$

with $e = \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$.

By inversion of TMATCH, there is p_0 such that $\Gamma \vdash e_0 \in p_0$ and $p_0 \leqslant_X p_1 \mid \ldots \mid p_n$. From Theorem 1, $(v, V) \lhd p_0$ and therefore $(v, V') \lhd (p_1 \mid \ldots \mid p_n)$ for some V', which leads to a contradiction.

NSMatch3. $\dfrac{E \vdash e_0 \Downarrow (v, V) \quad (v, V') \lhd p_i \quad E, \{x{:}\pi(v, V) \mid (x{:}\pi) \in V'\} \vdash e_i \Downarrow \mathsf{Err}}{E \vdash \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \Downarrow \mathsf{Err}}$

with $e = \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$.

Let $X = \mathbf{var}(p_1 \mid \ldots \mid p_n)$. By inversion of TMATCH, there are $p_0, r_1, \ldots, r_n$ such that $\Gamma \vdash e_0 \in p_0$ and $\Gamma, (p_0 \lhd_X p_i) \vdash e_i \in r_i$ for each $i = 1, \ldots, n$. From Theorem 1, $(v, V) \lhd p_0$. Since $(v, V') \lhd p_i$, we have $(\pi(v), \pi^{-1}V) \lhd (p_0 \lhd_X p_i)(x)$ for some $(x : \pi) \in V'$ by the definition of type inference. Therefore $\Gamma, (p_0 \lhd_X p_i) \vdash E, \{x : \pi(v) \mid (x : \pi) \in V'\}$. The result follows from the induction hypothesis. □

4. ALGORITHMS

This section describes algorithms necessary for implementing the type system defined in the previous section. The algorithms presented here implement linearity checking of patterns, substitution, subtype checking, type inference, and ambiguity checking.

4.1 Marking Automata

For describing our algorithms, we use a model called "marking automata," which can be seen as a slight variation of types in the binary representation given in the previous section.

Let us assume a finite set Σ of labels, which may be obtained by collecting all labels appearing in a given program. A *marking automaton* A is a tuple (S, I, δ, Ξ) where

—S is a finite set of *states*, ranged over by $s, t, \ldots$,
—I is a set of *initial states* ($I \subseteq S$),
—δ is a set of *transition rules* of the form either $s \to (s_1, s_2)$ or $s \to a$ where $s, s_1, s_2 \in S$ and $a \in \Sigma$, and
—Ξ is a mapping from states to sets of variables ($\Xi : S \to \mathcal{P}(\mathcal{X})$).

Given a marking automaton $A = (S, I, \delta, \Xi)$, we define the set, written $\pi_A s$, of *reachable states* from a given state s by a given path π as follows.

$$\begin{array}{lcl} \epsilon_A s & = & \{s\} \\ (j\pi)_A s & = & \bigcup\{\pi_A s_j \mid s \to (s_1, s_2) \in \delta\} \end{array}$$

We define $\pi_A S = \bigcup\{\pi_A s \mid s \in S\}$ for a set S of states. We simply say that a state s is reachable from s' when s is reachable from s' by some path. We write $\mathbf{var}_A(s)$ for the set of variables at the states reachable from s, and $\mathbf{var}_A(S) = \bigcup\{\mathbf{var}_A(s) \mid s \in S\}$. We also require the "linearity"—$\Xi(s) \cap (\mathbf{var}_A(s_1) \cup \mathbf{var}_A(s_2)) = \emptyset$ whenever $s \to (s_1, s_2) \in \delta$ (this is to ensure that any accepted marked value does not contain two markings of the same variable in the same path). In the sequel, we omit the subscript A appearing in these definitions whenever it is clear from the context.

The semantics of a marking automaton $A = (S, I, \delta, \Xi)$ is described by the matching relation $A \vdash (v, V) \lhd s$, read "marking automaton A accepts marked value (v, V) at state s." The matching relation is defined by the following set of rules.

$$\textsc{ACon}\ \frac{s \to a \in \delta}{A \vdash (a, \Xi(s) \times \{\epsilon\}) \lhd s} \qquad \textsc{APair}\ \frac{s \to (s_1, s_2) \in \delta \quad A \vdash (v_i, V_i) \lhd s_i \quad \forall\, i = 1, 2}{A \vdash ((v_1, v_2), 1V_1 \cup 2V_2 \cup \Xi(s) \times \{\epsilon\}) \lhd s}$$

When there is no ambiguity about which marking automaton we talk about, we write $(v, V) \lhd s$ instead of $A \vdash (v, V) \lhd s$. Also, we write $(v, V) \lhd A$ when $A \vdash (v, V) \lhd s$ for some $s \in I$. Note that each state is associated with a set of variables rather than a single variable. This is for encoding types like $x : y : (\top, \top)$ that may mark the same node with several variables. Any type (with a type definition) can be encoded by a marking automaton in a straightforward way. A concrete encoding procedure can be found in Appendix A. (Note that a marking automaton can trivially be encoded by a type with a type definition.) We lift all the type-related definitions (linearity, substitution, subtyping, inference, and ambiguity) to

marking automata. Thus, for instance, we write $A \leqslant B$ to denote that for all v and V, if $(v, V) \lhd A$, then $(v, W) \lhd B$ for some marking $W \subseteq V$ (*cf.* the definition of subtyping in Section 3.3.)

Some of the algorithms shown later use an "empty elimination" operation on a given marking automaton $A = (S, I, \delta, \Xi)$. This operation yields another marking automaton $A' = (S', I', \delta', \Xi')$ with $S' \subseteq S$ such that

—$A \vdash (v, V) \lhd s$ iff $A' \vdash (v, V) \lhd s$, for any (v, V) and $s \in S'$ (A and A' behave exactly the same at the same state),
—$A' \vdash (v, V) \lhd s$ for some (v, V) for any $s \in S'$ (any state in A' accepts some marked value),
—$I' = \{s \in I \mid A \vdash (v, V) \lhd s \text{ for some } (v, V)\}$ (A''s initial states are A's non-empty initial states), and
—$s \in \pi I'$ for some π for any $s \in S'$ (all states in A' are reachable from an initial state).

We do not describe a concrete procedure for empty elimination, but linear time algorithms can be found in the literature, e.g., [Comon et al. 1999].

In later proofs, we use the following lemma, which states that, whenever an empty-eliminated marking automaton accepts a marked value (v, V) at a state s, then, at any state s_0 that can reach s, it accepts some marked value containing (v, V) as a subpart.

LEMMA 8. *Let an empty-eliminated marking automaton given. Then,* $(v, V) \lhd s$ *and* $s \in \pi s_0$ *imply* $(v_0, V_0) \lhd s_0$ *with* $\pi(v_0, V_0) = (v, V)$ *for some* v_0 *and* V_0.

PROOF: Induction on the structure of π.

Case: $\pi = \epsilon$

The result immediately follows.

Case: $\pi = 1\pi'$

From $s \in (1\pi')s_0$, we have $s_0 \to (s_1, s_2) \in \delta$ and $s \in \pi' s_1$ for some s_1 and s_2. By the induction hypothesis, $(v_1, V_1) \lhd s_1$ for some v_1, V_1 with $\pi'(v_1, V_1) = (v, V)$. Since s_2 is not empty, $(v_2, V_2) \lhd s_2$ for some v_2 and V_2. Therefore, by letting $v_0 = (v_1, v_2)$ and $V_0 = \Xi(s) \times \{\epsilon\} \cup 1V_1 \cup 2V_2$, we have $(v_0, V_0) \lhd s_0$ with $1\pi'(v_0, V_0) = \pi'(1v_0, 1^{-1}V_0) = \pi'(v_1, V_1) = (v, V)$.

Case: $\pi = 2\pi'$

Similar to the previous case. □

4.2 Linearity

Checking linearity corresponds to check whether for every value a given pattern binds each pattern variable exactly once. Recall the condition given in Section 3.2: a pattern p is linear if $(v, V) \lhd p$ implies that $\mathbf{dom}(V) = \mathbf{var}(p)$ and V is a function, for any value v and any marking V. This check is performed on every pattern appearing in a given program.

Given a marking automaton $A = (S, I, \delta, \Xi)$, the linearity check algorithm is as follows.

(i) Empty-eliminate A.

(ii) Return "yes" iff all the following hold.
  (L1) for each $s \to (s_1, s_2) \in \delta$, we have $\mathbf{var}(s_1) \cup \mathbf{var}(s_2) \cup \Xi(s) = \mathbf{var}(s)$.
  (L1') if $s \to a \in \delta$, then $\Xi(s) = \mathbf{var}(s)$.
  (L2) for each $s \to (s_1, s_2) \in \delta$, we have $\mathbf{var}(s_1) \cap \mathbf{var}(s_2) = \emptyset$.
  (L3) for each $s \in I$, we have $\mathbf{var}(s) = \mathbf{var}(I)$.

Conditions (L1) and (L1') ensure that the same set of variables appears in each transitions from the same state. Condition (L2) ensures that, for each transition $s \to (s_1, s_2)$, no common variable appears both in s_1 and in s_2. Condition (L3) ensures that the same set of variables appears in all the initial states.

We can prove the following desirable property. (The above algorithm is similar to syntatic checks used in [Hosoya and Pierce 2002; Hosoya 2006] except that empty elimination is additionally performed. This addition precisely ensures completeness.)

LEMMA 9. *For an empty-eliminated marking automaton, $x \in \mathbf{var}(s)$ implies $(v, V) \lhd s$ and $x \in \mathbf{dom}(V)$ for some v, V.*

PROOF: By Lemma 8 with the easy fact that $(v, V) \lhd s$ implies $\Xi(s) \subseteq \mathbf{dom}(V)$. □

PROPOSITION 2. *Given a marking automaton A, the linearity-checking algorithm returns "yes" for A iff A is linear.*

PROOF: We first prove the "only if" direction. Assume that A is already empty-eliminated, and all (L1), (L1'), (L2), and (L3) hold. Then, to prove the result, it suffices to show that, for any v, V, if $(v, V) \lhd s$, then $\mathbf{dom}(V) = \mathbf{var}(s)$ and V is a function. The proof proceeds by induction on the structure of v.

Case: $v = a$

The result trivially holds by ACON.

Case: $v = (v_1, v_2)$

By APAIR, there are v_1, v_2, V_1, V_2 such that $s \to (s_1, s_2) \in \delta$ and $(v_i, V_i) \lhd s_i$ for each $i = 1, 2$ with $V = \Xi(s) \times \{\epsilon\} \cup 1V_1 \cup 2V_2$. By the induction hypothesis, $\mathbf{dom}(V_i) = \mathbf{var}(s_i)$ and V_i is a function for $i = 1, 2$. By (L1), $\mathbf{dom}(V) = \Xi(s) \cup \mathbf{dom}(V_1) \cup \mathbf{dom}(V_2) = \Xi(s) \cup \mathbf{var}(s_1) \cup \mathbf{var}(s_2) = \mathbf{var}(s)$. Also, by (L2), V is a function since $\Xi(s)$, V_1, and V_2 are pair-wise disjoint and V_1 and V_2 themselves are functions.

We next prove the "if" direction. Assume that A is already empty-eliminated. The result follows by proving (1) if (L2) does not hold, then A is not linear, (2) if (L2) holds and (L1) does not, then A is not linear, (2') if (L2) holds and (L1') does not, then A is not linear, and (3) if (L3) does not hold, then A is not linear.

(1) Since (L2) does not hold, let $s \to (s_1, s_2) \in \delta$ with $\mathbf{var}(s_1) \cap \mathbf{var}(s_2) \neq \emptyset$. Then, $x \in \mathbf{var}(s_1)$ and $x \in \mathbf{var}(s_2)$ for some x. Therefore by Lemma 9, $(v_i, V_i) \lhd s_i$ and $x \in \mathbf{dom}(V_i)$ for some v_i,V_i for each $i = 1, 2$, which implies $(v, V) \lhd s$ where $v = (v_1, v_2)$ and $V = \Xi(s) \times \{\epsilon\} \cup 1V_1 \cup 2V_2$ with $\mathbf{dom}(V_1) \cap \mathbf{dom}(V_2) \neq \emptyset$. Therefore V is not a function. Finally, since A is empty-eliminated, and $s \in \pi I$ for some π, by using Lemma 8, $(v', V') \lhd s'$ for some v' and V' with $\pi(v', V') = (v, V)$. This implies $V \subseteq V'$, and therefore V' is not a function.

(2) Since (L1) does not hold, let $s \to (s_1, s_2) \in \delta$ and $\mathbf{var}(s_1) \cup \mathbf{var}(s_2) \cup \Xi(s) \subsetneq \mathbf{var}(s)$ (note that we always have $\mathbf{var}(s_1) \cup \mathbf{var}(s_2) \cup \Xi(s) \subseteq \mathbf{var}(s)$). Since s_1 and s_2 are not empty, $(v_i, V_i) \lhd s_i$ for some v_i and V_i for each $i = 1, 2$, where we have $\mathbf{dom}(V_i) \subseteq \mathbf{var}(s_i)$. Therefore $(v, V) \lhd s$ with $V = 1V_1 \cup 2V_2 \cup \Xi(s) \times \{\epsilon\}$. We have $\mathbf{dom}(V) = \mathbf{dom}(V_1) \cup \mathbf{dom}(V_2) \cup \Xi(s) \subseteq \mathbf{var}(s_1) \cup \mathbf{var}(s_2) \cup \Xi(s) \subsetneq \mathbf{var}(s)$. Since A is empty-eliminated and, moreover, (L2) holds, this allows us to prove, for each s',

> if $s \in \pi s'$, then $(v', V') \lhd s'$ with $\mathbf{dom}(V') \subsetneq \mathbf{var}(s')$ for some v' and V'

by induction on the structure of π analogously to Lemma 8. Therefore $(v', V') \lhd A$ with $\mathbf{dom}(V') \subsetneq \mathbf{var}(I)$ for some v' and V'.

(2') Since (L1') does not hold, let $s \to a \in \delta$ and $\Xi(s) \subsetneq \mathbf{var}(s)$ (note that we always have $\Xi(s) \subseteq \mathbf{var}(s)$). Therefore $(v, V) \lhd s$ with $V = \Xi(s) \times \{\epsilon\}$. But $\mathbf{dom}(V) = \Xi(s) \subsetneq \mathbf{var}(s)$. By the same argument as in (2), we have $(v', V') \lhd A$ with $\mathbf{dom}(V') \subsetneq \mathbf{var}(I)$ for some v' and V'.

(3) Since (L3) does not hold, take $s \in I$ such that $\mathbf{var}(s) \subsetneq \mathbf{var}(I)$. Since s is not empty, $(v, V) \lhd s$. Then, the result follows from the east fact $\mathbf{dom}(V) \subseteq \mathbf{var}(s)$. □

4.3 Application of Type Substitution

Substitution is used for instantiating type variables in polymorphic types with concrete types. This happens at computing the result type of a call to polymorphic function (rule TApp in 3.5). The precise definition is given in Section 3.3. For brevity, we formalize here an algorithm for only single-variable substitutions. We can easily treat multiple variables simply by transforming a multi-variable substitution with successive applications of single-variable substitutions. (This will involve renaming of variables introduced by one substitution so that another substitution will not capture them.)

We present below an algorithm that, given two marking automata A and B and a variable x, calculates a marking automaton C representing $\{x \mapsto B\}A$. Let $A = (S_A, I_A, \delta_A, \Xi_A)$ and $B = (S_B, I_B, \delta_B, \Xi_B)$. Construct $C = (S_C, I_C, \delta_C, \Xi_C)$ with the following conditions such that

$$
\begin{aligned}
S_C &= S_C^{\times} \cup S_C^{\top} \\
&\quad \text{where } S_C^{\times} = S_A \times S_B \\
&\qquad\quad\ S_C^{\top} = \{(s, \top) \mid s \in S_A, x \notin \Xi_A(s)\} \\
I_C &= \{(s,t) \in S_C^{\top} \cup I_C^x \mid s \in I_A\} \\
&\quad \text{where } I_C^x = \{(s,t) \mid s \in S_A, x \in \Xi_A(s), t \in I_B, \} \\
\Xi_C((s,t)) &= \begin{cases} \Xi_A(s) \setminus \{x\} \cup \Xi_B(t) & ((s,t) \in S_C^{\times}) \\ \Xi_A(s) & ((s,t) \in S_C^{\top}). \end{cases}
\end{aligned}
$$

and

(T1). $(s,t) \to ((s_1,t_1),(s_2,t_2)) \in \delta_C$ iff $s \to (s_1, s_2) \in \delta_A$ and either
—$(s,t) \in S_C^{\times}$ and $t \to (t_1, t_2) \in \delta_B$, or
—$(s,t) \in S_C^{\top}$ and $(s_j, t_j) \in S_C^{\top} \cup I_C^x$ for $j = 1, 2$.

(T2). $(s,t) \to a \in \delta_C$ iff $s \to a \in \delta_A$ and either
—$(s,t) \in S_C^{\times}$ and $t \to a \in \delta_B$, or

—$(s,t) \in S_C^\top$.

The marking automaton produced by the algorithm contains two kinds of states, (s,t) in $S_C^\times$ and $(s,\top)$ in $S_C^\top$. Transitions are constructed as follows. Starting with a state $(s_0,\top) \in S_C^\top$ where $s_0 \in I_A$, we follow only A's transitions (keeping all variables in Ξ_A) until we reach a state s_1 with x in $\Xi_A(s_1)$. At that moment, we switch to a state (s_1,t_0) where $t_0 \in I_B$ (we write the set of such states by I_C^x) and, from then on, we follow both A's and B's transitions simultaneously. Thus, we effectively take the intersection of A (from s_1) and B (from t_0). In a state (s,t) in $S_C^\times$, we keep all variables in Ξ_A and Ξ_B except x. We need to take some care at the beginning where if an initial state s_0 of A already contains x, then we need to start with $(s_0,t_0) \in I_C^x$ where $t_0 \in I_B$.

PROPOSITION 3. *Let C be the marking automaton resulted from the substitution algorithm above for given marking automata A and B. Then, C represents $\{x \mapsto B\}A$.*

PROOF: Let $X = \{x\}$. To prove the result, it suffices to show the following for all v, V and $s \in S_A, t \in S_B$:

$$(v,V) \lhd (s,t) \tag{1}$$

iff there exist U and W such that

$$(v,W) \lhd s \tag{2}$$
$$V = W|_{\overline{X}} \cup U \tag{3}$$

where U satisfies

(a). in the case $(s,t) \in S_C^\times$, that $(v,U) \lhd t$, and

(b). in the case $(s,t) \in S_C^\top$, that $U = \bigcup_{l=1..k} \pi^l R^l$ where $\{(x : \pi^1), \ldots, (x : \pi^k)\} = W|_X$ and $(\pi^l(v), R^l) \lhd I_B$ for some $k, \pi^1, \ldots, \pi^k, R^1, \ldots, R^k$.

We first prove the "only if" direction. The proof proceeds by induction on the structure of v.

Case: $v = a$

When $(s,t) \in S_C^\times$, (1) together with (T2) implies all of the following.

$$s \to a \in \delta_A \tag{4}$$
$$t \to a \in \delta_B \tag{5}$$
$$V = (\Xi_A(s) \setminus X \cup \Xi_B(t)) \times \{\epsilon\} \tag{6}$$

Let $W = \Xi_A(s) \times \{\epsilon\}$ and $U = \Xi_B(t) \times \{\epsilon\}$. Then, from (4), we obtain (2), and from (5) and (6), we obtain (3) with the condition (a). When $(s,t) \in S_C^\top$, (1) with (T2) implies (4) and

$$V = \Xi_A(s) \times \{\epsilon\}. \tag{7}$$

Let $W = \Xi_A(s) \times \{\epsilon\}$ and $U = \emptyset$. (4) implies (2). From (7), we obtain (3) with (b) since $x \notin \Xi_A(s)$ and therefore $W|_X = \emptyset$.

Case: $v = (v_1, v_2)$

When $(s,t) \in S_C^{\times}$, (1) with (T1) implies all of the following

$$s \rightarrow (s_1, s_2) \in \delta_A \tag{8}$$
$$t \rightarrow (t_1, t_2) \in \delta_B \tag{9}$$
$$(v_j, V_j) \lhd (s_j, t_j) \quad \forall j = 1,2 \tag{10}$$
$$V = (\Xi_A(s) \setminus X \cup \Xi_B(t)) \times \{\epsilon\} \cup 1V_1 \cup 2V_2 \tag{11}$$

for some V_1, V_2 and $s_1, s_2 \in S_A, t_1, t_2 \in T_B$. From (10), by the induction hypothesis, for each $j = 1,2$, there exist W_j and U_j satisfying the following.

$$(v_j, W_j) \lhd s_j \tag{12}$$
$$V_j = W_j|_{\overline{X}} \cup U_j \tag{13}$$
$$(v_j, U_j) \lhd t_j \tag{14}$$

Let $W = \Xi_A(s) \times \{\epsilon\} \cup 1W_1 \cup 2W_2$ and $U = \Xi_B(t) \times \{\epsilon\} \cup 1U_1 \cup 2U_2$. Then, by (8) and (12), we obtain (2). From (9) and (14), the condition (a) follows. By (11) and (13), we obtain (3).

When $(s,t) \in S_C^{\top}$, (1) with (T1) implies all of (8), (10), and the following

$$(s_j, t_j) \in S_C^{\top} \cup I_C^x \quad \forall j = 1,2 \tag{15}$$
$$V = \Xi_A(s) \times \{\epsilon\} \cup 1V_1 \cup 2V_2 \tag{16}$$

for some V_1, V_2 and $s_1, s_2 \in S_A, t_1, t_2 \in S_B$. From (10), by the induction hypothesis, for each $j = 1,2$, there exist W_j and U_j such that (12) and (13) where either

$$(s_j, t_j) \in S_C^{\times} \quad (v_j, U_j) \lhd t_j \tag{17}$$

or, for some $k_j, \pi_j^1, \ldots, \pi_j^{k_j}, R_j^1, \ldots, R_j^{k_j}$,

$$(s_j, t_j) \in S_C^{\top} \quad U_j = \bigcup_{l=1..k_j} \pi_j^l R_j^l \quad (\pi_j^l(v_j), R_j^l) \lhd I_B \quad \{(x : \pi_j^1), \ldots, (x : \pi_j^{k_j})\} = W_j|_X. \tag{18}$$

Let $U = 1U_1 \cup 2U_2$ and $W = \Xi_A(s) \times \{\epsilon\} \cup 1W_1 \cup 2W_2$. From (8) and (12), we obtain (2). From (16), (13), and $x \notin \Xi_A(s)$, we obtain (3). For $j = 1,2$, when $(s_j, t_j) \in I_C^x$, from (17) together with the linearity restriction, we have $W_j|_X = \{x : \epsilon\}$. Further, from $t_j \in I_B$, we obtain, for some $k_j, \pi_j^1, \ldots, \pi_j^{k_j}, R_j^1, \ldots, R_j^{k_j}$,

$$U_j = \bigcup_{l=1..k_j} \pi_j^l R_j^l \quad (\pi_j^l(v_j), R_j^l) \lhd I_B \quad \{(x : \pi_j^1), \ldots, (x : \pi_j^{k_j})\} = W_j|_X. \tag{19}$$

Therefore, from (15), it must be either $(s_j, t_j) \in S_C^{\top}$ or $(s_j, t_j) \in I_C^x$ and, in either case, (19) holds. From this, the condition (b) follows.

We then prove the "if" direction by induction on the structure of v.

Case: $v = a$

When $(s,t) \in S_C^{\times}$, from (2), we have (4) with $W = \Xi_A(s) \times \{\epsilon\}$. Since we have (3) with the condition (a), we have (5) with $U = \Xi_B(t) \times \{\epsilon\}$, implying also (6). From (T2), (1) follows. When $(s,t) \in S_C^{\top}$, from (2), we have (4) with $W = \Xi_A(s) \times \{\epsilon\}$.

Since $x \notin \Xi_A(s)$, we have $W|_X = \emptyset$. Therefore the condition (b) implies $U = \emptyset$, from which we obtain (7). From (T2), (1) follows.

Case: $v = (v_1, v_2)$

When $(s,t) \in S_C^\times$, from (2), we have (8) for some $s_1, s_2 \in S_A$ and (12) for each $j = 1,2$ with $W = \Xi_A(s) \times \{\epsilon\} \cup 1W_1 \cup 2W_2$ for some W_1, W_2. Also, from the condition (a), we have (9) for some $t_1, t_2 \in S_B$ and (14) for $j = 1,2$ with $U = \Xi_B(t) \times \{\epsilon\} \cup 1U_1 \cup 2U_2$ for some U_1, U_2. Further, from (3), we obtain (11) by defining V_1 and V_2 as (13). By the induction hypothesis, (10) holds for $j = 1,2$. From (T1), (1) follows.

When $(s,t) \in S_C^\top$, again from (2), we have (8) for some $s_1, s_2 \in S_A$ and (12) for each $j = 1,2$ with $W = \Xi_A(s) \times \{\epsilon\} \cup 1W_1 \cup 2W_2$ for some W_1, W_2. Since $x \notin \Xi_A(s)$, the condition (b) implies that there are U_1 and U_2 satisfying $U = 1U_1 \cup 2U_2$ and (19) for some $k_j, \pi_j^1, \ldots, \pi_j^{k_j}, R_j^1, \ldots, R_j^{k_j}$ for $j = 1,2$. Further, from (3), we obtain (16) by defining V_1 and V_2 as (13). For each $j = 1,2$, there are two cases:

—$(x : \epsilon) \in W_j|_X$. In this case, $x \in \Xi_A(s_j)$. By the linearity condition, $k_j = 1$ and $\pi_j^1 = \epsilon$ with $U_j = R_j^1$. In addition, since $(\pi_j^1(v_j), R_j^1) \lhd I_B$, we have $(v_j, U_j) \lhd t_j$ for some $t_j \in I_B$. Thus, $(s_j, t_j) \in I_C^x$. By the induction hypothesis, (10) holds.

—$(x : \epsilon) \notin W_j|_X$. In this case, $x \notin \Xi_A(s_j)$. By letting $t_j = \top$, we have $(s_j, t_j) \in S_C^\top$. By the induction hypothesis, (10) holds.

Therefore, from (T1), we have $(s,t) \to ((s_1,t_1),(s_2,t_2)) \in \delta_C$. Hence, (1) follows. □

4.4 Subtyping

Subtyping is used whenever a value is transferred from one context to another, namely, at a function call (rule TApp), upon a function return (rule TFun), and for checking exhaustiveness of a pattern match (rule TMatch). The precise goal of subtype check is, given marking automata A and B, to check whether, whenever $(v,V) \lhd A$, we have $(v,W) \lhd B$ for some $W \subseteq V$. Let $A = (S_A, I_A, \delta_A, \Xi_A)$ and $B = (S_B, I_B, \delta_B, \Xi_B)$. Then, our subtyping algorithm is as follows.

(i) Construct $C = (S_C, I_C, \delta_C, \Xi_C)$ such that

$$\begin{aligned} S_C &= S_A \times \mathcal{P}(S_B) \\ I_C &= I_A \times \{I_B\} \\ \Xi_C((s,T)) &= \emptyset \end{aligned}$$

and

(S1). $(s,T) \to ((s_1,T_1),(s_2,T_2)) \in \delta_C$ iff

—$s \to (s_1,s_2) \in \delta_A$ and

—for each $t \in T$ where $\Xi_B(t) \subseteq \Xi_A(s)$, if $t \to (t_1,t_2) \in \delta_B$, then either $t_1 \in T_1$ or $t_2 \in T_2$

(S2). $(s,T) \to a \in \delta_C$ iff

—$s \to a \in \delta_A$ and

—for each $t \in T$ where $\Xi_B(t) \subseteq \Xi_A(s)$, we have $t \to a \notin \delta_B$.

(ii) Return "yes" iff C is empty (i.e., $\nexists (v,V).\ (v,V) \lhd C$).

That is, we first construct a marking automaton C that accepts marked values $(v, \emptyset)$ such that A accepts (v, V) for some V, but B does not accept (v, W) for any $W \subseteq V$. Then, we check whether the automaton C is empty. The automaton C contains states of the form $(s, \{t_1, \ldots, t_n\})$ such that $s \in S_A$ and each $t_i \in S_B$. Intuitively, the state $(s, \{t_1, \ldots, t_n\})$ accepts $(v, \emptyset)$ where s accepts (v, V) for some V, but any t_i does not accept (v, W) for any $W \subseteq V$. Hence, we set $I_A \times \{I_B\}$ as C's initial states. Rules (S1) and (S2) for constructing C's transition rules can be understood as follows. For rule (S1), consider a state (s, T) and a value (v_1, v_2). Suppose that (1) s accepts $((v_1, v_2), V)$, but (2) any $t \in T$ does not accept $((v_1, v_2), W)$ for any $W \subseteq V$. Condition (1) means that a transition rule $s \to (s_1, s_2)$ is in δ_A where s_j accepts (v_j, V_j) for some V_j for $j = 1, 2$ and $V = 1V_1 \cup 2V_2 \cup \Xi_A(s) \times \{\epsilon\}$. Condition (2) means that, for any $t \to (t_1, t_2)$ in δ_B where $t \in T$, either $\Xi_B(t) \not\subseteq \Xi_A(s)$ (in which case, $W \not\subseteq V$ whenever t accepts $((v_1, v_2), W)$), or t_1 does not accept (v_1, W_1) for any $W_1 \subseteq V_1$, or t_2 does not accept (v_2, W_2) for any $W_2 \subseteq V_2$. Rule (S2) is analogous.

The above presentation does not directly give an efficient algorithm for subtyping. One way of obtaining a practical algorithm is to adapt a subtyping algorithm for monomorphic types proposed in Hosoya, Vouillon, and Pierce [Hosoya et al. 2004]. Their algorithm computes essentially what our algorithm above does, minus the treatment of variables, i.e., the check "$\Xi_B(t) \subseteq \Xi_A(s)$," but is elaborated with various techniques for efficiency, including a lazy, top-down strategy for state exploration and a "working set" data structure for avoiding repeated computations. Thus, we can easily obtain an efficient subtyping algorithm for polymorphic types by just augmenting their algorithm with a rule for treating variables.[6]

PROPOSITION 4. *Given marking automata A and B, the subtyping algorithm returns "yes" for A and B iff $A \leqslant B$.*

PROOF: To prove the result, it suffices to show the following for all v and $s \in S_A, T \subseteq S_B$:

$$(v, V) \lhd s \tag{20}$$

$$\forall t \in T.\ \nexists W.\ (v, W) \lhd t \ \wedge\ W \subseteq V \tag{21}$$

for some V, iff

$$(v, \emptyset) \lhd (s, T). \tag{22}$$

We prove this by induction on the structure of v.

Case: $v = a$

(20) and (21) are equivalent to

$$s \to a \in \delta_A \quad \wedge \quad V = \Xi_A(s) \times \{\epsilon\} \tag{23}$$

$$\forall t \in T.\ t \to a \notin \delta_B \ \vee\ \Xi_B(t) \times \{\epsilon\} \not\subseteq V \tag{24}$$

[6]For readers familiar with the algorithm in [Hosoya et al. 2004], the additional rule would look like:

$$\frac{\Xi_B(t_1) \not\subseteq \Xi_A(s) \qquad s \leqslant t_2 | \ldots | t_n}{s \leqslant t_1 | t_2 | \ldots | t_n}$$

respectively. By the definition of δ_C, both of these hold if and only if $(s,T) \to a \in \delta_C$, that is, $(v,\emptyset) \lhd (s,T)$.

Case: $v = (v_1, v_2)$

(20) holds if and only if both of the following hold

$$(v_i, V_i) \lhd s_i \quad (\forall i = 1,2) \tag{25}$$

$$V = \Xi_A(s) \times \{\epsilon\} \cup 1V_1 \cup 2V_2 \tag{26}$$

for some V_1, V_2 and $s_1, s_2 \in S_A$. (21) is equivalent to that, for any $t \in T$, whenever there are $t_1, t_2 \in S_B$ such that

$$t \to (t_1, t_2) \in \delta_B \tag{27}$$

we have

$$\forall W_1, W_2.\ (\neg(v_1, W_1) \lhd t_1 \quad \vee \quad \neg(v_2, W_2) \lhd t_2 \quad \vee \quad \Xi_B(t) \times \{\epsilon\} \cup 1W_1 \cup 2W_2 \not\subseteq V). \tag{28}$$

(28) can be transformed to:

$$\Xi_B(t) \subseteq \Xi_A(s) \quad \Rightarrow \quad \nexists W_i.(v_i, W_i) \lhd t_i \wedge W_i \subseteq V_i \quad (\exists i = 1,2) \tag{29}$$

Let $T_i = \{t_i \mid t \in T \ \wedge\ \Xi_B(t) \subseteq \Xi_A(s) \ \wedge\ t \to (t_1,t_2) \in \delta_B \wedge\ \nexists W_i.\ ((v_i, W_i) \lhd t_i \ \wedge\ W_i \subseteq V_i)\}$ for $i = 1,2$. Then, "$\forall t \in T.\ \exists t_1, t_2.$ (27) implies (29)" can be rephrased to:

$$\forall t \in T.\ \exists t_1, t_2.\ \Xi_B(t) \subseteq \Xi_A(s) \ \wedge\ t \to (t_1,t_2) \in \delta_B \quad \Rightarrow \quad t_1 \in T_1 \vee t_2 \in T_2 \tag{30}$$

Further, for each $i = 1,2$, by the definition of T_i, we have,

$$\forall t_i \in T_i.\ \ \nexists W_i.\ (v_i, W_i) \lhd t_i \ \wedge\ W_i \subseteq V_i. \tag{31}$$

By the induction hypothesis, both (25) and (31) hold iff

$$(v_i, \emptyset) \lhd (s_i, T_i) \quad (\forall i = 1,2) \tag{32}$$

holds. From (S1), we have that (30) holds iff

$$(s,T) \to ((s_1,T_1),(s_2,T_2)) \in \delta_C \tag{33}$$

holds. Further, "(32)∧(33)" is equivalent to $(v,\emptyset) \lhd (s,T)$. □

4.5 Inference

Type inference is used for two purposes: inference of type arguments at a call to a polymorphic function and inference of types for term variables in patterns at a pattern match. The precise goal is as follows: given two marking automata A and B and a set $X = \{x_1, \ldots, x_n\}$ of variables, obtain an X-inference of B with respect to A, that is, a mapping $\{x_1 \mapsto D_{x_1}, \ldots, x_n \mapsto D_{x_n}\}$ such that, for each v, V, π, and x_i, we have that A accepts (v,V) and B accepts (v,W) for some W with $(x_i : \pi) \in W$ if and only if D_{x_i} accepts $\pi(v,V)$ (i.e., the extraction of (v,V) by the path π).

Let $A = (S_A, I_A, \delta_A, \Xi_A)$ and $B = (S_B, I_B, \delta_B, \Xi_B)$. We assume $X \subseteq \mathbf{var}(B)$ and $\mathbf{var}(I_A) \cap \mathbf{var}(I_B) = \emptyset$. Then, the inference algorithm is as follows.

(i) Construct $C = (S_C, I_C, \delta_C, \Xi_C)$ such that

$$\begin{aligned} S_C &= S_A \times S_B \\ I_C &= I_A \times I_B \\ \Xi_C(s,t) &= \Xi_A(s) \cup \Xi_B(t) \end{aligned}$$

and
(I1). $(s,t) \to ((s_1,t_1),(s_2,t_2)) \in \delta_C$ iff $s \to (s_1,s_2) \in \delta_A$ and $t \to (t_1,t_2) \in \delta_B$
(I2). $(s,t) \to a \in \delta_C$ iff $s \to a \in \delta_A$ and $s \to a \in \delta_B$.

(ii) Empty-eliminate C.

(iii) For each $x \in X$, construct $D_x = (S_{D_x}, I_{D_x}, \delta_{D_x}, \Xi_{D_x})$ such that

$$\begin{aligned} S_{D_x} &= S_C \\ I_{D_x} &= \{s \mid x \in \Xi_C(s)\} \\ \delta_{D_x} &= \delta_C \\ \Xi_{D_x}((s,t)) &= \Xi_C(s,t) \cap \mathbf{var}(I_A). \end{aligned}$$

That is, we first compute a product of A and B to obtain a specialization C of the "target" B with respect to the "domain" A. Thus, the automaton C behaves exactly the same as B except that it accepts only marked values that are also accepted by A. Therefore, whenever B matches a marked value (v,V) accepted by A and yields a binding of x to another marked value (u,U), the automaton C accepts the marked value (u,U) at some state (s,t) that marks x. Each result automaton D_x is essentially a copy of C where D_x starts from C's states that have x in their variable sets. The empty elimination performed in the second step is necessary to guarantee that each D_x accepts no more than the appropriate marked values. To see this, note first that, after the empty elimination of C, each state (s,t) is non-empty and reachable from an initial state. Therefore, whenever D_x has an initial state (s,t), or equivalently (empty-eliminated) C has a state (s,t) that marks x, there is some marked value (v,V) that is accepted by (empty-eliminated) C and produces a binding of x to another marked value (u,U) at the state (s,t). This means that both A and B accept the marked value (v,V) and B yields (u,U) at the state t.

The algorithm above can be seen as a straightforward adaptation of (monomorphic) type inference algorithms for patterns presented in [Hosoya and Pierce 2002; Hosoya 2003]. Those previous algorithms have treated features not considered here (such as first-match patterns and non-tail variables) whereas the present one treats polymorphic types (which is achieved just by allowing variables in the domain type).

LEMMA 10. *$(v,V) \lhd s$ and $(v,W) \lhd t$ for some V and W with $U = V \cup W$ if and only if $(v,U) \lhd (s,t)$.*

PROOF: Straightforward induction on the structure of v. □

PROPOSITION 5. *Let $D_{x_1}, \ldots, D_{x_n}$ be the marking automata resulted from the type inference algorithm from given marking automata A and B, and a set $X = \{x_1, \ldots, x_n\}$ of variables. Then, $\{x_1 \mapsto D_{x_1}, \ldots, x_n \mapsto D_{x_n}\}$ is an X-inference of B with respect to A.*

PROOF: To prove the result, it suffices to show the following for any u, U:

$$(v, V) \lhd s \text{ and } (w, W) \lhd t \text{ with } (x : \pi) \in W \text{ and } (u, U) = \pi(v, V)$$

for some x, π, v, w, V, W and $s \in S_A, t \in S_B$, if and only if

$$(u, U) \lhd D_x.$$

We prove the "only if" direction by induction on the structure of π.

Case: $\pi = \epsilon$

From Lemma 10, $(v, V \cup W) \lhd (s, t)$. Since $x \in \Xi_B(t) \subseteq \Xi_C((s, t))$, we have $(s, t) \in I_C$. Since (s, t) is not empty, $(s, t) \in I_{D_x}$ and therefore $(v, (V \cup W)|_{\mathbf{var}(I_A)}) \lhd D_x$, where $(V \cup W)|_{\mathbf{var}(I_A)} = V$. This implies $(u, U) \lhd D_x$.

Case: $\pi = j\pi'$

It must be that v has the form (v_1, v_2) and therefore $(u, U) = j\pi'(v, V) = (\pi' v_j, \pi'^{-1} j^{-1} V)$. From the assumption, we have $s \to (s_1, s_2) \in \delta_A$ and $t \to (t_1, t_2) \in \delta_B$ for some $s_1, s_2 \in S_A$ and $t_1, t_2 \in S_B$ where $(v_j, V') \lhd s_j$ for some $V' = j^{-1}V$ and $(v_j, W') \lhd t_j$ for some $W' = j^{-1}W$. Therefore, $(x : \pi') \in W'$. By the induction hypothesis, $(\pi' v_j, \pi'^{-1} V') \lhd D_x$, that is, $(u, U) \lhd D_x$.

We then prove the "if" direction. From $(u, U) \lhd D_x$, we have $(u, U) \lhd (s, t)$ for some $(s, t) \in I_{D_x}$, the latter implying that $x \in \Xi_C((s, t))$ and therefore $(x : \epsilon) \in U$. Also, since (s, t) is not eliminated in D, we have $(s, t) \in \pi I_C$ for some π. From Lemma 8, $(v, U') \lhd (p', q')$ for some $(p', q') \in I_C$ and some v and U' with $\pi(v, U') = (u, U)$. Therefore $(v, U') \lhd C$ and $(x : \pi) \in U'$. By Lemma 10, we have $(v, V) \lhd A$ and $(v, W) \lhd B$ for some V, W with $U' = V \cup W$. Since $x \in \mathbf{var}(I_B)$, we have $(x : \pi) \in W$. □

4.6 Ambiguity

Ambiguity is checked on the parameter types of each function definition (rule TFun). Once the parameter types are ensured to be unambiguous, the results of inference of type arguments at each function call (rule TApp) are guaranteed to be minimum (Proposition 1) if the types of the actual arguments are subtypes of the parameter types ignoring type varibles (rule TApp).

The precise goal is to find whether, given a marking automaton A, any value matched by A is marked in a unique way, that is, $V = W$ whenever A accepts (v, V) and (v, W). The algorithm for ambiguity check is as follows. Let $A = (S_A, I_A, \delta_A, \Xi_A)$.

(i) Construct $C = (S_C, I_C, \delta_C, \Xi_C)$ such that

$$\begin{aligned} S_C &= S_A \times S_A \\ I_C &= I_A \times I_A \\ \Xi_C((s, t)) &= (\Xi_A(s) \setminus \Xi_A(t)) \cup (\Xi_A(t) \setminus \Xi_A(s)) \end{aligned}$$

and

(A1). $(s, t) \to ((s_1, t_1), (s_2, t_2)) \in \delta_C$ iff $s \to (s_1, s_2) \in \delta_A$ and $t \to (t_1, t_2) \in \delta_A$

(A2). $(s, t) \to a \in \delta_C$ iff $s \to a \in \delta_A$ and $s \to a \in \delta_A$.

(ii) Empty-eliminate C; let $D = (S_D, I_D, \delta_D, \Xi_D)$ be the result.

(iii) Return "unambiguous" iff $\Xi_D(s, t) = \emptyset$ for each $(s, t) \in S_D$.

That is, we first take the self-product C of A, where we set the variables $\Xi_C((s,t))$ of each state (s,t) such that $\Xi_C((s,t)) = \emptyset$ if and only if $\Xi_A(s) = \Xi_A(t)$. The second and third steps can be understood as follows. For the empty elimination D of C, suppose $\Xi_D((s,t)) = \emptyset$ for all the states (s,t) and A accepts (v,V) and (v,W). Then, the first step ensures that, for each subnode of v, the states s' and t' assigned to this node in these two cases have the same set of variables: $\Xi_A(s') = \Xi_A(t')$. Therefore the whole markings V and W must also be the same. On the other hand, suppose $\Xi_D((s,t)) \neq \emptyset$ for some state (s,t), that is, $\Xi_A(s) \neq \Xi_B(t)$. Then, since D is empty-eliminated, there is some value v that is matched by D and produces a marking at the state (s,t). This means that A has two ways of matching the value v that put different marks on the same subnode at the states s and t.

The above algorithm is similarly to the ambiguity check algorithm for tree automata presented in [Hosoya 2003] in the sense that both are based on self-product. The difference is that the previous one checks for a unique *run* of an automaton whereas the present one for a unique *marking* (the former implies the latter but not necessarily conversely).

LEMMA 11. *$(v,V) \lhd s$ and $(v,W) \lhd t$ for some V and W with $U = V \backslash W \cup W \backslash V$ if and only if $(v,U) \lhd (s,t)$.*

PROOF: By induction on the structure of v.

Case: $v = a$

$(v,V) \lhd s$ and $(v,W) \lhd t$ if and only if

$$s \to a \in \delta_A \quad V = \Xi_A(s) \times \{\epsilon\} \tag{34}$$
$$t \to a \in \delta_A \quad W = \Xi_A(t) \times \{\epsilon\}. \tag{35}$$

From (A2), we have that (34) and (35) with $U = (V \setminus W) \cup (W \setminus V)$ hold iff

$$(s,t) \to a \in \delta_C \quad U = \Xi_C((s,t)) \times \{\epsilon\}.$$

holds. That is, $(v,U) \lhd (s,t)$.

Case: $v = (v_1, v_2)$

$(v,V) \lhd s$ and $(v,W) \lhd t$ if and only if

$$s \to (s_1, s_2) \in \delta_A \tag{36}$$
$$(v_j, V_j) \lhd s_j \quad (j = 1,2) \tag{37}$$
$$V = \Xi_A(s) \times \{\epsilon\} \cup 1V_1 \cup 2V_2 \tag{38}$$
$$t \to (t_1, t_2) \in \delta_A \tag{39}$$
$$(v_j, V_j) \lhd t_j \quad (j = 1,2) \tag{40}$$
$$W = \Xi_A(t) \times \{\epsilon\} \cup 1W_1 \cup 2W_2 \tag{41}$$

for some V_1, V_2, W_1, W_2 and $s_1, s_2, t_1, t_2 \in S_A$. By the induction hypothesis, (37) and (40) with $U_j = (V_j \setminus W_j) \cup (W_j \setminus V_j)$ hold iff

$$(v_j, U_j) \lhd (s_j, t_j) \quad (\forall j = 1,2) \tag{42}$$

holds. From (A1), (36) and (39) hold iff

$$(s,t) \to ((s_1,t_1),(s_2,t_2)) \in \delta_C \tag{43}$$

holds. Finally, (42), (43), (38), and (41) with $U = (V \setminus W) \cup (W \setminus V)$ altogether hold iff $(v, U) \lhd (s, t)$. □

PROPOSITION 6. *Given a marking automaton A and a set X of variables, the ambiguity-checking algorithm returns "unambiguous" for A and X iff A is X-unambiguous.*

PROOF: To prove the result, it suffices to show:

$$(v, V) \lhd I_A \quad \wedge \quad (v, W) \lhd I_A \quad \Rightarrow \quad V = W \quad (\forall v, V, W) \tag{44}$$

if and only if

$$\Xi_D((s, t)) = \emptyset \quad (\forall (s, t) \in S_D). \tag{45}$$

We first show the "only if" direction. Suppose that $\Xi_D((s, t)) \neq \emptyset$ for some $(s, t) \in S_D$. Since D is empty-eliminated, $(s, t) \in \pi I_D$ and $(u, U) \lhd (s, t)$ for some u and $U \neq \emptyset$. By Lemma 8, $(v, V') \lhd I_D$ (therefore $(v, V') \lhd I_C$) for some v,V' with $\pi(v, V') = (u, U)$, which implies $V' \neq \emptyset$. By Lemma 11, $(v, V) \lhd I_A$ and $(v, W) \lhd I_B$ for some V and W with $V \neq W$.

We then show the "if" direction. Suppose (45) with $(v, V) \lhd I_A$ and $(v, W) \lhd I_A$ where $V \neq W$. Then, by Lemma 11, $(v, U) \lhd I_C$ (therefore $(v, U) \lhd I_D$) for some $U \neq \emptyset$. This implies $\Xi_D((s', t')) \neq \emptyset$ for some $(s', t') \in S_D$, contradicting (45). □

5. EXTENSIONS

This section first discusses an extension of our type system with intersection and difference operations and then another with type and pattern variables capturing intermediate sequences.

5.1 Intersection and Difference Operators

As in ℂDuce [Benzaken et al. 2003], it would be useful to support intersection and difference operators on types. The former can combine two simultaneous constraints on data values. The latter can exclude data values that satisfy a given constraint.

Since our polymorphic types involve markings, we cannot directly use set intersections and differences for the semantics of the corresponding operators. Rather, we need to define them in terms of subtyping. The intersection can easily be defined as the greatest lower bound of types (with respect to $\leqslant$): for any v and V, we have that $(v, V) \lhd p_1 \cap p_2$ iff $(v, V_1) \lhd p_1$ and $(v, V_2) \lhd p_2$ for some V_1 and V_2 with $V = V_1 \cup V_2$. (Indeed, this definition coincides with the semantics of *intersection patterns* in ℂDuce.) Note that, from the definition of substitution, we have, for any v and V, that $(v, V) \lhd [x \mapsto p_1](x : p_2)$ iff $(v, V_1) \lhd p_1$ and $(v, V_2) \lhd p_2$ for some V_1 and V_2 with $V = V_1 \cup V_2$. This accounts for the fact that variables with type constraints are essentially intersection types. This also means that, by using our substitution algorithm, intersection types can be converted to types involving no intersection. Therefore, as a matter of facts, we effectively already support intersections.

For the difference operator, we could analogously define it as follows: for any v and V, we have that $(v, V) \lhd p_1 \setminus p_2$ iff $(v, V_1) \lhd p_1$ for some $V_1 \subseteq V$ and there is no $V_2 \subseteq V$ such that $(v, V_2) \lhd p_2$. However, unlike the case of intersections, such a type $p_1 \setminus p_2$ deos not seem to be expressible in general by types without the

difference operator. For example, let $p_1 = a$ and $p_2 = x : a$. Then, $p_1 \setminus p_2$ would accept a and yield any marking that does not contain $(x : \epsilon)$. Since there can be an infinite number of variables, such a constraint cannot be represented. One approach to solve this problem might be to take difference as a built-in operator, but we have not yet pursued this direction.

Though the general case fails, we can easily represent the difference in the case where p_2 contains no variables. (A concrete procedure can be obtained by a slight variation of our subtyping algorithm in Section 4.4.) This is enough for the type-checker to treat a first-match semantics of patterns, where we need to calculate the difference between the input type and the union of the types of the "preceding" patterns. (Indeed, this is the only case that we have identified where differences are useful.)

5.2 Marking Intermediate Sequences

As already mentioned before, our encoding of patterns and polytypes does not allow non-tail, non-individual sequences to be marked. In this section, we briefly discuss the design space occurring from relaxing this restriction.

First of all, it is well known how to achieve this for patterns. One approach is to reinterpret a variable in a pattern so that it captures the sequence obtained from concatenating all the elements marked by the variable [Hosoya 2003; Frisch et al. 2002]. For example, in order for the variable `y` to capture the intermediate sequence `b[],c[]` in the value `a[b[],c[],d[]]`, we use the following pattern in the internal representation

$$((\mathtt{a}, ((\mathtt{y} : (\mathtt{b}, \nu)), ((\mathtt{y} : (\mathtt{c}, \nu)), ((\mathtt{d}, \nu), \nu)))), \nu)$$

(depicted as the second tree in Figure 3). (Another approach is to introduce two variables $\mathtt{y}_B$ and $\mathtt{y}_E$ for each variable y and mark with these the starting and the ending nodes of the intermediate sequence to capture [Hosoya and Pierce 2002]. The first approach is more advantageous since it can capture also non-consecutive sequences.)

However, this encoding critically relies on the linearity condition of patterns. For polytypes, several intermediate sequences in the same sequence could be separately marked. To encode this in a marked internal value, we would need to introduce "separators" between those intermediate sequences and extend our automata to accommodate such marked values.

One design choice to avoid this complication would be to simply disallow multiple intermediate sequences in the same level to be marked. However, there is an orthogonal complication caused by allowing both patterns to extract intermediate sequences and polytypes to mark non-individual sequence *possibly in tail position*. That is, a pattern could extract only a part of such marked sequence. For example, we could write the pattern

```
match e with
  a[(b[],c[]) as y, d[]] -> ...
```

and feed to it the marked value

```
a[b[],X:(c[],d[])].
```

In this case, we need to remove the mark `X` for `y`'s binding. We can easily imagine that this will substantially complicate both the semantics of pattern matching and the type inference for patterns.

Our proposal (adopted by XDuce 0.5.0) is: allow patterns to extract arbitrary intermediate sequences but let polytypes mark only individual elements (possibly multiple ones in the same sequence). For example, the polytype `a[b[],X:c[],X:d[]]` can be encoded by

$$((\mathtt{a}, ((\mathtt{b}, \nu), (\mathtt{X} : (\mathtt{c}, \nu), (\mathtt{X} : (\mathtt{d}, \nu), \nu)))), \nu)$$

(note that `X` marks the two elements `c` and `d` separately) and, from a marked value of this type, the pattern `a[(b[],c[]) as y, d[]]` will extract the first two elements of `a` as follows:

$$((\mathtt{b}, \nu), (\mathtt{X} : (\mathtt{c}, \nu), \nu))$$

This extraction is depicted in Figure 3.

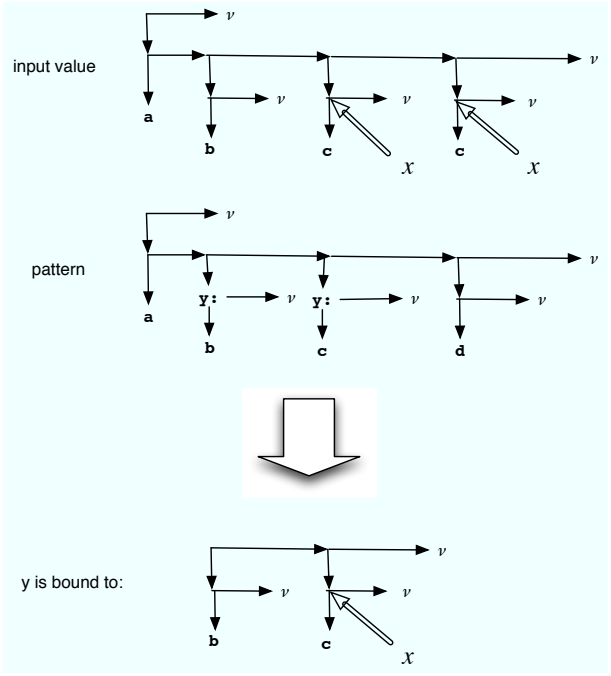

Fig. 3. Extraction of a non-tail sequence

This design choice seems to be acceptable from the programming point of view since the subpart of a value to be parameterized is typically a whole document (as in the examples in the introduction and in Section 2), which is a single element by the definition of XML [Bray et al. 2000]. This design is attractive also from the implementator's point of view since the technicality needed is minimum. Indeed, the change from our formalization in the last sections is very small and, in particular, there is no change in the parts purely related to polytypes (subtyping,

inference for type arguments, substitution, and ambiguity) except for the need to check the restriction. The only change happens in the semantics of patterns and, accordingly, the type inference for patterns so that it handles sequence-capturing variables (which can be done similarly to known techniques [Hosoya 2003; Frisch et al. 2002]).

6. RELATED WORK

There are very few pieces of work on static type systems for XML that support parametric polymorphism. In particular, we are not aware of a thorough study of polymorphism extending the semantic approach. In the work on XHaskell [Sulzmann and Lu 2006a], whose main focus is to implement subtyping as a coercion obtained from a constructive proof of a subtype relation, an attempt has been made to incorporate Hindley-Milner polymorphism [Sulzmann and Lu 2006b]. Like us, they adopt a definition of subtyping where type variables must basically correspond on both sides, though there are minor incompatibilities. For example, unlike us, XHaskell's type variables do not have constraints and cannot be promoted. On the other hand, our types have some restrictions on where type variables can occur as mentioned in Section 3.2, with which we disallow types like `(X as Any)*`, while XHaskell allows an equivalent type `X*`. More importantly, however, they have not justified their design choice with a formalization of the type system nor investigated some of algorithms that we treated such as inference of type arguments and check of marking-ambiguity.

A different approach that also tries to pursue the coexistence or juxtaposition of both XML and ML type systems in a same language is the one proposed by OCamlDuce [Frisch 2006] which ships an extension of the OCaml compiler in which classical ML type schemas may encapsulate XML types, whose values are then treated by primitives derived from the ℂDuce programming language (the OCamlDuce compiler is publicly available and actively maintained). While this eases the writing of polymorphic functions on XML values, this solution does not provide a full fledged polymorphism for XML functions. Indeed, both type systems (ML and XDuce) are kept apart, and a value is either seen as on the ML side—and can then be polymorphic—or on the XDuce side—and can then be precisely typed (with XDuce pattern matching for example). Simply put, OCamlDuce does not allow the programmer to express more than either XDuce or OCaml but only regroups both of them in a coherent framework.

In this work, though, we did not want to embed XDuce into some host polymorphic type system, but to study how to extend its salient features to polymorphic types. The same goal was pursued by Jérôme Vouillon in his recent work [Vouillon 2006a] by following an approach quite different from ours. As explained in Section 5.1, our system already accounts for intersection types and can be easily extended to include difference types. Vouillon instead gives up intersection and negation types and starts from a particular model of functions in order to avoid the circularity between typing and subtyping pointed out in Section 2.5 of [Frisch et al. 2008]. In particular, this is obtained by defining a subtyping relation via a deduction system that is then used to type the expressions of the language. This induces a model of values that, thanks to the absence of intersections (besides negations), is sound and complete with respects to the syntactically defined subtyping relation.

The advantage of giving up intersections and negations is twofold: like ℂDuce and unlike this work, it accounts for higher order function; while unlike ℂDuce and like this works it provides parametric polymorphism. Polymorphism is obtained at the expenses of some trade-offs in expressive power (e.g., in Vouillon's system one cannot inspect a polymorphic value, while this is possible in our framework) and, above all, at the expenses of type inference. As Vouillon explicitly states, type inference seems intractable thus, unlike this work, the type of function arguments must be explicitly given, yielding an explicit form of parametric polymorphism.[7] This of course makes the writing of programs more demanding, and clutters the application of polymorphic functions with type annotations (see [Vouillon 2006b] for a programming language oriented presentation of Vouillon's system). However it must be stressed that all these type annotations are used only at static time: like our system, Vouillon's one ensures that pattern matching can be compiled statically, without requiring any run-time type-check. It also impacts the internal representation of data, since in Vouillon's system the *boxing* of values (marking of a value so as to know its type at run-time) is not necessary thus yielding an efficient data model. As a subjective consideration, we reckon that Vouillon's work suffers from the original sin of starting from a subtyping relation that is given axiomatically by a deduction system. This makes the intuition underlying subtyping quite difficult to grasp (at least, for us). But, objectively, this is well counterbalanced by the fact of having higher-order functions, which makes the use of polymorphism far more interesting.

Polymorphism can trivially be treated by adopting so-called the data-binding approach. This approach, in general, is a handy method to attain, to some extent, static typing by mapping XML types and data values into the structure of an existing programming language. So, if the chosen programming language already supports parametric polymorphism, then we automatically achieve the goal. One such work is HaXML [Wallace and Runciman 1999], which maps DTDs to Haskell's polymorphic type system [Peyton Jones et al. 1993]. Another work close to this is XMLambda [Meijer and Shields 1999; Shields and Meijer 2001], which adds a more flexibility to the usual data-binding approach by using a novel typing discipline called type-index rows and parametric polymorphism adapted to this. Both their and our approaches can express a simple polymorphism as in a type `(A|X)` representing "at least choice `A`" or a type `(A,X)` representing "at least field `A`." However, our approach provides far more flexibilities with the use of our semantic subtyping. For example, consider a type `((A|B),X)` representing "either `A` or `B`, then followed by `X`." Our approach can regard this type as `(A,X)|(B,X)` thanks to our subtyping and therefore can treat the type `(A,C)|(B,C)` as an instance of this type. Their approach, however, cannot do the same because their encoding of regular expressions by disjoint union, tuples, lists, and so on does not allow such flexible type equivalence or subtyping.

To conclude our overview on work about polymorphism for XML transformations, we want to cite the work by Castagna and Nguyen [Castagna and Nguyen 2008]. This work is a drastic departure from what we have seen so far, since the authors

[7] In a personal communication Jérôme Vouillon conjectured that if he restricted its system to handle the same cases as ours, then the inference of type arguments should be possible as well.

claim that when transforming XML documents, having polymorphic functions is less important than having polymorphic operators that iterate on XML documents. To be useful, however, these operators must provide a kind of polymorphism that is out of reach of what currently exists. Therefore they define a language to define custom iterators that are just lightly typed (by some kind of soft typing). The application of these iterators is then precisely typed by performing an abstract execution of the iterator on the type of its input. This allows them to obtain a very precise typing for the application of the defined operators up to a point that it is possible to polymorphically type iterators that, say, capitalize all the tags of a document (that is, such an iterator can be applied to a document of any type schema and the type inference system deduces for its result the schema obtained by capitalizing all the elements of the type of the input). The proposed language can then either be used to extend by XML types and iterators exiting typed languages that are devoid of them, or as a compilation target to efficiently implement and precisely type other transformation languages such as XPath. But all of this is obtained at the price of a somewhat limited iterator language that is neither higher-order nor Turing-complete.

7. CONCLUSIONS

In this paper, we have presented a series of theoretical studies on parametric polymorphism for XML. Our type system smoothly extends the semantic approach that is already standard in monomorphic type systems for XML languages. The crucial part is to introduce *markings* so as to give a sensible interpretation to type variables and, at the same time, obtain practical typechecking algorithms.

The present work is, however, only a first step toward a full-fledged polymorphic type system for XML and various additional investigations are in order. Among others, treating high-order functions is the most important since otherwise polymorphism cannot be made the best use of. There are (at least) two different levels for such a support. In a more restrictive level, function types and the other types are completely stratified where function types can appear only at the top level. (Therefore, function types cannot be composed by unions or by pairs, for example.) A more flexible level allows function types to appear anywhere. The first approach should be easy except that inference of type arguments needs to be extended in order to handle type variables in contravariant positions. The second approach is much more ambitious since we would need to incorporate functions in the denotation of types and at the same time ensure the decidability of subtyping, etc. One way to address this issue might be to combine the treatment by Frisch, Castagna, and Benzaken of higher-order functions in the monomorphic case [Frisch et al. 2008] with polymorphism presented in this paper. A different way would be to fully embrace the semantic approach of [Frisch et al. 2008] by defining a semantics for types and polymorphism that makes it possible to discard the tricky behaviours that motivated this work. As a matter of fact, we noticed that all such tricky behaviours come from finite types as these can be split by variables so that bizarre relations pop up. We conjecture that if we interpret types as infinite sets of some domain and make universally quantified variables to range over all subsets of the domain (thus, on finite ones too), then (1) we avoid the tricky subtyping relations of Section 1 and (2) the subtyping algorithm can be easily derived from

the one in [Frisch et al. 2008] in which type variables are considered as fresh atomic types for which only reflexivity holds. We are currently actively exploring whether this conjecture holds.

Another important feature is XML attributes, which are not handled in the present work since unorderedness among attributes needs a different treatment than a simple tree automata formalism. For this, we hope to be able to combine existing ideas treating attributes for the monomorphic case [Hosoya and Murata 2003; Frisch et al. 2002].

REFERENCES

Aiken, A., Kozen, D., and Wimmers, E. L. 1995. Decidability of systems of set constraints with negative constraints. *Information and Computation 122,* 1, 30–44.

Alon, N., Milo, T., Neven, F., Suciu, D., and Vianu, V. 2001. XML with data values: Type-checking revisited. In *Proceedings of Symposium on Principles of Database Systems (PODS).*

Altheim, M. and McCarron, S. 2001. XHTML 1.1 — Module-based XHTML. `http://www.w3.org/TR/2001/REC-xhtml11-20010531/`.

Appel, A. W. and MacQueen, D. B. 1991. Standard ML of New Jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*. Springer-Verlag, 1–13.

Ausbrooks, R., Buswell, S., Dalmas, S., Devitt, S., Diaz, A., Hunter, R., Smith, B., Soiffer, N., Sutor, R., and Watt, S. 2003. Mathematical Markup Language (MathML) Version 2.0 (Second Edition). `http://www.w3.org/Math/`.

Benzaken, V., Castagna, G., and Frisch, A. 2003. CDuce: An XML-centric general-purpose language. In *Proceedings of the International Conference on Functional Programming (ICFP).* 51–63.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, C. Chambers, Ed. Vancouver, BC, 183–200.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E. 2000. Extensible markup language (XML^{TM}). `http://www.w3.org/XML/`.

Canning, P. S., Cook, W. R., Hill, W. L., Olthoff, W. G., and Mitchell, J. C. 1989. F-bounded polymorphism for object-oriented programming. In *Conference on Functional Programming Languages and Computer Architecture (FPCA).* 273–280.

Cardelli, L., Martini, S., Mitchell, J. C., and Scedrov, A. 1994. An extension of System F with subtyping. *Information and Computation 109,* 1–2, 4–56.

Castagna, G. and Nguyen, K. 2008. Typed iterators for XML. In *ICFP '08: 13th ACM-SIGPLAN International Conference on Funnctional Programming Languages.*

Castagna, G. and Pierce, B. C. 1995. Corrigendum: decidable bounded quantification. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 408.

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. 1999. Tree automata techniques and applications. Draft book; available electronically on `http://www.grappa.univ-lille3.fr/tata`.

Fallside, D. and Lafon, Y. 2004. XML protocol working group. `http://www.w3.org/2000/xp/Group/`.

Fankhauser, P., Fernández, M., Malhotra, A., Rys, M., Siméon, J., and Wadler, P. 2001. XQuery 1.0 Formal Semantics. `http://www.w3.org/TR/query-semantics/`.

Frisch, A. 2004. Théorie, conception et réalisation d'un langage de programmation adapté à XML. Ph.D. thesis, Universit Paris 7.

Frisch, A. 2006. OCaml + XDuce. *SIGPLAN Not. 41,* 9, 192–200.

Frisch, A., Castagna, G., and Benzaken, V. 2002. Semantic subtyping. In *Seventeenth Annual IEEE Symposium on Logic In Computer Science*. 137–146.

Frisch, A., Castagna, G., and Benzaken, V. 2008. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *The Journal of ACM*. To appear.

Gilleron, R., Tison, S., and Tommasi, M. 1999. Set constraints and automata. *Information and Computation 149,* 1, 1–41.

Harren, M., Raghavachari, M., Shmueli, O., Burke, M. G., Bordawekar, R., Pechtchanski, I., and Sarkar, V. 2005. XJ: Facilitating XML processing in Java. In *14th International Conference on World Wide Web (WWW2005)*. ACM Press, 278–287.

Hosoya, H. 2003. Regular expression pattern matching — a simpler design. Tech. Rep. 1397, RIMS, Kyoto University.

Hosoya, H. 2006. Regular expression filters for XML. *Journal of Functional Programming 16,* 6, 711–750. Short version appeared in Proceedings of Programming Technologies for XML (PLAN-X), pp.13–27, 2004.

Hosoya, H., Frisch, A., and Castagna, G. 2005. Parametric polymorphism for XML. In *The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 50–62.

Hosoya, H. and Murata, M. 2003. Boolean operations and inclusion test for attribute-element constraints. In *Eighth International Conference on Implementation and Application of Automata*. Lecture Notes in Computer Science, vol. 2759. Springer-Verlag, 201–212.

Hosoya, H. and Pierce, B. C. 2002. Regular expression pattern matching for XML. *Journal of Functional Programming 13,* 6, 961–1004. Short version appeared in Proceedings of The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 67–80, 2001.

Hosoya, H. and Pierce, B. C. 2003. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology 3,* 2, 117–148. Short version appeared in Proceedings of Third International Workshop on the Web and Databases (WebDB2000), volume 1997 of Lecture Notes in Computer Science, pp. 226–244, Springer-Verlag.

Hosoya, H., Vouillon, J., and Pierce, B. C. 2004. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems 27,* 1, 46–90. Short version appeared in Proceedings of the International Conference on Functional Programming (ICFP), pp.11-22, 2000.

Ishikawa, M. 2002. An XHTML + MathML + SVG Profile. `http://www.w3.org/TR/XHTMLplusMathMLplusSVG/xhtml-math-svg.html`.

Jackson, D., Ferraiolo, J., and Fujisawa, J. 2002. Scalable Vector Graphics (SVG) 1.1 Specification. `http://www.w3.org/TR/2002/CR-SVG11-20020430/`.

Kirkegaard, C. and Møller, A. 2006. Xact - xml transformations in java. In *Programming Language Technologies for XML (PLAN-X)*. 87.

Leroy, X., Doligez, D., Garrigue, J., Vouillon, J., and Rémy, D. 1996. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`.

Maneth, S., Perst, T., Berlea, A., and Seidl, H. 2005. XML type checking with macro tree transducers. In *Proceedings of Symposium on Principles of Database Systems (PODS)*. 283–294.

Maneth, S., Perst, T., and Seidl, H. 2007. Exact XML type checking in polynomial time. In *International Conference on Database Theory (ICDT)*. 254–268.

Meijer, E. and Shields, M. 1999. XMλ: A functional programming language for constructing and manipulating XML documents. Manuscript.

Milner, R., Tofte, M., and Harper, R. 1990. *The Definition of Standard ML*. The MIT Press.

Milo, T., Suciu, D., and Vianu, V. 2000. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 11–22.

Murata, M. 2001. Extended path expressions for XML. In *Proceedings of Symposium on Principles of Database Systems (PODS)*. 126–137.

Nottingham, M. and Sayre, R. 2005. The Atom Syndication Format (RFC 4287). `ftp://ftp.rfc-editor.org/in-notes/rfc4287.txt`.

OASIS. 2002. DocBook. http://www.docbook.org.

OASYS. 2007. Open Document Format for Office Applications (OpenDocument). http://www.oasis-open.org/committees/office/.

Peyton Jones, S. L., Hall, C. V., Hammond, K., Partain, W., and Wadler, P. 1993. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*.

Python XML Special Interest Group. 1998. The XML bookmark exchange language. http://pyxml.sourceforge.net/topics/xbel/.

Reynolds, J. C. 1983. Types, abstraction, and parametric polymorphism. *Information Processing 83*, 513–523.

Shields, M. and Meijer, E. 2001. Type-indexed rows. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. London, 261–275.

Stefénsson, K. 1994. Systems of set constraints with negative constraints are nexptime-complete. In *Proceedings of Ninth Annual IEEE Symposium on Logic in Computer Science*. 137–141.

Stroustrup, B. 2000. *The C++ Programming Language*. Addison-Wesley.

Suda, T. and Hosoya, H. 2005. Non-backtracking top-down algorithm for checking tree automata containment. In *Proceedings of Conference on Implementation and Applications of Automata (CIAA)*. 83–92.

Sulzmann, M. and Lu, K. Z. M. 2006a. A type-safe embedding of XDuce into ML. *Electric Notes in Theoretical Computer Science 148,* 2, 239–264.

Sulzmann, M. and Lu, K. Z. M. 2006b. Xhaskell. In *PLAN-X*. 92. Demonstration.

Tozawa, A. and Hagiya, M. 2003. XML schema containment checking based on semi-implicit techniques. In *8th International Conference on Implementation and Application of Automata*. Lecture Notes in Computer Science, vol. 2759. Springer-Verlag, 213–225.

Vansummeren, S. 2003. Unique pattern matching in strings.

Vouillon, J. 2006a. Polymorphic regular tree types and patterns. In *POPL*. 103–114.

Vouillon, J. 2006b. Polymorphism and xduce-style patterns. In *PLAN-X '06, 4th Workshop on Programming Language Technologies for XML*. 49–60.

Wallace, M. and Runciman, C. 1999. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*. ACM Sigplan Notices, vol. 34-9. ACM Press, N.Y., 148–159.

Appendix

A. AUTOMATA ENCODING

Let a type name s_0 defined under a type definition Δ. Our goal is to encode s_0 under Δ by a marking automaton.

Let $\{a_1, \ldots, a_n\}$ and X_0 be the set of labels and the set of variables, respectively, appearing in Δ. First, we transform the type definition Δ as follows.

(1) Add a new definition $s_\top \mapsto a_1 | \ldots | a_n | (s_\top, s_\top)$ to Δ, and replace all occurrences of $\top$ by $s_\top$;

(2) For each occurrences of (p_1, p_2), add new definitions $s_i \mapsto p_i$ to Δ and replace (p_1, p_2) with (s_1, s_2).

Then, we compute a mapping Λ from each type name to a set of atoms belonging to the type name, where an atom, written f, is a type of the form either $\vec{x} : a$ or $\vec{x} : (s, s)$ where $\vec{x} : p$ stands for $x_1 : \ldots : x_k : p$ for some $k \geq 0$. For each type name

in Δ, we set $\Lambda(s) = \mathrm{conv}(\Delta(s))$ where the function conv is defined as follows.

$$\begin{aligned}
\mathrm{conv}(a) &= \{a\} \\
\mathrm{conv}((s_1, s_2)) &= \{(s_1, s_2)\} \\
\mathrm{conv}(s) &= \mathrm{conv}(\Delta(s)) \\
\mathrm{conv}(p_1|p_2) &= \mathrm{conv}(p_1) \cup \mathrm{conv}(p_2) \\
\mathrm{conv}(x : p) &= \{x : f \mid f \in \mathrm{conv}(p)\} \\
\mathrm{conv}(\bot) &= \emptyset
\end{aligned}$$

This function never goes into an infinite loop since a recursive use of type name must go through a pair (Section 3.2). The following is obvious:

LEMMA 12. *Let* $\{f_1, \ldots, f_m\} = \Lambda(s)$. *Then,* $(v, V) \lhd \Delta(s)$ *iff* $(v, V) \lhd f_1|\ldots|f_m$.

From now, we will confuse between a vector $\vec{x}$ of variables and the set consisting of the variables in the vector since ordering among variables is semantically insignificant.

Then, we construct a marking automaton (S, I, δ, Ξ) such that

—$S = \{(s, \vec{x}) \mid \vec{x} : p \in \Lambda(s),\ s \in \mathbf{dom}(\Lambda)\}$,

—$I = \{(s_0, \vec{x}) \mid (s_0, \vec{x}) \in S\}$,

—$\delta = \{(s, \vec{x}) \to a \mid \vec{x} : a \in \Lambda(s)\} \cup \{(s, \vec{x}) \to ((s_1, \vec{x_1}), (s_2, \vec{x_2})) \mid \vec{x} : (s_1, s_2) \in \Lambda(s),\ (s_1, \vec{x_1}), (s_2, \vec{x_2}) \in S\}$, and

—$\Xi((s, \vec{x})) = \vec{x}$.

Intuitively, each state $(s, \vec{x})$ behaves exactly the same as the type name s except that $(s, \vec{x})$ matches a value only when it immediately marks the value with $\vec{x}$. Therefore the states $(s, \vec{x})$ with all variable vectors $\vec{x}$ together behave exactly the same as the type name s. The following is thus obvious.

LEMMA 13. $(v, V) \lhd (s, \vec{x})$ *for some* $\vec{x}$ *iff* $(v, V) \lhd s$.

For example, when the following type definition is given after preprocessing

$$\begin{aligned}
s_0 &= x : ((s_1, s_\top) \mid y : a) \mid y : (s_2, s_\top) \\
s_1 &= y : b \\
s_2 &= x : c
\end{aligned}$$

the produced marking automaton has the following transitions (omitting those for $s_\top$).

$$\begin{aligned}
(s_0, \{x\}) &\mapsto ((s_1, \{y\}), (s_\top, \emptyset)) \\
(s_0, \{x, y\}) &\mapsto a \\
(s_0, \{y\}) &\mapsto ((s_2, \{x\}), (s_\top, \emptyset)) \\
(s_1, \{y\}) &\mapsto b \\
(s_2, \{x\}) &\mapsto c
\end{aligned}$$

B. TRICKY SUBTYPING EXAMPLE

This section gives a bit more details on the tricky subtyping example shown in the introduction. The example is not allowed by our marking-based subtyping but is allowed by the placeholder-based subtyping, i.e., defined by "the subset relation holds for all substitutions."

The example was: for any type X,

$$(a, X) \subseteq (a, \overline{a}) \cup (X, a).$$

Here, $\overline{a}$ is a type representing the complement of a, which can easily be defined by using recursion. (We assume here that there are only a finite number of labels. To allow an infinite number of labels, we need to extend the type language.) Indeed, if X does not contain a, then the left hand side is included by the first clause on the right. If X does contain a, then all the values on the left except (a, a) is included by the first clause on the right and the value (a, a) is included by the second clause.

One may wonder where classical subtyping rules (in particular, "a variable X is a subtype of only X itself and the top type $\top$") do not work. Let us have a closer look at the above relation. It can be transformed to

$$\begin{aligned} & (a \subseteq a \cup X \ \vee \ X \subseteq \emptyset) && (46)\\ \wedge \ & (a \subseteq a \ \vee \ X \subseteq a) && (47)\\ \wedge \ & (a \subseteq X \ \vee \ X \subseteq \overline{a}) && (48)\\ \wedge \ & (a \subseteq \emptyset \ \vee \ X \subseteq \overline{a} \cup a) && (49) \end{aligned}$$

according to the subtyping algorithm in [Hosoya et al. 2004]. The first and the second clauses hold because $a \subseteq a \cup X$ and $a \subseteq a$ obviously hold, respectively. The fourth clause holds since $X \subseteq \overline{a} \cup a$ holds. This is also not surprising since we could promote X to $\top$ and continue to check $\top \subseteq \overline{a} \cup a$ (which succeeds). However, the third clause is tricky. This holds because of the set-theoretic property that, for any X, we have either $a \in X$ or $a \notin X$. However, no classical subtyping rule allows us to verify this clause.

Our definition of subtyping does not permit the above example since no occurrence of X on the right corresponds to the occurrence of X on the left.

We can generalize the above example for any number of singletons as follows (we use $(n+1)$-ary tuples for simplicity but they can of course be encoded by pairs).

$$\begin{aligned} (a_1, \ldots, a_n, X) \subseteq \quad & (a_1, \ldots, a_n, \overline{a_1 | \ldots | a_n}) \\ \cup \ & \textstyle\bigcup_{i=1}^{n} (a_1, \ldots, a_{i_1}, X, a_{i+1}, \ldots, a_n, a_i) \end{aligned}$$

Acknowledgments

We would like to express our best gratitude to Jérôme Vouillon, Benjamin Pierce, Vladimir Gapeyev, and Naoki Kobayashi for precious comments and useful discussions. We thank anonymous referees of POPL'05 whose comments and suggestions have greatly improved the presentation of this paper. This work was partly supported by The Inamori Foundation, Japan Society for the Promotion of Science, and the European FET contract "MyThS," IST-2001-32617.