Parametric Polymorphism for XML

Haruo Hosoya
The University of Tokyo
hahosoya@is.s.u-tokyo.ac.jp

Alain Frisch
INRIA
Alain.Frisch@inria.fr

Giuseppe Castagna
École Normale Supérieure de Paris
Giuseppe.Castagna@ens.fr

ABSTRACT

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. In this paper, we propose a different method that smoothly extends the semantic approach yet is algorithmically easier. In this, we 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, and so on. As a result, we achieve a sensible system that directly represents a usual expected behavior of polymorphic type systems—"values of variable 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

Keywords: 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 [18; 11; 3; 21, etc] and of theoretical frameworks [23; 22, etc]. 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 [20, 2], Haskell [24], C++ [9], and GJ [4], and already established as an important ingredient for code reusability. Not surprisingly, this usefulness can be extended to XML processing, not only because one may use, in XML, operations such as `map` and `fold` that are standard in functional languages, but also because some 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 "parametric schemas" and typecheck them. A typical example is SOAP [10], which provides a generic type for "envelopes" for enclosing "content" data whose type is parameterized.[1] Concretely, suppose that we want to write a generic function for wrapping a given data d in a SOAP envelope as follows:

```
<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. Therefore an appropriate type for this function must be polymorphic:

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

Here, $\mathrm{SoapEnv}(X)$ is the parameterized type for SOAP envelopes and may be defined by

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

in the notation of regular expression types [19]. (The form tag[...] corresponds to the XML structure `<tag>...</tag>` and the comma is a concatenation operation. See Section 2.1 for more details on types.)

The need of polymorphism for XML is quite clear, and also certified by the fact that this feature has repeatedly been requested to and discussed in various working groups of standards (e.g., RELAX NG [7] and XQuery [11]). Despite that, it has almost never been studied formally. This

[1]The specification of SOAP itself does not define types at all (probably because parametric types are not supported by any schema language), but the above requirement is described informally in a natural language.

POPL'05, January 12–14, 2005, Long Beach, California, USA.

is probably because the "semantic" approach—the one used by most of current research on type systems for XML—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:

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

Although the subtype relation, which is equivalent to the tree automata containment problem [19], has a high worst-case complexity (EXPTIME-complete), several algorithms are known to work in practice [19, 28]. 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 for the type variables:

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

However, the subtype relation defined in this way is substantially more difficult than the monomorphic case. This problem can be reduced to the satisfiability problem for set constraint systems with *negative constraints* [1, 13]; this problem is known to be NEXPTIME-complete [27] and, so far, no practical algorithm is known. It still is an open question whether the above subtype problem can be reduced to an easier problem. However, we have noticed some tricky behavior that makes us believe that the problem is not easy to solve. For example, with the definition given above, the following relation holds:

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

Here, $a[\,]$ denotes a type representing the singleton set consisting of the value `<a></a>`. We assume the type A to represent the complement of the type $a[\,]$ (which is possible to define since types are essentially regular). We can prove that this relation holds by a set-theoretical analysis of cases on the type S to be substituted for X. (We defer a concrete proof to Appendix A.) Note that the example above 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 example can further be generalized to a "finite" type rather than a "singleton" type. See Appendix A.)

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, and (3) yields practical typechecking algorithms. The key idea in our approach is to interpret type variables not as "place holders" for performing substitutions, but as *markings* in documents for indicating their parameterized 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. 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 problem simply to a slight variation of the tree automata containment problem (Section 4.2) and to obtain a reasonably efficient algorithm by incorporating various known algorithmic techniques [19, 28].

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 parameterized 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 parameterized subpart in a result value of a function must come from some parameterized 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 a variable type is never reconstructed"—in a reminiscence of Reynold's parametricity theory [25]. Note that we would not have such a property if we adopted the "placeholder" subtyping since, by using the example relation given above, a value that that has the concrete type $a[\,]$ could be given the variable type X and viceversa.

Another by-product from our interpretation of polymorphic types is that their semantics is mathematically identical to *pattern matches* since 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 provides us 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 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 [14], as shown in Section 4.3 and 4.4.

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. Section 5 discusses related work and Section 6 concludes this paper and hints at future work. Appendix A discusses in more detail the above-mentioned tricky example allowed in the "place-holder" subtyping. For space limitation, all proofs of the presented theorems are elided from this paper. They can be found in the full version of this paper [15].

2. BASIC IDEAS

In this section, we explain our polymorphic type system by example, starting with a monomorphic system and then describing our ideas for adding polymorphism.

2.1 Monomorphic system

We start with considering a minimal, functional language designed for XML processing, in the notation of XDuce [18]. The language basically 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 key1)(BibDB as d) : BibResult =
  match d with
    db[BibEntry* as l]  ->  iter(key1)(l)

fun iter (String as key1)(BibEntry* as l) : BibResult =
  match l with
    ()  ->  notfound[]
  | entry[key[String as key2], content[Any as c]],
                                          Any as rest
       ->  if key1 = key2 then found[c]
           else iter(key1)(rest)
```

We first define the type `BibDB` to be values of label `db` that contains a repetition of data of type `BibEntry`. The `BibEntry` type is in turn defined as 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. We then define the type `BibResult` to be 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 `key1` 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[BibEntry*]` and binds the variable `l` to the subpart of the input value corresponding to `BibEntry*`, 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 `key1` and `l`. The function `iter` takes a string `key1` 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. We extract the content `key2` 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 `key1` and the extracted `key2` are equal, then we return `c` enclosed by a `found` label; 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. Thus, types can also be concatenations (`T1,T2`), unions (`T1|T2`), repetitions (`T*`), and the empty sequence type (`()`). We also allow `Any` to denote any value. 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. See [19, 14] for more details.)

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`.) A pattern match form first tries matching the input value `e` against the patterns `P1` through `Pn` in this order, and then evaluates the body expression corresponding to the first matching pattern 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, we take a nondeterministic semantics of patterns, where we choose an arbitrary match when a pattern can have more than one possible match for a given input value. (More discussions on pattern matching can be found in [14, 17, 12].)

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. For some important places, we check subtyping. Those places include 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` 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. Moreover, the feasibility is guaranteed by the exact correspondence to finite tree automata, whose containment problem is known to be decidable. (More details are given in [19].)

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 `key2`, `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 [17, 14, 12]. 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

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 above example 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 key1)(DB{X} as d) : Result{X} =
  match d with
    db[Entry{X}* as l]  ->  iter{X}(key1)(l)

fun iter {X}(String as key1)(Entry{X}* as l) : Result{X} =
  match l with
    ()  ->  notfound[]
  | entry[key[String as key2], content[Any as c]],
                                          Any as rest
        ->  if key1 = key2 then found[c]
            else iter{X}(key1)(rest)
```

Changes from the previous program are that what used to be `Content` is now replaced by the type variable `X`, that all the definitions of type names and functions now take the type parameters, written `{X}`, and that 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, but they actually need not be written explicitly. As a result of parameterizing 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.

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,

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

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 like

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

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, this would tempt us to construct a type system that requires overly ambitious algorithmics, e.g., subtyping 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`).

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

These conditions will remind many readers of the well-known parametricty property of the polymorphic lambda calculus [25]. 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 of the following form.

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

where each v_i is marked `X`. Similarly, the result type `Result{X}` denotes a set of marked values that have either the form `found[`v`]` where v is marked `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, it results in a marked value that conforms to the declared result type (if it terminates). Since each operation never modifies a subpart marked with `X` if its result preserves this marking, 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 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` iff any value of type `S` is also of type `T`. However, this definition turns out to be too strong. For example, consider the following function

```
fun f {X} (a[X] as x) : a[Any] = x
```

which simply returns the input value without performing any operation. However, the content of the label `a` has type `Any` for the output, whereas it has type `X` for the input. With the above simple definition of subtyping, this function does not typecheck since the subtype relation $\texttt{a[X]} \leqslant \texttt{a[Any]}$ does not hold (the right hand side requires that no marking is present). Nevertheless, this function is reasonable to accept since it fulfills our safety conditions. That is, each subpart of the result corresponding to `X` must be identical to some subpart of the input corresponding to `X`. But such a subpart corresponding to `X` does not exist in the result; hence, the safety property vacuously holds.

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` iff, 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 least non-variable upper bounds, which often appears in type checking algorithms of systems that combine subtyping and polymorphism (cf. $F_{\leqslant}$ [6]).

The typing rule for polymorphic function applications is as usual. That is, we check that the argument type is a subtype of the parameter type with the type parameters replaced by the type arguments. Then, the whole function application is given the declared result type with the type parameters 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, where, in particular, the second argument's type `DB{Contact}` is a subtype of $[\texttt{X} \mapsto \texttt{Contact}]\ \texttt{DB\{X\}} = \texttt{DB\{Contact\}}$ (since they are identical). 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 with our semantic view to polymorphism. First, according to our safety property, the definition of `search` function declares that, for any input value (as the second argument) of the form

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

the result (if any) has the form `found[`v_i`]` for some $1 \leq i \leq n$, or the form `notfound[]`. Therefore, since the value `contactDB` has the above form of `db` where each v_i has type `Contact`, the result of the function `search` returns for this input has type either `found[Contact]` or `notfound[]`, i.e., type `Result{Contact}`.

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. For the same example, we 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 this if we supported higher-order functions. See [15] for a related discussion.)

A question that may arise here is: does such a minimum type always exist? The answer is *no*, with only the specification above. For example, there does not exists 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 to this issue is to reject such an ambiguous parameter type. Then, under this restriction, we can prove that a minimum type argument specified before always exists (Section 3.4).

In our type system, we additionally support type variables with type constraints of the form `T as X`. This form of type denotes values of type `T` with a marking of `X` at the top. Thus, a bare type variable `X` now can be rewritten by `Any as X`. Such type constraints have an effect somewhat similar to *bounded quantification*, which often appears in type systems with both subtyping and polymorphism. For example, a function of type usually written $\forall \texttt{X} \leqslant \texttt{T}.\ \texttt{a[X]} \to \texttt{b[X]}$, where the type parameter is constrained to be a subtype of `T`, can be simulated by:

```
fun f {X} (a[T as X] as x) : b[T as X] = ...
```

Of course, both are not exactly the same. For example, different occurrences of type variables may have different "bounds." Also, type arguments passed to the function are not *required* to be a subtype of `T`, but substitution of any type for `X` is ensured to be a subtype of `T`. In fact, what is closer to our `as` notation is intersection types. See [15] for more details.

3. FORMAL SYSTEM

The subsequent two sections formulate our polymorphic type system outlined in the previous section. In this section, we focus on the semantic aspect of the formal system, and, 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 [19, 12]. 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, that contains at least a special label ν. (Internal) values are then defined by:

$$v \quad ::= \quad a \quad | \quad (v, v)$$

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. Variables are divided into two sets, *term variables* and *type variables*. *Paths* are defined by:

$$\pi \quad ::= \quad 1\pi \quad | \quad 2\pi \quad | \quad \epsilon$$

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

When we compose two marked values in a pair, the original marks 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, the original marks 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. We encode those types by internal types also based on 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*, where types are defined by the following.

$$p \quad ::= \quad a \quad | \quad (p, p) \quad | \quad p \,|\, p \quad | \quad x : p \quad | \quad 1 \quad | \quad 0 \quad | \quad s$$

That is, a type can be 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 be the set of variables appearing in p and all types associated with the type names reachable from p. We make two restrictions on types. First, any recursive use of type name must go through a pair. Second, for any occurrence of $x : p$, we require that $x \notin \mathbf{var}(p)$. This 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 [5], but we do not further pursue this direction in this paper.)

Under a fixed 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.

$$\frac{}{(a, \emptyset) \lhd a}\ \textsc{MCon} \qquad \frac{(v_i, V_i) \lhd p_i \quad \text{for } i = 1, 2}{(v_1, V_1) \otimes (v_2, V_2) \lhd (p_1, p_2)}\ \textsc{MPair}$$

$$\frac{(v, V) \lhd p_i \quad i = 1 \text{ or } 2}{(v, V) \lhd p_1 | p_2}\ \textsc{MAlt} \qquad \frac{(v, V) \lhd p}{(v, V \cup \{x : \epsilon\}) \lhd x : p}\ \textsc{MVar}$$

$$\frac{}{(v, \emptyset) \lhd 1}\ \textsc{MAll} \qquad \frac{(v, V) \lhd \Delta(s)}{(v, V) \lhd s}\ \textsc{MName}$$

Translation from external types to internal types is anal-

ogous 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 labeled type, 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 1. 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 1, and so on. For example, the external type

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

can be translated to the internal type

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

with definitions

$$\begin{aligned} s_1 &\mapsto \nu \mid ((\mathtt{entry}, s_2), s_1) \\ s_2 &\mapsto ((\mathtt{key}, s_3), ((\mathtt{content}, \mathtt{X} : 1), \nu)) \\ s_3 &\mapsto (1, \nu) \end{aligned}$$

(Note that the bare variable `X` is encoded by `X` : 1 since `X` is an abbreviation for `Any as X`.) We refer the reader to the literature (e.g., [19]) for a formalization of a general translation scheme.

We call types containing only term variables *pattern* and those containing only type variables *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 a substantial complexity to the type and run-time system since behaviors of pattern matches could now 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 has a lot to study.

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 that p is *linear*: $(v, V) \lhd p$ implies that $\mathbf{dom}(V) = \mathbf{var}(p)$ and V is a function (i.e., $\pi = \pi'$ for any $x : \pi, x : \pi' \in V$). See the full version of this paper [15] for an algorithm for checking linearity.

Note that our internal representation allows a mark to be put only on a single node. A consequence of this is that the external representation has the restriction that values can have a mark only on a *tail-sequence*, i.e., a sequence that ends at the tail of the whole sequence. For example, in the value `a[b[],c[],d[]]`, we can put a mark on the sequence `c[],d[]` but cannot on `b[],c[]`. We consider that this restriction is rather undesirable and therefore better to be eliminated by a cleverer encoding of variables. In the case of patterns, we can use known encoding techniques relying on the linearity condition [19, 14], but in the case of polytypes, we would need a different technique. We leave this issue for future work. (The "tail-variable" restriction on polytypes might actually be acceptable in many cases since the subpart of a value to be parameterized is typically on the whole content of a label, as in the examples in the introduction and in Section 2.)

The subtype relation, written $p \leqslant q$, is defined as, for all v and V, if $(v, V) \lhd p$, then $(v, W) \lhd q$ for some $W \subseteq V$. We present an algorithm for checking subtyping in Section 4.2.

3.3 Substitution

Later, we will formalize our inference scheme in a way that obtains a substitution of types for variables. An *X-substitution* σ is a mapping from a set X of variables to types.

In the definition of the application σp of a substitution σ to a type p, we do not use a usual syntactic way, but instead adopt a "semantic" way. This is because we have type constraints on variables for which we need to perform an intersection operation rather than a simple replacement. For example, when we have a type $p = (x : 1, 1)$ and a substitution $\sigma = \{x \mapsto (y : 1, a)\}$, we can easily obtain $\sigma p = ((y : b, a), 1)$ by replacing $x : 1$ with $(y : 1, a)$. However, in the case that we have a constraint on x as in $p = (x : (b, 1), 1)$, this interacts with $(y : 1, a)$; in this example, we need to take an intersection between $(b, 1)$ and the type $(y : 1, a)$, which results in $\sigma p = ((y : b, a), 1)$. This situation is even more complicated when the type constraint contains other variables and we need to do a simultaneous substitution for multiple variables. (Another complication would also arise when the type p is recursively defined.)

Thus, we define an application σp of a substitution σ by what the type σp should satisfy semantically. Let us first see an example: $p = (x : (b, 1), 1)$ and $\sigma = \{x \mapsto (y : 1, a)\}$. The type σp satisfies the following. For each marked value in p, which has the form $((b, v), w)$ where v and w are any values and (b, v) is marked x, if the x-marked subpart (b, v) matches $\sigma(x) = (y : 1, a)$ with a marking U, then the type σp contains a marked value $((b, v), w)$ where the mark x is replaced by the new marking U; the type σp contains only such marked values. Therefore each marked value in σp has the form $((b, a), w)$ where w is any value and b is marked y. Syntactically, σp can be written $((y : b, a), 1)$ as in the last paragraph.

Below, we generalize the above notion of substitution for the case where X contains an arbitrary number of variables and p may contain extra variables not in X. An X-substitution σ applied to a type p, written σp, is a type satisfying the following.

$$(v, V) \lhd \sigma p$$

iff

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

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 contains the marked value (v, V) where V contains the marking after replacing each x_i-mark by the new marking U_i pushed down by π_i; the type σp contains only such marked values. Though the definition does not tell, such σp always exists and can be computed algorithmically. A concrete algorithm can be found in the full version of the paper [15]. 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 [15]).

We extend the subtype relation between types to that between 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 polymorphic function and each pattern match, we perform a form of type inference. Although the inference may appear different for these two cases, it 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, where q may contain type variables in X. 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, where q may contain term variables in X. Below, we 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 that of the inference for type arguments.

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 the following for each $x \in X$.

$$(u, U) \lhd (p \lhd_X q)(x)$$
$$\text{iff}$$
$$\exists v, V, W, \pi.\quad (v, V) \lhd p \quad (v, W \cup \{x : \pi\}) \lhd q \quad (u, U) = \pi^{-1}(v, V)$$

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 contains the subnode with all the original marks V pulled up by π (note that we lose marks on the nodes that are not descendents of $\pi(v)$); the inferred type $(p \lhd_X q)(x)$ contains only such marked values. Section 4.3 proves that such $p \lhd_X q$ always exists and can be found deterministically.

For example, let the domain $p = ((y : 1, a), 1)$ and the target $q = (x : (b, 1), 1)$, 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 y. Values of the form $((v, a), w)$ match q when $v = b$ and yield a marking of x on the subnode (b, a). Therefore the inferred type for x contains (b, a) where b is marked y (since it is marked so in the original marked value). Syntactically, the type can be written $(y : b, a)$.

We use the above definition directly as the specification of the inference for patterns. For the inference of type arguments, however, we want to use a different specification as outlined in Section 2.2: a *minimum* X-substitution σ such that $p \leqslant \sigma q$. Let us call an X-substitution σ satisfying $p \leqslant \sigma q$ *solution*. Our intention is to use the same inference algorithm both for patterns and for type arguments. However, the first inference specification given above does not necessarily yield a solution. For example, in the previous paragraph, from the domain $p = ((y : 1, a), 1)$ and the target $q = (x : (b, 1), 1)$, the inference has 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 $q = (x : 1, b)|(a, x : 1)$ (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 do not know which case of $(x : 1, b)$ and $(a, x : 1)$ would be taken at run time and therefore the type for x should 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 typable as possible. Further, the *minimum* one 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 would complicate it).

Fortunately, by imposing two 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 or smaller than any solution: we require that q yields a unique marking for any value, formally, for all value v, we have that $(v, V) \lhd q$ and $(v, V') \lhd q$ imply $V|_X = V'|_X$. When this holds, we say that q is X-*unambiguous*. We will show an algorithm for ambiguity check in Section 4.4.

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 $(x : 1)|1$ is ambiguous but, for any type argument, the solution $\{x \mapsto 0\}$ is always minimum. If one considers that our requirement is too restrictive, an alternative design might be to require explicit type arguments when the parameter types are ambiguous.

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 $p \leqslant \sigma q$.*

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.

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

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 a function call may be to a polymorphic one, but we do not provide a form to explicitly supply type arguments—they are always inferred. In any match expression **match** e **with** $p_1 \to e_1 \mid \ldots \mid p_n \to e_n$, 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. For a function **fun** $f\{X\}(x : p_1) : p_2 = e$, we require that $\mathbf{var}(p_1) \cup \mathbf{var}(p_2) \subseteq X$. Also, we assume a fixed set Φ of functions from now on, and no two different functions of the same name are declared in Φ. 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 are well-typed") for $\vdash \phi$ for all $\phi \in \Phi$. The typing relations are defined by the following set of rules.

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

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

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

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

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.

$$\textsc{SVar}\ \frac{}{E \vdash x \Downarrow E(x)} \qquad \textsc{SCon}\ \frac{}{E \vdash a \Downarrow (a, \emptyset)}$$

$$\textsc{SPair}\ \frac{E \vdash e_i \Downarrow (v_i, V_i) \quad \text{for } i = 1, 2}{E \vdash (e_1, e_2) \Downarrow (v_1, V_1) \otimes (v_2, V_2)}$$

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

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

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 design choice is just for simplifying the formalization. Changing the semantics to the first match should be straightforward, except that the type system would require taking "difference" between types. See [15] for further discussions.

Finally, note that no construct "creates" marks. Therefore, when we start from an entry-point expression and proceed evaluation, we will never see any marking at all during execution. This means that we actually do not need marked values and all run-time mechanisms related to marking in an actual implementation—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 parameterized subpart in the result value comes from some parameterized subpart in the input value"—is best explained in terms of our evaluation semantics.

3.7 Correctness

The correctness of our type system consists 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").

Type preservation can be proved by standard induction on the derivation of the evaluation relation. We write $\Gamma \vdash E$ if

$\mathbf{dom}(\Gamma) = \mathbf{dom}(E)$, and $E(x) = (v, V)$ implies $(v, V') \lhd \Gamma(x)$ 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 1 (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* $W' \subseteq W$.

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 $E \vdash e \Downarrow \bot$ to explicitly state that e encounters a run-time error. We omit the rules for the failure relation from this paper (the definition is straightforward: $E \vdash e \Downarrow \bot$ when one of the necessary premises fails), but these can be found in the full version [15].

THEOREM 2 (PROGRESS). *Let* $\vdash \Phi$. *If* $\Gamma \vdash E$ *and* $\Gamma \vdash e \in p$, *then* $E \vdash e \Downarrow \bot$ *never holds.*

4. ALGORITHMS

This section describes algorithms necessary for implementing the type system defined in the previous section. The algorithms presented here are subtype checking, inference, and ambiguity checking. For space limitation, algorithms for checking linearity of patterns and for substitution are omitted here, but can be found in [15].

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*,
- 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 that $\Xi(s) \cap (\mathbf{var}_A(s_1) \cup \mathbf{var}_A(s_2)) = \emptyset$ 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.

$$\text{ACON}\ \frac{s \to a \in \delta}{A \vdash (a, \Xi(s) \times \{\epsilon\}) \lhd s}$$

$$\text{APAIR}\ \frac{s \to (s_1, s_2) \in \delta \quad A \vdash (v_i, V_i) \lhd s_i \quad \text{for } i = 1, 2}{A \vdash ((v_1, v_2), 1V_1 \cup 2V_2 \cup \Xi(s) \times \{\epsilon\}) \lhd s}$$

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 $x : y : (1, 1)$ 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 [15]. (Note that a marking automaton can trivially be encoded by a type with a type definition.) We transfer all the type-related definitions (linearity, substitution, subtyping, inference, and ambiguity) to marking automata. (Thus, we write in a way like $A \leqslant B$.)

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., [8].

4.2 Subtyping

The goal here 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.

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

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

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 $t \to a \notin \delta_B$.

2. 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 [19]. Their algorithm computes essentially what our algorithm above does minus the treatment of variables, 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.[2]

PROPOSITION 2. *The subtyping algorithm returns "yes" iff $A \leqslant B$.*

4.3 Inference

Consider marking automata A and B and a set $X = \{x_1, \ldots, x_n\}$ of variables. Our purpose here is to 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 A accepts (v, V) and B accepts (v, W) with a mark $x_i : \pi \in W$ 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.

[2]For readers familiar with the algorithm in [19], the additional rule would look like:

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

1. 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 $s \to a \in \delta_B$.

2. Empty-eliminate C

3. 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 [17, 14]. 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).

PROPOSITION 3. *$\{x_1 \mapsto D_{x_1}, \ldots, x_n \mapsto D_{x_n}\}$ is an X-inference of B with respect to A.*

4.4 Ambiguity

The ambiguity check aims 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). Let $A = (S_A, I_A, \delta_A, \Xi_A)$. The ambiguity check algorithm is as follows.

1. 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 $s \to a \in \delta_A$.

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

3. 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.

PROPOSITION 4. *The ambiguity-checking algorithm returns "unambiguous" iff A is X-unambiguous.*

5. RELATED WORK

There have been very few work on static type systems for XML that support parametric polymorphism. In particular, we are not aware of any attempt to extend the semantic approach. However, 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 [29], which maps DTDs to Haskell's polymorphic type system [24]. Another work close to this is XMLambda [21, 26], 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.

6. 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 only a first step toward a full-fledged polymorphic type system for XML and various additional investigations are in order. In particular, we have not treated type variables in non-tail positions for a technical reason. We consider that the restriction can be eliminated by improving the current naive encoding of types in automata. Another important extension is for XML attributes. For this, we plan to investigate whether we can combine existing ideas treating attributes for the monomorphic case [16, 12].

7. REFERENCES

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

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

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

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

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

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

[7] J. Clark and M. Murata. RELAX NG. `http://www.relaxng.org`, 2001.

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

[9] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

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

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

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

[13] R. Gilleron, S. Tison, and M. Tommasi. Set constraints and automata. *Information and Computation*, 149(1):1–41, 1999.

[14] H. Hosoya. Regular expression pattern matching — a simpler design. Technical Report 1397, RIMS, Kyoto University, 2003.

[15] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. `http://arbre.is.s.u-tokyo.ac.jp/~hahosoya/papers/polyx.ps`, 2004. Full version.

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

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

[18] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003. 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.

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

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

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

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

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

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

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

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

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

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

[29] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34-9 of *ACM Sigplan Notices*, pages 148–159, N.Y., Sept. 27–29 1999. ACM Press.

APPENDIX

A. 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.

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).

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

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.