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
Giuseppe.Castagna@ens.fr

July 16, 2004

Abstract

Although several type systems have been investigated for XML, parametric polymorphism is rarely 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, moreover, the type system obtained in this way is characterized by a hardly useful expressiveness. In this paper, we propose a lighter-weight way of extending the semantic approach, where we interpret type variables as *markings* in data values for indicating the parameterized subparts. As a result, we can construct a sensible polymorphic type system both with a semantic flavor and a set of practical algorithms needed for typechecking. Most of these algorithms can be obtained by local modifications to existing ones for a monomorphic system.

1 Introduction

Recently, static typing for XML processing has actively been investigated in the contexts both of concrete language designs [17, 10, 3, 20, etc] and of theoretical frameworks [22, 21, 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 [19, 2], Haskell [23], C++ [8], 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 [9], which provides a generic type for "envelopes" for enclosing "content" data whose type is parameterized.[1] 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>
```

[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.

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:

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

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 [18]. (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 its occasional evocation in discussion forums of various working groups of standards (e.g., RELAX NG [6] and XQuery [10]). Despite that, it has almost never been studied formally. This 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. 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:[2]

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

Although the subtype relation, which is equivalent to the tree automata containment problem [18], has a high worst-case complexity (EXPTIME-complete), several algorithms are known to work in practice [18, 27]. 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, 12]; this problem is known to be NEXPTIME-complete [26] and, so far, no practical algorithm is known. Moreover, this subtype relation seems to have an unnecessary expressive power; by closely looking at this relation, we can find useless examples that satisfy this (Appendix B). (It is still open if this subtype problem can be reduced to an easier problem, but we are quite pessimistic in this respect.)

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 useless 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. By consistently using this interpretation of types, we can construct a sensible polymorphic type system with a semantic flavor and, at the same time, obtain practical algorithms for typechecking by slightly modifying existing ones used in monomorphic type systems. For example, since each type denotes simply a set, we can still define the subtype relation essentially by the subset relation, without involving quantification. (We actually

[2] Note that *subtype polymorphism* is already treated. Throughout this paper, we mean by polymorphism only *parametric polymorphism*, following the tradition of functional programming community. Also, by "monomorphic," we refer to type systems with subtyping but *without* parametric polymorphism.

need to add a little more flexibility, as we will discuss in Section 2.2.) This allows us to reduce the subtyping problem to a slight variation of the tree automata containment problem (Section 4.4) and thus obtain a reasonably efficient algorithm by incorporating various known algorithmic techniques [18, 27]. (A little more specifically, our "marking" subtyping requires that variable occurrences on both sides correspond each other, whereas the "place holder" subtyping does not. This freedom allows somewhat unintuitive properties that hold set-theoretically, and handling such cases appears to be the hard core in the algorithmics. For example, for any set X, we have $(\{a\} \subseteq X) \vee (X \subseteq \overline{\{a\}})$. This property is critically used in the tricky example shown in Appendix B.)

A by-product of interpreting polymorphic types in this way 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 (Section 2.2). Both of these can be obtained by slight modifications to existing algorithms [13], as shown in Section 4.5 and 4.6.

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 touches on future work. Appendix A gives some addendum algorithms omitted from the main part of the paper. Appendix B shows an tricky example allowed in the above-mentioned naive extension of the semantic approach.

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 [17]. 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. In most part of this paper, for simplicity, we concentrate on the treatment of element structures of XML data. See Section 5 and 7 for discussion of XML attributes, higher-order functions, and other additional features.

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 key)(BibDB as d) : BibResult =
  match d with
    db[BibEntry* as l]  ->  iter(key)(l)
```

```
fun iter (String as key)(BibEntry* as l) : BibResult =
  match l with
    ()  ->  notfound[]
  | entry[key[String as key2], content[Any as c]],
                                              Any as rest
        ->  if key = key2 then found[c]
            else iter(key)(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 key 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, we proceed to evaluate the corresponding body, where we call the function iter with key and l. The function iter takes a string key 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. 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 key 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.[3] 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 [18, 13] 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 has the type of the pattern (that is, the type obtained after eliminating all the binders from the pattern), and then forms bindings of each variable to the subpart of the value 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 some input value. (More discussions on pattern matching can be found in [13, 16, 11].)

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

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

`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 any 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 decidability. (More details are given in [18].)

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 [16, 13, 11]. 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 variables over which the parameter types and the result type are quantified. 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 key)(DB{X} as d) : Result{X} =
  match d with
    db[Entry{X}* as l]  ->  iter{X}(key)(l)

fun iter {X}(String as key)(Entry{X}* as l) : Result{X} =
  match l with
    ()  ->  notfound[]
  | entry[key[String as key2], content[Any as c]],
                                  Any as rest
        ->  if key = key2 then found[c]
            else iter{X}(key)(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 name 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[4],

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

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 of the result value corresponding to `X` (i.e., the content of the `found`) is exactly the same as *some* subpart of the input value corresponding to `X` (i.e., the content of some `content`).[5]

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

In order to embody our safety property, we employ a "marking" semantics formalized in the following way. 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.

$$\begin{array}{l}\texttt{entry[key["..."], content[}v_1\texttt{]],}\\ \texttt{...}\\ \texttt{entry[key["..."], content[}v_n\texttt{]]}\end{array}$$

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

[5] This second condition might remind us a well-known *parametricity* property of the polymorphic lambda calculus [24]. However, the parametricity that our type system might be thought to have would be slightly different. In the polymorphic lambda calculus, any subpart of an input given type X will never be "touched" by the function body. In our case, the pattern matching construct is actually allowed to snoop such a subpart. But once such a subpart is "modified," the resulting value is never given type X.

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.

Most part of our polymorphic type system is 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 can 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 `a[X]` $\leqslant$ `a[Any]` does not hold (the right hand side requires no marking). Nevertheless, this function is safe in the sense of 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 upper bounds, which often appears in type systems that combine subtyping and polymorphism (cf. $F_{\leqslant}$ [5]).

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 $[\mathtt{X} \mapsto \mathtt{Contact}]\,\mathtt{DB\{X\}} = \mathtt{DB\{Contact\}}$ (since they are identical). We then give the type $[\mathtt{X} \mapsto \mathtt{Contact}]\,\mathtt{Result\{X\}} = \mathtt{Result\{Contact\}}$ 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

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

Hence, we obtain $\mathtt{T} = \mathtt{Contact}$. (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 support higher-order functions. See Section 5.1 for a related discussion.)

A question that may arise here is: does such a minimum type always exist? The answer is *no*, with the specification above only. For example, find a minimum `T` such that

$$\mathtt{a[c[]],b[d[]]} \;\leqslant\; [\mathtt{X} \mapsto \mathtt{T}]\;\mathtt{(a[Any],b[X]\ \mid\ a[X],b[Any])}.$$

This fails since 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. More precisely, we check *unambiguity* of a parameter type `U`:

> for any two marked values `v` and `w` of type `U`, if the values *without* markings are identical, then the values *with* markings are also identical.

Then, under this requirement, 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 $\forall \mathtt{X} \leqslant \mathtt{T}.\ \mathtt{a[X]} \rightarrow \mathtt{b[X]}$, 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. We continue discussing this in Section 5.2.

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 [18, 11]. 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. We will distinguish between these by whether they contain type variables or term variables.

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 countably infinite 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 countably infinite 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 $\mathbf{dom}(V) = \{x \mid x : \pi \in V\}$.

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 not on `b[],c[]`. We consider that this restriction is acceptable in many cases, where 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). Still, this restriction is rather artificial and would probably be better to be eliminated by some cleverer encoding. We leave this issue for future work.

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 countably infinite 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).

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.

$$\textsc{MCon}\ \frac{}{(a, \emptyset) \lhd a} \qquad \textsc{MPair}\ \frac{(v_i, V_i) \lhd p_i \quad \text{for } i = 1, 2}{(v_1, V_1) \otimes (v_2, V_2) \lhd (p_1, p_2)} \qquad \textsc{MAlt}\ \frac{(v, V) \lhd p_i \quad i = 1 \text{ or } 2}{(v, V) \lhd p_1 | p_2} \qquad \textsc{MVar}\ \frac{(v, V) \lhd p}{(v, V \cup \{x : \epsilon\}) \lhd x : p}$$

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

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 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

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

(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., [18]) 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, it is not clear what role type variables in patterns should have (for example, should patterns check markings at run time?). Therefore, for the present formalization, we treat the simplest case where no type variable appears in patterns.

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 Section 4.2for an algorithm for checking linearity.

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 4.4.

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. Unlike the usual syntactic way of defining the application σp of a substitution σ to a type p, we define it by what the type σp should satisfy semantically. Let us first see a simple case: $p = (x : 1, 1)$ and $\sigma = \{x \mapsto (y : 1, a)\}$. The type σp satisfies the following. For each marked value in p, which has the form (v, w) where v and w are any values and v is marked x, if the x-marked subpart v matches $\sigma(x) = (y : 1, a)$ with a marking U, then the type σp contains a marked value (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 $((v', a), w)$ where v' and w are any values and v' is marked y. Syntactically, σp can be written $((y : 1, a), 1)$. This result fits well with the usual syntactic definition of substitution, which would simply replace $x : 1$ in p by $(y : 1, a)$. Note, however, that if p contains a constraint on x, e.g., $(x : (b, 1), 1)$, then this constraint interacts with $(y : 1, a)$, resulting in $\sigma p = ((y : b, a), 1)$. In other words, intersection is taken on the type constraint $(b, 1)$ on x and the type $(y : 1, a)$ for the replacement.

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.

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

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 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. It is not obvious from the definition that such σp always exists. This will be proved in Section 4.3. Also, since we have no guarantee that such σp is syntactically unique, we assume some strategy that picks up one of types satisfying the above condition (e.g., the algorithm presented in Section 4.3).

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 function applications, 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 pattern matches, 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 directly yields the specification of our inference for patterns, whereas combining both yields that 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 $x \in X$.

$$
\begin{array}{rl}
 & (u, U) \lhd (p \lhd_X q)(x) \\
 & \quad\quad\text{iff} \\
\exists v, V, W, \pi. & (v, V) \lhd p \\
 & (v, W \cup \{x : \pi\}) \lhd q \\
 & (u, U) = \pi(v, V)
\end{array}
$$

That is, for each marked value (v, V) in p, if v matches q with 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 4.5 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 one: a *minimum* X-substitution σ such that $p \leqslant \sigma q$. The first inference specification given above does not give a minimum solution nor even a minimal one (where we mean by *solution* an X-substitution σ satisfying $p \leqslant \sigma q$). To see this, 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). Then, the inference yields $\{x \mapsto (a|b)\}$ because there are two possibilities that the value in p matches q and therefore the type inferred for x captures both values that can be bound to x. This result is not minimal since there are other substitutions 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. This 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 substitutions would complicate it).

Fortunately, by imposing two restrictions on p and q, we can ensure that the result $p \lhd_X q$ of inference is a minimum solution. The first restriction is that p is a subtype of q *ignoring* variables in X, more formally, 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 $p \leqslant_X q$ when this holds. This restriction is to ensure that $p \lhd_X q$ is a solution. The second restriction is that q yields a unique marking for any value, more formally, for all value v, we have that $(v, V) \lhd q$ and $(v, V') \lhd q$ imply $V|_X = V'|_X$. We say that q is X-*unambiguous* when this holds. This restriction is to ensure that $p \lhd_X q$ is smaller than any solution. We will show an algorithm for ambiguity check in Section 4.6.

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: The result 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, 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 $(v, V') \lhd \sigma q$ with $V' \subseteq V$. By the definition of substitution, there are V'', $U_1, \ldots, U_n$ such that $V' = V''|_{\overline{X}} \cup \bigcup_i \pi_i U_i$ where

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

Since q is X-unambiguous w.r.t. p, we know that $V''|_X = (W \cup \{x : \pi\})|_X$ and therefore $(x_i : \pi_i) = (x : \pi)$ for some i. 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

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

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

$$\begin{array}{l} (\pi_i(v), \sigma(x_i)) \lhd U_i \\ U_i \subseteq \pi_i^{-1} V \end{array}$$

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

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

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. $\square$

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

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

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

$$\begin{array}{rcl} e & ::= & x \mid a \mid (e, e) \mid f(e) \mid \\ & & \textbf{match}\ e\ \textbf{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \end{array}$$

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** $p_1 \rightarrow e_1 \mid \ldots \mid$

$p_n \rightarrow e_n$, 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. 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.

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.

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

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

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

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

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 type of the domain 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.[6]

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

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

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

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 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. We will discuss this in Section 5.2.

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 subpart marked x in the result value comes from some subpart marked x in the input value"—is best explained in terms of our evaluation semantics.

3.7 Correctness

The goal of this subsection is to prove the correctness of our type system consisting of two main theorems, *type soundness* ("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 $i = 1, 2$, then $(p_1, p_2) \leqslant (q_1, q_2)$.*

Proof: Let $(v, V) \lhd (p_1, p_2)$. Then, by MPair, $(v_i, V_i) \lhd p_i$ for $i = 1, 2$ with $v = (v_1, v_2)$ and $V = 1V_1 \cup 2V_2$. For $i = 1, 2$, from the assumption $p_i \leqslant q_i$, we obtain $(v_i, V_i') \lhd q_i$ with $V_i' \subseteq V_i$. Therefore $(v, 1V_1' \cup 2V_2') \lhd (p_1, p_2)$, where $1V_1' \cup 2V_2' \subseteq 1V_1 \cup 2V_2$. □

[6] Our evaluation semantics adopts a so-called big-step semantics, as opposed to a more popular 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.

Lemma 2 *If $p \leqslant q$, then $\sigma p \leqslant \sigma q$.*

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

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

From the assumption $p \leqslant q$, we have $(v, V'') \lhd q$ with $V'' \subseteq V'$. Since $V''|_X \subseteq V'|_X$, let K be the subset of $\{1, \ldots, n\}$ such that $\{(x_k : \pi_k) \mid k \in K\} = V''|_X$. Then, by the definition of substitution, $(v, W) \lhd q$ where $W = V''|_{\overline{X}} \cup \bigcup_{k \in K} \pi_k U_k$. The result follows since $W \subseteq V$. □

For a given target, the inference gives a bigger substitution from a bigger 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, W, 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 $(v, V') \lhd q$ with $V' \subseteq V$. Therefore, again by the definition of type inference, $(u, U') \lhd (p \lhd_X r)$ where $U' = \pi^{-1} V'$. The result follows since $U' \subseteq U$. □

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

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

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

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

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

In an inference, the following 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,

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

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

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

Again by the definition of type inference,

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

where $U' = \pi^{-1}V'$.

Note that $U'|_Y = \pi^{-1}(V'|_Y) = \{(y_i, \pi'_i) \mid \pi_i = \pi\pi'_i\}$. Also, if $\pi_i = \pi\pi'_i$, then, from $u = \pi(v)$ and $(\pi_i(v), U_i) \lhd \sigma(y_i)$, we have $(\pi'_i(u), U_i) \lhd \sigma(y_i)\pi'_i(u)$. From these, by the definition of substitution, we obtain

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

where $U'' = U'|_{\overline{Y}} \cup \bigcup\{\pi'_i U_i \mid \pi_i = \pi\pi'_i\}$.

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 $\sigma = \{x \mapsto \emptyset\}$. 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 strengthening the type environment preserves typability of an expression with its type becoming smaller. Here, we write $\Gamma' \leqslant \Gamma$ when $\Gamma'(x) \leqslant \Gamma(x)$ for any $x \in \mathbf{dom}(\Gamma)$.

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

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

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

The result follows immediately by TCON.

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

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

TPair $\dfrac{\Gamma \vdash e_i \in p_i \quad \text{for } i = 1, 2}{\Gamma \vdash (e_1, e_2) \in (p_1, p_2)}$ 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 $\dfrac{\Gamma \vdash e_0 \in p_0 \qquad X = \mathbf{var}(p_1 \mid \ldots \mid p_n) \qquad p_0 \leqslant_X p_1 \mid \ldots \mid p_n \qquad \Gamma, (p_0 \lhd_X p_i) \vdash e_i \in r_i}{\Gamma \vdash \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n \in r_1 \mid \ldots \mid r_n}$

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

From the induction hypothesis, $\Gamma' \vdash e_0 \in p'_0$ and $p'_0 \leqslant p_0$. 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, $(p'_0 \lhd_X p_i) \leqslant (p_0 \lhd_X p_i)$. Therefore by the induction hypothesis, $\Gamma', (p'_0 \lhd_X p_i) \vdash e_i \in r'_i$ and $r'_i \leqslant r_i$. 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 *If $\Gamma \vdash e \in p$, then $\sigma\Gamma \vdash e \in p'$ and $p' \leqslant \sigma p$ for some p'.*

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

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

The result follows immediately by TCon.

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

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

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

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

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

From the induction hypothesis, $\sigma\Gamma \vdash e_1 \in p_1'$ and $p_1' \leqslant \sigma p_1$. With $p_1 \leqslant_X p_2$ and Lemma 2, we have $p_1' \leqslant_X \sigma p_2$. We have $\sigma p_2 = p_2$ since $\mathbf{var}(p_2) \cap \mathbf{dom}(\sigma) = \emptyset$. 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 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 $\dfrac{\Gamma \vdash e_0 \in p_0 \qquad X = \mathbf{var}(p_1 \mid \ldots \mid p_n) \qquad p_0 \leqslant_X p_1 \mid \ldots \mid p_n \qquad \Gamma, (p_0 \lhd_X p_i) \vdash e_i \in r_i}{\Gamma \vdash \textbf{match}\ e_0\ \textbf{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n \in r_1 \mid \ldots \mid r_n}$
with $e = \textbf{match}\ e_0\ \textbf{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$ and $p = r_1 \mid \ldots \mid r_n$.

From the induction hypothesis, $\sigma\Gamma \vdash e_0 \in p_0'$ and $p_0' \leqslant \sigma p_0$. 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, $\sigma\Gamma, \sigma(p_0 \lhd_X p_i) \vdash e_i \in r_i'$ and $r_i' \leqslant r_i$. 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 $p' = r_1'' \mid \ldots \mid r_n'' \leqslant p$. □

Type soundness 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 soundness) *Let $\vdash \Phi$. If $\Gamma \vdash e \in p$ and $E \vdash e \Downarrow (w, W)$ with $\Gamma \vdash E$, then $(w, W) \lhd p$.*

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 $w = (a, \emptyset)$.

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

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

From TPair, $\Gamma \vdash e_i \in p_i$ for $i = 1, 2$ with $p = (p_1, p_2)$. By the induction hypothesis, $(v_i, V_i) \lhd p_i$. The result follows from MPair.

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

By TApp, $\textbf{fun } f\{X\}(x : p_1) : q = e_2$ and $\Gamma \vdash e_1 \in r$ with $r \leqslant_X p_1$ where $p = (r \lhd_X p_1)q$. By the induction hypothesis, $(v, V) \lhd r$. 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 $V' \subseteq V$. Therefore $x : (r \lhd_X p_1)p_1 \vdash x : (v, V)$. 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, $x : (r \lhd_X p_1)p_1 \vdash e_2 \in (r \lhd_X p_1)p_2$ and by the induction hypothesis, $(w, W) \lhd (r \lhd_X p_1)p_2$. Since we have $(r \lhd_X p_1)p_2 \leqslant (r \lhd_X p_1)q$ from $p_2 \leqslant q$ and Lemma 2, the result follows.

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

Let $X = \textbf{var}(p_1 \mid \ldots \mid p_n)$. From TMatch, we have $\Gamma \vdash e_0 \in p_0$ and $\Gamma, (p_0 \lhd_X p_i) \vdash e_i \in r_i$ with $p = r_1 \mid \ldots \mid r_n$. By induction hypothesis, $(v, V) \lhd p_0$. Since $(v, V') \lhd p_i$, we have $(\pi(v), \pi^{-1}V) \lhd (p_0 \lhd_X p_i)(x)$ for $(x : \pi) \in V'$ by the definition of type inference. Therefore $\Gamma, (p_0 \lhd_X p_i) \vdash E, \{x : \pi(v, V) \mid x : \pi \in V'\}$. From the induction hypothesis, $(w, W) \lhd r_i$. The result follows since $r_i \leqslant p$. □

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 an additional relation to explicitly state that e encounters a run-time error. Thus, the *failure relation* $E \vdash e \Downarrow \bot$ is defined by the following set of rules.

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

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

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

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

NSMatch2
$$\frac{E \vdash e_0 \Downarrow (v, V) \qquad \nexists V'.\ (v, V') \lhd p_i \qquad \text{for } i = 1, \ldots, n}{E \vdash \textbf{match } e_0 \textbf{ with } p_1 {\rightarrow} e_1 \mid \ldots \mid p_n {\rightarrow} e_n \Downarrow \bot}$$

SMatch3
$$\frac{E \vdash e_0 \Downarrow (v, V) \qquad (v, V') \lhd p_i \qquad E, \{x : \pi(v, V) \mid x : \pi \in V'\} \vdash e_i \Downarrow \bot}{E \vdash \textbf{match } e_0 \textbf{ with } p_1 {\rightarrow} e_1 \mid \ldots \mid p_n {\rightarrow} e_n \Downarrow \bot}$$

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

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

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

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

From TPair, $\Gamma \vdash e_i \in p_i$ where $p = (p_1, p_2)$. The result follows from the induction hypothesis.

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

The result follows similar to the case NSPair.

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

From TApp, $\Gamma \vdash e_1 \in r$ 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 $\vdash$ **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 $\dfrac{E \vdash e_0 \Downarrow \bot}{E \vdash \textbf{match } e_0 \textbf{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \Downarrow \bot}$
with $e = \textbf{match } e_0 \textbf{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$.

The result follows similar to the case NSPair.

NSMatch2 $\dfrac{E \vdash e_0 \Downarrow (v, V) \qquad v \not\lhd (p_1 \mid \ldots p_n)}{E \vdash \textbf{match } e_0 \textbf{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \Downarrow \bot}$
with $e = \textbf{match } e_0 \textbf{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$.

From TMatch, we have $\Gamma \vdash e_0 \in p_0$ 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.

SMatch3 $\dfrac{E \vdash e_0 \Downarrow (v, V) \qquad (v, V') \lhd p_i \qquad E, \{x : \pi(v, V) \mid x : \pi \in V'\} \vdash e_i \Downarrow \bot}{E \vdash \textbf{match } e_0 \textbf{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \Downarrow \bot}$
with $e = \textbf{match } e_0 \textbf{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$.

Let $X = \textbf{var}(p_1 \mid \ldots \mid p_n)$. From TMatch, we have $\Gamma \vdash e_0 \in p_0$ and $\Gamma, (p_0 \lhd_X p_i) \vdash e_i \in r_i$. 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 $(x : \pi) \in V'$ 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. $\square$

4 Algorithms

This section describes algorithms necessary for implementing the type system defined in the previous section. The algorithms presented here are 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*,
- I is a set of *initial states* ($I \subseteq S$),
- δ is a set of *transition rules* of the form either $s \rightarrow (s_1, s_2)$ or $s \rightarrow a$ where $s, s_1, s_2 \in S$ and $a \in \Sigma$, and
- Ξ is a mapping from states to sets of variables ($\Xi : S \rightarrow \mathcal{P}(\mathcal{X})$).

We write $\mathbf{var}(s)$ for the set of variables at the states reachable from s, and $\mathbf{var}(S) = \bigcup\{\mathbf{var}(s) \mid s \in S\}$. We also require that $\Xi(s) \cap (\mathbf{var}(s_1) \cup \mathbf{var}(s_2)) = \emptyset$ whenever $s \to (s_1, s_2) \in \delta$.

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.

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

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 is given in Appendix A. (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 $A \leqslant B$.)

We define the set, written πs, of *reachable states* from a given state s by a given path π as follows.

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

We define $\pi S = \bigcup\{\pi s \mid s \in S\}$ for a set S of states.

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), 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., [7].

In later proofs, we use the following lemma, which states that, when 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 *Suppose that all states are non-empty in A. 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, $(v_1, V_1) \lhd s_1$ 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

The goal of linearity check is to find whether a given pattern binds each pattern variable exactly once for any value. Given a marking automaton $A = (S, I, \delta, \Xi)$, the linearity check algorithm is as follows.

1. Empty-eliminate A.
2. Return "yes" iff all the following hold.
   (a) for each $s \to (s_1, s_2) \in \delta$, we have $\mathbf{var}(s_1) \cup \mathbf{var}(s_2) \cup \Xi(s) = \mathbf{var}(s)$.
   (b) for each $s \to (s_1, s_2) \in \delta$, we have $\mathbf{var}(s_1) \cap \mathbf{var}(s_2) = \emptyset$.
   (c) for each $s \in I$, we have $\mathbf{var}(s) = \mathbf{var}(I)$.

Condition (a) ensures that the same set of variables appears in each transitions from the same state. Condition (b) ensures that, for each transition $s \to (s_1, s_2)$, no common variable appears both in s_1 and in s_2. Condition (c) ensures that the same set of variables appears in all the initial states.

We can prove the following desirable property. (Note that, unlike a usual linearity check as in ML that only ensures the "only if" direction, we also guarantee the "if" direction.)

Proposition 2 *The linearity-checking algorithm returns "yes" iff A is linear, that is, $(v, V) \lhd A$ implies that $\mathbf{dom}(V) = \mathbf{var}(I)$ and V is a function.*

Proof: We first prove the "only if" direction. Assume that A is already empty-eliminated, and all (a), (b), and (c) hold. Then, the result suffices to show that, 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, $s \to (s_1, s_2) \in \delta$ and $(v_i, V_i) \lhd s_i$ for $i = 1, 2$ 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 (a), $\mathbf{dom}(V) = \Xi(s) \cup \mathbf{dom}(V_1) \cup \mathbf{dom}(V_2) = \Xi(s) \cup \mathbf{var}(s_1) \cup \mathbf{var}(s_2) = \mathbf{var}(s)$. Also, by (b), 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. By Lemma 8, we have the following.

$x \in \mathbf{var}(s)$ implies $(v, V) \lhd s$ and $x \in \mathbf{dom}(V)$ for some v,V.

The result follows by proving (1) if (b) does not hold, then A is not linear, (2) if (b) holds and (a) does not, then A is not linear, and (3) if (c) does not holds, then A is not linear. Below, we prove the first two since the third is trivial.

1. 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 (4.2), $(v_i, V_i) \lhd s_i$ and $x \in \mathbf{dom}(V_i)$ for some v_i,V_i for $i = 1, 2$, 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. Let $s \to (s_1, s_2) \in \delta$ and $\mathbf{var}(s_1) \cup \mathbf{var}(s_2) \cup \Xi(s) \subsetneq \mathbf{var}(s)$. Since s_1 and s_2 are not empty, $(v_i, V_i) \lhd s_i$ with $\mathbf{dom}(V_i) \subseteq \mathbf{var}(s_i)$ for some v_i and V_i for $i = 1, 2$. Therefore $(v, V) \lhd s$ with $V = 1V_1 \cup 2V_2 \cup \Xi(s) \times \{\epsilon\}$. Therefore $\mathbf{dom}(V) = \mathbf{dom}(V_1) \cup \mathbf{dom}(V_2) \cup \Xi(s) \subseteq \mathbf{var}(s_1) \cup \mathbf{var}(s_2) \cup \Xi(s) \subsetneq \mathbf{var}(s)$. Since A is empty-eliminated, (b) 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. Therefore $(v', V') \lhd A$ with $\mathbf{dom}(V') \subsetneq \mathbf{var}(I)$ for some v' and V'. $\square$

4.3 Substitution

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 below present an algorithm that, given 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 $B = (S_B, I_B, \delta_B, \Xi_B)$. Then, construct $C = (S_C, I_C, \delta_C, \Xi_C)$ such that (where $\top \notin S_A \cup S_B$)

$$\begin{array}{rcl} S_C & = & S_A \times S_B \cup \{(s,\top) \mid s \in S_A, x \notin \Xi_A(s)\} \\ I_C & = & \{(s,t) \mid s \in I_A, t \in I_B, x \in \Xi_A(s)\} \\ & & \cup\{(s,\top) \mid s \in I_A, x \notin \Xi_A(s)\} \\ \Xi_C((s,t)) & = & \begin{cases} \Xi_A(s) \setminus \{x\} \cup \Xi_B(t) & (t \neq \top) \\ \Xi_A(s) & (t = \top \wedge x \notin \Xi_A(s)). \end{cases} \end{array}$$

and

- $(s,t) \to ((s_1,t_1),(s_2,t_2)) \in \delta_C$ iff $s \to (s_1,s_2) \in \delta_A$ and either
  - $t \neq \top$ and $t \to (t_1,t_2) \in \delta_B$, or
  - $t = \top$ and either $x \notin \Xi_A(s_j) \wedge t_j = \top$ or $x \in \Xi_A(s_j) \wedge t_j \in I_B$ for $j = 1,2$.
- $(s,t) \to a \in \delta_C$ iff $s \to a \in \delta_A$ and either
  - $t \neq \top$ and $t \to a \in \delta_B$, or
  - $t = \top$.

The marking automaton produced by the algorithm contains two kinds of states, (s,t) and $(s,\top)$ (where $s \in S_A$ and $t \in S_B$). Transitions are constructed as follows. Starting with a state $(s_0,\top)$ 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 where we find the variable x in $\Xi_A(s_1)$. At that moment, we switch to a state (s_1,t_0) where $t_0 \in I_B$ and, from then on, 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 of the form (s,t) (where $t \neq \top$), we keep all variables in Ξ_A and Ξ_B except x. We need a little 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) where $t_0 \in I_B$.

Proposition 3 *C represents $\{x \mapsto B\}A$.*

PROOF: Let $X = \{x\}$. The result suffices to show that

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

iff

$$(v,W) \lhd s \tag{2}$$

$$V = W|_{\overline{X}} \cup U \tag{3}$$

where

- $t \neq \top$ and $(v,U) \lhd t$, or
- $t = \top$ and $U = \bigcup_{l=1..k} \pi^l R^l$ where $(\pi^l(v), R^l) \lhd I_B$ and $\{(x:\pi^1),\ldots,(x:\pi^k)\} = W|_X$.

The proof proceeds by induction on the structure of v.

Case: $v = a$

When $t \neq \top$, (1) holds if and only if

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

Thus, (4) is equivalent to (2) with $W = \Xi_A(s) \times \{\epsilon\}$, and (5) and (6) is equivalent to (3) with $U = \Xi_B(t) \times \{\epsilon\}$. When $t = \top$ and $x \notin \Xi_A(s)$, (1) holds if and only if (4) and

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

Again, (4) is equivalent to (2) with $W = \Xi_A(s) \times \{\epsilon\}$. (7) is equivalent to (3) since $x \notin \Xi_A(s)$ and therefore $W|_X = \emptyset$.

Case: $v = (v_1, v_2)$

When $t \neq \top$, (1) holds if and only if

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

By the induction hypothesis, (10) if and only if, for $j = 1, 2$,

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

Thus, (8) and (12) are equivalent to (2) with $W = \Xi_A(s) \times \{\epsilon\} \cup 1W_1 \cup 2W_2$. Also, (9) and (11) are equivalent to (3) with $U = \Xi_B(t) \times \{\epsilon\} \cup 1U_1 \cup 2U_2$. When $t = \top$ and $x \notin \Xi_A(s)$, (1) holds if and only if (8), (12), and

$$V = \Xi_A(s) \times \{\epsilon\} \cup 1V_1 \cup 2V_2 \tag{14}$$
$$x \notin \Xi(s_j) \wedge t_j = \top \quad \vee \quad x \in \Xi(s_j) \wedge t_j \in I_B \quad j = 1, 2 \tag{15}$$
$$V_j = W_j|_{\overline{X}} \cup U_j \tag{16}$$

where

$$t_j \neq \top \quad (v_j, U_j) \lhd t_j \tag{17}$$

or

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

Again, (8) and (12) are equivalent to (2) with $W = \Xi_A(s) \times \{\epsilon\} \cup 1W_1 \cup 2W_2$. Finally, (14), (15), and (16) are equivalent to (3) since combining (15), (17), and (18) yields

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

(note that $W_j|_X = \{x : \pi\}$ when $x \in \Xi_A(s_j)$ and $t_j \in I_B$ since x never occurs after x occurs) and therefore $U = 1U_1 \cup 2U_2$. □

4.4 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

   - $(s, T) \to ((s_1, T_1), (s_2, T_2)) \in \delta_C$ 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 $t_2 \in T_2$
   - $(s, T) \to a \in \delta_C$ 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., $\not\exists (v, V).\ (v, V) \lhd C$).

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. The rules for constructing C's transition rules can be understood as follows. For the first rule, 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 $W_2 \subseteq V_2$. The second rule 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 [18]. 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.[7]

Proposition 4 *The subtyping algorithm returns "yes" iff $A \leqslant B$.*

PROOF: The result suffices to show that

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

$$\forall t \in T.\ \not\exists W.\ (v, W) \lhd t \ \wedge\ W \subseteq V \tag{21}$$

[7]For readers familiar with the algorithm in [18], the additional rule would look like:

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

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

$$s \to a \in \delta_A \quad \wedge \quad V = \Xi_A(s) \times \{\epsilon\} \tag{23}$$
$$\forall t \in T.\ \ t \to a \notin \delta_B \ \vee\ \Xi_B(t) \times \{\epsilon\} \not\subseteq V \tag{24}$$

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) is equivalent to

$$(v_1, V_1) \lhd s_1 \quad \wedge \quad (v_2, V_2) \lhd s_2 \quad \wedge \quad V = \Xi_A(s) \times \{\epsilon\} \cup 1V_1 \cup 2V_2 \tag{25}$$

and (21) is equivalent to that, $\forall t \in T$,

$$t \to (t_1, t_2) \in \delta_B \tag{26}$$

implies

$$\forall W_1, W_2.\ (\neg(v_1, W_1) \lhd t_1 \quad \vee \quad \neg(v_2, W_2) \lhd t_2 \quad \vee \quad \Xi_B(t) \times \{\epsilon\} \cup 1W_1 \cup 2W_2 \not\subseteq V). \tag{27}$$

(27) can be transformed to:

$$\Xi_B(t) \subseteq \Xi_A(s) \quad \Rightarrow \quad \nexists W_i.(v_i, W_i) \lhd t_i \wedge W_i \subseteq V_i \quad (\exists i = 1,2) \tag{28}$$

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, "$\forall t \in T.$ (26) implies (28)" can be rephrased to:

$$\forall t \in T.\ \Xi_B(t) \subseteq \Xi_A(s) \ \wedge\ t \to (t_1, t_2) \in \delta_B \quad \Rightarrow \quad t_1 \in T_1 \vee t_2 \in T_2 \tag{29}$$

Further, we have, for each $i = 1, 2$,

$$\forall t_i \in T_i.\ \ \nexists W_i.\ (v_i, W_i) \lhd t_i \ \wedge\ W_i \subseteq V_i. \tag{30}$$

By the induction hypothesis and the definition of δ_C, (25), (29), and (30) altogether are equivalent to

$$(s,T) \to ((s_1,T_1),(s_2,T_2)) \in \delta_C \quad \wedge \quad (v_i, \emptyset) \lhd (s_i, T_i) \quad (\forall i = 1,2), \tag{31}$$

that is, $(v,\emptyset) \lhd (s,T)$. □

4.5 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.

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

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

   and

   - $(s,t) \to ((s_1,t_1),(s_2,t_2)) \in \delta_C$ iff $s \to (s_1,s_2) \in \delta_A$ and $t \to (t_1,t_2) \in \delta_B$
   - $(s,t) \to a \in \delta_C$ 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 such that

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

That is, we first compute 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. The converse also holds after empty-eliminating C. This is because, after this, each state (s,t) is non-empty and reachable from an initial state. Therefore, when we have a state (s,t) that marks x, there is some marked value (v,V) that is accepted by C and yields 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 [16, 13]. 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 9 *$(v,V) \lhd s$ and $(v,W) \lhd t$ with $U = V \cup W$ if and only if $(u,U) \lhd (s,t)$.*

Proof: Straightforward induction on the structure of v. □

Proposition 5 *$\{x_1 \mapsto D_{x_1}, \ldots, x_n \mapsto D_{x_n}\}$ is an X-inference of B with respect to A.*

Proof: The result suffices to show: $(v,V) \lhd s$ and $(w,W) \lhd t$ with $x : \pi \in W$ and $(u,U) = \pi(v,V)$ 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 9, $(v, V \cup W) \lhd (s,t)$. 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 $t \to (t_1,t_2) \in \delta_B$ where $(v_j, V') \lhd s_j$ with $V' = j^{-1}V$ and $(v_j, W') \lhd t_j$ with $W' = j^{-1}W$. 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 $(u,U) \lhd (s,t)$ with $(s,t) \in I_{D_x}$ and therefore $(u,U) \lhd (s,t)$ with $x \in \Xi_C((s,t))$. This implies $x : \epsilon \in U$. Also, since (s,t) is not eliminated in D, we have $(s,t) \in \pi I_C$. From Lemma 8, $(v,U') \lhd (p',q')$ with $(p',q') \in I_C$ and $\pi(v,U') = (u,U)$ for some v and U'. Therefore $(v,U') \lhd C$ and $x : \pi \in U'$. By Lemma 9, we have $(v,V) \lhd A$ and $(v,W) \lhd B$ with $U' = V \cup W$. Since $x \in \mathbf{var}(I_B)$, we have $x : \pi \in W$. □

4.6 Ambiguity

The ambiguity check aims to find whether, given a marking automaton A, any value accepted 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

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

   and

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

2. Empty-eliminate C; let D 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 $\Xi_A(s) = \Xi_A(t)$. 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, 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 accepted by D and produces a marking at the state (s,t). This means that A has two ways of accepting the value v that put different marks on the same subnode at the states s and t.

Lemma 10 *$(v,V) \lhd s$ and $(v,W) \lhd t$ with $U = V \setminus W \cup W \setminus V$ 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

$$s \to a \in \delta_A \quad V = \Xi_A(s) \times \{\epsilon\} \tag{32}$$
$$t \to a \in \delta_A \quad W = \Xi_A(t) \times \{\epsilon\}. \tag{33}$$

By letting $U = (V \setminus W) \cup (W \setminus V)$, (32) and (33) are equivalent to

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

That is, $(v,U) \lhd (p,q)$.

Case: $v = (v_1, v_2)$

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

$$s \to (s_1,s_2) \in \delta_A \quad (v_j,V_j) \lhd p_j \quad (j=1,2) \quad V = \Xi_A(s) \times \{\epsilon\} \cup 1V_1 \cup 2V_2 \tag{34}$$
$$t \to (t_1,t_2) \in \delta_A \quad (v_j,V_j) \lhd q_j \quad (j=1,2) \quad W = \Xi_A(t) \times \{\epsilon\} \cup 1W_1 \cup 2W_2. \tag{35}$$

Let $U_j = (V_j \setminus W_j) \cup (W_j \setminus V_j)$. By the induction hypothesis, (34) and (35) are equivalent to $(v_j, U_j) \lhd (s_j, t_j)$ for $j = 1, 2$, which is in turn equivalent to $(v, U) \lhd (s,t)$ where $U = (V \setminus W) \cup (W \setminus V)$. □

Proposition 6 *The ambiguity-checking algorithm returns "unambiguous" iff A is X-unambiguous.*

Proof: The result suffices to show:

$$(v, V) \lhd I_A \quad \wedge \quad (v, W) \lhd I_A \quad \Rightarrow \quad V = W \quad (\forall v, V, W) \tag{36}$$

if and only if

$$\Xi_D((s,t)) = \emptyset \quad (\forall (s,t) \in S_D). \tag{37}$$

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 $(v, U) \lhd (s,t)$ for some $U \neq \emptyset$. By Lemma 8, $(v, V') \lhd I_D$ (therefore $(v, V') \lhd I_C$) with $\pi(v, V') = (u, U)$, which implies $V' \neq \emptyset$. By Lemma 10, $(v, V) \lhd I_A$ and $(v, W) \lhd I_B$ with $V \neq W$ for some V and W.

We then show the "if" direction. Suppose (37) with $(v, V) \lhd I_A$ and $(v, W) \lhd I_A$ where $V \neq W$. Then, by Lemma 10, $(v, U) \lhd I_C$ (therefore $(v, U) \lhd I_D$) with $U \neq \emptyset$. This implies $\Xi_D((s', t')) \neq \emptyset$ for some $(s', t') \in S_D$, contradicting (37). □

5 Extensions

This section discusses some possibilities of extending the type system described in the previous two sections.

5.1 Higher-order functions

Since polymorphism is often used in conjunction with higher-order functions (e.g., `map`, `fold`, `sort`, etc.), supporting these is quite important. 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 second approach is technically more ambitious since it is not straightforward to incorporate functions in the denotation of types and at the same time ensure the decidability of subtyping, etc. One attempt would be to combine a treatment by Frisch, Castagna, and Benzaken of higher-order functions in the monomorphic case [11] with our polymorphism presented in this paper. We have not yet worked out this direction, however.

The first approach mentioned above is easier. We may define subtyping by using usual inference rules for function types (contravariant in argument types and covariant in result types) and use our semantic definition for the other types. One issue is that, in the inference of type arguments, it may not always be desirable to find minimum solutions. That is, without higher-order functions, type parameters always appear in positive positions in the result type and therefore minimum type arguments always minimizes the result type. With higher-order functions, however, type parameters may occur in negative positions and therefore we need to maximize type arguments in order to minimize the result type. (If a type parameter occurs both in positive and negative positions, then we need to check that the minimum and the maximum solutions are equal.) Fortunately, obtaining maximum solutions is easy: it is always the substitution that maps each variable to the top type 1, as shown by the following proposition.

Proposition 7 *Let $p \leqslant_X q$ and let $\sigma_1(x) = 1$ for $x \in X$. Then, σ_1 is the maximum X-substitution σ satisfying $p \leqslant \sigma q$.*

Proof: The result suffices to show (1) $p \leqslant \sigma q$ implies $\sigma \leqslant \sigma_1$ and (2) $p \leqslant \sigma_1 q$. (1) is trivial. (2) follows from $p \leqslant_X q$. □

5.2 Intersection and difference operation

As in $\mathbb{C}$Duce [3], 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 by the great lower bound of types (w.r.t. $\leqslant$): $(v, V) \lhd p_1 \cap p_2$ iff $(v, V_1) \lhd p_1$ and $(v, V_2) \lhd p_2$ with $V = V_1 \cup V_2$. (Indeed, this definition coincides with the semantics of *intersection patterns* in $\mathbb{C}$Duce.) Note that, from the definition of substitution, we have that $(v, V) \lhd [x \mapsto p_1](x : p_2)$ iff $(v, V_1) \lhd p_1$ and $(v, V_2) \lhd p_2$ and $V = V_1 \cup V_2$. 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 we effectively already support intersections.

For the difference operator, we could analogously define it: $(v, V) \lhd p_1 \setminus p_2$ 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 as $p_1 \setminus p_2$ seems not expressible in general by types with no 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. (We could take difference as a built-in operator, but we have not yet pursued this direction.) Even if the general case fails, we can easily represent the difference when 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 typechecker 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.)

6 Related Work

There have been very few work on a static type system for XML that supports 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 somewhat attain 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 [28], which maps DTDs to Haskell's polymorphic type system [23]. Another work close to this is XMLambda [20, 25], 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. One interesting question would be whether and how much the host language's polymorphism could help relaxing well known inflexibilities proper to the data-binding approach but not existent in the semantic approach ("spurious structure introduced by encoding regular expressions"). We do not yet know the answer.

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 only a first step toward a full-fledged polymorphic type system for XML and various other 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 [15, 11].

References

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

[2] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13. Springer-Verlag, August 91.

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

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

[5] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994.

[6] James Clark and Makoto Murata. RELAX NG. `http://www.relaxng.org`, 2001.

[7] Hubert Comon, Max Dauchet, Rémy Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Draft book; available electronically on `http://www.grappa.univ-lille3.fr/tata`, 1999.

[8] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[9] David Fallside and Yves Lafon. XML protocol working group. `http://www.w3.org/2000/xp/Group/`, 2004.

[10] Peter Fankhauser, Mary Fernández, Ashok Malhotra, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 Formal Semantics. `http://www.w3.org/TR/query-semantics/`, 2001.

[11] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *Seventeenth Annual IEEE Symposium on Logic In Computer Science*, pages 137–146, 2002.

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

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

[14] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for XML. `http://arbre.is.s.u-tokyo.ac.jp/~hahosoya/papers/polyx.ps`, 2004.

[15] Haruo Hosoya and Makoto 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.

[16] Haruo Hosoya and Benjamin 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.

[17] Haruo Hosoya and Benjamin 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.

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

[19] Xavier Leroy, Damien Doligez, Jacques Garrigue, Jérôme Vouillon, and Dider Rémy. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[20] Erik Meijer and Mark Shields. XMλ: A functional programming language for constructing and manipulating XML documents. Manuscript, 1999.

[21] Tova Milo, Dan Suciu, and Victor 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.

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

[23] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 1993.

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

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

[26] Kjartan 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.

[27] Akihiko Tozawa and Masami 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.

[28] Malcolm Wallace and Colin 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., September 27–29 1999. ACM Press.

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. We first preprocess 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 1 by $s_\top$.
2. For each occurrences of (p_1, p_2), add new definitions $s_1 \mapsto p_1$ and $s_2 \mapsto p_2$ to Δ and replace (p_1, p_2) with (s_1, s_2).

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

- $S = \{(s, X) \mid s \in \mathbf{dom}(\Delta),\ X \subseteq X_0\}$,
- $I = \{(s_0, X) \mid X \subseteq X_0\}$,
- $\delta = \{(s, X) \to f \mid f \in \mathrm{atoms}_X(s)\}$, and
- $\Xi((s, X)) = X$.

Here, f has the form either a or $((s, X), (s, X))$. The function $\mathrm{atoms}_X(p)$ is defined as follows.

$$\begin{array}{lcll}
\mathrm{atoms}_\emptyset(a) & = & \{a\} & \\
\mathrm{atoms}_X(a) & = & \emptyset & (X \neq \emptyset) \\
\mathrm{atoms}_\emptyset((s_1, s_2)) & = & \{((s_1, X_1), (s_2, X_2)) \mid X_1 \subseteq X_0,\ X_2 \subseteq X_0\} & \\
\mathrm{atoms}_X((s_1, s_2)) & = & \emptyset & (X \neq \emptyset) \\
\mathrm{atoms}_X(s) & = & \mathrm{atoms}_X(\Delta(s)) & \\
\mathrm{atoms}_X(p_1|p_2) & = & \mathrm{atoms}_X(p_1) \cup \mathrm{atoms}_X(p_2) & \\
\mathrm{atoms}_X(x : p) & = & \mathrm{atoms}_{X \setminus \{x\}}(p) & (x \in X) \\
\mathrm{atoms}_X(x : p) & = & \emptyset & (x \notin X)
\end{array}$$

As usual, it is not necessary to fully construct the automaton as above since $\mathrm{atoms}_X(s)$ is the empty set for most $X \subseteq X_0$.

B Tricky subtyping example

This section attempts to show that subtyping defined by "quantifying the subset relation over all substitutions" is needlessly powerful. The following subset relation holds 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.) 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 1") do not work. Let us have a closer look at the above relation. It can be transformed to

$$
\begin{aligned}
& (a \subseteq a \cup X \ \vee \ X \subseteq \emptyset) && (38)\\
\wedge \quad & (a \subseteq a \ \vee \ X \subseteq a) && (39)\\
\wedge \quad & (a \subseteq X \ \vee \ X \subseteq \overline{a}) && (40)\\
\wedge \quad & (a \subseteq \emptyset \ \vee \ X \subseteq \overline{a} \cup a) && (41)
\end{aligned}
$$

according to the subtyping algorithm in [18]. 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 1 and continue to check $1 \subseteq \overline{a} \cup a$ (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.

Acknowledgments

I would like to express our best gratitude to Jérôme Vouillon, Benjamin Pierce, Vladimir Gapeyev, and Naoki Kobayashi for precious comments and useful discussions. 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.