Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 323 (2016) 235–251

www.elsevier.com/locate/entcs

Type Soundness for Path Polymorphism ⋆

Andrés Viso[a,1] Eduardo Bonelli[b,2] Mauricio Ayala-Rincón[c,3]

[a] *Consejo Nacional de Investigaciones Científicas y Técnicas – CONICET*
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires – UBA
Buenos Aires, Argentina

[b] *Consejo Nacional de Investigaciones Científicas y Técnicas – CONICET*
Departamento de Ciencia y Tecnología
Universidad Nacional de Quilmes – UNQ
Bernal, Argentina

[c] *Departamentos de Matemática e Ciência da Computação*
Universidade de Brasília – UnB
Brasília D.F., Brasil

Abstract

Path polymorphism is the ability to define functions that can operate uniformly over arbitrary recursively specified data structures. Its essence is captured by patterns of the form $x\,y$ which decompose a compound data structure into its parts. Typing these kinds of patterns is challenging since the type of a compound should determine the type of its components. We propose a static type system (*i.e.* no run-time analysis) for a pattern calculus that captures this feature. Our solution combines type application, constants as types, union types and recursive types. We address the fundamental properties of Subject Reduction and Progress that guarantee a well-behaved dynamics. Both these results rely crucially on a notion of *pattern compatibility* and also on a coinductive characterisation of subtyping.

Keywords: λ-Calculus, Pattern Matching, Path Polymorphism, Static Typing

1 Introduction

Applicative representation of data structures in functional programming languages consists in applying variable arity constructors to arguments. Examples are:

$$s = \mathtt{cons}\,(\mathtt{vl}\,v_1)\,(\mathtt{cons}\,(\mathtt{vl}\,v_2)\,\mathtt{nil})$$
$$t = \mathtt{node}\,(\mathtt{vl}\,v_3)\,(\mathtt{node}\,(\mathtt{vl}\,v_4)\,\mathtt{nil}\,\mathtt{nil})\,(\mathtt{node}\,(\mathtt{vl}\,v_5)\,\mathtt{nil}\,\mathtt{nil})$$

⋆ Work partially funded by the international project DeCOPA STIC-AmSud 146/2012, CONICET, CAPES, CRNS.

[1] Email: aeviso@dc.uba.ar

[2] Email: eabonelli@gmail.com

[3] Email: ayala@unb.br

http://dx.doi.org/10.1016/j.entcs.2016.06.015

These are data structures that hold values, prefixed by the constructor `vl` for "value" ($v_{1,2}$ in the first case, and $v_{3,4,5}$ in the second). Consider the following function for updating the values of any of these two structures by applying some user-supplied function f to it:

$$\begin{array}{lll} \mathsf{upd} = f \to_{\{f:A\supset B\}} & (\ \mathtt{vl}\, z \to_{\{z:A\}} & \mathtt{vl}\,(f\, z) \\ & \mid x\, y \ \to_{\{x:C,y:D\}} & (\mathsf{upd}\, f\, x)\,(\mathsf{upd}\, f\, y) \\ & \mid w \quad \to_{\{w:E\}} & w) \end{array} \tag{1}$$

Both $\mathsf{upd}\,(+1)\,s$ and $\mathsf{upd}\,(+1)\,t$ may be evaluated. The expression to the right of "=" is called an *abstraction* and consists of a unique *branch*; this branch in turn is formed from a pattern (f), a user-specified type declaration for the variables in the pattern ($\{f : A \supset B\}$), and a body (in this case the body is itself another abstraction that consists of three branches). Type declarations bind variables in both the pattern and the body. An argument to an abstraction is matched against the patterns, in the order in which they are written, and the appropriate body is selected. Notice the pattern $x\, y$. This pattern embodies the essence of what is known as *path polymorphism* [17,19] since it abstracts a path being "split". The starting point of this paper is how to type a calculus, let us call it CAP for *Calculus of Applicative Patterns*, that admits such examples. CAP may be seen as the static patterns fragment of PPC where instead of the usual abstraction we have alternatives. We next show why the problem is challenging, explain our contribution and also discuss why the current literature falls short of addressing it. We do so with an introduction-by-example approach, for the full syntax and semantics of the calculus refer to Sec. 2.

Preliminaries on typing patterns expressing path polymorphism

Consider these two simple examples:

$$(\mathtt{nil} \to 0)\,\mathtt{cons} \qquad\qquad (\mathtt{vl}\, x \to_{\{x:\mathsf{Nat}\}} x + 1)\,(\mathtt{vl}\,\mathtt{true}) \tag{2}$$

They should clearly not be typable. In the first case, the abstraction is not capable of handling `cons`. This is avoided by introducing singleton types in the form of the constructors themselves: `nil` is given type $\mathfrak{nil}$ while `cons` is given type $\mathfrak{cons}$; these are then compared. In the second case, x in the pattern is required to be Nat yet the type of the argument to `vl` in `vl true` is Bool. This is avoided by introducing type application [24] into types: `vl` x is assigned a type of the form $\mathfrak{vl}$ @ Nat while `vl true` is assigned type $\mathfrak{vl}$ @ Bool; these are then compared.

Consider next the pattern $x\, y$ of `upd`. It can be instantiated with different applicative terms in each recursive call to `upd`. For example, suppose $A = B =$ Nat, that v_1 and v_2 are numbers and consider $\mathsf{upd}\,(+1)\,s$. The following table illustrates some of the terms with which x and y are instantiated during the evaluation of $\mathsf{upd}\,(+1)\,s$:

	x	y
$\mathtt{upd}\,(+1)\,s$	$\mathtt{cons}\,(\mathtt{vl}\,v_1)$	$\mathtt{cons}\,(\mathtt{vl}\,v_2)\,\mathtt{nil}$
$\mathtt{upd}\,(+1)\,(\mathtt{cons}\,(\mathtt{vl}\,v_1))$	$\mathtt{cons}$	$\mathtt{vl}\,v_1$
$\mathtt{upd}\,(+1)\,(\mathtt{cons}\,(\mathtt{vl}\,v_2)\,\mathtt{nil})$	$\mathtt{cons}\,(\mathtt{vl}\,v_2)$	$\mathtt{nil}$

The type assigned to x (and y) should encompass all terms in its respective column. This suggests adopting a union type for x. On the assumption that the programmer has provided an exhaustive coverage, the type of x in upd is:

$$\mu\alpha.(\mathfrak{vl} \mathbin{@} A) \oplus (\alpha \mathbin{@} \alpha) \oplus (\mathfrak{cons} \oplus \mathfrak{node} \oplus \mathfrak{nil})$$

Here μ is the recursive type constructor and $\oplus$ the union type constructor. The variable y in the pattern $x\,y$ will also be assigned the same type. Note that upd itself is assigned type $(A \supset B) \supset (F_A \supset F_B)$, where F_X is $\mu\alpha.(\mathfrak{vl} \mathbin{@} X) \oplus (\alpha \mathbin{@} \alpha) \oplus (\mathfrak{cons} \oplus \mathfrak{node} \oplus \mathfrak{nil})$. Thus variables in applicative patterns will be assigned union types.

Recursive types are useful to give static semantics to fixpoint combinators, which embodies the essence of recursion and thus *path polymorphism*. Together with unions, they allow to model recursively defined data types. Combining these ideas with type application allows to define data types in a more intuitive manner, like for example lists and trees

$$\mu\alpha.\mathfrak{nil} \oplus (\mathfrak{cons} \mathbin{@} A \mathbin{@} \alpha) \qquad \mu\alpha.\mathfrak{nil} \oplus (\mathfrak{node} \mathbin{@} A \mathbin{@} \alpha \mathbin{@} \alpha)$$

The advantage of this approach is that the type expression reflects the structure of the terms that inhabit it (*cf.* Fig. 3). This will prove to be convenient for our proposed notion of *pattern compatibility*.

Compatibility is the key for ensuring Safety (Subject Reduction, SR for short, and Progress). Consider the following example:

$$(\mathtt{vl}\,x \to_{\{x:\mathsf{Bool}\}} \mathtt{if}\ x\ \mathtt{then}\ 1\ \mathtt{else}\ 0) \mid (\mathtt{vl}\,y \to_{\{y:\mathsf{Nat}\}} y + 1) \tag{3}$$

Although there is a branch capable of handling a term such as $\mathtt{vl}\,4$, namely the second one, evaluation in CAP takes place in left-to-right order following standard practice in functional programming languages. Since the term $\mathtt{vl}\,4$ *also* matches the pattern $\mathtt{vl}\,x$, we would obtain the (incorrect) reduct $\mathtt{if}\ 4\ \mathtt{then}\ 1\ \mathtt{else}\ 0$. We thus must relate the types of $\mathtt{vl}\,x$ and $\mathtt{vl}\,y$ in order to avoid failure of SR. Since $\mathtt{vl}\,y$ is an instance of $\mathtt{vl}\,x$, we require the type of the latter to be a subtype of the type of the former since it will always have priority: $\mathfrak{vl} \mathbin{@} \mathsf{Nat} \preceq \mathfrak{vl} \mathbin{@} \mathsf{Bool}$. Fortunately, this is not the case since $\mathsf{Nat} \not\preceq \mathsf{Bool}$, rendering this example untypable.

Consider now, a term such as:

$$f \to_{\{f:A\supset B\}} \begin{array}{ll} (\ \mathtt{vl}\,z & \to_{\{z:A\}} \quad \mathtt{vl}\,(f\,z) \\ \mid x\,y & \to_{\{x:C,y:D\}} x\,y) \end{array} \tag{4}$$

This function takes an argument f and pattern-matches with a data structure to apply f only when this data structure is an application with the constructor `vl` on the left-hand side. Assigning x in the second branch the type $C = \mathbb{vl}$ is a potential source of failure of SR since the function would accept arguments of type $\mathbb{vl} \mathbin{@} D$. Our proposed notion of compatibility will check the *types* occurring at offending positions in the *type* of both patterns. In this case, if $C = \mathbb{vl}$ then $C \mathbin{@} D \preceq \mathbb{vl} \mathbin{@} A$ is enforced. Note that if C were a type such as $\mu\alpha.\mathbb{vl} \oplus \alpha \mathbin{@} \alpha$, then also the same condition would be enforced.

Let us return to example (1). The type declarations would be $C = D = \mu\alpha.(\mathbb{vl} \mathbin{@} A) \oplus (\alpha \mathbin{@} \alpha) \oplus (\mathbb{cons} \oplus \mathbb{node} \oplus \mathbb{nil})$ and $E = \mathbb{cons} \oplus \mathbb{node} \oplus \mathbb{nil}$. We now illustrate how compatibility determines any possible source of failure of SR. Let us call p, q and r the three patterns of the innermost abstraction of (1), resp. Since pattern p does not subsume q, we determine the (maximal) positions in both patterns which are sources of failure of subsumption. In this case, it is that of `vl` in p and x in q. We now consider the *subtype* at that position in $\mathbb{vl} \mathbin{@} A$, the type of p, and the *subtype* at the same position in $F_A \mathbin{@} F_A$, the type of q: the first is $\mathbb{vl}$ and the second is F_A. Since F_A does not admit $\mathbb{vl}$ (*cf.* Def. 3.5), these branches are immediately declared compatible. In the case of p and r, ϵ is the offending position in the failure of p subsuming r: since the type application constructor @ located at position ϵ in $\mathbb{vl} \mathbin{@} A$ is not admitted by E, the type of r, these branches are immediately declared compatible. Finally, a similar analysis between q and r entails that these are compatible too. The type system and its proof of Safety will therefore assure us that this example preserves typability.

Summary of contributions:

- A typing discipline for CAP. We statically guarantee safety for path polymorphism in its purest form (other, more standard forms of polymorphism such as parametric polymorphism which we believe to be easier to handle, are out of the scope of this paper).
- Invertibility of subtyping of recursive types. This is crucial for the proof of safety (*cf.* next item). It relies on an equivalent coinductive formulation for which invertibility implies invertibility of subtyping of recursive types.
- A proof of safety for the resulting system. It relies on the syntactic notion of pattern compatibility mentioned above, hence no runtime analysis is required.

Related work

The literature on (typed) pattern calculi is extensive; we mention the most relevant ones (see [17, 19] for a more thorough listing). In [2] the constructor calculus is proposed. It has a different notion of pattern matching: it uses a case construct $\{c_1 \mapsto s_1, \ldots, c_n \mapsto s_n\} \cdot t$ in which certain occurrences of the constructors c_i in t are replaced by their corresponding terms. [24] studies typing to ensure that these constructor substitutions never block on a constant not in their domain. Recursive types are not considered (nor is path polymorphism). Two further closely related

efforts merit comments: the first is the work by Jay and Kesner and the second is that of the ρ-calculus by Kirchner and colleagues.

In [18,19] the Pure Pattern Calculus (PPC) is studied. It allows patterns to be computed dynamically (they may contain free variables). A type system for a PPC like calculus is given in [17] however neither recursive nor union types are considered. [17] also studies a simple static pattern calculus. However, there are numerous differing aspects w.r.t. this work among which we can mention the following. First, the typed version of [17] (the *Query Calculus*) omits recursive types and union types. Then, although it admits a form of path polymorphism, this is at the cost of matching types at runtime and thus changing the operational semantics of the untyped calculus; our system is purely static, no runtime analysis is required.

The ρ-calculus [9] is a generic pattern matching calculus parametrized over a matching theory. There has been extensive work exploring numerous extensions [5, 10–13, 22]. None addresses path polymorphism however. Indeed, none of the above allow patterns of the form $x\,y$. This limitation seems to be due to the alternative approach to typing $\mathsf{c}\,x$ adopted in the literature on the ρ-calculus where c is assigned a *fixed* functional type. This approach seems incompatible with path polymorphism, as we see it, in that it suggests no obvious way of typing patterns of the form $x\,y$ where x denotes an arbitrary piece of unstructured *data*. Additional differences with our work are:

- [12]: It does not introduce union types. No runtime matching error detection takes place (this is achieved via Progress in our paper).
- [10]: It deals with an untyped ρ-calculus. Hence no SR.
- [5, 11]: Neither union nor recursive types are considered.

Structure of the paper. Sec. 2 introduces the terms and operational semantics of CAP. The typing system is developed in Sec. 3 together with a precise definition of compatibility. Sec. 4 studies Safety: SR and Progress. For the benefit of the reviewing process a full report with all details of the proofs is available online [27].

2 Syntax and Operational Semantics of CAP

We assume given an infinite set of term variables $\mathbb{V}$ and constants $\mathbb{C}$. The syntax of CAP consists of four syntactic categories, namely **patterns** $(p, q, \ldots)$, **terms** $(s, t, \ldots)$, **data structures** $(d, e, \ldots)$ and **matchable forms** $(m, n, \ldots)$:

$$
\begin{array}{lll}
p ::= x & \text{(matchable)} & t ::= x \quad \text{(variable)} \\
\quad \mid \mathsf{c} & \text{(constant)} & \quad \mid \mathsf{c} \quad \text{(constant)} \\
\quad \mid p\,p & \text{(compound)} & \quad \mid t\,t \quad \text{(application)} \\
 & & \quad \mid p \to_\theta t \mid \ldots \mid p \to_\theta t \quad \text{(abstraction)} \\
d ::= \mathsf{c} & \text{(constant)} & m ::= d \quad \text{(data structure)} \\
\quad \mid d\,t & \text{(compound)} & \quad \mid p \to_\theta t \mid \ldots \mid p \to_\theta t \quad \text{(abstraction)}
\end{array}
$$

The set of patterns, terms, data structures and matchable forms are denoted $\mathbb{P}$, $\mathbb{T}$, $\mathbb{D}$ and $\mathbb{M}$, resp. Variables occurring in patterns are called **matchables**. We often abbreviate $p_1 \to_{\theta_1} s_1 \mid \ldots \mid p_n \to_{\theta_n} s_n$ with $(p_i \to_{\theta_i} s_i)_{i \in 1..n}$. The θ_i are typing contexts annotating the type assignments for the variables in p_i (*cf.* Sec. 3).

Definition 2.1 The **free variables** of a term (notation $\mathsf{fv}(t)$) and **free matchables** of a pattern ($\mathsf{fm}(p)$) are defined inductively as follows:

$$\begin{aligned} \mathsf{fv}(x) &\triangleq \{x\} \\ \mathsf{fv}(\mathsf{c}) &\triangleq \varnothing \\ \mathsf{fv}(r\,u) &\triangleq \mathsf{fv}(r) \cup \mathsf{fv}(u) \\ \mathsf{fv}((p_i \to_{\theta_i} s_i)_{i \in 1..n}) &\triangleq \textstyle\bigcup_{i \in 1..n} (\mathsf{fv}(s_i) \setminus \mathsf{fm}(p_i)) \end{aligned} \qquad \begin{aligned} \mathsf{fm}(x) &\triangleq \{x\} \\ \mathsf{fm}(\mathsf{c}) &\triangleq \varnothing \\ \mathsf{fm}(p\,q) &\triangleq \mathsf{fm}(p) \cup \mathsf{fm}(q) \end{aligned}$$

Positions in patterns and terms are defined as expected and denoted $\pi, \pi', \ldots$ (ϵ denotes the root position). We write $\mathsf{pos}(s)$ for the set of positions of s and $s|_\pi$ for the subterm of s occurring at position π.

A **substitution** ($\sigma, \sigma_i, \ldots$) is a partial function from term variables to terms. If it assigns u_i to x_i, $i \in 1..n$, then we write $\{u_1/x_1, \ldots, u_n/x_n\}$. Its domain ($\mathsf{dom}\,(\sigma)$) is $\{x_1, \ldots, x_n\}$. Also, $\{\}$ is the identity substitution. We write σs for the result of applying σ to term s. **Matchable forms** are required for defining the **matching operation**, described next.

Given a pattern p and a term s, the matching operation $\{\!\{s/p\}\!\}$ determines whether s matches p. It may have one of three outcomes: success, fail (in which case it returns the special symbol `fail`) or undetermined (in which case it returns the special symbol `wait`). We say $\{\!\{s/p\}\!\}$ is **decided** if it is either successful or it fails. In the former it yields a substitution σ; in this case we write $\{\!\{s/p\}\!\} = \sigma$. The disjoint union of matching outcomes is given as follows ("$\triangleq$" is used for definitional equality):

$$\begin{aligned} \mathtt{fail} \uplus o &\triangleq \mathtt{fail} \\ o \uplus \mathtt{fail} &\triangleq \mathtt{fail} \\ \sigma_1 \uplus \sigma_2 &\triangleq \sigma_1 \cup \sigma_2 \end{aligned} \qquad \begin{aligned} \mathtt{wait} \uplus \sigma &\triangleq \mathtt{wait} \\ \sigma \uplus \mathtt{wait} &\triangleq \mathtt{wait} \\ \mathtt{wait} \uplus \mathtt{wait} &\triangleq \mathtt{wait} \end{aligned}$$

where o denotes any possible output and $\sigma_1 \cup \sigma_2$ denotes the standard union of substitutions assuming that their domains are disjoint. To ensure this always holds we assumed patterns to be linear (at most one occurrence of any matchable). The matching operation is defined as follows, where the defining clauses below are evaluated from top to bottom [4]:

$$\begin{aligned} \{\!\{u/x\}\!\} &\triangleq \{u/x\} && \\ \{\!\{\mathsf{c}/\mathsf{c}\}\!\} &\triangleq \{\} && \\ \{\!\{u\,v/p\,q\}\!\} &\triangleq \{\!\{u/p\}\!\} \uplus \{\!\{v/q\}\!\} && \text{if } u\,v \text{ is a } \textit{matchable form} \\ \{\!\{u/p\}\!\} &\triangleq \mathtt{fail} && \text{if } u \text{ is a } \textit{matchable form} \\ \{\!\{u/p\}\!\} &\triangleq \mathtt{wait} && \end{aligned}$$

[4] Specialization of the matching operation introduced in [19] to static patterns.

For example: $\{\!\{x \to s/\mathtt{c}\}\!\} = \mathtt{fail}$; $\{\!\{\mathtt{d}/\mathtt{c}\}\!\} = \mathtt{fail}$; $\{\!\{x/\mathtt{c}\}\!\} = \mathtt{wait}$ and $\{\!\{\mathtt{c}\,\mathtt{c}/x\,\mathtt{d}\}\!\} = \mathtt{fail}$. We now turn to the only reduction axiom of CAP:

$$\frac{\{\!\{u/p_i\}\!\} = \mathtt{fail} \text{ for all } i < j \quad \{\!\{u/p_j\}\!\} = \sigma_j \quad j \in 1..n}{(p_i \to_{\theta_i} s_i)_{i\in 1..n}\, u \to \sigma_j s_j}\ (\beta)$$

It may be applied under any context and states that if the argument u to an abstraction $(p_i \to_{\theta_i} s_i)_{i\in 1..n}$ fails to match all patterns p_i with $i < j$ and successfully matches pattern p_j (producing a substitution σ_j), then the term $(p_i \to_{\theta_i} s_i)_{i\in 1..n}\, u$ reduces to $\sigma_j s_j$.

The following example illustrates the use of the reduction rule and the matching operation:

$$\begin{aligned}
&(\mathtt{true} \to 1 \mid \mathtt{false} \to 0)\,((\mathtt{true} \to \mathtt{false} \mid \mathtt{false} \to \mathtt{true})\,\mathtt{true}) \\
&\to (\mathtt{true} \to 1 \mid \mathtt{false} \to 0)\,\{\!\{\mathtt{true}/\mathtt{true}\}\!\}\,\mathtt{false} \\
&= (\mathtt{true} \to 1 \mid \mathtt{false} \to 0)\,\mathtt{false} \\
&\to \{\!\{\mathtt{false}/\mathtt{false}\}\!\}\,0 \\
&= 0
\end{aligned} \tag{5}$$

Note that in $(\mathtt{true} \to 1 \mid \mathtt{false} \to 0)\,\mathtt{false}$, the second branch is selected since $\{\!\{\mathtt{false}/\mathtt{true}\}\!\} = \mathtt{fail}$.

Proposition 2.2 *Reduction in CAP is confluent (CR).*

This result follows from a straightforward application of the CR proof technique presented in [19] to our calculus. The key step is proving that the matching operation satisfies the *Rigid Matching Condition (RMC)* proposed in the cited work. Our contribution is on the typed variant of the calculus.

3 Typing System

This section presents μ-types, the finite type expressions that shall be used for typing terms in CAP, their associated notions of equivalence and subtyping and then the typing schemes. Also, further examples and definitions associated to compatibility are included.

3.1 Types

In order to ensure that patterns such as $x\,y$ decompose only data structures rather than arbitrary terms, we shall introduce two sorts of typing expressions: *types* and *datatypes*, the latter being strictly included in the former.

We assume given countably infinite sets $\mathcal{V}_D$ of **datatype variables** $(\alpha, \beta, \ldots)$, $\mathcal{V}_A$ of **type variables** $(X, Y, \ldots)$ and $\mathcal{C}$ of **type constants** $(\mathbb{c}, \mathbb{d}, \ldots)$. We define $\mathcal{V} \triangleq \mathcal{V}_A \cup \mathcal{V}_D$ and use metavariables $V, W, \ldots$ to denote an arbitrary element in it. Likewise, we write $a, b, \ldots$ for elements in $\mathcal{V} \cup \mathcal{C}$. The sets $\mathcal{T}_D$ of **μ-datatypes** and

$$\frac{}{\vdash A \oplus A \simeq_\mu A}\ (\text{E-UNION-IDEM}) \qquad \frac{}{\vdash A \oplus B \simeq_\mu B \oplus A}\ (\text{E-UNION-COMM})$$

$$\frac{}{\vdash A \oplus (B \oplus C) \simeq_\mu (A \oplus B) \oplus C}\ (\text{E-UNION-ASSOC})$$

$$\frac{}{\vdash \mu V.A \simeq_\mu \{\mu V.A/V\}\, A}\ (\text{E-FOLD})$$

$$\frac{\vdash A \simeq_\mu \{A/V\}\, B \quad \mu V.B \text{ contractive}}{\vdash A \simeq_\mu \mu V.B}\ (\text{E-CONTR})$$

Fig. 1. Type equivalence for μ-types (sample)

$\mathcal{T}$ of **μ-types**, resp., are inductively defined as follows:

$D ::=$	α	(datatype variable)	$A ::=$	X	(type variable)
$\mid$	$\mathbb{c}$	(atom)	$\mid$	D	(datatype)
$\mid$	$D \mathbin{@} A$	(compound)	$\mid$	$A \supset A$	(type abstraction)
$\mid$	$D \oplus D$	(union)	$\mid$	$A \oplus A$	(union)
$\mid$	$\mu\alpha.D$	(recursion)	$\mid$	$\mu X.A$	(recursion)

Remark 3.1 A type of the form $\mu\alpha.A$ is excluded since it may produce invalid unfoldings. For example, $\mu\alpha.\alpha \supset \alpha = (\mu\alpha.\alpha \supset \alpha) \supset (\mu\alpha.\alpha \supset \alpha)$, since α is a datatype variable and type abstraction is not a datatype. On the other hand, types of the form $\mu X.D$ are not necessary since they denote the solution to the equation $X = D$, hence X is a variable representing a datatype.

We consider $\oplus$ to bind tighter than $\supset$, while @ binds tighter than $\oplus$. Therefore $D \mathbin{@} A \oplus A' \supset B$ means $((D \mathbin{@} A) \oplus A') \supset B$. Additionally, when referring to a finite series of consecutive unions such as $A_1 \oplus \ldots \oplus A_n$ we will use the simplified notation $\bigoplus_{i \in 1..n} A_i$. This notation is not strict on how subexpressions A_i are associated hence, in principle, it refers to any of all possible associations. In the next section we present an equivalence relation on μ-types that will identify all these associations. We often write $\mu V.A$ to mean either $\mu\alpha.D$ or $\mu X.A$. A **non-union μ-type** A is a μ-type of one of the following forms: α, $\mathbb{c}$, $D \mathbin{@} A$, X, $A \supset B$ or $\mu V.A$ with A a non-union μ-type. We assume μ-types are **contractive**: $\mu V.A$ is contractive if V occurs in A only under a type constructor $\supset$ or @, if at all. We henceforth redefine $\mathcal{T}$ to be the set of **contractive μ-types**. μ-types come equipped with a notion of equivalence $\simeq_\mu$ and subtyping $\preceq_\mu$.

Definition 3.2 (i) $\simeq_\mu$ is the least congruence closed under the schemes in Fig. 1.

(ii) $\preceq_\mu$ is defined in Fig. 2 where a subtyping context Σ is a set of assumptions over type variables of the form $V \preceq_\mu W$ with $V, W \in \mathcal{V}$.

(E-CONTR) actually encodes two rules, one for datatypes ($\mu\alpha.D$) and one for

$$\frac{}{\Sigma \vdash A \preceq_\mu A}\ (\textsc{s-refl}) \qquad \frac{}{\Sigma, V \preceq_\mu W \vdash V \preceq_\mu W}\ (\textsc{s-hyp}) \qquad \frac{\vdash A \simeq_\mu B}{\Sigma \vdash A \preceq_\mu B}\ (\textsc{s-eq})$$

$$\frac{\Sigma \vdash A \preceq_\mu B \quad \Sigma \vdash B \preceq_\mu C}{\Sigma \vdash A \preceq_\mu C}\ (\textsc{s-trans}) \qquad \frac{\Sigma \vdash D \preceq_\mu D' \quad \Sigma \vdash A \preceq_\mu A'}{\Sigma \vdash D \mathbin{@} A \preceq_\mu D' \mathbin{@} A'}\ (\textsc{s-comp})$$

$$\frac{\Sigma \vdash A \preceq_\mu A' \quad \Sigma \vdash B \preceq_\mu B'}{\Sigma \vdash A' \supset B \preceq_\mu A \supset B'}\ (\textsc{s-func}) \qquad \frac{\Sigma \vdash A \preceq_\mu C \quad \Sigma \vdash B \preceq_\mu C}{\Sigma \vdash A \oplus B \preceq_\mu C}\ (\textsc{s-union-l})$$

$$\frac{\Sigma \vdash A \preceq_\mu B}{\Sigma \vdash A \preceq_\mu B \oplus C}\ (\textsc{s-union-r1}) \qquad \frac{\Sigma \vdash A \preceq_\mu C}{\Sigma \vdash A \preceq_\mu B \oplus C}\ (\textsc{s-union-r2})$$

$$\frac{\Sigma, V \preceq_\mu W \vdash A \preceq_\mu B \quad W \notin \mathsf{fv}(A) \quad V \notin \mathsf{fv}(B)}{\Sigma \vdash \mu V.A \preceq_\mu \mu W.B}\ (\textsc{s-rec})$$

Fig. 2. Strong subtyping for μ-types (Sample)

arbitrary types ($\mu X.A$). Likewise for (E-FOLD). The relation resulting from dropping (E-CONTR) [3, 6] is called weak type equivalence [8] and is known to be too weak to capture equivalence of its coinductive formulation (required for our proof of invertibility of subtyping *cf.* Prop. 3.12); for example, types $\mu X.A \supset A \supset X$ and $\mu X.A \supset X$ cannot be equated. We can now use notation $\bigoplus_{i \in 1..n} A_i$ on contractive μ-types to denote several consecutive applications of the binary operator $\oplus$ irrespective of how they are associated. All such associations yield equivalent μ-types. Regarding the subtyping rules, we adopt those for union of [28]. It should be noted that the naïve variant of (S-REC) in which $\Sigma \vdash \mu V.A \preceq_\mu \mu V.B$ is deduced from $\Sigma \vdash A \preceq_\mu B$, is known to be unsound [1]. We often abbreviate $\vdash A \preceq_\mu B$ as $A \preceq_\mu B$.

3.2 Typing Schemes

A **typing context** Γ (or θ) is a partial function from term variables to μ-types; $\Gamma(x) = A$ means that Γ maps x to A. We have two typing judgments, one for patterns $\theta \vdash_{\mathsf{p}} p : A$ and one for terms $\Gamma \vdash s : A$. Accordingly, we have two sets of typing rules: Fig. 3, top and bottom. We write $\theta \rhd_{\mathsf{p}} p : A$ to indicate that the typing judgment $\theta \vdash_{\mathsf{p}} p : A$ is derivable (likewise for $\Gamma \rhd s : A$). The typing schemes speak for themselves except for two of them which we now comment. The first is (T-APP). Note that we do not require the A_i to be non-union types. This allows examples such as (5) to be typable (the outermost instance of (T-APP) is with $n = 1$ and $A_1 = \mathsf{Bool} = \mathtt{true} \oplus \mathtt{false}$). Regarding (T-ABS) it requests a number of conditions. First of all, each of the patterns p_i must be typable under the typing context θ_i, $i \in 1..n$. Also, the set of free matchables in each p_i must be exactly the domain

Patterns

$$\frac{\theta(x) = A}{\theta \vdash_{\mathsf{p}} x : A}\ (\text{P-MATCH}) \qquad \frac{}{\theta \vdash_{\mathsf{p}} \mathsf{c} : \mathbb{c}}\ (\text{P-CONST}) \qquad \frac{\theta \vdash_{\mathsf{p}} p : D \quad \theta \vdash_{\mathsf{p}} q : A}{\theta \vdash_{\mathsf{p}} p\, q : D \mathbin{@} A}\ (\text{P-COMP})$$

Terms

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ (\text{T-VAR}) \qquad \frac{}{\Gamma \vdash \mathsf{c} : \mathbb{c}}\ (\text{T-CONST}) \qquad \frac{\Gamma \vdash r : D \quad \Gamma \vdash u : A}{\Gamma \vdash r\, u : D \mathbin{@} A}\ (\text{T-COMP})$$

$$\frac{[p_i : A_i]_{i \in 1..n} \text{ compatible} \quad (\theta_i \vdash_{\mathsf{p}} p_i : A_i)_{i \in 1..n} \quad (\mathsf{dom}\,(\theta_i) = \mathsf{fm}(p_i))_{i \in 1..n} \quad (\Gamma, \theta_i \vdash s_i : B)_{i \in 1..n}}{\Gamma \vdash (p_i \to_{\theta_i} s_i)_{i \in 1..n} : \bigoplus_{i \in 1..n} A_i \supset B}\ (\text{T-ABS})$$

$$\frac{\Gamma \vdash r : \bigoplus_{i \in 1..n} A_i \supset B \quad \Gamma \vdash u : A_k \quad k \in 1..n}{\Gamma \vdash r\, u : B}\ (\text{T-APP})$$

$$\frac{\Gamma \vdash s : A \quad \vdash A \preceq_{\mu} A'}{\Gamma \vdash s : A'}\ (\text{T-SUBS})$$

Fig. 3. Typing rules for patterns and terms

of θ_i. Another condition, indicated by $(\Gamma, \theta_i \vdash s_i : B)_{i \in 1..n}$, is that the bodies of each of the branches s_i, $i \in 1..n$, be typable under the context extended with the corresponding θ_i. More noteworthy is the condition that the list $[p_i : A_i]_{i \in 1..n}$ be *compatible*, which we now discuss in further detail.

3.3 Compatibility

Let us say that a pattern p **subsumes** a pattern q, written $p \lhd q$ if there exists a substitution σ s.t. $\sigma p = q$. Consider an abstraction $(p \to_{\theta} s \mid q \to_{\theta'} t)$ and two judgments $\theta \vdash_{\mathsf{p}} p : A$ and $\theta' \vdash_{\mathsf{p}} q : B$. We consider two cases depending on whether p subsumes q or not.

As already mentioned in example (3) of the introduction, if p subsumes q, then the branch $q \to_{\theta'} t$ will never be evaluated since the argument will already match p. Indeed, for any term u of type B in matchable form, the application will reduce to $\{\!\{u/p\}\!\}\, s$. Thus, in this case, in order to ensure SR we demand that $B \preceq_{\mu} A$.

Suppose p does not subsume q (*i.e.* $p \not\lhd q$). We analyze the cause of failure of subsumption in order to determine whether requirements on A and B must be put forward. In some cases no requirements are necessary. For example in:

$$\begin{aligned} f \to_{\{f : A \supset B\}} \ & (\ \mathtt{vl}\, z \ \to_{\{z : A\}} \ \mathtt{vl}\,(f\, z) \\ & \mid \mathtt{vl}'\, y \to_{\{y : B\}} \mathtt{vl}'\, y) \end{aligned} \tag{6}$$

no relation between A and B is required since the branches are mutually disjoint. In other cases, however, $B \preceq_\mu A$ is required; we seek to characterize them. We focus on those cases where p fails to subsume q, and $\pi \in \mathsf{pos}(p) \cap \mathsf{pos}(q)$ is an offending position in both patterns. The following table exhaustively lists them:

	$p\|_\pi$	$q\|_\pi$	
(a)		y	restriction required
(b)	c	d	no overlapping ($q \not\lhd p$)
(c)		$q_1\, q_2$	no overlapping
(d)		y	restriction required
(e)	$p_1\, p_2$	d	no overlapping

In cases (b), (c) and (e), no extra condition on the types of p and q is necessary either, since their respective sets of possible arguments are disjoint; example (6) corresponds to the first of these. The cases where A and B must be related are (a) and (d): for those we require $B \preceq_\mu A$. The first of these has already been illustrated in the introduction (3), the second one is illustrated as follows:

$$f \to_{\{f:D\supset A\supset C\}} g \to_{\{g:B\supset C\}} \begin{array}{ll} (\, x\, y & \to_{\{x:D,y:A\}} \; f\, x\, y \\ \mid z & \to_{\{z:B\}} \quad g\, z) \end{array} \tag{7}$$

The problematic situation is when $B = D' \mathbin{@} B'$, *i.e.* the type of z is another compound, which may have no relation at all with $D \mathbin{@} A$. Compatibility ensures $B \preceq_\mu D \mathbin{@} A$.

We now formalize these ideas.

Definition 3.3 Given a pattern $\theta \vdash_{\mathsf{p}} p : A$ and $\pi \in \mathsf{pos}(p)$, we say A *admits a symbol* $\odot$ (with $\odot \in \mathcal{V} \cup \mathcal{C} \cup \{\supset, @\}$) *at position* π iff $\odot \in A\|_\pi$, where:

$$\begin{array}{rll} a\|_\epsilon \triangleq \{a\} & & \\ (A_1 \star A_2)\|_\epsilon \triangleq \{\star\}\,, & & \star \in \{\supset, @\} \\ (A_1 \star A_2)\|_{i\pi} \triangleq A_i\|_\pi, & & \star \in \{\supset, @\}\,, i \in \{1,2\} \\ (A_1 \oplus A_2)\|_\pi \triangleq A_1\|_\pi \cup A_2\|_\pi & & \\ (\mu V.A')\|_\pi \triangleq (\{\mu V.A'/V\}\, A')\|_\pi & & \end{array}$$

Note that $\theta \rhd_{\mathsf{p}} p : A$ and contractiveness of A, implies $A\|_\pi$ is well-defined for $\pi \in \mathsf{pos}(p)$.

Whenever subsumption between two patterns fails, any mismatching position is a leaf in the syntactic tree of one of the patterns. Otherwise, both of them would have a type application constructor in that position and there would be no failure of subsumption.

Definition 3.4 The *maximal positions* in a set of positions P are:

$$\mathsf{maxpos}(P) \triangleq \left\{\pi \in P \mid \nexists \pi' \in P.\pi' = \pi\pi'' \wedge \pi'' \neq \epsilon\right\}$$

The *mismatching positions* between two patterns are:

$$\mathsf{mmpos}(p,q) \triangleq \{\pi \mid \pi \in \mathsf{maxpos}(\mathsf{pos}(p) \cap \mathsf{pos}(q)) \wedge p|_\pi \not\lhd q|_\pi\}$$

Definition 3.5 We say $p : A$ is *compatible* with $q : B$, written $p : A \lll q : B$, iff the following two conditions hold:

(i) $p \lhd q \implies B \preceq_\mu A$.

(ii) $p \not\lhd q \implies (\forall \pi \in \mathsf{mmpos}(p,q)\,.A\|_\pi \cap B\|_\pi \neq \varnothing) \implies B \preceq_\mu A$.

A list of patterns $[p_i : A_i]_{i\in 1..n}$ is compatible if $\forall i,j \in 1..n.i < j \implies p_i : A_i \lll p_j : A_j$.

As an example, recall the function $\mathtt{upd}$ (*cf.* (1)). Consider the first pattern of its inner abstraction, namely $\mathtt{vl}\,z$, and the second one, namely $x\,y$. Given that $\mathtt{vl}\,z \not\lhd x\,y$, in order to determine whether the former is compatible with the latter we need to check item (ii) of Def. 3.5. Since $\mathsf{mmpos}(\mathtt{vl}\,z, x\,y) = \{1\}$, we analyze $(\mathbb{vl} \mathbin{@} A)\|_1 \cap (C \mathbin{@} D)\|_1$. Note that $(\mathbb{vl} \mathbin{@} A)\|_1 = \{\mathbb{vl}\}$ and thus, it is enough to check whether $\mathbb{vl} \in (C \mathbin{@} D)\|_1 = C\|_\epsilon$ or not, to know if the restriction $C \mathbin{@} D \preceq_\mu \mathbb{vl} \mathbin{@} A$ must be imposed. Since $C = \mu\alpha.(\mathbb{vl} \mathbin{@} A) \oplus (\alpha \mathbin{@} \alpha) \oplus (\mathbb{cons} \oplus \mathbb{node} \oplus \mathbb{nil})$, we deduce $C\|_\epsilon = \{@, \mathbb{cons}, \mathbb{node}, \mathbb{nil}\}$. Thus, $(\mathbb{vl} \mathbin{@} A)\|_1 \cap (C \mathbin{@} D)\|_1 = \emptyset$ and item (ii) holds trivially.

As a further example, suppose we wish to apply $\mathtt{upd}$ to data structures holding values of different types: say $\mathtt{vl}$ prefixed values are numbers and $\mathtt{vl}'$ prefixed values are functions over numbers. Note that $\mathtt{upd}$ cannot be typed as it stands. The reason is that the last branch would have to handle values of functional type and hence would receive type $\mathbb{cons} \oplus \mathbb{node} \oplus \mathbb{nil} \oplus \mathbb{vl}' \oplus (\mathsf{Nat} \supset \mathsf{Nat})$. This fails to be a datatype due to the presence of the component of functional type. As a consequence, $x\,y$ cannot be typed since it requires an applicative type @. The remedy is to add an additional branch to $\mathtt{upd}$ capable of handling values prefixed by $\mathtt{vl}'$:

$$\begin{array}{l} \mathtt{upd}' = f \to_{\{f:\mathsf{Nat}\supset B_1\}} g \to_{\{g:(\mathsf{Nat}\supset\mathsf{Nat})\supset B_2\}} \\ \qquad \begin{array}{lll} (\ \mathtt{vl}\,z & \to_{\{z:\mathsf{Nat}\}} & \mathtt{vl}\,(f\,z) \\ \mid \mathtt{vl}'\,z & \to_{\{z:\mathsf{Nat}\supset\mathsf{Nat}\}} & \mathtt{vl}'\,(g\,z) \\ \mid x\,y & \to_{\{x:C,y:D\}} & (\mathtt{upd}'\,f\,x)\,(\mathtt{upd}'\,f\,y) \\ \mid w & \to_{\{w:E\}} & w) \end{array} \end{array} \tag{8}$$

The type of $\mathtt{upd}'$ is $(\mathsf{Nat} \supset B_1) \supset ((\mathsf{Nat} \supset \mathsf{Nat}) \supset B_2) \supset (F_{\mathsf{Nat},\mathsf{Nat}\supset\mathsf{Nat}} \supset F_{B_1,B_2})$, where $F_{X,Y}$ is

$$\mu\alpha.(\mathbb{vl} \mathbin{@} X) \oplus (\mathbb{vl}' \mathbin{@} Y) \oplus (\alpha \mathbin{@} \alpha) \oplus (\mathbb{cons} \oplus \mathbb{node} \oplus \mathbb{nil})$$

This is quite natural: the type system establishes a clear distinction between semi-structured data, susceptible to path polymorphism, and "unstructured" data represented here by base and functional types.

3.4 Basic Metatheory of Typing

We present some technical lemmas that will be useful in the proof of safety.

Lemma 3.6 (Generation Lemma) *Let Γ be a typing context and A a type.*

(i) If $\Gamma \rhd x : A$ then $\exists A'$ s.t. $A' \preceq_\mu A$ and $x : A' \in \Gamma$.

(ii) If $\Gamma \rhd \mathsf{c} : A$ then $\mathbb{c} \preceq_\mu A$.

(iii) If $\Gamma \rhd r\, u : A$ then:
(a) either $\exists D, A'$ s.t. $D \mathbin{@} A' \preceq_\mu A$, $\Gamma \rhd r : D$ and $\Gamma \rhd u : A'$;
(b) or $\exists A_1, \ldots, A_n, A', k \in 1..n$ s.t. $A' \preceq_\mu A$, $\Gamma \rhd r : \bigoplus_{i \in 1..n} A_i \supset A'$, and $\Gamma \rhd u : A_k$.

(iv) If $\Gamma \rhd (p_i \to_{\theta_i} s_i)_{i \in 1..n} : A$ then $\exists A_1, \ldots, A_n, B$ s.t. $\bigoplus_{i \in 1..n} A_i \supset B \preceq_\mu A$, $[p_i : A_i]_{i \in 1..n}$ is compatible, $\mathsf{dom}(\theta_i) = \mathsf{fm}(p_i)$, $\theta_i \rhd_{\mathsf{p}} p_i : A_i$ and $\Gamma, \theta_i \rhd s_i : B$ for every $i \in 1..n$.

The following lemma is useful to deduce the shape of the type when we know the term is a data structure. Essentially it states that every data structure that can be given a type, can also be typed with a more specific non-union datatype.

Lemma 3.7 (Typing for Data Structures) *Suppose $\Gamma \rhd d : A$, for d a data structure. Then $\exists D$ datatype such that D is a non-union type, $D \preceq_\mu A$ and $\Gamma \rhd d : D$. Moreover,*

(i) *If $d = \mathsf{c}$, then $D \simeq_\mu \mathbb{c}$.*

(ii) *If $d = d'\, t$, then $\exists D', A'$ such that $D \simeq_\mu D' \mathbin{@} A'$, $\Gamma \rhd d' : D'$ and $\Gamma \rhd t : A'$.*

Some results on compatibility follow, the crucial one being Lem. 3.9. This next lemma shows that matching failure is enough to guarantee that the type of the argument is not a subtype of that of the pattern.

Lemma 3.8 *Given $\Gamma \rhd u : B$, $\theta \rhd_{\mathsf{p}} p : A$. If $\{\!\{u/p\}\!\} = \mathtt{fail}$, then $B \not\preceq_\mu A$.*

Define $\mathcal{P}_{\mathsf{comp}}(p : A, q : B) \triangleq \forall \pi \in \mathsf{mmpos}(p, q)\,.A\|_\pi \cap B\|_\pi \neq \varnothing$, so that compatibility can alternatively be characterized as:

$$p : A \lll q : B \quad \text{iff} \quad \mathcal{P}_{\mathsf{comp}}(p : A, q : B) \implies B \preceq_\mu A$$

The Compatibility Lemma should be interpreted in the context of an abstraction. Assume an argument u of type B is passed to a function where there are (at least) two branches, defined by patterns p and q, the latter having the same type as u. If the argument matches the first pattern of (potentially) a different type A, then $\mathcal{P}_{\mathsf{comp}}(p : A, q : B)$ must hold. Since patterns in a well-typed abstraction are compatible, whenever p comes before q we get $B \preceq_\mu A$, and thus $\Gamma \rhd u : A$ too.

Lemma 3.9 (Compatibility Lemma) *Suppose $\Gamma \rhd u : B$, $\theta \rhd_{\mathsf{p}} p : A$, $\theta' \rhd_{\mathsf{p}} q : B$ and $\{\!\{u/p\}\!\}$ is successful. Then, $\mathcal{P}_{\mathsf{comp}}(p : A, q : B)$ holds.*

We write $\Gamma \vdash \sigma : \theta$ to indicate that $\mathsf{dom}(\sigma) = \mathsf{dom}(\theta)$ and $\Gamma \vdash \sigma(x) : \theta(x)$, for all $x \in \mathsf{dom}(\sigma)$.

The following lemma assures that the substitution yielded by a successful match preserves the types of the variables in the pattern.

Lemma 3.10 (Type of Successful Match) *Suppose* $\{\!\{u/p\}\!\} = \sigma$ *is successful,* $\mathsf{dom}\,(\theta) = \mathsf{fm}(p)$, $\theta \rhd_{\mathsf{p}} p : A$ *and* $\Gamma \rhd u : A$. *Then* $\Gamma \rhd \sigma : \theta$.

Finally, we have the standard Substitution Lemma.

Lemma 3.11 (Substitution Lemma) *Suppose* $\Gamma, \theta \rhd s : A$ *and* $\Gamma \rhd \sigma : \theta$. *Then* $\Gamma \rhd \sigma s : A$.

Type safety, addressed in the next section, also relies on $\preceq_\mu$ enjoying the fundamental property of *invertibility* of non-union types:

Proposition 3.12 (i) *If* $D \mathbin{@} A \preceq_\mu D' \mathbin{@} A'$, *then* $D \preceq_\mu D'$ *and* $A \preceq_\mu A'$.
(ii) *If* $A \supset B \preceq_\mu A' \supset B'$, *then* $A' \preceq_\mu A$ *and* $B \preceq_\mu B'$.

To prove this we appeal to the standard tree interpretation of terms and formulate an equivalent coinductive definition of equivalence and subtyping.

For the latter, invertibility of non-union types is proved coinductively, entailing Prop. 3.12 (*cf.* [27]).

4 Safety

Subject Reduction (Prop. 4.1) and Progress (Prop. 4.2) are addressed next.

Proposition 4.1 (Subject Reduction) *If* $\Gamma \rhd s : A$ *and* $s \to s'$, *then* $\Gamma \rhd s' : A$.

Proof. By induction on s. The non-trivial case is when $s = (p_i \to_{\theta_i} s_i)_{i \in 1..n}\, u$ and $s' = \{\!\{u/p_k\}\!\} s_k$ for some $k \in 1..n$ such that $\{\!\{u/p_k\}\!\} = \sigma$ and $\{\!\{u/p_i\}\!\} = \mathtt{fail}$ for every $i < k$. By Generation Lemma (iii.b), there exists $C_1, \ldots, C_m, A'$ such that $A' \preceq_\mu A$, $\Gamma \rhd (p_i \to_{\theta_i} s_i)_{i \in 1..n} : \bigoplus_{j \in 1..m} C_m \supset A'$ and:

$$\Gamma \rhd u : C_{k'} \tag{9}$$

for some $k' \in 1..m$. Applying once again the Generation Lemma, item (iv) this time, to $\Gamma \rhd (p_i \to_{\theta_i} s_i)_{i \in 1..n} : \bigoplus_{j \in 1..m} C_m \supset A'$, we get $\exists A_1, \ldots, A_n, B$ such that:

$$\bigoplus_{i \in 1..n} A_i \supset B \preceq_\mu \bigoplus_{j \in 1..m} C_m \supset A' \tag{10}$$

$\mathsf{dom}\,(\theta_i) = \mathsf{fm}(p_i)$, $[p_i : A_i]_{i \in 1..n}$ is compatible, $\theta_i \rhd_{\mathsf{p}} p_i : A_i$ and $\Gamma, \theta_i \rhd s_i : B$ for every $i \in 1..n$.

From (10), by invertibility of subtyping for non-union types, we have $B \preceq_\mu A'$ and

$$\bigoplus_{j \in 1..m} C_m \preceq_\mu \bigoplus_{i \in 1..n} A_i \tag{11}$$

We want to show that $\Gamma \rhd u : A_k$. For that we need to distinguish two cases:

(i) If u is in matchable form, we have two possibilities:
  (a) u is a data structure: then, by the Typing for Data Structures lemma, there exists a non-union datatype D such that $D \preceq_\mu C_{k'}$ and $\Gamma \rhd u : D$.
  (b) u is an abstraction: then, by Generation Lemma (iv), there exists types C', C'' such that $C' \supset C'' \preceq_\mu C_{k'}$ and $\Gamma \rhd u : C' \supset C''$.

Then, in both cases there exists a non-union type, say C, such that $C \preceq_\mu C_{k'}$ and $\Gamma \rhd u : C$. Then, from (11) we get:

$$C \preceq_\mu \bigoplus_{i \in 1..n} A_i$$

and, since C is non-union, $C \preceq_\mu A_l$ for some $l \in 1..n$. Hence, by subsumption $\Gamma \rhd u : A_l$.

If $k = l$ we are done, so assume $k \neq l$. Recall the conditions for the reduction rule, where $\{\!\{u/p_i\}\!\} = \texttt{fail}$ for every $i < k$. Then, by Lem. 3.8, we have $A_l \not\preceq_\mu A_i$. Thus, it must be the case that $k < l$. By Lem. 3.9 with hypothesis $\Gamma \rhd u : A_l$, $\theta_k \rhd_{\mathsf{p}} p_k : A_k$, $\theta_l \rhd_{\mathsf{p}} p_l : A_l$ and $\{\!\{u/p_k\}\!\} = \sigma$ we get that $\mathcal{P}_{\mathsf{comp}}(p_k : A_k, p_l : A_l)$ holds. Additionally, we already saw that the list $[p_i : A_i]_{i \in 1..n}$ is compatible, thus $p_k : A_k \lll p_l : A_l$ and by definition $A_l \preceq_\mu A_k$. Finally we conclude by subsumption once again, $\Gamma \rhd u : A_k$.

(ii) If u is not in matchable form, then $p_k = x$ and by the premises of the reductions rule we need $\{\!\{u/p_i\}\!\} = \texttt{fail}$ for every $i < k$. Thus, necessarily $k = 1$. Moreover, since $x \lhd p_i$ for every $i \in 1..n$, by compatibility we have $A_i \preceq_\mu A_k$. Then, from (11) we get

$$C_{k'} \preceq_\mu \bigoplus_{j \in 1..m} C_j \preceq_\mu \bigoplus_{i \in 1..n} A_i \preceq_\mu A_k$$

Thus, by subsumption, $\Gamma \rhd u : A_k$.

Finally, in either case we have $\Gamma \rhd u : A_k$. Now Lem. 3.10 and 3.11 with $\Gamma, \theta_k \rhd s_k : B$ entails $\Gamma \rhd s' : B$ and we conclude by subsumption, $\Gamma \rhd s' : A$ (recall $B \preceq_\mu A' \preceq_\mu A$). □

Let the set of **values** be defined as $v ::= x\, v_1 \ldots v_n \mid \mathsf{c}\, v_1 \ldots v_n \mid (p_i \to_{\theta_i} s_i)_{i \in 1..n}$. The following auxiliary property guarantees the success of matching for well-typed closed values (note that values are already in matchable form).

Proposition 4.2 (Progress) *If $\rhd s : A$ and s is not a value, then $\exists s'$ s.t. $s \to s'$.*

The proof is by induction on the term analyzing those subterms that can still be reduced to a value. Full details are available in the complete report [27].

5 Conclusions

A type system is proposed for a calculus that supports path polymorphism and two fundamental properties are addressed, namely Subject Reduction and Progress. The type system includes type application, constants as types, union and recursive types. Both properties rely crucially on a notion of pattern *compatibility* and on invertibility of subtyping of μ-types. This last result is proved via a coinductive semantics for the finite μ-types. Regarding future work an outline of possible avenues follows.

- A syntax directed, alternative formulation of the system has been developed [16]. Based on this, a type-checking algorithm for CAP is defined and implemented (also in [16]). By adapting extant techniques [15,20,21,23] we are

able to produce efficient equivalence and subtype checking algorithms for the relations presented in this article.

- We already mentioned the addition of parametric polymorphism (presumably in the style of $F_{<:}$ [7,14,25]). We believe this should not present major difficulties.
- Strong normalization requires devising a notion of positive/negative occurrence in the presence of strong μ-type equality, which is known not to be obvious [4, page 515].
- A more ambitious extension is that of *dynamic patterns*, namely patterns that may be computed at run-time, PPC being the prime example of a calculus supporting this feature.

References

[1] Amadio, R. M. and L. Cardelli, *Subtyping recursive types*, ACM Trans. Program. Lang. Syst. **15** (1993), pp. 575–631.

[2] Arbiser, A., A. Miquel and A. Ríos, *The lambda-calculus with constructors: Syntax, confluence and separation*, J. Funct. Program. **19** (2009), pp. 581–631.

[3] Ariola, Z. M. and J. W. Klop, *Equational term graph rewriting*, Fundam. Inform. **26** (1996), pp. 207–240.

[4] Barendregt, H., W. Dekkers and R. Statman, editors, "Lambda Calculus with Types," Perspectives in Logic, CUP, 2013.

[5] Barthe, G., H. Cirstea, C. Kirchner and L. Liquori, *Pure patterns type systems*, in: A. Aiken and G. Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003* (2003), pp. 250–261.
URL http://doi.acm.org/10.1145/640128.604152

[6] Brandt, M. and F. Henglein, *Coinductive axiomatization of recursive type equality and subtyping*, Fundam. Inform. **33** (1998), pp. 309–338.

[7] Cardelli, L., S. Martini, J. C. Mitchell and A. Scedrov, *An extension of System F with subtyping*, in: T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, LNCS **526**, Springer Berlin Heidelberg, 1991 pp. 750–770.
URL http://dx.doi.org/10.1007/3-540-54415-1_73

[8] Cardone, F., *An algebraic approach to the interpretation of recursive types*, in: J.-C. Raoult, editor, *CAAP*, LNCS **581** (1992), pp. 66–85.

[9] Cirstea, H. and C. Kirchner, *The rewriting calculus - part i and ii*, Logic Journal of the IGPL **9** (2001), pp. 339–410.

[10] Cirstea, H., C. Kirchner and L. Liquori, *Matching power*, in: A. Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, LNCS **2051** (2001), pp. 77–92.
URL http://dx.doi.org/10.1007/3-540-45127-7_8

[11] Cirstea, H., C. Kirchner and L. Liquori, *The rho cube*, in: F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 (ETAPS 2001), Genova, Italy, April 2-6, 2001, Proceedings*, LNCS **2030** (2001), pp. 168–183.
URL http://dx.doi.org/10.1007/3-540-45315-6_11

[12] Cirstea, H., C. Kirchner and L. Liquori, *Rewriting calculus with(out) types*, Electr. Notes Theor. Comput. Sci. **71** (2002), pp. 3–19.
URL http://dx.doi.org/10.1016/S1571-0661(05)82526-5

[13] Cirstea, H., L. Liquori and B. Wack, *Rewriting calculus with fixpoints: Untyped and first-order systems*, in: S. Berardi, M. Coppo and F. Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, LNCS **3085** (2003), pp. 147–161.
URL http://dx.doi.org/10.1007/978-3-540-24849-1_10

[14] Colazzo, D. and G. Ghelli, *Subtyping recursion and parametric polymorphism in kernel fun*, Information and Computation **198** (2005), pp. 71–147.
URL http://www.sciencedirect.com/science/article/pii/S0890540105000167

[15] Di Cosmo, R., F. Pottier and D. Rémy, *Subtyping recursive types modulo associative commutative products*, in: *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*, TLCA'05 (2005), pp. 179–193.
URL http://dx.doi.org/10.1007/11417170_14

[16] Edi, J., "Chequeo de tipos eficiente para Path Polymorphism," Master's thesis, Universidad de Buenos Aires, Buenos Aires, Argentina (2015).

[17] Jay, C. B., "Pattern Calculus: Computing with Functions and Structures," Springer, 2009.

[18] Jay, C. B. and D. Kesner, *Pure pattern calculus*, in: P. Sestoft, editor, *ESOP*, LNCS **3924** (2006), pp. 100–114.

[19] Jay, C. B. and D. Kesner, *First-class patterns*, J. Funct. Program. **19** (2009), pp. 191–225.

[20] Jha, S., J. Palsberg and T. Zhao, *Efficient type matching*, in: M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002 (ETAPS 2002) Grenoble, France, April 8-12, 2002, Proceedings*, LNCS **2303** (2002), pp. 187–204.
URL http://dx.doi.org/10.1007/3-540-45931-6_14

[21] Kozen, D., J. Palsberg and M. I. Schwartzbach, *Efficient recursive subtyping*, Mathematical Structures in Computer Science **5** (1995), pp. 113–125.
URL http://dx.doi.org/10.1017/S0960129500000657

[22] Liquori, L. and B. Wack, *The polymorphic rewriting-calculus: [type checking vs. type inference]*, Electr. Notes Theor. Comput. Sci. **117** (2005), pp. 89–111.
URL http://dx.doi.org/10.1016/j.entcs.2004.06.027

[23] Palsberg, J. and T. Zhao, *Efficient and flexible matching of recursive types*, Inf. Comput. **171** (2001), pp. 364–387.
URL http://dx.doi.org/10.1006/inco.2001.3090

[24] Petit, B., *Semantics of typed lambda-calculus with constructors*, Logical Methods in Computer Science **7** (2011).

[25] Pierce, B. C., "Types and Programming Languages," MIT Press, Cambridge, MA, USA, 2002.

[26] ten Eikelder, H., *Some algorithms to decide the equivalence of recursive types*, Technical Report 91/31, Eindhoven University of Technology (1991).

[27] Viso, A., E. Bonelli and M. Ayala-Rincón, *Type soundness for path polymorphism*, Extended report (2015), http://arxiv.org/abs/1601.03271.

[28] Vouillon, J., *Subtyping union types*, in: J. Marcinkowski and A. Tarlecki, editors, *CSL*, LNCS **3210** (2004), pp. 415–429.