Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Typed path polymorphism ☆

Mauricio Ayala-Rincón [a,*], Eduardo Bonelli [b,c,e,*], Juan Edi [d,*], Andrés Viso [d,e,*]

[a] Universidade de Brasília, Brazil
[b] Universidad Nacional de Quilmes, Argentina
[c] Stevens Institute of Technology, United States of America
[d] Universidad de Buenos Aires, Argentina
[e] CONICET, Argentina

ARTICLE INFO

Article history:
Received 15 December 2017
Received in revised form 30 January 2019
Accepted 23 February 2019
Available online xxxx

Keywords:
λ-calculus
Pattern matching
Path polymorphism
Static typing
Type checking

ABSTRACT

Path polymorphism enables the definition of functions uniformly applicable to arbitrary recursively specified data structures. Path polymorphic function declarations rely on *patterns* of the form $x\,y$ (*i.e.* the application of two variables), which decompose a data structure into its parts. We propose a static type system for a calculus that captures this feature, combining constants as types, union types and recursive types. The fundamental properties of Subject Reduction and Progress are addressed to guarantee well-behaved dynamics; they rely crucially on a novel notion of *pattern compatibility*. We also introduce an efficient type-checking algorithm by formulating a syntax-directed variant of the type system. This involves algorithms for checking type equivalence and subtyping, both based on coinductive characterizations of those relations.

1. Introduction

In the lambda-calculus, functions are represented as expressions of the form $\lambda x.t$, x being the formal parameter and t the body. Such a function may be applied to any term, regardless of its form. This is captured by the β-reduction rule: $(\lambda x.t)\,s \to_\beta \{s/x\}t$, where $\{s/x\}t$ stands for the result of replacing all free occurrences of x in t with s. The β-reduction rule places no requirements on the form of s, it can be any term. *Pattern calculi* [10,22,19,23,20] are generalizations of the β-reduction rule in which abstractions $\lambda x.t$ are replaced by $\lambda p.t$ where p is called a *pattern*. An example is $\lambda\langle x, y\rangle.x$ where the pattern p is $\langle x, y\rangle$, representing a pair of terms; this function projects the first component of a pair. The outermost application in an expression such as $(\lambda\langle x, y\rangle.x)\,s$ will only be able to reduce if s is of the form $\langle s_1, s_2\rangle$, for any expressions s_1 and s_2; computation will otherwise focus on s.

The addition of constants to the lambda-calculus allows data and hence also patterns to be expressed using constants and application (so called *applicative notation*). For example, the list `[1,2]` is expressed as the following term s:

```
cons(l 1)(cons(l 2)nil)
```

constructed from constants `cons`, `nil` and `l`. Note the use of a constant `l` (for leaf) to inject values of standard types, in this case integers, into the tree-like structure. Another example is the term t capturing a binary tree:

☆ Work partially funded by the international projects DeCOPA STIC-AmSud 146/2012, CONICET, CAPES, CNRS; and ECOS-Sud A12E04, CONICET, CNRS.
* Corresponding authors.
E-mail addresses: ayala@unb.br (M. Ayala-Rincón), eabonelli@gmail.com (E. Bonelli), jedi@dc.uba.ar (J. Edi), aeviso@dc.uba.ar (A. Viso).

https://doi.org/10.1016/j.tcs.2019.02.018
0304-3975/

$\mathtt{node}\,(\mathtt{l}\,1)\,(\mathtt{node}\,(\mathtt{l}\,2)\,\mathtt{nil}\,\mathtt{nil})\,(\mathtt{node}\,(\mathtt{l}\,3)\,\mathtt{nil}\,\mathtt{nil})$

Applicative notation applies to patterns too: $\lambda\langle x, y\rangle.x$ is written as $\lambda \mathtt{p}\,x\,y.x$, $\mathtt{p}$ being a constant. Note that $\mathtt{p}$ is applied to the variable x and the resulting term is then applied to y.

Consider the following function for updating the values of any of these two structures by applying some user-supplied function f to it:

$$\begin{aligned}\mathsf{upd} = f \to\ & (\ \mathtt{l}\,z \to \mathtt{l}\,(f\,z)\\ & \mid x\,y \to (\mathsf{upd}\,f\,x)\,(\mathsf{upd}\,f\,y)\\ & \mid w \to w)\end{aligned} \tag{1}$$

Let $(+1)$ stand for the successor function. The expression $\mathsf{upd}\,(+1)$ is applicable to both s and t, returning an updated structure in which all numbers in the leaves have been incremented by one. 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 consisting solely of the variable f, and a body (in this case the body is itself another abstraction that consists of three branches). 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 *path polymorphism* [20,18]: it abstracts a path in a data structure viewed as a tree, being "split". The starting point of this paper is how to devise a calculus, which we dub *Calculus of Applicative Patterns* (CAP), that allows examples such as the one above to be typed. We next discuss the challenges that arise from devising such a type system. We do so by exhibiting examples of CAP terms, thus simultaneously serving as a gentle introduction to its syntax; the full syntax and semantics of the calculus are provided in Sec. 2.

Preliminaries on typing patterns expressing path polymorphism Consider these two expressions:

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

They should not be typable. In the first case, the abstraction is not capable of handling $\mathtt{cons}$. This will be avoided by introducing *singleton types* in the form of the constructors themselves: $\mathtt{nil}$ will be given type $\mathtt{nil}$ while $\mathtt{cons}$ will be given type $\mathtt{cons}$, and then they will be compared. In the second case, x in the pattern $\mathtt{l}\,x$ is required to be of type Nat yet the type of the argument to $\mathtt{l}$ in $\mathtt{l}\,\mathtt{true}$ is Bool. This will be avoided by introducing *type application* [26] into types: $\mathtt{l}\,x$ will be assigned a type of the form $\mathbb{l}\,@\,\mathsf{Nat}$ while $\mathtt{l}\,\mathtt{true}$ is assigned type $\mathbb{l}\,@\,\mathsf{Bool}$, then they will be compared.

CAP actually requires that the user supply type declarations for variables in patterns. The following variation of (1) includes such type declarations:

$$\begin{aligned}\mathsf{upd} = f \to_{\{f:A\supset B\}}\ & (\ \mathtt{l}\,z \to_{\{z:A\}} \mathtt{l}\,(f\,z)\\ & \mid x\,y \to_{\{x:C,\,y:D\}} (\mathsf{upd}\,f\,x)\,(\mathsf{upd}\,f\,y)\\ & \mid w \to_{\{w:E\}} w)\end{aligned} \tag{2}$$

For example, the type declaration for f states that it denotes a function from type A to type B. Types A and B will be specified below (the same applies to types C, D and E), once we know how to determine the type of the result of upd itself. We next address what the types of x and y in $x\,y$ should be, and hence the type of $x\,y$ itself. The pattern $x\,y$ can be instantiated to different terms in each recursive call to upd. For example, 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
$\mathsf{upd}\,(+1)\,s$	$\mathtt{cons}\,(\mathtt{l}\,1)$	$\mathtt{cons}\,(\mathtt{l}\,2)\,\mathtt{nil}$
$\mathsf{upd}\,(+1)\,(\mathtt{cons}\,(\mathtt{l}\,1))$	$\mathtt{cons}$	$\mathtt{l}\,1$
$\mathsf{upd}\,(+1)\,(\mathtt{cons}\,(\mathtt{l}\,2)\,\mathtt{nil})$	$\mathtt{cons}\,(\mathtt{l}\,2)$	$\mathtt{nil}$

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

$$\mu\alpha.(\mathbb{l}\,@\,A) \oplus (\alpha\,@\,\alpha) \oplus (\mathtt{cons} \oplus \mathtt{node} \oplus \mathtt{nil})$$

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. Finally, upd itself is assigned type $(A \supset B) \supset (F_A \supset F_B)$, where F_X is $\mu\alpha.(\mathbb{l}\,@\,X) \oplus (\alpha\,@\,\alpha) \oplus (\mathtt{cons} \oplus \mathtt{node} \oplus \mathtt{nil})$.

Typing data structures Type recursion, in combination with union, singleton and applicative type constructors allows one to define data types for structures like lists and trees. In the case of s and t they may be assigned, resp., the types below:

$$\mu\alpha.\mathtt{nil} \oplus (\mathtt{cons}\,@\,(\mathbb{l}\,@\,A)\,@\,\alpha) \qquad \mu\alpha.\mathtt{nil} \oplus (\mathtt{node}\,@\,(\mathbb{l}\,@\,A)\,@\,\alpha\,@\,\alpha)$$

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

Subtyping As mentioned, $\mathsf{upd}(+1)$ should be applicable to both s and t. For that, we must ensure that the types of s and t are adequate in the sense that they are in accordance with the type F_{Nat}. A suitable notion of *subtyping* will allow $\mathsf{upd}(+1)$ to be applicable to both s and t. Indeed, both of the above mentioned types, in which A is replaced with Nat, are subtypes of F_{Nat}:

$$\begin{aligned} \mu\alpha.\mathrm{nil} \oplus (\mathrm{cons} \mathbin{@} (\mathbb{l} \mathbin{@} A) \mathbin{@} \alpha) &\preceq_\mu F_{\mathsf{Nat}} \\ \mu\alpha.\mathrm{nil} \oplus (\mathrm{node} \mathbin{@} (\mathbb{l} \mathbin{@} A) \mathbin{@} \alpha \mathbin{@} \alpha) &\preceq_\mu F_{\mathsf{Nat}} \end{aligned}$$

The properties of the subtyping relation will allow us to prove Safety for CAP: it enjoys *Subject Reduction* (SR) and *Progress*. The former states that reduction of typed terms produces typed terms; the latter that typable, closed terms that are not values can always reduce. However, it turns out that although subtyping is important for this result, it is the novel notion of *pattern compatibility*, described next, that is crucial for proving both of these properties.

Pattern compatibility Consider the following example:

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

Although there is a branch capable of handling a term such as $\mathtt{l}\,4$, namely the second one, evaluation in CAP takes place in left-to-right order following standard practice in functional programming languages. Since the term $\mathtt{l}\,4$ *also* matches the pattern $\mathtt{l}\,x$, we would obtain the (incorrect) reduct $\mathtt{if}\ 4\ \mathtt{then}\ 1\ \mathtt{else}\ 0$. We thus must relate the types of $\mathtt{l}\,x$ and $\mathtt{l}\,y$ in order to avoid failure of SR. Since $\mathtt{l}\,y$ is an instance of $\mathtt{l}\,x$, we require the type of the former to be a subtype of the type of the latter since it will always have priority: $\mathbb{l} \mathbin{@} \mathsf{Nat} \preceq \mathbb{l} \mathbin{@} \mathsf{Bool}$. Fortunately, this is not the case since $\mathsf{Nat} \not\preceq \mathsf{Bool}$, rendering this example untypable. Examples like (2) require a more refined analysis (*cf.* Sec. 3.3). In a nutshell, compatibility focuses on offending positions between patterns, and whenever there is a structural overlap among their potential arguments, a subtyping restriction on their respective types is imposed.

Summary of contributions This work builds on [4,17]. The former presents a static typing discipline for CAP that guarantees safety for path polymorphism. This result relies on the syntactic notion of pattern compatibility mentioned above, hence no runtime analysis is required. The latter addresses the efficient implementation of a type-checking algorithm for the proposed system. This is done in two stages:

- The first stage presents a naïve but correct, high-level description of a type-checking algorithm, the principal aim being clarity. We propose coinductive and invertible presentations of the equivalence and subtyping relations, as well as a syntax-directed variant of the system. This leads to algorithms for checking equivalence and subtyping modulo associative, commutative and idempotent (ACI) unions, both based on the invertibility of the functional generating the associated notions.
- The second stage builds on ideas from the first algorithm with the aim of improving efficiency. μ-types are interpreted as infinite n-ary trees and represented using automata, avoiding having to explicitly handle unfoldings of recursive types, and leading to a significant improvement in the complexity of the key steps of the type-checking process, namely equality and subtype checking.

Related work CAP is a restriction of a more general pattern calculus known as the *Pure Pattern Calculus* or PPC [20]. PPC allows not only path polymorphism but also pattern polymorphism: patterns may be created at run-time. For literature on (typed) pattern calculi the reader is referred to [4]. A recent book on pattern calculi is [18]; it addresses many interesting aspects including path, pattern and parametric polymorphism. The main difference with our work is that the latter requires performing type-checking at run-time (*cf.* Chapter 9.5, "Typed Static Pattern Calculus"); here we focus exclusively on static typing. Also, efficient type-checking algorithms are not discussed in [18]. The algorithms for checking equality of recursive types or subtyping of recursive types have been extensively studied in [1,24,6,21] among others. Additionally, in [25] the authors studied the possibilities of incorporating associative and commutative (AC) products to the equality check, on an automata-based approach that the authors themselves claimed was not extensible to subtyping [29]. Later on, [14] presented another automata-based algorithm for subtyping that properly handles AC products with a complexity cost of $\mathcal{O}(n^2 n'^2 d^{5/2})$, where n and n' are the sizes of the analyzed types, and d is a bound on the arity of the involved products.

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. Sec. 6 proposes invertible generating functions for coinductive notions of type-equivalence and subtyping. Sec. 5.1 introduces a syntax-directed type system for CAP. Sec. 7 studies a more efficient type-checking algorithm based on automata. Finally, we conclude in Sec. 8. See the extended report [3] for full proofs and further details. An implementation of the algorithms described here is available online [16].

2. Syntax and operational semantics of CAP

We assume given countably infinite sets $\mathbb{V}$ of term variables and $\mathbb{C}$ of constants. 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** $(m, n, \ldots)$:

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

$$
\begin{array}{lcll}
p & ::= & x & \text{(matchable)} \\
 & | & \mathtt{c} & \text{(constant)} \\
 & | & p\ p & \text{(compound)} \\
d & ::= & \mathtt{c} & \text{(constant)} \\
 & | & d\,t & \text{(compound)}
\end{array}
\qquad
\begin{array}{lcll}
t & ::= & x & \text{(variable)} \\
 & | & \mathtt{c} & \text{(constant)} \\
 & | & t\,t & \text{(application)} \\
 & | & p \to_\theta t \mid \cdots \mid p \to_\theta t & \text{(abstraction)} \\
m & ::= & d & \text{(data structure)} \\
 & | & p \to_\theta t \mid \cdots \mid p \to_\theta t & \text{(abstraction)}
\end{array}
$$

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 matchables 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:*

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

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 $\pi \in \mathsf{pos}(s)$, *e.g.* $\mathsf{pos}\big((\mathtt{l}\,x \to_{\{x:\mathsf{Nat}\}} x)\,(\mathtt{l}\,\mathtt{true})\big) = \{\epsilon, 1, 2, 11, 12, 111, 112, 12, 21, 22\}$.

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):

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

where o denotes any possible outcome and $\sigma_1 \cup \sigma_2$ denotes the standard union of substitutions assuming that their domains are disjoint. To ensure this always holds we assume patterns to be linear (at most one occurrence of any matchable). The matching operation is defined as follows, where the clauses below are evaluated from top to bottom[1]:

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

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:

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

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:

$$
\begin{array}{ll}
 & (\mathtt{true} \to 1 \mid \mathtt{false} \to 0)\,((\mathtt{true} \to \mathtt{false} \mid \mathtt{false} \to \mathtt{true})\,\mathtt{true}) \\
\to & (\mathtt{true} \to 1 \mid \mathtt{false} \to 0)\,\mathtt{false} \\
\to & 0
\end{array}
\tag{4}
$$

Note that in $(\mathtt{true} \to 1 \mid \mathtt{false} \to 0)\,\mathtt{false}$, the second branch is selected since $\{\{\mathtt{false}/\mathtt{true}\}\} = \mathtt{fail}$. A further example showing how a data structure is decomposed by the matching operation is:

[1] It corresponds to a specialization of the matching operation introduced in [20] to static patterns.

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

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

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

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

Fig. 1. Strong type equivalence for μ-types (sample).

$$(\texttt{nil} \to \texttt{nothing} \mid \texttt{cons}\,(\texttt{l}\,x)\,xs \to \texttt{just}\,x)\,(\texttt{cons}\,(\texttt{l}\,\texttt{1})\,(\texttt{cons}\,(\texttt{l}\,\texttt{2})\,\texttt{nil})) \to \texttt{just}\,\texttt{1}$$

where:

$$\{\{\texttt{cons}\,(\texttt{l}\,\texttt{1})\,(\texttt{cons}\,(\texttt{l}\,\texttt{2})\,\texttt{nil})/\texttt{nil}\}\} = \texttt{fail}$$
$$\{\{\texttt{cons}\,(\texttt{l}\,\texttt{1})\,(\texttt{cons}\,(\texttt{l}\,\texttt{2})\,\texttt{nil})/\texttt{cons}\,(\texttt{l}\,x)\,xs\}\} = \{\texttt{1}/x, \texttt{cons}\,(\texttt{l}\,\texttt{2})\,\texttt{nil}/xs\}$$

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

This result follows from a straightforward application of the CR proof technique presented in [20] to our calculus. The key step is proving that the matching operation satisfies the *Rigid Matching Condition (RMC)* proposed in the cited work.

3. Typing system

This section presents μ-types, the finite type expressions that are 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 introduce two sorts of typing expressions: *types* and *datatypes*, the latter being a subset of the former.

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** ($\mathbb{c}, \mathbb{d}, \ldots$); the sets $\mathcal{T}_D$ of **μ-datatypes** and $\mathcal{T}$ of **μ-types**, resp., are inductively defined as follows:

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

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 $\mathcal{V} \cup \mathcal{C}$.

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 represents 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 $\oplus_{i \in 1..n} A_i$. 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, $A' \supset A''$ or $\mu V.B$ with B 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 $\preceq_\mu$.

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

$$\frac{}{\Sigma, V \preceq_\mu W \vdash V \preceq_\mu W}\,(\text{S-HYP}) \qquad \frac{\Sigma, V \preceq_\mu W \vdash A \preceq_\mu B \quad W \notin \mathsf{fv}(A) \quad V \notin \mathsf{fv}(B)}{\Sigma \vdash \mu V.A \preceq_\mu \mu W.B}\,(\text{S-REC})$$

$$\frac{\Sigma \vdash D \preceq_\mu D' \quad \Sigma \vdash A \preceq_\mu A'}{\Sigma \vdash D\,@\,A \preceq_\mu D'\,@\,A'}\,(\text{S-COMP}) \qquad \frac{\Sigma \vdash A' \preceq_\mu A \quad \Sigma \vdash B \preceq_\mu B'}{\Sigma \vdash A \supset B \preceq_\mu A' \supset B'}\,(\text{S-FUNC})$$

$$\frac{\Sigma \vdash A \preceq_\mu C \quad \Sigma \vdash B \preceq_\mu C}{\Sigma \vdash A \oplus B \preceq_\mu C}\,(\text{S-UNION-L}) \qquad \frac{\vdash A \simeq_\mu B}{\Sigma \vdash A \preceq_\mu B}\,(\text{S-EQ})$$

$$\frac{\Sigma \vdash A \preceq_\mu B}{\Sigma \vdash A \preceq_\mu B \oplus C}\,(\text{S-UNION-R1}) \qquad \frac{\Sigma \vdash A \preceq_\mu C}{\Sigma \vdash A \preceq_\mu B \oplus C}\,(\text{S-UNION-R2})$$

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

Patterns

$$\frac{}{\{x : A\} \vdash_{\mathsf{p}} x : A}\,(\text{P-MATCH}) \qquad \frac{}{\varnothing \vdash_{\mathsf{p}} c : \mathbb{c}}\,(\text{P-CONST}) \qquad \frac{\theta_1 \vdash_{\mathsf{p}} p : D \quad \theta_2 \vdash_{\mathsf{p}} q : A}{\theta_1, \theta_2 \vdash_{\mathsf{p}} p\,q : D\,@\,A}\,(\text{P-COMP})$$

Terms

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

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

$$\frac{\Gamma \vdash r : \oplus_{i\in 1..n} A_i \supset B \quad \Gamma \vdash u : A_k \quad k \in 1..n}{\Gamma \vdash r\,u : B}\,(\text{T-APP}) \qquad \frac{\Gamma \vdash s : A \quad \vdash A \preceq_\mu A'}{\Gamma \vdash s : A'}\,(\text{T-SUBS})$$

Fig. 3. Typing rules for patterns and terms.

Definition 3.2 *(μ-Type Equivalence and μ-Subtyping). μ-*Type Equivalence*, $\simeq_\mu$, is the least congruence closed under the rules in Fig. 1. μ-*Subtyping*, $\preceq_\mu$, is the least preorder*[2] *closed under the rules 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 arbitrary types ($\mu X.A$). Likewise for (E-FOLD). The relation resulting from dropping (E-CONTR) [2,7] is called weak type equivalence [9] 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 $\oplus_{i\in 1..n} A_i$ 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 write θ_1, θ_2 for the typing context that, given x, behaves as θ_1 if $x \in \mathsf{dom}(\theta_1)$ and as θ_2 if $x \in \mathsf{dom}(\theta_2)$; here θ_1 and θ_2 are assumed to have disjoint domains. We have typing judgments for patterns, $\theta \vdash_{\mathsf{p}} p : A$, and for terms, $\Gamma \vdash s : A$. Moreover, if σ is a substitution, the judgement $\Gamma \vdash \sigma : \theta$ holds if $\mathsf{dom}(\sigma) = \mathsf{dom}(\theta)$ and $\Gamma \vdash \sigma(x) : \theta(x)$, for all $x \in \mathsf{dom}(\sigma)$. Fig. 3 (top and bottom) depicts two sets of typing rules, one defining $\theta \vdash_{\mathsf{p}} p : A$ and the other defining $\Gamma \vdash s : A$. We write $\rhd\, \theta \vdash_{\mathsf{p}} p : A$ to indicate that $\theta \vdash_{\mathsf{p}} p : A$ is derivable (likewise for $\rhd\, \Gamma \vdash s : A$). The typing schemes for patterns are straightforward. Worth noticing, though, is that variables in contexts are handled linearly: $\mathsf{dom}(\theta) = \mathsf{fv}(p)$ whenever $\rhd\, \theta \vdash_{\mathsf{p}} p : A$. The typing rules for terms mostly speak for themselves except for two of them which we now comment. The first is (T-APP) where A_i are not required to be non-union types, thus allowing examples such as (4) 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, $i \in 1..n$. 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

[2] Reflexive and transitive binary relation.

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

noteworthy is the condition $\mathsf{cmp}([p_i : A_i]_{i\in 1..n})$, *i.e.* that the list of annotated patterns $[p_1 : A_1, \ldots, p_n : A_n]$ or $[p_i : A_i]_{i\in 1..n}$ for short, be *compatible*, a notion we discuss in Sec. 3.3. A summary of the judgements presented so far are:

Judgement	Description	Ref.
$\vdash A \simeq_\mu B$	Types A and B are equivalent	Fig. 1
$\Sigma \vdash A \preceq_\mu B$	A is a subtype of B under type assumptions Σ	Fig. 2
$\theta \vdash_{\mathsf{p}} p : A$	Pattern p has type A under typing context θ	Fig. 3
$\Gamma \vdash s : A$	Term s has type A under typing context Γ	Fig. 3
$\Gamma \vdash \sigma : \theta$	Substitution σ has type θ under typing context Γ	Sec. 3.2

3.3. Pattern compatibility

Let us say that pattern p **subsumes** 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 imposed. In some cases no requirements are necessary. For example in:

$$f \to_{\{f:C\supset D\}} \begin{array}{lll} (& \mathtt{c}\, z & \to_{\{z:C\}} \quad \mathtt{c}\,(f\, z) \\ \mid & \mathtt{d}\, y & \to_{\{y:E\}} \quad \mathtt{d}\, y) \end{array} \tag{5}$$

no relation between A (the type of $\mathtt{c}\, z$) and B (the type of $\mathtt{d}\, y$) is required since the branches are mutually disjoint. In other cases, however, $B \preceq_\mu A$ is required; we seek to characterize these other cases. We focus on those 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:

	$p\vert_\pi$	$q\vert_\pi$	
(a)	$\mathtt{c}$	y	restriction required
(b)	$\mathtt{c}$	$\mathtt{d}$	no overlapping ($q \not\lhd p$)
(c)	$\mathtt{c}$	$q_1\, q_2$	no overlapping
(d)	$p_1\, p_2$	y	restriction required
(e)	$p_1\, p_2$	$\mathtt{d}$	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 (5) 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), for the second one consider:

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

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 $\rhd \theta \vdash_{\mathsf{p}} p : A$ 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:*

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

Note that $\rhd \theta \vdash_{\mathsf{p}} p : A$ 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:*

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

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

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. *Type assignment* $p : A$ overlaps *with* $q : B$ *iff:*

$$\mathsf{ovl}(p : A, q : B) \triangleq \forall \pi \in \mathsf{mmpos}(p,q)\,.A\|_\pi \cap B\|_\pi \neq \varnothing$$

Type assignment $p : A$ *is* compatible *with* $q : B$ *iff:*

$$\mathsf{cmp}(p : A, q : B) \triangleq \mathsf{ovl}(p : A, q : B) \implies B \preceq_\mu A$$

A list of type assignments $[p_i : A_i]_{i\in 1..n}$ *is compatible iff:*

$$\mathsf{cmp}\big([p_i : A_i]_{i\in 1..n}\big) \triangleq \forall i, j \in 1..n.i < j \implies \mathsf{cmp}\big(p_i : A_i, p_j : A_j\big)$$

We conclude this section with an example that illustrates how the type system establishes a clear distinction between semi-structured data, susceptible to path polymorphism, and "unstructured" data (represented below by base and functional types). Suppose we wish to apply the upd function (2) from the Introduction to data structures holding values of different types: say `l` prefixed values of type A_1 and `l2` prefixed values of type $A_2 \supset A_3$. Note that 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 $\mathtt{cons} \oplus \mathtt{node} \oplus \mathtt{nil} \oplus \mathtt{l2} \oplus (A_2 \supset A_3)$. 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 upd capable of handling values prefixed by `l2`:

$$\begin{array}{lllll}\mathsf{upd}' = f \to_{\{f:A_1\supset B\}} g \to_{\{g:(A_2\supset A_3)\supset B\}} & (\ \mathtt{l}\,z & \to_{\{z:A_1\}} & \mathtt{l}\,(f\,z)\\ & \mid\ \mathtt{l2}\,z & \to_{\{z:A_2\supset A_3\}} & \mathtt{l2}\,(g\,z)\\ & \mid\ x\,y & \to_{\{x:C,y:D\}} & (\mathsf{upd}'\,f\,x)\,(\mathsf{upd}'\,f\,y)\\ & \mid\ w & \to_{\{w:E\}} & w)\end{array}$$

The type of upd′ is $(A_1 \supset B) \supset ((A_2 \supset A_3) \supset B) \supset (F_{A_1,A_2\supset A_3} \supset F_{B,B})$ with

$$F_{X,Y} = \mu\alpha.(\mathtt{l} \mathbin{@} X) \oplus (\mathtt{l2} \mathbin{@} Y) \oplus (\alpha \mathbin{@} \alpha) \oplus (\mathtt{cons} \oplus \mathtt{node} \oplus \mathtt{nil})$$

3.4. Basic metatheory of typing

We present some technical lemmas that will be useful in the proof of safety and type-checking.

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

1. *If* $\rhd \Gamma \vdash x : A$ *then* $\exists A'$ *s.t.* $A' \preceq_\mu A$ *and* $x : A' \in \Gamma$.
2. *If* $\rhd \Gamma \vdash c : A$ *then* $\mathtt{c} \preceq_\mu A$.
3. *If* $\rhd \Gamma \vdash r\,u : A$ *then, either:*
   (a) $\exists D, A'$ *s.t.* $D \mathbin{@} A' \preceq_\mu A$, $\rhd \Gamma \vdash r : D$ *and* $\rhd \Gamma \vdash u : A'$*; or*
   (b) $\exists A_1, \ldots, A_n, A', k \in 1..n$ *s.t.* $A' \preceq_\mu A$, $\rhd \Gamma \vdash r : \oplus_{i\in 1..n} A_i \supset A'$, *and* $\rhd \Gamma \vdash u : A_k$.
4. *If* $\rhd \Gamma \vdash (p_i \to_{\theta_i} s_i)_{i\in 1..n} : A$ *then* $\exists A_1, \ldots, A_n, B$ *s.t.* $\oplus_{i\in 1..n} A_i \supset B \preceq_\mu A$, $\mathsf{cmp}\big([p_i : A_i]_{i\in 1..n}\big)$, $\rhd \theta_i \vdash_{\mathsf{p}} p_i : A_i$ *and* $\rhd \Gamma, \theta_i \vdash s_i : B$ *for every* $i \in 1..n$.

Lemma 3.7 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 typed, can also be typed with a more specific non-union datatype.

Lemma 3.7 *(Typing for Data Structures). Let* $\rhd \Gamma \vdash d : A$ *for* $d \in \mathbb{D}$. *Then,* $\exists D \in \mathcal{T}_D$ *s.t.* D *is a non-union type,* $D \preceq_\mu A$ *and* $\rhd \Gamma \vdash d : D$. *Moreover,*

1. *If* $d = c$, *then* $D \simeq_\mu \mathtt{c}$.
2. *If* $d = d'\,t$, *then* $\exists D', A'$ *s.t.* $D \simeq_\mu D' \mathbin{@} A'$, $\rhd \Gamma \vdash d' : D'$, *and* $\rhd \Gamma \vdash t : A'$.

Some results on compatibility follow, the crucial one being Lemma 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. *Let* $\rhd \Gamma \vdash u : B$, $\rhd \theta \vdash_{\mathsf{p}} p : A$ *and* $\{\{u/p\}\} = \mathtt{fail}$, *then* $B \not\preceq_\mu A$.

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

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, guarded 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 $\mathsf{ovl}(p : A, q : B)$ must hold. Since patterns in a well-typed abstraction are compatible, whenever p comes before q we get $B \preceq_\mu A$, and thus $\rhd \Gamma \vdash u : A$ too.

Lemma 3.9 *(Compatibility Lemma). Let* $\rhd \Gamma \vdash u : B$, $\rhd \theta \vdash_{\mathsf{p}} p : A$, $\rhd \theta' \vdash_{\mathsf{p}} q : B$ *and* $\{\{u/p\}\}$ *be successful. Then,* $\mathsf{ovl}(p : A, q : B)$ *holds.*

Recall from Sec. 3.2 that $\Gamma \vdash \sigma : \theta$ indicates 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). Let* $\rhd \Gamma \vdash u : A$, $\rhd \theta \vdash_{\mathsf{p}} p : A$ *and* $\{\{u/p\}\} = \sigma$. *Then,* $\rhd \Gamma \vdash \sigma : \theta$.

Finally, we recall the standard Substitution Lemma.

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

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.

1. *If* $D \mathbin{@} A \preceq_\mu D' \mathbin{@} A'$, *then* $D \preceq_\mu D'$ *and* $A \preceq_\mu A'$.
2. *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 strong equivalence and subtyping. For the latter, invertibility of non-union types is proved coinductively, entailing Proposition 3.12 (*cf.* [3]). Further details on the coinductive presentation of type equivalence and subtyping are supplied in Sec. 6.

4. Safety

This section addresses Safety. We start with Subject Reduction.

Proposition 4.1. *If* $\rhd \Gamma \vdash s : A$ *and* $s \to s'$, *then* $\rhd \Gamma \vdash s' : A$.

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 Lemma 3.6 (3b), there exists $C_1, \ldots, C_m$, A' such that $A' \preceq_\mu A$, $\rhd \Gamma \vdash (p_i \to_{\theta_i} s_i)_{i \in 1..n} : \oplus_{j \in 1..m} C_m \supset A'$ and:

$$\rhd \Gamma \vdash u : C_{k'}$$

for some $k' \in 1..m$. Applying Lemma 3.6, once again, item (4) this time, to $\rhd \Gamma \vdash (p_i \to_{\theta_i} s_i)_{i \in 1..n} : \oplus_{j \in 1..m} C_m \supset A'$, we get $\exists A_1, \ldots, A_n, B$ such that:

$$\oplus_{i \in 1..n} A_i \supset B \preceq_\mu \oplus_{j \in 1..m} C_m \supset A' \tag{6}$$

$\mathsf{cmp}\big([p_i : A_i]_{i \in 1..n}\big)$, $\rhd \theta_i \vdash_{\mathsf{p}} p_i : A_i$ and $\rhd \Gamma, \theta_i \vdash s_i : B$ for every $i \in 1..n$.

From (6), by invertibility of subtyping for non-union types, $B \preceq_\mu A'$ and:

$$\oplus_{j \in 1..m} C_m \preceq_\mu \oplus_{i \in 1..n} A_i \tag{7}$$

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

1. 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 $\rhd \Gamma \vdash u : D$.
   (b) u is an abstraction: then, by Lemma 3.6 (4), there exists types C', C'' such that $C' \supset C'' \preceq_\mu C_{k'}$ and $\rhd \Gamma \vdash u : C' \supset C''$.

   Then, in both cases there exists a non-union type, say C, such that $C \preceq_\mu C_{k'}$ and $\rhd \Gamma \vdash u : C$. Then, from (7) we get:

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

and, since C is non-union, $C \preceq_\mu A_l$ for some $l \in 1..n$. Hence, by subsumption $\rhd \Gamma \vdash u : A_l$.
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 Lemma 3.8, we have $A_l \not\preceq_\mu A_i$. Thus, it must be the case that $k < l$. By Lemma 3.9 with hypothesis $\rhd \Gamma \vdash u : A_l$, $\rhd \theta_k \vdash_{\mathsf{p}} p_k : A_k$, $\rhd \theta_l \vdash_{\mathsf{p}} p_l : A_l$ and $\{\{u/p_k\}\} = \sigma$ we get that $\mathsf{ovl}(p_k : A_k, p_l : A_l)$ holds. Additionally, we already saw that $\mathsf{cmp}([p_i : A_i]_{i\in 1..n})$ holds, thus $\mathsf{cmp}(p_k : A_k, p_l : A_l)$ and by definition $A_l \preceq_\mu A_k$. Finally we conclude by subsumption once again, $\rhd \Gamma \vdash u : A_k$.

2. 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 (7) we get

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

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

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

Let the set of **values** be $v ::= x\, v_1 \dots v_m \mid \subset v_1 \dots v_m \mid (p_i \to_{\theta_i} s_i)_{i\in 1..n}$ with $m \geq 0$. Progress states that reduction for closed typed terms does not get stuck before reaching a value.

Proposition 4.2. *If $\rhd \varnothing \vdash s : A$ 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.

5. Towards type-checking

Obtaining an implementation of type-checking based on the type system of Fig. 3 is not immediate for two reasons.

The first is that the system is not syntax-directed due to the presence of the subsumption typing rule. Given a term s and assuming it is typable, there may be more than one typing derivation for it. A syntax-directed presentation of the type system of Fig. 3 can be obtained by dropping subsumption and "hard-wiring" it back into (T-APP). It is presented in Sec. 5.1. Judgements in this new presentation will take the form $\Gamma \mapsto s : A$ to distinguish them for the ones of the non syntax-directed presentation.

The second reason that implementing a type-checker for CAP is not immediate is the non-trivial nature of deciding $\preceq_\mu$. To address the latter we will resort to a coinductive presentation of subtyping. It is well-known (*cf.* Sec. 6) that if such a characterization can be provided in which the so called generating function is *invertible*, then a simple algorithm can be given for deciding the relation. It should be mentioned that an invertible coinductive presentation also provides a convenient mechanism for reasoning about subtyping; this fact will be used to prove that the syntax and non syntax-directed presentations are in correspondence.

The solutions to these two obstacles, as described above, will indeed produce a type-checking algorithm for CAP, however it will be an inefficient one. Our last contribution to type-checking for CAP is to switch from a representation of types based on trees to one based on *term automata*. This shall be the subject of Sec. 7.

The rest of this section develops the syntax-directed presentation of our type-system.

5.1. Syntax-directed typing

As mentioned, a syntax-directed presentation for typing in CAP, inferring judgments of the form $\Gamma \mapsto s : A$, may be obtained from the rules of Fig. 3 by dropping subsumption and "hard-wiring" it back in into (T-APP). Unfortunately, the so obtained naïve syntax-directed variant:

$$\Gamma \mapsto r : (\oplus_{i\in 1..n} A_i) \supset B \quad \Gamma \mapsto u : A' \quad A' \preceq_\mu A_k, \text{ for some } k \in 1..n \Gamma \mapsto r\, u : B \text{(T-APP-AL)}'$$

fails to capture all the required terms. In other words, there are Γ, s and A such that $\Gamma \vdash s : A$ but no $A' \preceq_\mu A$ such that $\Gamma \mapsto s : A'$. For example, take $\Gamma(x) \triangleq (\mathbb{c} \oplus \mathbb{e} \supset \mathbb{d}) \oplus (\mathbb{c} \oplus \mathbb{f} \supset \mathbb{d})$, $s \triangleq x\subset$ and $A \triangleq \mathbb{d}$. Using (T-APP) we can indeed prove the judgement $\Gamma \vdash x\subset : \mathbb{d}$. This follows from the fact that $(\mathbb{c} \oplus \mathbb{e} \supset \mathbb{d}) \oplus (\mathbb{c} \oplus \mathbb{f} \supset \mathbb{d}) \preceq_\mu \mathbb{c} \supset \mathbb{d}$ and hence $\Gamma \vdash x : \mathbb{c} \supset \mathbb{d}$. However, from $\Gamma \mapsto r : A$ and $A \preceq_\mu \oplus_{i\in 1..n} A_i \supset B$, in general, we cannot infer that A is a functional type due to the presence of union types. A complete syntax directed presentation is obtained by dropping (T-SUBS) from Fig. 3 and replacing (T-ABS) and (T-APP) by (T-ABS-AL) and (T-APP-AL), resp. The full set of rules for the typing system is presented in Fig. 4. It is complete in the following sense:

Proposition 5.1.

1. *If $\rhd \Gamma \mapsto s : A$, then $\rhd \Gamma \vdash s : A$.*

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

$$\frac{(\theta_i \vdash_{\mathsf{p}} p_i : A_i)_{i\in 1..n} \quad \mathsf{cmp}([p_i : A_i]_{i\in 1..n}) \quad (\Gamma, \theta_i \vdash s_i : B_i)_{i\in 1..n}}{\Gamma \vdash (p_i \to_{\theta_i} s_i)_{i\in 1..n} : \oplus_{i\in 1..n} A_i \supset \oplus_{i\in 1..n} B_i}\,(\text{T-ABS-AL})$$

$$\frac{\Gamma \vdash r : A \quad A \simeq_\mu \oplus_{i\in 1..n}(A_i \supset B_i) \quad A_i \neq \oplus \quad \Gamma \vdash u : C \quad (\vdash C \preceq_\mu A_i)_{i\in 1..n}}{\Gamma \vdash r\,u : \oplus_{i\in 1..n} B_i}\,(\text{T-APP-AL})$$

Fig. 4. Syntax-directed typing rules for terms.

$$\begin{array}{rcl}
\mathsf{tcp}(\Gamma, x) & \triangleq & \mathtt{if}\ x \in \mathsf{dom}(\Gamma)\ \mathtt{then}\ \Gamma(x)\ \mathtt{else}\ \mathtt{fail} \\
\mathsf{tcp}(\Gamma, \mathtt{c}) & \triangleq & \mathbb{c} \\
\mathsf{tcp}(\Gamma, p\,q) & \triangleq & \mathtt{let}\ A = \mathsf{tcp}(\Gamma, p), B = \mathsf{tcp}(\Gamma, q)\ \mathtt{in} \\
 & & \quad \mathtt{if}\ A\ \text{is a datatype}\ \mathtt{then}\ A\,@\,B\ \mathtt{else}\ \mathtt{fail} \\
 & & \\
\mathsf{tc}(\Gamma, x) & \triangleq & \mathtt{if}\ x \in \mathsf{dom}(\Gamma)\ \mathtt{then}\ \Gamma(x)\ \mathtt{else}\ \mathtt{fail} \\
\mathsf{tc}(\Gamma, \mathtt{c}) & \triangleq & \mathbb{c} \\
\mathsf{tc}(\Gamma, (p_i \to_{\theta_i} s_i)_{i\in 1..n}) & \triangleq & \mathtt{let}\ A_i = \mathsf{tcp}(\theta_i, p_i), B_i = \mathsf{tc}(\Gamma, \theta_i, s_i)\ \mathtt{in} \\
 & & \quad \mathtt{if}\ \forall i \in 1..n.\forall j \in i+1..n.\mathsf{compatible}(p_i : A_i, p_j : A_j) \\
 & & \quad\ \mathtt{then}\ \oplus_{i\in 1..n} A_i \supset \oplus_{i\in 1..n} B_i\ \mathtt{else}\ \mathtt{fail} \\
\mathsf{tc}(\Gamma, r\,u) & \triangleq & \mathtt{let}\ A = \mathsf{tc}(\Gamma, r), C = \mathsf{tc}(\Gamma, u)\ \mathtt{in} \\
 & & \quad \mathtt{if}\ A\ \text{is a datatype}\ \mathtt{then}\ A\,@\,C \\
 & & \quad\ \mathtt{else}\ \mathtt{let}\ \oplus_{i\in 1..n}(A_i \supset B_i) = \mathsf{unfold}(A)\ \mathtt{in} \\
 & & \quad\quad \mathtt{if}\ \forall i \in 1..n.\mathsf{subtype}(\varnothing, C, A_i) \\
 & & \quad\quad\ \mathtt{then}\ \oplus_{i\in 1..n} B_i\ \mathtt{else}\ \mathtt{fail}
\end{array}$$

Fig. 5. Type-checking for patterns and terms of CAP.

2. *If* $\triangleright\, \Gamma \vdash s : A$, *then* $\exists A'$ *such that* $A' \preceq_\mu A$ *and* $\triangleright\, \Gamma \mapsto s : A'$.

The proof is by induction on the derivation of $\Gamma \mapsto s : A$, in the first item, and by induction on the derivation of $\Gamma \vdash s : A$, in the second. Moreover, in the proof of the second item we will resort to the coinductive presentation of subtyping (*cf.* Sec. 6) to reason about the form of the types. Consider the case where the derivation of $\Gamma \vdash s : A$ ends in an instance of (T-APP). Then $\Gamma \vdash r : \oplus_{i\in 1..n} A_i \supset A$ and $\Gamma \vdash u : A_k$ for some $k \in 1..n$. By the induction hypothesis we have $\Gamma \mapsto r : B$ with $B \preceq_\mu \oplus_{i\in 1..n} A_i \supset A$. Without loss of generality, we can assume $B = \oplus_{j\in 1..m} B_j$ with $B_j \neq \oplus$. Moreover, by contractiveness, we can further assume that $B \simeq_\mu \oplus_{j\in 1..m} B_j$ and $B_j \neq \mu, \oplus$ for every $j \in 1..m$. In order to continue we need to relate the B_j with $\oplus_{i\in 1..n} A_i \supset A$. This, is one example of where the invertible coinductive presentation is put to use; in this case to determine that $B_j \preceq_\mu \oplus_{i\in 1..n} A_i \supset A$ for every $j \in 1..m$.

In the sequel of this section we assume we have an algorithm for checking whether a type A is a subtype of type B. We write $\mathsf{subtype}(\varnothing, A, B)$ for this algorithm (the role of $\varnothing$ will be explained later). From the syntax-directed presentation we may obtain a simple type-checking function $\mathsf{tc}(\Gamma, s)$ (Fig. 5) such that $\mathsf{tc}(\Gamma, s) = A$ iff $\Gamma \mapsto s : A'$, for some $A' \simeq_\mu A$. The interesting clause is that of application, where the decision of whether (T-COMP-AL) or (T-APP-AL) may be applied depends on the result of the recursive call. If the term r is assigned a datatype, then a new compound datatype is built; if its type can be rewritten as a union of functional types, then a proper type is constructed with each of the co-domains of the latter, as established in rule (T-APP-AL). The expression $\mathsf{unfold}(A)$, in the clause defining $\mathsf{tc}(\Gamma, r\,u)$, is the result of unfolding type A using rules (E-REC-L-AL) and (E-REC-R-AL) until the result is an equivalent type $A' = \oplus_{i\in 1..n} A'_i$ with $A'_i \neq \mu, \oplus$, and then simply verifying that $A'_i = \supset$ for all $i \in 1..n$.

$$\begin{array}{rcl}
\mathsf{unfold}(A \supset B) & \triangleq & A \supset B \\
\mathsf{unfold}(\oplus_{i\in 1..n} A_i) & \triangleq & \mathtt{let}\ \oplus_{j\in 1..m_i}(A_{ij} \supset B_{ij}) = \mathsf{unfold}(A_i)\ \mathtt{foreach}\ i \in 1..n\ \mathtt{in} \\
 & & \quad \oplus_{\substack{i\in 1..n \\ j\in 1..m_i}} (A_{ij} \supset B_{ij}) \qquad \text{if } n > 1 \text{ and } (A_i \neq \oplus)_{i\in 1..n} \\
\mathsf{unfold}(\mu V.A) & \triangleq & \mathsf{unfold}(\{\mu V.A/V\}\,A) \\
\mathsf{unfold}(_) & \triangleq & \mathtt{fail}
\end{array}$$

Termination is guaranteed by contractiveness of μ-types. In the worst case it requires exponential time due to the need to unfold types until the desired equivalent form is obtained (*e.g.* $\mu X_1 \ldots \mu X_n . X_1 \supset \ldots X_n \supset \mathbb{c}$).

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

Compatibility between branches is verified by checking $\mathsf{ovl}(p : A, q : B)$:

$$\begin{aligned}
\mathsf{compatible}(p : A, q : B) &\triangleq (\mathtt{not}\ \mathsf{overlap}(p : A, q : B))\ \mathtt{or}\ \mathsf{subtype}(\varnothing, B, A)\\
\mathsf{overlap}(p_1\ p_2 : A, q_1\ q_2 : B) &\triangleq \mathtt{let}\ A = A_1\ @\ A_2, B = B_1\ @\ B_2\ \mathtt{in}\\
&\qquad \mathsf{overlap}(p_1 : A_1, q_1 : B_1)\ \mathtt{and}\ \mathsf{overlap}(p_2 : A_2, q_2 : B_2)\\
\mathsf{overlap}(p : A, q : B) &\triangleq (p = x)\ \mathtt{or}\ (p = q = \mathtt{c})\ \mathtt{or}\ (A\|_\epsilon \cap B\|_\epsilon \neq \varnothing)
\end{aligned}$$

In overlap we may assume that it has already been checked that p has type A and q has type B. Therefore, if these are compound patterns they can only be assigned application types, and union types may only appear at leaf positions of a pattern. We use this correspondence to traverse both pattern and type simultaneously in linear time, which means the worst-case execution time of the compatibility check is governed by the complexity of subtyping.

6. Coinductive presentation of subtyping

Coinductive characterizations of subsets of $\mathcal{T} \times \mathcal{T}$ whose generating function Φ is *invertible* admit a simple (although not necessarily efficient) algorithm for subtype membership checking and consists in "running Φ backwards" [27, Sec. 21.5]. This strategy is supported by the fact that contractiveness of μ-types guarantees a finite state space to explore (*i.e.* unfolding these types results in regular trees); invertibility further guarantees that there is at most one way in which a member of $\nu\Phi$, the greatest fixed-point of Φ, can be generated. Invertibility of $\Phi : \wp(\mathcal{T} \times \mathcal{T}) \to \wp(\mathcal{T} \times \mathcal{T})$ means that for any $\langle A, B\rangle \in \mathcal{T}$, the set $\{\mathcal{X} \in \wp(\mathcal{T} \times \mathcal{T}) \mid \langle A, B\rangle \in \Phi(\mathcal{X})\}$ is either empty or contains a unique member. We say that a set $S \in \wp(\mathcal{T} \times \mathcal{T})$ is Φ-dense if $S \subseteq \Phi(S)$. For monotone Φ, the greatest fixed-point of Φ (written $\nu\Phi$), is the largest Φ-dense set. We next introduce invertible generating functions for equivalence and subtyping.

6.1. Equivalence checking

A coinductive presentation of type equality over μ-types is given by the rules in Fig. 6. The resulting relation $\simeq^{\mathrm{co}}_\mu$ is defined by means of these rules, interpreted coinductively as indicated by the double lines. Rules (E-REFL-AL), (E-COMP-AL) and (E-FUNC-AL) are standard. We describe the remaining ones. The rule (E-UNION-AL) makes use of functions f and g to encode the ACI nature of $\oplus$. Letters $\mathcal{C}, \mathcal{D}$, used in rules (E-REC-L-AL) and (E-REC-R-AL), denote contexts of the form:

$$A_1 \oplus \ldots A_{i-1} \oplus \Box \oplus A_{i+1} \oplus \ldots \oplus A_n$$

where $\Box$ denotes the hole of the context, $A_j \neq \oplus$ for all $j \in 1..n \setminus i$ and $A_l \neq \mu$ for all $l \in 1..i-1$. Note that, in particular, $\mathcal{C}$ may take the form $\Box$. These contexts help identify the first occurrence of a μ constructor within a maximal union. For instance, consider types $\mathtt{a} \oplus (\mu\alpha.\mathtt{b} \supset \alpha)$ and $\mathtt{a} \oplus (\mathtt{b} \supset \mu\alpha.\mathtt{b} \supset \alpha)$ and the derivation:

$$\dfrac{\dfrac{}{\mathtt{a} \simeq^{\mathrm{co}}_\mu \mathtt{a}}\ (\text{E-REFL-AL}) \qquad \dfrac{\dfrac{}{\mathtt{b} \simeq^{\mathrm{co}}_\mu \mathtt{b}}\ (\text{E-REFL-AL}) \qquad \dfrac{\dfrac{\mathtt{b} \supset \mu\alpha.\mathtt{b} \supset \alpha \simeq^{\mathrm{co}}_\mu \mathtt{b} \supset \mu\alpha.\mathtt{b} \supset \alpha}{\mathtt{b} \supset \mu\alpha.\mathtt{b} \supset \alpha \simeq^{\mathrm{co}}_\mu \mu\alpha.\mathtt{b} \supset \alpha}\ (\text{E-REC-R-AL})}{\mu\alpha.\mathtt{b} \supset \alpha \simeq^{\mathrm{co}}_\mu \mu\alpha.\mathtt{b} \supset \alpha}\ (\text{E-REC-L-AL})}{\mathtt{b} \supset \mu\alpha.\mathtt{b} \supset \alpha \simeq^{\mathrm{co}}_\mu \mathtt{b} \supset \mu\alpha.\mathtt{b} \supset \alpha}\ (\text{E-FUNC-AL})}{\mathtt{a} \oplus (\mathtt{b} \supset \mu\alpha.\mathtt{b} \supset \alpha) \simeq^{\mathrm{co}}_\mu (\mathtt{b} \supset \mu\alpha.\mathtt{b} \supset \alpha) \oplus \mathtt{a}}\ (\text{E-UNION-AL})}{\mathtt{a} \oplus (\mu\alpha.\mathtt{b} \supset \alpha) \simeq^{\mathrm{co}}_\mu (\mathtt{b} \supset \mu\alpha.\mathtt{b} \supset \alpha) \oplus \mathtt{a}}\ (\text{E-REC-L-AL})$$

Note that there is only one applicable rule at each step due to the conditions on rules (E-UNION-AL) and (E-REC-R-AL). This allows us to guarantee the *invertibility of the generating function* for $\simeq^{\mathrm{co}}_\mu$. Moreover, $\simeq^{\mathrm{co}}_\mu$ coincides with $\simeq_\mu$. These two properties will allow us to check $A \simeq_\mu B$ by using the invertibility of the generating function (implicit in the rules of Fig. 6) for $\simeq^{\mathrm{co}}_\mu$.

Proposition 6.1. *$A \simeq_\mu B$ iff $A \simeq^{co}_\mu B$.*

The proof of Proposition 6.1 relies on an intermediate relation $\simeq_{\mathfrak{T}}$ over the possibly infinite trees resulting from the complete unfolding of μ-types. This relation is defined using the same rules as in Fig. 6 except for two important differences: 1) the relation is defined over regular (infinite) trees, and 2) rules (E-REC-L-AL) and (E-REC-R-AL) are dropped.

As mentioned above, we can resort to invertibility of the generating function to check for $\simeq^{\mathrm{co}}_\mu$. Fig. 7 presents the algorithm. It uses $\mathtt{seq}\ e_1 \ldots e_n$ which sequentially evaluates each of its arguments, returning the value of the first of these that does not fail. Evaluation of $\mathsf{eqtype}(\emptyset, A, B)$ can have one of two outcomes: `fail`, meaning that $A \not\simeq^{\mathrm{co}}_\mu B$, or a set $S \in \wp(\mathcal{T} \times \mathcal{T})$ that is Φ-dense with $\langle A, B\rangle \in S$, proving that $A \simeq^{\mathrm{co}}_\mu B$.

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

$$\frac{}{a \simeq^{co}_{\mu} a}\ (\text{E-REFL-AL})$$

$$\frac{D \simeq^{co}_{\mu} D' \quad A \simeq^{co}_{\mu} A'}{D\,@\,A \simeq^{co}_{\mu} D'\,@\,A'}\ (\text{E-COMP-AL}) \qquad \frac{A \simeq^{co}_{\mu} A' \quad B \simeq^{co}_{\mu} B'}{A \supset B \simeq^{co}_{\mu} A' \supset B'}\ (\text{E-FUNC-AL})$$

$$\frac{\mathscr{C}[\{\mu V.A/V\}\,A] \simeq^{co}_{\mu} B}{\mathscr{C}[\mu V.A] \simeq^{co}_{\mu} B}\ (\text{E-REC-L-AL}) \qquad \frac{A \simeq^{co}_{\mu} \mathscr{D}[\{\mu W.B/W\}\,B] \quad A \neq \mathscr{C}[\mu V.C]}{A \simeq^{co}_{\mu} \mathscr{D}[\mu W.B]}\ (\text{E-REC-R-AL})$$

$$\frac{(A_i \simeq^{co}_{\mu} B_{f(i)})_{i\in 1..n} \quad f : 1..n \to 1..m \qquad (A_i)_{i\in 1..n}, (B_j)_{j\in 1..m} \neq \mu, \oplus \quad n+m>2}{(A_{g(j)} \simeq^{co}_{\mu} B_j)_{j\in 1..m} \quad g : 1..m \to 1..n}\ (\text{E-UNION-AL})$$
$$\oplus_{i\in 1..n} A_i \simeq^{co}_{\mu} \oplus_{j\in 1..m} B_j$$

Fig. 6. Coinductive axiomatization of type equality for contractive μ-types.

```
eqtype(S, A, B) ≜
  if ⟨A, B⟩ ∈ S
    then S
    else let S_0 = S ∪ {⟨A, B⟩} in
      case ⟨A, B⟩ of
        ⟨a, a⟩ →
          S_0
        ⟨A' @ A'', B' @ B''⟩ →
          if A', B' are datatypes
            then let S_1 = eqtype(S_0, A', B') in
              eqtype(S_1, A'', B'')
            else fail
        ⟨A' ⊃ A'', B' ⊃ B''⟩ →
          let S_1 = eqtype(S_0, A', B') in
            eqtype(S_1, A'', B'')
        ⟨𝒞[μV.A'], B⟩ →
          eqtype(S_0, 𝒞[{μV.A'/V} A'], B)
        ⟨A, 𝒟[μW.B']⟩ →
          eqtype(S_0, A, 𝒟[{μW.B'/W} B'])
        ⟨⊕_{i∈1..n} A_i, ⊕_{j∈1..m} B_j⟩ →
          let S_1 = (seq eqtype(S_0, A_1, B_1), ..., eqtype(S_0, A_1, B_m)) in
          ...
          let S_n = (seq eqtype(S_{n-1}, A_n, B_1), ..., eqtype(S_{n-1}, A_n, B_m)) in
          let S_{n+1} = (seq eqtype(S_n, A_1, B_1), ..., eqtype(S_n, A_n, B_1)) in
          ...
          let S_{n+m-1} = (seq eqtype(S_{n+m-2}, A_1, B_{m-1}), ..., eqtype(S_{n+m-2}, A_n, B_{m-1})) in
            seq eqtype(S_{n+m-1}, A_1, B_m), ..., eqtype(S_{n+m-1}, A_n, B_m)
        otherwise →
          fail
```

Fig. 7. Equivalence checking algorithm.

6.2. Subtype checking

The approach to subtype checking is similar to that of type equivalence. Consider the relation $\preceq^{co}_{\mu}$ over μ-types defined in Fig. 8. It captures $\preceq_{\mu}$:

Proposition 6.2. *$A \preceq_{\mu} B$ iff $A \preceq^{co}_{\mu} B$.*

The proof strategy is similar to that of Proposition 6.1. In this case we resort to a proper subtyping relation for infinite trees that essentially results from dropping rules (S-REC-L-AL) and (S-REC-R-AL) in Fig. 8.

Unfortunately, the generating function determined by the rules in Fig. 8, let us call it $\Phi_{\preceq^{co}_{\mu}}$, is not invertible. Notice that (S-UNION-R-AL) overlaps with itself. For example, $\mathbb{c} \preceq^{co}_{\mu} (\mathbb{c} \oplus \mathbb{d}) \oplus (\mathbb{e} \oplus \mathbb{c})$ belongs to two $\Phi_{\preceq^{co}_{\mu}}$-saturated sets (*i.e.* sets $\mathcal{X}$ such that $\mathcal{X} \subseteq \Phi_{\preceq^{co}_{\mu}}(\mathcal{X})$):

$$\begin{aligned}\mathcal{X}_1 &= \{\langle \mathbb{c}, (\mathbb{c} \oplus \mathbb{d}) \oplus (\mathbb{e} \oplus \mathbb{c})\rangle, \langle \mathbb{c}, (\mathbb{c} \oplus \mathbb{d})\rangle, \langle \mathbb{c}, \mathbb{c}\rangle\}\\ \mathcal{X}_2 &= \{\langle \mathbb{c}, (\mathbb{c} \oplus \mathbb{d}) \oplus (\mathbb{e} \oplus \mathbb{c})\rangle, \langle \mathbb{c}, (\mathbb{e} \oplus \mathbb{c})\rangle, \langle \mathbb{c}, \mathbb{c}\rangle\}\end{aligned}$$

However, since this is the only source of non-invertibility we easily derive a subtype membership checking function subtype that, in the case of (S-UNION-R-AL), simply checks all cases (*cf.* [3]).

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

$$\frac{}{a \preceq^{\mathrm{co}}_{\mu} a}\ (\text{S-REFL-AL})$$

$$\frac{D \preceq^{\mathrm{co}}_{\mu} D' \quad A \preceq^{\mathrm{co}}_{\mu} A'}{D \mathbin{@} A \preceq^{\mathrm{co}}_{\mu} D' \mathbin{@} A'}\ (\text{S-COMP-AL}) \qquad \frac{A' \preceq^{\mathrm{co}}_{\mu} A \quad B \preceq^{\mathrm{co}}_{\mu} B'}{A \supset B \preceq^{\mathrm{co}}_{\mu} A' \supset B'}\ (\text{S-FUNC-AL})$$

$$\frac{\{\mu V.A/V\}\, A \preceq^{\mathrm{co}}_{\mu} B}{\mu V.A \preceq^{\mathrm{co}}_{\mu} B}\ (\text{S-REC-L-AL}) \qquad \frac{A \preceq^{\mathrm{co}}_{\mu} \{\mu W.B/W\}\, B \quad A \neq \mu}{A \preceq^{\mathrm{co}}_{\mu} \mu W.B}\ (\text{S-REC-R-AL})$$

$$\frac{(A_i \preceq^{\mathrm{co}}_{\mu} B)_{i\in 1..n} \quad n > 1 \quad B \neq \mu \quad (A_i)_{i\in 1..n} \neq \oplus}{\oplus_{i\in 1..n} A_i \preceq^{\mathrm{co}}_{\mu} B}\ (\text{S-UNION-L-AL})$$

$$\frac{A \preceq^{\mathrm{co}}_{\mu} B_k \text{ for some } k \in 1..m \quad m > 1 \quad A \neq \mu, \oplus \quad (B_j)_{j\in 1..m} \neq \oplus}{A \preceq^{\mathrm{co}}_{\mu} \oplus_{j\in 1..m} B_j}\ (\text{S-UNION-R-AL})$$

Fig. 8. Coinductive axiomatization of subtyping for contractive μ-types.

7. Towards efficient type-checking

The algorithms presented so far are clear but inefficient. The number of recursive calls in eqtype and subtype depends on the size of the type, and unfolding recursive types may increment their size exponentially. This section draws from ideas in [21,25,14] and adopts a dag-representation of recursive types which are encoded as *term automata* (described below). Associativity is handled by resorting to n-ary unions, commutativity and idempotence of $\oplus$ is handled by how types are decomposed in their automaton representation (*cf.* check in Fig. 12). The algorithm itself is tc of Fig. 5 except that:

1. μ-types are represented with term automata, rendering unfold linear.
2. The subtyping algorithm is optimized, based on the new representation of types and following ideas from [25,14].

The end product is an algorithm with complexity $\mathcal{O}(n^7 d)$ where n is the size of the input (*i.e.* that of Γ plus t) and d is the maximum arity of the n-ary unions occurring in Γ and t. Note that all the information needed to type t is either in the context or annotated within the term itself. Thus, a linear relation can be established between the size of the input and the size of the resulting type; and we can think of n as the size of the latter.

7.1. Term automata

μ-types may be understood as finite dags since their infinite unfolding produce regular (infinite) trees. We further simplify the types whose dags we consider by flattening the union type constructor and switching to an alphabet where unions are n-ary: $\mathfrak{L}^* \triangleq \{a^0 \mid a \in \mathcal{V} \cup \mathcal{C}\} \cup \{@^2, \supset^2\} \cup \{\oplus^n \mid n > 1\}$ and we let $\mathfrak{T}^*$ stand for possibly infinite trees whose nodes are symbols in $\mathfrak{L}^*$. μ-types may be interpreted in $\mathfrak{T}^*$ simply by unfolding and then considering maximal union types as their underlying n-ary union types [13]. We write $[\![\bullet]\!]^*$ for this function and use meta-variables $\mathcal{A}, \mathcal{B}, \ldots$ when referring to elements of $\mathfrak{T}^*$. Types in $\mathfrak{T}^*$ may be represented as *term automata* [1].

Definition 7.1. *A **term automaton** is a tuple $\mathcal{M} = \langle Q, \Sigma, q_0, \delta, \ell\rangle$ where:*

1. *Q is a finite set of states.*
2. *Σ is an alphabet where each symbol has an associated arity.*
3. *q_0 is the initial state.*
4. *$\delta : Q \times \mathbb{N} \to Q$ is a partial transition function between states, defined over the arity of the symbol associated by ℓ to the origin state.*
5. *$\ell : Q \to \Sigma$ is a total labelling function over states.*

We write $\mathcal{M}_A$ for the automaton associated to type A. $\mathcal{M}_A$ recognizes all paths from the root of A to any of its sub-expressions. Fig. 9 illustrates an example type, namely $\mathsf{List}_A = \mu\alpha.\mathtt{nil} \oplus (\mathtt{cons} \mathbin{@} A \mathbin{@} \alpha)$, represented as an infinite tree and as a term automaton $\mathcal{M}_{\mathsf{List}_A}$. If q_0 is the initial state of $\mathcal{M}_{\mathsf{List}_A}$ and $\widehat{\delta}$ denotes the natural extension of δ to sequences of symbols, then $\ell(\widehat{\delta}(q_0, 211)) = \mathtt{cons}$. As mentioned, the regular structure of trees arising from types yields automata with a finite number of states.

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

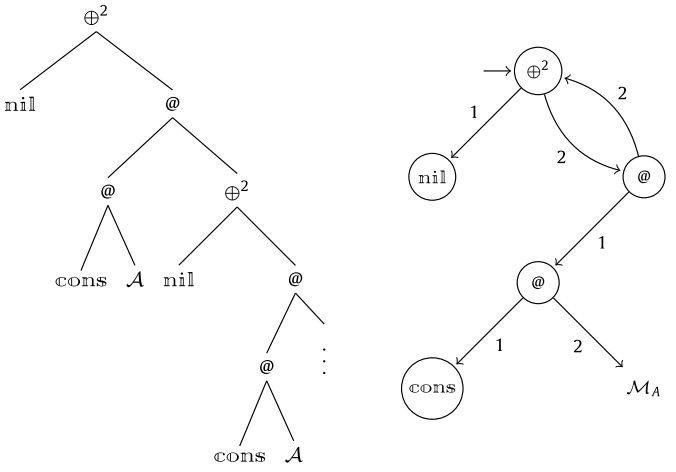

Fig. 9. The type List_A represented as an infinite tree and as a term automaton.

$$\frac{}{a \preceq^{\mathcal{R}}_{\mathfrak{T}^*} a}\ (\text{S-REFL-UP}) \qquad \frac{\mathcal{D}\ (\preceq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{D}' \quad \mathcal{A}\ (\preceq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{A}'}{\mathcal{D}\,@\,\mathcal{A} \preceq^{\mathcal{R}}_{\mathfrak{T}^*} \mathcal{D}'\,@\,\mathcal{A}'}\ (\text{S-COMP-UP})$$

$$\frac{\mathcal{A}'\ (\preceq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{A} \quad \mathcal{B}\ (\preceq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{B}'}{\mathcal{A} \supset \mathcal{B} \preceq^{\mathcal{R}}_{\mathfrak{T}^*} \mathcal{A}' \supset \mathcal{B}'}\ (\text{S-FUNC-UP})$$

$$\frac{(\mathcal{A}_i\ (\preceq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{B}_{f(i)})_{i\in 1..n} \quad f : 1..n \to 1..m \quad (\mathcal{A}_i)_{i\in 1..n}, (\mathcal{B}_j)_{j\in 1..m} \neq \oplus}{\oplus_i^n \mathcal{A}_i \preceq^{\mathcal{R}}_{\mathfrak{T}^*} \oplus_j^m \mathcal{B}_j}\ (\text{S-UNION-UP})$$

$$\frac{(\mathcal{A}_i\ (\preceq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{B})_{i\in 1..n} \quad (\mathcal{A}_i)_{i\in 1..n} \neq \oplus \quad \mathcal{B} \neq \oplus}{\oplus_i^n \mathcal{A}_i \preceq^{\mathcal{R}}_{\mathfrak{T}^*} \mathcal{B}}\ (\text{S-UNION-L-UP})$$

$$\frac{\mathcal{A}\ (\preceq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{B}_k \text{ for some } k \in 1..m \quad \mathcal{A} \neq \oplus \quad (\mathcal{B}_j)_{j\in 1..m} \neq \oplus}{\mathcal{A} \preceq^{\mathcal{R}}_{\mathfrak{T}^*} \oplus_j^m \mathcal{B}_j}\ (\text{S-UNION-R-UP})$$

Fig. 10. Subtyping relation *up-to* $\mathcal{R}$ over $\mathfrak{T}^*$.

7.2. Subtyping and subtype checking

We next present a coinductive notion of subtyping over $\mathfrak{T}^*$. $\preceq^{\mathcal{R}}_{\mathfrak{T}^*}$ is a binary relation defined *up-to* a set of hypothesis $\mathcal{R}$ (Fig. 10). For $\mathcal{R} = \varnothing$, $\preceq^{\mathcal{R}}_{\mathfrak{T}^*}$ coincides with $\preceq_\mu$ modulo the interpretation of types as trees, and hence with $\preceq^{\text{co}}_\mu$ (Proposition 6.2).

Proposition 7.2. *$A \preceq_\mu B$ iff $[\![A]\!]^* \preceq^{\varnothing}_{\mathfrak{T}^*} [\![B]\!]^*$.*

So we can use $\preceq^{\varnothing}_{\mathfrak{T}^*}$ to determine whether types are related via $\preceq_\mu$: take two types, construct their automaton representation and check whether these are related via $\preceq^{\varnothing}_{\mathfrak{T}^*}$. Moreover, our formulation of $\preceq^{\mathcal{R}}_{\mathfrak{T}^*}$ will prove convenient for proving correctness of our subtyping algorithm.

7.2.1. Algorithm description

The algorithm that checks whether types are related by the new subtyping relation builds on ideas from [14]. Our presentation is more general than required for subtyping; this general scheme will also be applicable to type equivalence, as we shall later see. Call $p \in \mathfrak{T}^* \times \mathfrak{T}^*$ *valid* if $p \in \preceq^{\varnothing}_{\mathfrak{T}^*}$. The algorithm consists of two phases. The aim of the first one is to construct a set $U \subseteq \mathfrak{T}^* \times \mathfrak{T}^*$ that delimits the universe of pairs of types that will later be refined to obtain a set of only valid pairs. It starts off with an initial pair (*cf.* Fig. 11, buildUniverse) and then explores pairs of sub-terms of both types in this pair by decomposing the type constructors (*cf.* Fig. 11, children). Note that, given p, the algorithm may add invalid pairs in order to prove the validity of p. The second phase shall be in charge of eliminating these invalid pairs. Note that the first phase can easily be adapted to other relations by simply redefining function children. U may be interpreted as a directed

```
buildUniverse(p_0) :
  U = ∅
  W = {p_0}
  while W ≠ ∅ :
    p := takeOne(W)
    if p ∉ U
      insert(p, U)
      foreach q ∈ children(p)
        insert(q, W)
  return U
```

```
children(p) :
  case p of
    ⟨D @ A, D' @ B⟩ →
        {⟨D, D'⟩, ⟨A, B⟩}
    ⟨A' ⊃ A'', B' ⊃ B''⟩ →
        {⟨B', A'⟩, ⟨A'', B''⟩}
    ⟨⊕_i^n A_i, ⊕_j^m B_j⟩ →
        {⟨A_i, B_j⟩ | i ∈ 1..n, j ∈ 1..m}
    ⟨⊕_i^n A_i, B⟩, B ≠ ⊕ →
        {⟨A_i, B⟩ | i ∈ 1..n}
    ⟨A, ⊕_j^m B_j⟩, A ≠ ⊕ →
        {⟨A, B_j⟩ | j ∈ 1..m}
    otherwise →
        ∅
```

Fig. 11. Pseudo-code of the first phase of the algorithm (building the universe).

```
gfp^≤(p) :
  W = buildUniverse(p)
  S = ∅
  F = ∅
  while W ≠ ∅ :
    q := takeOne(W)
    if check(q, F)
      then insert(q, S)
      else invalidate(q, S, F, W)
  return p ∈ S

invalidate(p, S, F, W) :
  insert(p, F)
  foreach q ∈ parents(p) ∩ S
    move(q, S, W)
```

```
check(p, F) :
  case p of
    ⟨a, a⟩ →
        true
    ⟨D @ A, D' @ B⟩ →
        ⟨D, D'⟩ ∉ F and ⟨A, B⟩ ∉ F
    ⟨A' ⊃ A'', B' ⊃ B''⟩ →
        ⟨B', A'⟩ ∉ F and ⟨A'', B''⟩ ∉ F
    ⟨⊕_i^n A_i, ⊕_j^m B_j⟩ →
        ∀i.∃j.⟨A_i, B_j⟩ ∉ F
    ⟨⊕_i^n A_i, B⟩, B ≠ ⊕ →
        ∀i.⟨A_i, B⟩ ∉ F
    ⟨A, ⊕_j^m B_j⟩, A ≠ ⊕ →
        ∃m.⟨A, B_m⟩ ∉ F
    otherwise →
        false
```

Fig. 12. Pseudo-code of the second phase (relation refinement).

graph where an edge from pair p to $q \in \mathsf{children}(p)$ is added, meaning that q might belong to the support set of p in the final relation $\preceq^{\varnothing}_{\mathfrak{T}^*}$. In this case we say that p is a **parent** of q. Since types represented as term automata could have cycles, a pair may be encountered more than once during the creation of U, meaning that it has more than one parent in the graph. Set $u(p)$ to be the **incoming degree** of p, *i.e.* the number of parents.

During the second phase (Fig. 12, $\mathsf{gfp}^{\preceq}$) the following sets are maintained, all of which conform a partition of U:

- W: pairs whose validity has yet to be determined
- S: pairs considered conditionally valid
- F: invalid pairs

The algorithm repeatedly takes elements in W and, in each iteration, transfers to S the selected pair p if its validity can be proved assuming valid only those pairs which have not been discarded up until now (*i.e.* those in $W \cup S$). Otherwise, p is transferred to F and all of its parents in S need to be reconsidered (their validity up-to W may have changed). Thus these parents are moved back to W (*cf.* Fig. 12, invalidate). Intuitively, S contains elements in $\preceq^{W}_{\mathfrak{T}^*}$. The process ends when W is empty. The only aspect of this second phase specific to $\preceq^{W}_{\mathfrak{T}^*}$ is function check, which may be redefined to be other suitable *up-to* relations.

7.2.2. Correctness

It is based on the fact that S may be considered a set of valid pairs *assuming the validity of those in* W. More generally, the following holds:

Proposition 7.3. *The algorithm preserves the following invariant:*

- $\langle W, S, F\rangle$ *is a partition of* U
- F *is composed solely of invalid pairs*
- $S \subseteq \Phi_{\preceq^{W}_{\mathfrak{T}^*}}(S)$

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

$$\frac{\mathcal{A}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{A}' \quad \mathcal{B}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{B}'}{\mathcal{A} \supset \mathcal{B} \simeq^{\mathcal{R}}_{\mathfrak{T}^*} \mathcal{A}' \supset \mathcal{B}'}\ (\text{E-FUNC-UP})$$

$$\frac{(\mathcal{A}_i\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{B}_{f(i)})_{i\in 1..n} \quad f : 1..n \to 1..m \quad (\mathcal{A}_i)_{i\in 1..n}, (\mathcal{B}_j)_{j\in 1..m} \neq \oplus}{(\mathcal{A}_{g(j)}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{B}_j)_{j\in 1..m} \quad g : 1..m \to 1..n}\ (\text{E-UNION-UP})$$
$$\oplus^n_i \mathcal{A}_i \simeq^{\mathcal{R}}_{\mathfrak{T}^*} \oplus^m_j \mathcal{B}_j$$

$$\frac{(\mathcal{A}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^*} \cup \mathcal{R})\ \mathcal{B}_j)_{j\in 1..m} \quad \mathcal{A} \neq \oplus \quad (\mathcal{B}_j)_{j\in 1..m} \neq \oplus}{\mathcal{A} \simeq^{\mathcal{R}}_{\mathfrak{T}^*} \oplus^m_j \mathcal{B}_j}\ (\text{E-UNION-R-UP})$$

Fig. 13. Equivalence relation *up-to* $\mathcal{R}$ over $\mathfrak{T}^*$ (sample).

When the main cycle ends we know that W is empty, and therefore that $S \subseteq \Phi_{\preceq^{\varnothing}_{\mathfrak{T}^*}}(S)$. The coinduction principle then yields $S \subseteq \preceq^{\varnothing}_{\mathfrak{T}^*}$ (*i.e.* every pair in S is valid) and therefore we are left to verify whether the original pair of types is in S or F.

Corollary 7.4. $\mathcal{A} \preceq^{\varnothing}_{\mathfrak{T}^*} \mathcal{B}$ *iff* $\mathsf{gfp}^{\preceq}(\langle \mathcal{A}, \mathcal{B}\rangle) = \mathtt{true}$.

7.2.3. Complexity

The first phase consists of identifying relevant pairs of sub-terms in both types being evaluated. If we call N and N' the size of such types (considering nodes and edges in their representations) we have that the size and cost of building the universe U can be bounded by $\mathcal{O}(NN')$. As we shall see, the total cost of the algorithm is governed by operations in the second phase.

As stated in [14], since any pair can be invalidated at most once (in which case $u(p)$ nodes are transferred back to W for reconsideration) the amount of iterations in the refinement stage can be bounded by

$$\sum_{p\in U} 1 + \sum_{p\in U} u(p) \quad = \quad \sum_{p\in U}(1 + u(p)) \quad = \quad size(U)$$

Assuming that set operations can be performed in constant time, the cost of evaluating each pair in the main loop is that of deciding whether to suspend or invalidate it and, in the latter case, the cost of the invalidation process. The decision whether to transfer the pair is computed in the function check, which always performs a constant amount of operations on non-union types. The worst case involves checking pairs of the form $\langle \oplus^n_i A_i, \oplus^m_j B_i\rangle$, which can be resolved by maintaining in each node a table indicating, for every A_i, the amount of pairs $\langle A_i, B_j\rangle$ that have not yet been invalidated. Using this approach, this check can be performed in $\mathcal{O}(d)$ operations, where d is a bound on the size of both unions. Whenever a pair is invalidated, all tables present in its immediate parents are updated as necessary.

Finally we resort to an argument introduced in [14] to argue that the cost of invalidating an element can be seen as $\mathcal{O}(1)$: a new iteration will be performed for each of the $u(p)$ pairs added to W when invalidating p. Because of this, a more precise bound for the cost of the complete execution of the algorithm can be obtained if we consider the cost of adding each of these elements to W as part of the iteration itself, yielding an amortized cost of $\mathcal{O}(d)$ operations for each iteration. This gives a total cost of $\mathcal{O}(size(U)d)$ for the subtyping check, *i.e.* $\mathcal{O}(NN'd)$ in terms of the input automata.

Let us call n and n' the amount of constructors in types A and B, respectively. N and N' are the sizes of automata representing these types, and can consequently be bounded by $\mathcal{O}(n^2)$ and $\mathcal{O}(n'^2)$. Therefore, the complexity of the algorithm can be expressed as $\mathcal{O}(n^2 n'^2 d)$.

7.3. Equivalence checking

In this section we adapt the previous algorithm to obtain one for equivalence checking with the same complexity cost. The equivalence relation $\simeq^{\mathcal{R}}_{\mathfrak{T}^*}$ *up-to* $\mathcal{R}$ results from replacing rules (S-FUNC-UP), (S-UNION-UP), and (S-UNION-R-UP) from Fig. 10 by their counterparts in Fig. 13.

Proposition 7.5. $A \simeq_\mu B$ *iff* $[\![A]\!]^* \simeq^{\varnothing}_{\mathfrak{T}^*} [\![B]\!]^*$.

The algorithm is the result of adapting the scheme presented for subtyping to the new relation $\simeq^{\mathcal{R}}_{\mathfrak{T}^*}$. This is done by redefining functions children and check from the first and second phase respectively (*cf.* Fig. 14). For the former the only difference is on rule (E-FUNC-UP), where we need to add pair $\langle \mathcal{A}', \mathcal{B}'\rangle$ instead of $\langle \mathcal{B}', \mathcal{A}'\rangle$, added for subtyping. This could have been avoided by resorting the symmetry of $\simeq^{\mathcal{R}}_{\mathfrak{T}^*}$, but is meant to emphasize the fact that phase one can easily be adapted if needed. For the refinement phase we need to properly check the premises of rules (E-UNION-UP) and (E-UNION-R-UP), while the others remain the same.

With these considerations is easy to see that, in each iteration, S consists of pairs in the relation $\simeq^{W}_{\mathfrak{T}^*}$, getting $S \subseteq \simeq^{\varnothing}_{\mathfrak{T}^*}$ at the end of the process.

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

```
children(p) :
  case p of
    ⟨D@A, D'@B⟩ →
        {⟨D, D'⟩, ⟨A, B⟩}
    ⟨A' ⊃ A'', B' ⊃ B''⟩ →
        {⟨A', B'⟩, ⟨A'', B''⟩}
    ⟨⊕_i^n A_i, ⊕_j^m B_j⟩ →
        {⟨A_i, B_j⟩ | i ∈ 1..n, j ∈ 1..m}
    ⟨⊕_i^n A_i, B⟩, B ≠ ⊕ →
        {⟨A_i, B⟩ | i ∈ 1..n}
    ⟨A, ⊕_j^m B_j⟩, A ≠ ⊕ →
        {⟨A, B_j⟩ | j ∈ 1..m}
    otherwise →
        ∅
```

```
check(p, F) :
  case p of
    ⟨a, a⟩ →
        true
    ⟨D@A, D'@B⟩ →
        ⟨D, D'⟩ ∉ F and ⟨A, B⟩ ∉ F
    ⟨A' ⊃ A'', B' ⊃ B''⟩ →
        ⟨A', B'⟩ ∉ F and ⟨A'', B''⟩ ∉ F
    ⟨⊕_i^n A_i, ⊕_j^m B_j⟩ →
        ∀i.∃j.⟨A_i, B_j⟩ ∉ F and ∀j.∃i.⟨A_i, B_j⟩ ∉ F
    ⟨⊕_i^n A_i, B⟩, B ≠ ⊕ →
        ∀i.⟨A_i, B⟩ ∉ F
    ⟨A, ⊕_j^m B_j⟩, A ≠ ⊕ →
        ∀j.⟨A, B_j⟩ ∉ F
    otherwise →
        false
```

Fig. 14. Pseudo-code of first (left) and second (right) phase for equivalence checking.

Proposition 7.6. *The algorithm preserves the following invariant:*

- $\langle W, S, F\rangle$ *is a partition of* U
- F *is composed solely of invalid pairs*
- $S \subseteq \Phi_{\simeq^W_{\mathfrak{T}^*}}(S)$

Correctness of the algorithmic presentation of type equivalence follows just like for subtyping (Corollary 7.4), where $\mathsf{gfp}^{\simeq}$ behaves similar to $\mathsf{gfp}^{\preceq}$ except that the check function is adapted for type equivalence.

Corollary 7.7. $\mathcal{A} \simeq^{\varnothing}_{\mathfrak{T}^*} \mathcal{B}$ *iff* $\mathsf{gfp}^{\simeq}(\langle \mathcal{A}, \mathcal{B}\rangle) = \mathtt{true}$.

For the complexity analysis, recall that uppercase N denotes the size of the automata representing a type. Notice that the size of the built universe is the same as before and overall cost is governed by phase two, which has at most $\mathcal{O}(NN')$ iterations. For the cost of each iteration it is enough to analyze the complexity of check, since the rest of the scheme remains the same. As we remarked before, the only difference in check between subtyping and equality is in the cases involving unions. Here the worst case is when checking rule (E-UNION-UP) that requires the existence of two functions f and g relating elements of each type. This can be done in linear time by maintaining tables with the count of non-invalidated pairs of descendants, as indicated in Sec. 7.2.3. Thus, the cost of an iteration is $\mathcal{O}(d)$, resulting in an overall cost of $\mathcal{O}(NN'd)$ as before.

7.4. Type checking

Let us revisit type-checking (tc). As already discussed, it linearly traverses the input term, the most costly operations being those that deal with application and abstraction. These cases involve calling subtype. Notice that these calls do not depend directly on the input to tc. However, a linear correspondence can be established between the size of the types being considered in subtype and the input to the algorithm, since such expressions are built from elements of Γ (the input context) or from annotations in the input term itself. Consider for instance $\mathsf{subtype}(\varnothing, A, B)$ with a and b the size of each type resp. This has complexity $\mathcal{O}(a^2b^2d)$ and, from the discussion above, we can refer to it as $\mathcal{O}(n^4d)$, where n is the size of the input to tc (*i.e.* that of Γ plus t). Similarly, we may say that unfold is linear in n.

We now analyze the application and abstraction cases in detail:

Application First it performs a linear check on the type to verify if it is a datatype. If so it returns. Otherwise, a second linear check is required (unfold) in order to then perform as many calls to subtype as elements there are in the union of the functional types. This yields a local complexity of $\mathcal{O}(n^4d^2)$.

Abstraction First there are as many calls to tcp (the algorithm for type-checking patterns) as branches the abstraction has. Note that tcp has linear complexity in the size of its input and this call is instantiated with arguments p_i and θ_i which occur in the original term. All these calls, taken together, may thus be considered to have linear time complexity with respect to the input of tcp. Then it is necessary to perform a quadratic number (in the number of branches) of checks on compatibility. We have already analyzed that compatibility in the worst case involves checking subtyping. If we assume a linear number of branches w.r.t. the input, we obtain a total complexity of $\mathcal{O}(n^6d)$ for this case.

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

Finally, the total complexity of tc is governed by the case of the abstraction, and is therefore $\mathcal{O}(n^7 d)$.

7.5. Prototype implementation

A prototype in Scala is available [16]. It implements tc and resorts to the efficient algorithm for subtyping and type equivalence described above. It also includes further optimizations. For example, following a suggestion in [14], the order in which elements in W are selected for evaluation relies on detecting strongly connected components, using Tarjan's [15] algorithm of linear cost and topologically sorting them in reverse order. In the absence of cycles this results in evaluating every pair only after all its children have already been considered. For cyclic types, pairs for which no order can be determined are encapsulated within the same strongly connected component.

8. Conclusions

A type system is proposed for a calculus that supports path polymorphism and the fundamental property of Safety is addressed. The system includes type application, constants as types, union and recursive types. This result relies crucially on a novel notion of *pattern compatibility* and on the invertibility of subtyping for μ-types.

For the proposed system, efficient type-checking is also addressed. This requires the formulation of a syntax-directed variant of the system. Also, invertible coinductive formalizations of type equivalence and subtyping are devised. The first naïve but correct type-checking algorithm proved to be inefficient (exponential in the size of the type). However, efficiency is considerably improved by adopting a representation of type expressions based on automata techniques. Indeed, the final algorithm has polynomial complexity in the size of the input.

Regarding future work an outline of possible avenues follows. These are aimed at enhancing the expressiveness of CAP itself and then adapting the techniques presented here to obtain efficient type checking algorithms.

- The addition of parametric polymorphism (*e.g.* in the style of $F_{<:}$ [8,27,11]).
- 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 [5, page 515].
- Exploring connections between our use of term automata for subtype checking and tree automata [12].
- 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] R.M. Amadio, L. Cardelli, Subtyping recursive types, ACM Trans. Program. Lang. Syst. 15 (4) (1993) 575–631.
[2] Z.M. Ariola, J.W. Klop, Equational term graph rewriting, Fund. Inform. 26 (3/4) (1996) 207–240.
[3] M. Ayala-Rincón, E. Bonelli, J. Edi, A. Viso, Typed path polymorphism. Extended report, https://drive.google.com/open?id=1d4d8mA05KLgxBuCPbuGZRSE3T2GglPYz, 2017.
[4] M. Ayala-Rincón, E. Bonelli, A. Viso, Type soundness for path polymorphism, Electron. Notes Theor. Comput. Sci. 323 (2016) 235–251.
[5] H.P. Barendregt, W. Dekkers, R. Statman, Lambda Calculus with Types. Perspectives in Logic, Cambridge University Press, 2013.
[6] M. Brandt, F. Henglein, Coinductive axiomatization of recursive type equality and subtyping, in: P. de Groote (Ed.), Proceedings of the TLCA '97, Nancy, France, April 2–4, 1997, in: Lecture Notes in Computer Science, vol. 1210, Springer, 1997, pp. 63–81.
[7] M. Brandt, F. Henglein, Coinductive axiomatization of recursive type equality and subtyping, Fund. Inform. 33 (4) (1998) 309–338.
[8] L. Cardelli, S. Martini, J.C. Mitchell, A. Scedrov, An extension of system F with subtyping, in: T. Ito, A.R. Meyer (Eds.), Proceedings of the TACS '91, Sendai, Japan, September 24–27, 1991, in: Lecture Notes in Computer Science, vol. 526, Springer, 1991, pp. 750–770.
[9] F. Cardone, An algebraic approach to the interpretation of recursive types, in: J.-C. Raoult (Ed.), CAAP, in: Lecture Notes in Computer Science, vol. 581, Springer, 1992, pp. 66–85.
[10] H. Cirstea, C. Kirchner, The rewriting calculus – part I and II, Log. J. IGPL 9 (3) (2001) 339–410.
[11] D. Colazzo, G. Ghelli, Subtyping recursion and parametric polymorphism in kernel fun, Inform. and Comput. 198 (2) (2005) 71–147.
[12] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, M. Tommasi, Tree automata techniques and applications, http://tata.gforge.inria.fr, 2008.
[13] B. Courcelle, Fundamental properties of infinite trees, Theoret. Comput. Sci. 25 (1983) 95–169.
[14] R. Di Cosmo, F. Pottier, D. Rémy, Subtyping recursive types modulo associative commutative products, in: P. Urzyczyn (Ed.), Proceedings of the TLCA 2005, Nara, Japan, April 21–23, 2005, in: Lecture Notes in Computer Science, vol. 3461, Springer, 2005, pp. 179–193.
[15] P.J. Downey, R. Sethi, R.E. Tarjan, Variations on the common subexpression problem, J. ACM 27 (4) (1980) 758–771.
[16] J. Edi, A. Viso, Prototype implementation of efficient type-checker in scala, https://github.com/juanedi/cap-typechecking, 2016.
[17] J. Edi, A. Viso, E. Bonelli, Efficient type checking for path polymorphism, in: T. Uustalu (Ed.), 21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18–21, 2015, Tallinn, Estonia, in: LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, vol. 69, 2015, pp. 6:1–6:23.
[18] B. Jay, Pattern Calculus – Computing with Functions and Structures, Springer, 2009.
[19] B. Jay, D. Kesner, Pure pattern calculus, in: P. Sestoft (Ed.), ESOP, in: Lecture Notes in Computer Science, vol. 3924, Springer, 2006, pp. 100–114.
[20] B. Jay, D. Kesner, First-class patterns, J. Funct. Programming 19 (2) (2009) 191–225.
[21] T. Jim, J. Palsberg, Type Inference in Systems of Recursive Types with Subtyping, 1997.
[22] W. Kahl, Basic pattern matching calculi: a fresh view on matching failure, in: Y. Kameyama, P.J. Stuckey (Eds.), Proceedings of the Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7–9, 2004, in: Lecture Notes in Computer Science, vol. 2998, Springer, 2004, pp. 276–290.
[23] J.W. Klop, V. van Oostrom, R.C. de Vrijer, Lambda calculus with patterns, Theoret. Comput. Sci. 398 (1–3) (2008) 16–31.
[24] D. Kozen, J. Palsberg, M.I. Schwartzbach, Efficient recursive subtyping, Math. Structures Comput. Sci. 5 (1) (1995) 113–125.
[25] J. Palsberg, T. Zhao, Efficient and flexible matching of recursive types, Inform. and Comput. 171 (2) (2001) 364–387.

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018

[26] B. Petit, Semantics of typed lambda-calculus with constructors, Log. Methods Comput. Sci. 7 (1) (2011).
[27] B.C. Pierce, Types and Programming Languages, MIT Press, 2002.
[28] J. Vouillon, Subtyping union types, in: J. Marcinkowski, A. Tarlecki (Eds.), CSL, in: Lecture Notes in Computer Science, vol. 3210, Springer, 2004, pp. 415–429.
[29] T. Zhao, Type Matching and Type Inference for Object-Oriented Systems, Ph.D. thesis, Computer Science, Purdue University, 2002.

Please cite this article in press as: M. Ayala-Rincón et al., Typed path polymorphism, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.02.018