View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Dagstuhl Research Online Publication Server

Efficient Type Checking for Path Polymorphism*

Juan Edi[1], Andrés Viso[2], and Eduardo Bonelli[3]

1 Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Pabellón I, Ciudad Universitaria, Buenos Aires C1428EGA, Argentina
jedi@dc.uba.ar

2 Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina and
Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Pabellón I, Ciudad Universitaria, Buenos Aires C1428EGA, Argentina
aeviso@gmail.com

3 Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina and
Departamento de Ciencia y Tecnología, Universidad Nacional de Quilmes, Roque Sáenz Peña 352, Bernal B1876BXD, Argentina
eabonelli@gmail.com

Abstract

A type system combining type application, constants as types, union types (associative, commutative and idempotent) and recursive types has recently been proposed for statically typing *path polymorphism*, the ability to define functions that can operate uniformly over recursively specified applicative data structures. A typical pattern such functions resort to is $x\,y$ which decomposes a compound, in other words any applicative tree structure, into its parts. We study type-checking for this type system in two stages. First we propose algorithms for checking type equivalence and subtyping based on coinductive characterizations of those relations. We then formulate a syntax-directed presentation and prove its equivalence with the original one. This yields a type-checking algorithm which unfortunately has exponential time complexity in the worst case. A second algorithm is then proposed, based on automata techniques, which yields a polynomial-time type-checking algorithm.

1998 ACM Subject Classification F.4.1 Mathematical Logic and Formal Languages: Lambda Calculus and Related Systems; F.3.2 Logics and Meanings of Programs: Semantics of Programming Languages; D.3.3 Programming Languages: Language Constructs and Features

Keywords and phrases λ-calculus, pattern matching, path polymorphism, type checking

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.6

1 Introduction

The *lambda-calculus* plays an important role in the study of programming languages (PLs). Programs are represented as syntactic terms and execution by repeated simplification of these terms using a reduction rule called *β-reduction*. The study of the lambda-calculus has produced deep results in both the theory and the implementation of PLs. Many variants of the lambda-calculus have been introduced with the purpose of studying specific PL features.

* A full version of the paper is available at [9], https://arxiv.org/abs/1704.09026.

© Juan Edi, Andrés Viso, and Eduardo Bonelli;
licensed under Creative Commons License CC-BY
21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editor: Tarmo Uustalu; Article No. 6; pp. 6:1–6:24

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

One such feature of interest is *pattern-matching*. Pattern-matching is used extensively in PLs as a means for writing more succinct and, at the same time, elegant programs. This is particularly so in the functional programming community, but by no means restricted to that community.

In the standard 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 expressed by the above mentioned 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. Note that, in this rule, no requirement on the form of s is placed. *Pattern calculi* 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 $\lambda\langle x, y\rangle.x$ for projecting the first component of a pair, the pattern p being $\langle x, y\rangle$. An expression such as $(\lambda\langle x, y\rangle.x)\, s$ will only be able to reduce if s indeed is of the form $\langle s_1, s_2\rangle$; it will otherwise be blocked.

Patterns may be catalogued in at least two dimensions. One is their *structure* and another their *time of creation*. The structure of patterns may be very general. Such is the case of variables: any term can match a variable, as in the standard lambda-calculus. The structure of a pattern may also be very specific. Such is the case when arbitrary terms are allowed to be patterns [13, 17]. Regarding the time of creation, patterns may either be *static* or *dynamic*. Static patterns are those that are created at compile time, such as the pattern $\langle x, y\rangle$ mentioned above. Dynamic patterns are those that may be generated at run-time [10, 11]. For example, consider the term $\lambda x.(\lambda(x\, y).y)$; note that it has an occurrence of a pattern $x\, y$ with a free variable, namely the x in $x\, y$, that is bound to the outermost lambda. If this term is applied to a constant $\mathtt{c}$, then one obtains $\lambda \mathtt{c}\, y.y$. However, if we apply it to the constant $\mathtt{d}$, then we obtain $\lambda \mathtt{d}\, y.y$. Both patterns $\mathtt{c}\, y$ and $\mathtt{d}\, y$ are created during execution. Note that one could also replace the x in the pattern $x\, y$ with an abstraction. This leads to computations that evaluate to patterns.

Expressive pattern features may easily break desired properties, such as confluence, and are not easy to endow with type systems. This work is an attempt at devising type systems for such expressive pattern calculi. We originally set out to type-check the *Pure Pattern Calculus* (PPC) [10, 11]. PPC is a lambda-calculus that embodies the essence of dynamic patterns by stripping away everything inessential to the reduction and matching process of dynamic patterns. It admits terms such as $\lambda x.(\lambda(x\, y).y)$. We soon realized that typing PPC was too challenging and noticed that the static fragment of PPC, which we dub *Calculus of Applicative Patterns* (CAP), was already challenging in itself. CAP also admits patterns such as $x\, y$ however all variables in this pattern are considered *bound*. Thus, in a term such as $\lambda(x\, y).s$ both occurrences of x and y are bound in s, disallowing reduction inside patterns. Such patterns, however, allow arguments that are applications to be decomposed, as long as these applications encode *data structures*. They are therefore useful for writing functions that operate on *semi-structured data*.

The main obstacle for typing CAP is dealing in the type system with a form of polymorphism called *path polymorphism* [10, 11], that arises from these kinds of patterns. We next briefly describe path polymorphism and the requirements it places on typing considerations.

Path Polymorphism. In CAP data structures are trees. These trees are built using application and variable arity constants or constructors. Examples of two such trees follow, where the first one represents a list and the second a binary tree:

$$\mathtt{cons}\,(\mathtt{vl}\,1)\,(\mathtt{cons}\,(\mathtt{vl}\,2)\,\mathtt{nil})$$
$$\mathtt{node}\,(\mathtt{vl}\,3)\,(\mathtt{node}\,(\mathtt{vl}\,4)\,\mathtt{nil}\,\mathtt{nil})\,(\mathtt{node}\,(\mathtt{vl}\,5)\,\mathtt{nil}\,\mathtt{nil})$$

The constructor vl is used to tag values (1 and 2 in the first case, and 3, 4 and 5 in the

second). A "map" function for updating the values of any of these two structures by applying some user-supplied function f follows, where type annotations are omitted for clarity:

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

The expression $\mathsf{upd}\,(+1)$ may thus be applied to any of the two data structures illustrated above. For example, we can evaluate $\mathsf{upd}\,(+1)\,\mathtt{cons}\,(\mathtt{vl}\,1)\,(\mathtt{cons}\,(\mathtt{vl}\,2)\,\mathtt{nil})$ and also $\mathsf{upd}\,(+1)\,\mathtt{node}\,(\mathtt{vl}\,3)\,(\mathtt{node}\,(\mathtt{vl}\,4)\,\mathtt{nil}\,\mathtt{nil})\,(\mathtt{node}\,(\mathtt{vl}\,5)\,\mathtt{nil}\,\mathtt{nil})$. The expression to the right of "=" is called an *abstraction* (or *case*) and consists of a unique *branch*; this branch in turn is formed from a pattern (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 $x\, y$. During evaluation of $\mathsf{upd}\,(+1)\,\mathtt{cons}\,(\mathtt{vl}\,1)\,(\mathtt{cons}\,(\mathtt{vl}\,2)\,\mathtt{nil})$ the variables x and y may be instantiated with different applicative terms in each recursive call to upd. For example:

	x	y
$\mathsf{upd}\,(+1)\,s$	$\mathtt{cons}\,(\mathtt{vl}\,1)$	$\mathtt{cons}\,(\mathtt{vl}\,2)\,\mathtt{nil}$
$\mathsf{upd}\,(+1)\,(\mathtt{cons}\,(\mathtt{vl}\,1))$	$\mathtt{cons}$	$\mathtt{vl}\,1$
$\mathsf{upd}\,(+1)\,(\mathtt{cons}\,(\mathtt{vl}\,2)\,\mathtt{nil})$	$\mathtt{cons}\,(\mathtt{vl}\,2)$	$\mathtt{nil}$

The type assigned to x and y should encompass all terms in its respective column.

Singleton Types and Type Application. A further consideration in typing CAP is that terms such as the ones depicted below should clearly not be typable.

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

In the first case, $\mathtt{cons}$ will never match $\mathtt{nil}$. The type system will resort to singleton types in order to determine this: $\mathtt{cons}$ will be assigned a type of the form $\mathbb{cons}$ which will fail to match $\mathbb{nil}$. The second expression in (2) breaks *Subject Reduction* (SR): reduction will produce $\mathtt{true} + 1$. Applicative types of the form $\mathbb{vl} \mathbin{@} \mathbb{true}$ will allow us to check for these situations, @ being a new type constructor that applies datatypes to arbitrary types. Also, note the use of typing environments (the expression $\{x : \mathsf{Nat}\}$) to declare the types of the variables of patterns in branches. These are supplied by the programmer.

Union and Recursive Types. On the assumption that the programmer has provided an exhaustive coverage, the type assigned by CAP to the variable x in the pattern $x\, y$ in upd is:

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

Here μ is the recursive type constructor and $\oplus$ the union type constructor. $\mathbb{vl}$ is the singleton type used for typing the constant $\mathtt{vl}$ and @ denotes type application, as mentioned above. The union type constructor is used to collect the types of all the branches. The variable y in the pattern $x\, y$ will also be assigned the same type as x. Thus variables in applicative patterns are assigned union types. upd itself is assigned type $(A \supset B) \supset (F_A \supset F_B)$, where F_X is $\mu\alpha.(\mathbb{vl} \mathbin{@} X) \oplus (\alpha \mathbin{@} \alpha) \oplus (\mathbb{cons} \oplus \mathbb{node} \oplus \mathbb{nil})$.

Type-Checking for CAP. Based on these, and other similar considerations, we proposed *typed CAP* [18], referred to simply as CAP in the sequel. The system consists of typing rules that combine singleton types, type application, union types, recursive types and subtyping. Also it enjoys several properties, the salient one being safety (subject reduction and progress). Safety relies on a notion of *typed pattern compatibility* based on subtyping that guarantees that examples such as (2–right) and the following one do not break safety:

$$((\mathtt{vl}\,x \to_{\{x:\mathsf{Bool}\}} \mathtt{if}\ x\ \mathtt{then}\ 1\ \mathtt{else}\ 0) \mid (\mathtt{vl}\,y \to_{\{y:\mathsf{Nat}\}} y+1))\,(\mathtt{vl}\,4) \tag{3}$$

Assumptions on associativity and commutativity of typing operators in [18], make it non-trivial to deduce a type-checking algorithm from the typing rules. The proposed type system is, moreover, not syntax-directed. Also, it relies on coinductive notions of type equivalence and subtyping which in the presence of recursive subtypes are not obviously decidable. A practical implementation of CAP is instrumental since a robust theoretical analysis without such an implementation is of little use.

Goal and Summary of Contributions. This paper addresses this implementation. It does so 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 an invertible presentation of the coinductive notions of type-equivalence and subtyping of [18] and also a syntax-directed variant of the presentation in [18]. This leads to algorithms for checking subtyping membership and equivalence modulo associative, commutative and idempotent (ACI) unions, both based on an invertible presentation of the functional generating the associated coinductive 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. For literature on (typed) pattern calculi the reader is referred to [18]. The algorithms for checking equality of recursive types or subtyping of recursive types have been studied in the 1990s by Amadio and Cardelli [1]; Kozen, Palsberg, and Schwartzbach [14]; Brandt and Henglein [3]; Jim and Palsberg [12] among others. Additionally, Zhao and Palsberg [15] 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 [20]. Later on Di Cosmo, Pottier, and Rémy [6] 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 reviews the syntax and operational semantics of CAP, its type system and the main properties. Further details may be consulted in [18]. Sec. 3 proposes invertible generating functions for coinductive notions of type-equivalence and subtyping that lead to inefficient but elegant algorithms for checking these relations. Sec. 4 proposes a syntax-directed type system for CAP. Sec. 5 studies a more efficient type-checking algorithm based on automaton. Finally, we conclude in Sec. 6. Full details of all omitted proofs may be found in an extended report [9]. An implementation of the algorithms described here is available online [8].

2 Review of CAP

2.1 Syntax and Operational Semantics

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

$$
\begin{array}{llll}
p & ::= & x & \text{(matchable)} \\
 & | & \mathsf{c} & \text{(constant)} \\
 & | & p\,p & \text{(compound)}
\end{array}
\qquad
\begin{array}{llll}
t & ::= & x & \text{(variable)} \\
 & | & \mathsf{c} & \text{(constant)} \\
 & | & t\,t & \text{(application)} \\
 & | & p \to_\theta t \mid \ldots \mid p \to_\theta t & \text{(abstraction)}
\end{array}
$$

$$
\begin{array}{llll}
d & ::= & \mathsf{c} & \text{(constant)} \\
 & | & d\,t & \text{(compound)}
\end{array}
\qquad
\begin{array}{llll}
m & ::= & d & \text{(data structure)} \\
 & | & p \to_\theta t \mid \ldots \mid p \to_\theta t & \text{(abstraction)}
\end{array}
$$

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 variables in p_i (*cf.* Sec. 2.3). The **free variables** of a term t (notation $\mathsf{fv}(t)$) are defined as expected; in a pattern p we call them **free matchables** ($\mathsf{fm}(p)$). All free matchables in each p_i are assumed to be bound in their respective bodies s_i. Positions in patterns and terms are defined as expected and denoted $\pi, \pi', \ldots$ (ϵ denotes the root position). We write $\mathsf{pos}(s)$ for the set of positions of s and $s|_\pi$ for the subterm of s occurring at position π.

A **substitution** $(\sigma, \sigma_i, \ldots)$ is a partial function from term variables to terms. If it assigns u_i to x_i, $i \in 1..n$, then we write $\{u_1/x_1, \ldots, u_n/x_n\}$. Its domain ($\mathsf{dom}\,(\sigma)$) is $\{x_1, \ldots, x_n\}$. Also, $\{\}$ is the identity substitution. We write σs for the result of applying σ to term s. We say a pattern p **subsumes** a pattern q, written $p \lhd q$ if there exists σ such that $\sigma p = q$. **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):

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

where o denotes any possible output and $\sigma_1 \uplus \sigma_2 \triangleq \sigma$ if the domains of σ_1 and σ_2 are disjoint. This always holds given that patterns are assumed to be linear (at most one occurrence of any matchable). The matching operation is defined as follows, where the defining clauses below are evaluated from top to bottom[1]:

[1] This is simplification to the static patterns case of the matching operation introduced in [10].

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

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

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

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

For instance, consider the function

$$
\mathsf{head} = ((\mathtt{nil} \to_{\{\}} \mathtt{nothing}) \mid (\mathtt{cons}\,x\,xs \to_{\{x:\mathsf{Nat},xs:\mu\alpha.\mathbb{nil}\oplus\mathbb{cons}@\mathsf{Nat}@\alpha\}} \mathtt{just}\,x))
$$

Then, $\mathsf{head}\,\mathtt{nil} \to \mathtt{nothing}$ with $\{\!\{\mathtt{nil}/\mathtt{nil}\}\!\} = \{\}$, while $\mathsf{head}\,(\mathtt{cons}\,4\,\mathtt{nil}) \to \mathtt{just}\,4$ since $\{\!\{\mathtt{cons}\,x\,\mathtt{nil}/\mathtt{nil}\}\!\} = \mathtt{fail}$ and $\{\!\{\mathtt{cons}\,4\,\mathtt{nil}/\mathtt{cons}\,x\,xs\}\!\} = \{4/x, \mathtt{nil}/xs\}$.

▶ **Proposition 1.** *Reduction in CAP is confluent [18].*

2.2 Types

In order to ensure that patterns such as $x\,y$ decompose only data structures rather than arbitrary terms, we shall introduce two sorts of typing expressions: *types* and *datatypes*, the latter being strictly included in the former. We assume given countably infinite sets $\mathcal{V}_D$ of **datatype variables** $(\alpha, \beta, \ldots)$, $\mathcal{V}_A$ of **type variables** $(X, Y, \ldots)$ and $\mathcal{C}$ of **type constants** $(\mathbb{c}, \mathbb{d}, \ldots)$. We define $\mathcal{V} \triangleq \mathcal{V}_A \cup \mathcal{V}_D$ and use meta-variables $V, W, \ldots$ to denote an arbitrary element in it. Likewise, we write $a, b, \ldots$ for elements in $\mathcal{V} \cup \mathcal{C}$. The sets $\mathcal{T}_D$ of **μ-datatypes** and $\mathcal{T}$ of **μ-types**, resp., are inductively defined as follows:

$$
\begin{array}{lcll}
D & ::= & \alpha & \text{(datatype variable)} \\
& \mid & \mathbb{c} & \text{(atom)} \\
& \mid & D\ @\ A & \text{(compound)} \\
& \mid & D \oplus D & \text{(union)} \\
& \mid & \mu\alpha.D & \text{(recursion)}
\end{array}
\qquad
\begin{array}{lcll}
A & ::= & X & \text{(type variable)} \\
& \mid & D & \text{(datatype)} \\
& \mid & A \supset A & \text{(type abstraction)} \\
& \mid & A \oplus A & \text{(union)} \\
& \mid & \mu X.A & \text{(recursion)}
\end{array}
$$

▶ Remark. A type of the form $\mu\alpha.A$ is not valid in general since it may produce invalid unfoldings. For example, $\mu\alpha.\alpha \supset \alpha = (\mu\alpha.\alpha \supset \alpha) \supset (\mu\alpha.\alpha \supset \alpha)$, which fails to preserve sorting. On the other hand, types of the form $\mu X.D$ are not necessary since they denote the solution to the equation $X = D$, hence X is a variable representing a datatype, a role already fulfilled by α.

We consider $\oplus$ to bind tighter than $\supset$, while @ binds tighter than $\oplus$. *E.g.* $D\ @\ A \oplus A' \supset B$ means $((D\ @\ A) \oplus A') \supset B$. We write $A \neq \oplus$ to mean that the root symbol of A is different from $\oplus$; and similarly with the other type constructors. Expressions such as $A_1 \oplus \ldots \oplus A_n$ will be abbreviated $\bigoplus_{i\in 1..n} A_i$; this is sound since μ-types will be considered modulo associativity of $\oplus$. A type of the form $\bigoplus_{i\in 1..n} A_i$ where each $A_i \neq \oplus$, $i \in 1..n$, is called a **maximal union**.

$$\frac{}{\vdash A \simeq_\mu A}\ (\text{E-REFL}) \qquad \frac{\vdash A \simeq_\mu B \quad \vdash B \simeq_\mu C}{\vdash A \simeq_\mu C}\ (\text{E-TRANS}) \qquad \frac{\vdash A \simeq_\mu B}{\vdash B \simeq_\mu A}\ (\text{E-SYMM})$$

$$\frac{\vdash D \simeq_\mu D' \quad \vdash A \simeq_\mu A'}{\vdash D @ A \simeq_\mu D' @ A'}\ (\text{E-COMP}) \qquad \frac{\vdash A \simeq_\mu A' \quad \vdash B \simeq_\mu B'}{\vdash A \supset B \simeq_\mu A' \supset B'}\ (\text{E-FUNC})$$

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

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

$$\frac{\vdash A \simeq_\mu A' \quad \vdash B \simeq_\mu B'}{\vdash A \oplus B \simeq_\mu A' \oplus B'}\ (\text{E-UNION}) \qquad \frac{\vdash A \simeq_\mu B}{\vdash \mu V.A \simeq_\mu \mu V.B}\ (\text{E-REC})$$

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

Figure 1 Type equivalence for μ-types.

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, $\mathbb{c}$, $D @ A'$, 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. For instance, $\mu X.X \supset \mathbb{c}$, $\mu X.X \supset X$ and $\mu X.\mathbb{c} @ X \oplus X$ are contractive while $\mu X.X$ and $\mu X.X \oplus X$ are not. We henceforth redefine $\mathcal{T}$ to be the set of **contractive μ-types**.

μ-types come equipped with a notion of **type equivalence** $\simeq_\mu$ (Fig. 1) and **subtyping** $\preceq_\mu$ (Fig. 2). In Fig. 2 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-REC) actually encodes two rules, one for datatypes ($\mu\alpha.D$) and one for arbitrary types ($\mu X.A$). Likewise for (E-FOLD) and (E-CONTR). Regarding the subtyping rules, we adopt those for union of [19]. 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 $A \preceq_\mu B$.

2.3 Typing and Safety

A **typing context** Γ (or θ) is a partial function from term variables to μ-types; $\Gamma(x) = A$ means that Γ maps x to A. We have two typing judgments, one for patterns $\theta \vdash_{\mathsf{p}} p : A$ and one for terms $\Gamma \vdash s : A$. Accordingly, we have two sets of typing rules: Fig. 3, top and bottom. We write $\rhd\, \theta \vdash_{\mathsf{p}} p : A$ to indicate that the typing judgment $\theta \vdash_{\mathsf{p}} p : A$ is derivable (likewise for $\rhd\, \Gamma \vdash s : A$). The typing schemes speak for themselves except for two of them which we now comment. The first is (T-APP). Note that we do not impose any additional restrictions on A_i, in particular it may be a union-type itself. This implies that the argument u can have a union type too. Regarding (T-ABS) it requests a number of conditions. First of all, each of the patterns p_i must be typable under the typing context θ_i, $i \in 1..n$. Also, the set of free matchables in each p_i must be exactly the domain of θ_i. Another condition, indicated by $(\Gamma, \theta_i \vdash s_i : B)_{i \in 1..n}$, is that the bodies of each of the branches s_i, $i \in 1..n$, must

$$\frac{}{\Sigma \vdash A \preceq_\mu A}\ (\text{S-REFL}) \qquad \frac{}{\Sigma, V \preceq_\mu W \vdash V \preceq_\mu W}\ (\text{S-HYP}) \qquad \frac{\vdash A \simeq_\mu B}{\Sigma \vdash A \preceq_\mu B}\ (\text{S-EQ})$$

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

$$\frac{\Sigma \vdash A \preceq_\mu A' \quad \Sigma \vdash B \preceq_\mu B'}{\Sigma \vdash A' \supset B \preceq_\mu A \supset B'}\ (\text{S-FUNC}) \qquad \frac{\Sigma \vdash A \preceq_\mu C \quad \Sigma \vdash B \preceq_\mu C}{\Sigma \vdash A \oplus B \preceq_\mu C}\ (\text{S-UNION-L})$$

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

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

Figure 2 Strong subtyping for μ-types.

be typable under the context extended with the corresponding θ_i. More noteworthy is the condition that the list $[p_i : A_i]_{i\in 1..n}$ be *compatible*.

Compatibility is a condition that ensures that Subject Reduction is not violated. We briefly recall it; see [18] for further details and examples. As already mentioned in example (3) of the introduction, if p_i subsumes p_j (*i.e.* $p_i \lhd p_j$) with $i < j$, then the branch $p_j \to_{\theta_j} s_j$ will never be evaluated since the argument will already match p_i. Thus, in this case, in order to ensure SR we demand that $A_j \preceq_\mu A_i$. If p_i does not subsume p_j (*i.e.* $p_i \not\lhd p_j$) with $i < j$ we analyze the cause of failure of subsumption in order to determine whether requirements on A_i and A_j must be put forward, focusing on those cases where $\pi \in \mathsf{pos}(p_i) \cap \mathsf{pos}(p_j)$ is an offending position in both patterns. The following table exhaustively lists them:

	$p_i\|_\pi$	$p_j\|_\pi$	
(a)		y	restriction required
(b)	$\mathtt{c}$	$\mathtt{d}$	no overlapping ($p_j \not\lhd p_i$)
(c)		$q_1\, q_2$	no overlapping
(d)		y	restriction required
(e)	$q_1\, q_2$	$\mathtt{d}$	no overlapping

In cases (b), (c) and (e), no extra condition on the types of p_i and p_j is necessary, since their respective sets of possible arguments are disjoint. The cases where A_i and A_j must be related are (a) and (d): for those we require $A_j \preceq_\mu A_i$. In summary, the cases requiring conditions on their types are: 1) $p_i \lhd p_j$; and 2) $p_i \not\lhd p_j$ and $p_j \lhd p_i$.

▶ **Definition 2.** Given a pattern $\theta \vdash_{\mathsf{p}} p : A$ and $\pi \in \mathsf{pos}(p)$, we say A *admits a symbol* $\odot$ *(with* $\odot \in \mathcal{V} \cup \mathcal{C} \cup \{\supset, @\}$*) at position* π iff $\odot \in A\|_\pi$, where:

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

Patterns

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

Terms

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

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

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

Figure 3 Typing rules for patterns and terms.

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

▶ **Definition 3.** The *maximal positions* in a set of positions P are:

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

The *mismatching positions* between two patterns are defined below where, recall from the introduction, $p|_\pi$ stands for the sub-pattern at position π of p:

$$\mathsf{mmpos}(p, q) \triangleq \{\pi \mid \pi \in \mathsf{maxpos}(\mathsf{pos}(p) \cap \mathsf{pos}(q)) \wedge p|_\pi \ntrianglelefteq q|_\pi\}$$

For instance, given patterns $\mathtt{nil}$ and $\mathtt{cons}\,x\,xs$ with set of positions $\{\epsilon\}$ and $\{\epsilon, 1, 2, 11, 12\}$ respectively, we have $\mathsf{maxpos}(\mathtt{nil}) = \{\epsilon\}$ and $\mathsf{maxpos}(\mathtt{cons}\,x\,xs) = \{11, 12\}$, while the only mismatching position among them is the root, *i.e.* $\mathsf{mmpos}(\mathtt{nil}, \mathtt{cons}\,x\,xs) = \{\epsilon\}$.

▶ **Definition 4.** Define the *compatibility predicate* as

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

We say $p : A$ is *compatible* with $q : B$, notation $p : A \lll q : B$, iff

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

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

Following the example, consider types $\mathbb{nil}$ and $\mathbb{cons} \,@\, \mathsf{Nat} \,@\, (\mu\alpha.\mathbb{nil} \oplus \mathbb{cons} \,@\, \mathsf{Nat} \,@\, \alpha)$ for patterns $\mathtt{nil}$ and $\mathtt{cons}\,x\,xs$ respectively. Compatibility requires no further restriction in this case since $\mathsf{mmpos}(\mathtt{nil}, \mathtt{cons}\,x\,xs) = \{\epsilon\}$ and

$$\mathbb{nil}\|_\epsilon = \{\mathbb{nil}\} \qquad (\mathbb{cons} \,@\, \mathsf{Nat} \,@\, (\mu\alpha.\mathbb{nil} \oplus \mathbb{cons} \,@\, \mathsf{Nat} \,@\, \alpha))\|_\epsilon = \{@\}$$

hence $\mathcal{P}_{\mathsf{comp}}$ is false and the property gets validated trivially.

On the contrary, recall example (3) on Sec. 1. $\mathtt{vl}\,x : \mathsf{vl} @ \mathsf{Bool} \lll \mathtt{vl}\,y : \mathsf{vl} @ \mathsf{Nat}$ requires $\mathsf{vl} @ \mathsf{Nat} \preceq_\mu \mathsf{vl} @ \mathsf{Bool}$ since $\mathsf{mmpos}(\mathtt{vl}\,x, \mathtt{vl}\,y) = \varnothing$ (*i.e.* $\mathcal{P}_{\mathsf{comp}}$ is trivially true). This actually fails because $\mathsf{Nat} \not\preceq_\mu \mathsf{Bool}$. Thus, this pattern combination is rejected by rule (T-ABS).

Types are preserved along reduction. The proof relies crucially on compatibility.

▶ **Proposition 5** (Subject Reduction). *If $\rhd \Gamma \vdash s : A$ and $s \to s'$, then $\rhd \Gamma \vdash s' : A$.*

Let the set of **values** be defined as $v ::= x\,v_1 \ldots v_n \mid \mathsf{c}\,v_1 \ldots v_n \mid (p_i \to_{\theta_i} s_i)_{i \in 1..n}$. The following property guarantees that no functional application gets stuck. Essentially this means that, in a well-typed closed term, a function which is applied to an argument has at least one branch that is capable of handling it.

▶ **Proposition 6** (Progress). *If $\rhd \vdash s : A$ and s is not a value, then $\exists s'$ such that $s \to s'$.*

3 Checking Equivalence and Subtyping

As mentioned in the related work, there are roughly two approaches to implementing equivalence and subtype checking in the presence of recursive types, one based on automata theory and another based on coinductive characterizations of the associated relations. The former leads to efficient algorithms [15] while the latter is more abstract in nature and hence closer to the formalism itself although they may not be as efficient. In the particular case of subtyping for recursive types in the presence of ACI operators, the automata approach of [15] is known not to be applicable [20] while the coinductive approach, developed in this section, yields a correct algorithm. In Sec. 5 we explore an alternative approach for subtyping based on automata inspired from [6]. We next further describe the reasoning behind the coinductive approach.

Preliminaries. 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" [16, 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.

3.1 Equivalence Checking

Fig. 4 presents a coinductive definition of type equality over μ-types. This relation $\simeq_{\vec{\mu}}$ is defined by means of rules that are interpreted coinductively (indicated by the double lines). 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. In turn, this allows us to guarantee the invertibility of the generating function associated to these rules.

$$\frac{}{a \simeq_{\vec{\mu}} a} \text{(E-REFL-AL)}$$

$$\frac{D \simeq_{\vec{\mu}} D' \quad A \simeq_{\vec{\mu}} A'}{D \mathbin{@} A \simeq_{\vec{\mu}} D' \mathbin{@} A'} \text{(E-COMP-AL)} \qquad \frac{A \simeq_{\vec{\mu}} A' \quad B \simeq_{\vec{\mu}} B'}{A \supset B \simeq_{\vec{\mu}} A' \supset B'} \text{(E-FUNC-AL)}$$

$$\frac{\mathcal{C}[\{\mu V.A/V\}\,A] \simeq_{\vec{\mu}} B}{\mathcal{C}[\mu V.A] \simeq_{\vec{\mu}} B} \text{(E-REC-L-AL)}$$

$$\frac{A \simeq_{\vec{\mu}} \mathcal{D}[\{\mu W.B/W\}\,B] \quad A \neq \mathcal{C}[\mu V.C]}{A \simeq_{\vec{\mu}} \mathcal{D}[\mu W.B]} \text{(E-REC-R-AL)}$$

$$\frac{\begin{array}{ll} A_i \simeq_{\vec{\mu}} B_{f(i)} & f : 1..n \to 1..m \\ A_{g(j)} \simeq_{\vec{\mu}} B_j & g : 1..m \to 1..n \end{array} \quad A_i, B_j \neq \mu, \oplus \quad n+m > 2}{\bigoplus_{i \in 1..n} A_i \simeq_{\vec{\mu}} \bigoplus_{j \in 1..m} B_j} \text{(E-UNION-AL)}$$

Figure 4 Coinductive axiomatization of type equality for contractive μ-types.

▶ **Proposition 7.** *The generating function associated with the rules of Fig. 4 is invertible.*

Moreover, $\simeq_{\vec{\mu}}$ coincides with $\simeq_{\mu}$:

▶ **Proposition 8.** *$A \simeq_{\vec{\mu}} B$ iff $A \simeq_{\mu} B$.*

The proof of Prop. 8 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. 4 except for two important differences: 1) the relation is defined over regular trees in $\mathfrak{T}$, and 2) rules (E-REC-L-AL) and (E-REC-R-AL) are dropped.

Thus we can resort to invertibility of the generating function to check for $\simeq_{\vec{\mu}}$. Fig. 5 presents the algorithm. It uses `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 $\texttt{eqtype}(\emptyset, A, B)$ can have one of two outcomes: `fail`, meaning that $A \not\simeq_{\vec{\mu}} B$, or a set $S \in \wp(\mathcal{T} \times \mathcal{T})$ that is Φ-dense with $(A, B) \in S$, proving that $A \simeq_{\vec{\mu}} B$.

3.2 Subtype Checking

The approach to subtype checking is similar to that of type equivalence. First consider the relation $\preceq_{\vec{\mu}}$ over μ-types defined in Fig. 6. It captures $\preceq_{\mu}$:

▶ **Proposition 9.** *$A \preceq_{\vec{\mu}} B$ iff $A \preceq_{\mu} B$.*

The proof strategy is similar to that of Prop. 8. 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. 6.

Unfortunately, the generating function determined by the rules in Fig. 6, let us call it $\Phi_{\preceq_{\vec{\mu}}}$, is not invertible. Notice that (S-UNION-R-AL) overlaps with itself. For example, $\mathbb{c} \preceq_{\vec{\mu}} (\mathbb{c} \oplus \mathbb{d}) \oplus (\mathbb{e} \oplus \mathbb{c})$ belongs to two $\Phi_{\preceq_{\vec{\mu}}}$-saturated sets (*i.e.* sets $\mathcal{X}$ such that $\mathcal{X} \subseteq \Phi_{\preceq_{\vec{\mu}}}(\mathcal{X})$):

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

```
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
```

Figure 5 Equivalence checking algorithm.

$$\frac{}{a \preceq_{\vec{\mu}} a}\ (\text{S-REFL-AL})$$

$$\frac{D \preceq_{\vec{\mu}} D' \quad A \preceq_{\vec{\mu}} A'}{D \mathbin{@} A \preceq_{\vec{\mu}} D' \mathbin{@} A'}\ (\text{S-COMP-AL}) \qquad \frac{A' \preceq_{\vec{\mu}} A \quad B \preceq_{\vec{\mu}} B'}{A \supset B \preceq_{\vec{\mu}} A' \supset B'}\ (\text{S-FUNC-AL})$$

$$\frac{\{\mu V.A/V\}\, A \preceq_{\vec{\mu}} B}{\mu V.A \preceq_{\vec{\mu}} B}\ (\text{S-REC-L-AL}) \qquad \frac{A \preceq_{\vec{\mu}} \{\mu W.B/W\}\, B \quad A \neq \mu}{A \preceq_{\vec{\mu}} \mu W.B}\ (\text{S-REC-R-AL})$$

$$\frac{A_i \preceq_{\vec{\mu}} B \text{ for all } i \in 1..n \quad n > 1 \quad B \neq \mu \quad A_i \neq \oplus}{\bigoplus_{i \in 1..n} A_i \preceq_{\vec{\mu}} B}\ (\text{S-UNION-L-AL})$$

$$\frac{A \preceq_{\vec{\mu}} B_k \text{ for some } k \in 1..m \quad m > 1 \quad A \neq \mu, \oplus \quad B_j \neq \oplus}{A \preceq_{\vec{\mu}} \bigoplus_{j \in 1..m} B_j}\ (\text{S-UNION-R-AL})$$

Figure 6 Coinductive axiomatization of subtyping for contractive μ-types.

```
subtype(S, A, B) ≜
    if ⟨A, B⟩ ∈ S
      then S
      else let S₀ = S ∪ {⟨A, B⟩} in
        case ⟨A, B⟩ of
          ⟨a, a⟩ →
            S₀
          ⟨A′ @ A″, B′ @ B″⟩ →
            if A′, B′ are datatypes
              then let S₁ = subtype(S₀, A′, A″) in
                subtype(S₁, B′, B″)
              else fail
          ⟨A′ ⊃ A″, B′ ⊃ B″⟩ →
            let S₁ = subtype(S₀, B′, A′) in
              subtype(S₁, A″, B″)
          ⟨μV.A′, B⟩ →
            subtype(S₀, {μV.A′/V} A′, B)
          ⟨A, μW.B′⟩ →
            subtype(S₀, A, {μW.B′/W} B′)
          ⟨⊕_{i∈1..n} A_i, B⟩ →
            let S₁ = subtype(S₀, A₁, B) in
            let S₂ = subtype(S₁, A₂, B) in
            ...
            let S_{n−1} = subtype(S_{n−2}, A_{n−1}, B) in
              subtype(S_{n−1}, A_n, B)
          ⟨A, ⊕_{j∈1..m} B_j⟩ →
            seq subtype(S₀, A, B₁), ..., subtype(S₀, A, B_m)
          otherwise →
            fail
```

Figure 7 Subtype checking algorithm.

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 (Fig. 7).

4 Type Checking

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. This requires "hard-wiring" it back in into (T-APP). Unfortunately, the naïve syntax-directed variant:

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

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 $\Gamma(x) \triangleq (\mathbb{c} \oplus \mathbb{e} \supset \mathbb{d}) \oplus (\mathbb{c} \oplus \mathbb{f} \supset \mathbb{d})$, $s \triangleq x\,\mathsf{c}$ and $A \triangleq \mathbb{d}$. More generally, from $\Gamma \mapsto r : A$ and $A \preceq_\mu \bigoplus_{i \in 1..n} A_i \supset B$ we cannot infer

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

$$\frac{\begin{array}{c}[p_i : A_i]_{i\in 1..n} \text{ compatible} \\ (\theta_i \vdash_{\mathsf{p}} p_i : A_i)_{i\in 1..n} \quad (\mathsf{dom}\,(\theta_i) = \mathsf{fm}(p_i))_{i\in 1..n} \quad (\Gamma, \theta_i \mapsto s_i : B_i)_{i\in 1..n}\end{array}}{\Gamma \mapsto (p_i \rightarrow_{\theta_i} s_i)_{i\in 1..n} : \bigoplus_{i\in 1..n} A_i \supset \bigoplus_{i\in 1..n} B_i}\,(\text{T-ABS-AL})$$

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

Figure 8 Syntax-directed typing rules for terms.

that A is a functional type due to the presence of union types. A complete (Prop. 10) 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., of Fig. 8.

▶ **Proposition 10.**
1. *If* $\Gamma \mapsto s : A$*, then* $\Gamma \vdash s : A$*.*
2. *If* $\Gamma \vdash s : A$*, then* $\exists A'$ *such that* $A' \preceq_\mu A$ *and* $\Gamma \mapsto s : A'$*.*

From this we may obtain a simple type-checking function $\mathsf{tc}(\Gamma, s)$ (Fig. 9-top) 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 $i \in 1..n$.

$$\begin{array}{lll}\mathsf{unfold}(A) & \triangleq & \texttt{if } A = A' \supset A'' \texttt{ then } A \\ & & \texttt{else if } A = \oplus_{i\in 1..n} A_i \texttt{ and } n > 1 \texttt{ and } A_i \neq \oplus \texttt{ then} \\ & & \quad \texttt{let } \oplus_{j\in 1..m_i}(A_{ij} \supset B_{ij}) = \mathsf{unfold}(A_i) \texttt{ foreach } i \in 1..n \texttt{ in} \\ & & \quad\quad \bigoplus_{\substack{i\in 1..n \\ j\in 1..m_i}} (A_{ij} \supset B_{ij}) \\ & & \texttt{else if } A = \mu V.A' \texttt{ then } \mathsf{unfold}(\{\mu V.A/V\}\, A) \\ & & \texttt{else fail}\end{array}$$

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.* $\mu X_1 \ldots \mu X_n . X_1 \supset \ldots X_n \supset \mathbb{c}$).

Compatibility between branches is verified by checking if $\mathcal{P}_{\mathsf{comp}}(p : A, q : B)$ holds:

$$\mathsf{compatible}(p : A, q : B) \quad \triangleq \quad (\texttt{not } \mathsf{pcomp}(p : A, q : B)) \texttt{ or } \mathsf{subtype}(\varnothing, B, A)$$

In pcomp 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

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

$$
\begin{array}{rcl}
\mathsf{tcp}(\Gamma, x) & \triangleq & \Gamma(x) \\
\mathsf{tcp}(\Gamma, \mathsf{c}) & \triangleq & \mathbb{c} \\
\mathsf{tcp}(\Gamma, p\, q) & \triangleq & \texttt{let } A = \mathsf{tcp}(\Gamma, p), B = \mathsf{tcp}(\Gamma, q) \texttt{ in} \\
& & \quad \texttt{if } A \text{ is a datatype} \\
& & \quad\quad \texttt{then } A \mathbin{@} B \\
& & \quad\quad \texttt{else fail}
\end{array}
$$

Figure 9 Type-checking CAP.

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.

$$
\begin{array}{rcl}
\mathsf{pcomp}(p : A, q : B) & \triangleq & \texttt{if } p = p_1\, p_2 \textsf{ and } q = q_1\, q_2 \texttt{ then} \\
& & \quad \texttt{let } A = A_1 \mathbin{@} A_2, B = B_1 \mathbin{@} B_2 \texttt{ in} \\
& & \quad\quad \mathsf{pcomp}(p_1 : A_1, q_1 : B_1) \textsf{ and } \mathsf{pcomp}(p_2 : A_2, q_2 : B_2) \\
& & \texttt{else} \\
& & \quad\quad (p = x) \textsf{ or } (p = q = \mathsf{c}) \textsf{ or } (A\|_\epsilon \cap B\|_\epsilon \neq \varnothing)
\end{array}
$$

5 Towards Efficient Type-Checking

The algorithms presented so far are clear but inefficient. The number of recursive calls in eqtype and subtype is not bounded (it depends on the size of the type) and unfolding recursive types may increment their size exponentially. This section draws from ideas in [6, 12, 15] 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. 13). The algorithm itself is tc of Fig. 9 except that:

1. The representation of μ-types are now term automata. This renders unfold linear.
2. The subtyping algorithm is optimized, based on the new representation and following ideas from [6, 15].

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-unions occurring in Γ and t. Note that all the information needed to type t is either in the context or annotated within the

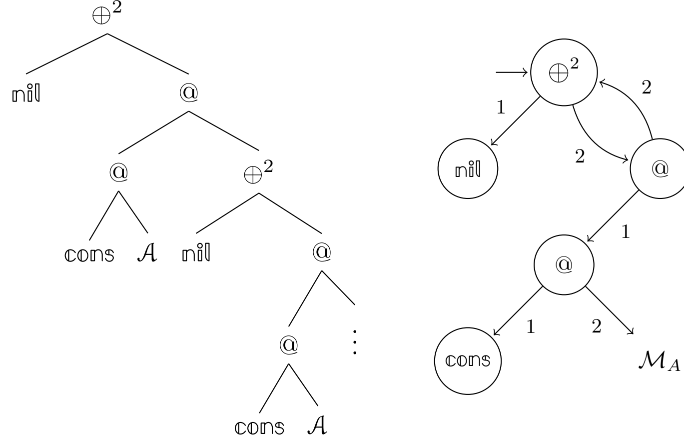

Figure 10 The type List_A represented as an infinite tree and as a term automaton.

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.

5.1 Term Automata

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

▶ **Definition 11.** 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 $1..k$, where k is the arity of the symbol associated by ℓ to the origin state.
5. $\ell : Q \to \Sigma$ is a total function that associates a symbol in Σ to each state.

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. 10 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.

5.2 Subtyping and Subtype Checking

We next present a coinductive notion of subtyping over $\mathfrak{T}^{\mathfrak{n}}$. It is a binary relation $\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}$ *up-to* a set of hypothesis $\mathcal{R}$ (Fig. 11). For $\mathcal{R} = \varnothing$, $\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}$ coincides with $\preceq_\mu$, modulo application of our translation $[\![\bullet]\!]^{\mathfrak{n}}$.

$$\frac{}{a \preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathsf{n}}} a} \text{ (S-REFL-UP)}$$

$$\frac{\mathcal{D}\ (\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathsf{n}}} \cup \mathcal{R})\ \mathcal{D}' \quad \mathcal{A}\ (\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathsf{n}}} \cup \mathcal{R})\ \mathcal{A}'}{\mathcal{D} \mathbin{@} \mathcal{A} \preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathsf{n}}} \mathcal{D}' \mathbin{@} \mathcal{A}'} \text{ (S-COMP-UP)}$$

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

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

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

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

Figure 11 Subtyping relation *up-to* $\mathcal{R}$ over $\mathfrak{T}^{\mathsf{n}}$.

▶ **Proposition 12.** *$A \preceq_\mu B$ iff $[\![A]\!]^{\mathsf{n}} \preceq^{\varnothing}_{\mathfrak{T}^{\mathsf{n}}} [\![B]\!]^{\mathsf{n}}$.*

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

5.2.1 Algorithm Description

The algorithm that checks whether types are related by the new subtyping relation builds on ideas from [6]. 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 $p \in \mathfrak{T}^{\mathsf{n}} \times \mathfrak{T}^{\mathsf{n}}$ *valid* if $p \in \preceq^{\varnothing}_{\mathfrak{T}^{\mathsf{n}}}$. The algorithm consists of two phases. The aim of the first one is to construct a set $U \subseteq \mathfrak{T}^{\mathsf{n}} \times \mathfrak{T}^{\mathsf{n}}$ 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. 12, buildUniverse) and then explores pairs of sub-terms of both types in this pair by decomposing the type constructors (*cf.* Fig. 12, 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 graph where an edge from pair p to q means that q might belong to the support set of p in the final relation $\preceq^{\varnothing}_{\mathfrak{T}^{\mathsf{n}}}$. In this case we say that p is a **parent** of q. Since types could have cycles, a pair may be added to U more than once and hence have more than one parent. Set $u(p)$ to be the **incoming degree** of p, *i.e.* the number of parents.

During the second phase (Fig. 13, gfp) the following sets are maintained, all of which conform a partition of U:

```
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 →
            ∅
```

Figure 12 Pseudo-code of the first phase of the algorithm (construction of the universe 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. 13, invalidate). Intuitively, S contains elements in $\preceq^{W}_{\mathfrak{T}^{\mathfrak{n}}}$. The process ends when W is empty. The only aspect of this second phase specific to $\preceq^{W}_{\mathfrak{T}^{\mathfrak{n}}}$ is function check, which may be redefined to be other suitable *up-to* relations.

5.2.2 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 13.** *The algorithm preserves the following invariant:*

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

When the main cycle ends we know that W is empty, and therefore that $S \subseteq \Phi_{\preceq^{\emptyset}_{\mathfrak{T}^{\mathfrak{n}}}}(S)$. The coinduction principle then yields $S \subseteq \preceq^{\emptyset}_{\mathfrak{T}^{\mathfrak{n}}}$ (*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.

5.2.3 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

```
gfp(𝒜, ℬ) :
    W = buildUniverse(⟨𝒜, ℬ⟩)
    S = ∅
    F = ∅
    while W ≠ ∅ :
      p := takeOne(W)
      if check(p, F)
        then insert(p, S)
        else invalidate(p, 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
```

Figure 13 Pseudo-code of the second phase (relation refinement).

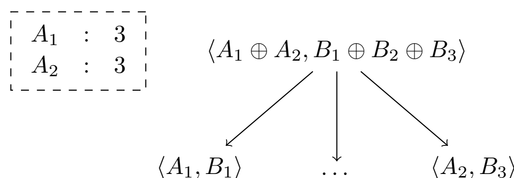

Figure 14 Verification of invalidated descendants.

by $\mathcal{O}(NN')$. As we shall see, the total cost of the algorithm is governed by the amount of operations in the second phase.

As stated in [6], 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

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

Assuming that set operations can be performed in constant time, the cost of evaluating each node in the main loop is that of deciding whether to suspend or invalidate the pair and, in the later case, the cost of the invalidation process. The decision of where to transfer the node is computed in the function check, which always performs a constant amount of operations for pairs of non-union types. The worst case involves checking pairs of the form $\langle \oplus_i^n A_i, \oplus_j^m B_i \rangle$, 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 [6] 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

$$\frac{}{a \simeq^{\mathcal{R}}_{\mathfrak{T}^n} a} \text{(E-REFL-UP)}$$

$$\frac{\mathcal{D}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{D}' \quad \mathcal{A}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{A}'}{\mathcal{D} @ \mathcal{A} \simeq^{\mathcal{R}}_{\mathfrak{T}^n} \mathcal{D}' @ \mathcal{A}'} \text{(E-COMP-UP)}$$

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

$$\frac{\begin{array}{ccc} \mathcal{A}_i\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{B}_{f(i)} & f : 1..n \to 1..m & \\ \mathcal{A}_{g(j)}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{B}_j & g : 1..m \to 1..n & \mathcal{A}_i, \mathcal{B}_j \neq \oplus \end{array}}{\bigoplus_i^n \mathcal{A}_i \simeq^{\mathcal{R}}_{\mathfrak{T}^n} \bigoplus_j^m \mathcal{B}_j} \text{(E-UNION-UP)}$$

$$\frac{\mathcal{A}_i\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R})\ \mathcal{B} \text{ for all } i \in 1..n \quad \mathcal{A}_i \neq \oplus \quad \mathcal{B} \neq \oplus}{\bigoplus_i^n \mathcal{A}_i \simeq^{\mathcal{R}}_{\mathfrak{T}^n} \mathcal{B}} \text{(E-UNION-L-UP)}$$

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

Figure 15 Equivalence relation *up-to* $\mathcal{R}$ over $\mathfrak{T}^n$.

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 leaves a total cost of $\mathcal{O}(size(U)d)$ for the subtyping check, expressed as $\mathcal{O}(NN'd)$ in terms of the size 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 $\mathcal{O}(n^2n'^2d)$.

5.3 Equivalence Checking

In this section we adapt the previous algorithm to obtain one proper of equivalence checking with the same complexity cost. Fig. 15 introduces an equivalence relation *up-to* $\mathcal{R}$ over $\mathfrak{T}^n$ which can be used to compute $\simeq_\mu$ via the translation $[\![\bullet]\!]^n$.

▶ **Lemma 14.** $A \simeq_\mu B$ *iff* $[\![A]\!]^n \simeq^{\varnothing}_{\mathfrak{T}^n} [\![B]\!]^n$.

The algorithm is the result of adapting the scheme presented for subtyping to the new relation $\simeq^{\mathcal{R}}_{\mathfrak{T}^n}$. This is done by redefining functions children and check from the first and second phase respectively (*cf.* Fig. 16). 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. We could have omitted this by using the same rule for functional types as before and resorting to the symmetry of the resulting relation (which does not depend on this rule), but we wanted 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.

```
children(p) :
    case p of
      ⟨𝒟 @ 𝒜, 𝒟′ @ ℬ⟩ →
          {⟨𝒟, 𝒟′⟩, ⟨𝒜, ℬ⟩}
      ⟨𝒜′ ⊃ 𝒜″, ℬ′ ⊃ ℬ″⟩ →
          {⟨𝒜′, ℬ′⟩, ⟨𝒜″, ℬ″⟩}
      ⟨⊕ᵢⁿ𝒜ᵢ, ⊕ⱼᵐℬⱼ⟩ →
          {⟨𝒜ᵢ, ℬⱼ⟩ | i ∈ 1..n, j ∈ 1..m}
      ⟨⊕ᵢⁿ𝒜ᵢ, ℬ⟩, ℬ ≠ ⊕ →
          {⟨𝒜ᵢ, ℬ⟩ | i ∈ 1..n}
      ⟨𝒜, ⊕ⱼᵐℬⱼ⟩, 𝒜 ≠ ⊕ →
          {⟨𝒜, ℬⱼ⟩ | 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
      ⟨⊕ᵢⁿ𝒜ᵢ, ⊕ⱼᵐℬⱼ⟩ →
          ∀i.∃j. ⟨𝒜ᵢ, ℬⱼ⟩ ∉ F and ∀j.∃i. ⟨𝒜ᵢ, ℬⱼ⟩ ∉ F
      ⟨⊕ᵢⁿ𝒜ᵢ, ℬ⟩, ℬ ≠ ⊕ →
          ∀i. ⟨𝒜ᵢ, ℬ⟩ ∉ F
      ⟨𝒜, ⊕ⱼᵐℬⱼ⟩, 𝒜 ≠ ⊕ →
          ∀j. ⟨𝒜, ℬⱼ⟩ ∉ F
```

Figure 16 Pseudo-code of first (left) and second (right) phase for equivalence checking.

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

▶ **Proposition 15.** *The algorithm preserves the following invariant:*
- $\langle W, S, F\rangle$ *is a partition of* U
- F *is composed solely of invalid pairs*
- $S \subseteq \Phi_{\simeq^{W}_{\mathfrak{T}^n}}(S)$

For the complexity analysis, notice that the size of the built universe is the same as before and phase one 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. 5.2.3. Thus, the cost of an iteration is $\mathcal{O}(d)$, resulting in an overall cost of $\mathcal{O}(NN'd)$ as before.

5.4 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 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 of the algorithm 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 $\mathcal{O}(n^4 d^2)$.

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 $\mathcal{O}(n^6 d)$ for this case.

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

5.5 Prototype implementation

A prototype in Scala is available [8]. It implements tc but resorts to the efficient algorithm for subtyping and type equivalence described above. It also includes further optimizations. For example, following a suggestion in [6], the order in which elements in W are selected for evaluation relies on detecting strongly connected components, using Tarjan's [7] 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.

6 Conclusions

We address efficient type-checking for path polymorphism. We start off with the type system of [18] which includes singleton types, union types, type application and recursive types. The union type constructor is assumed associative, commutative and idempotent. First we formulate a syntax-directed presentation. Then we devise invertible coinductive presentations of type-equivalence and subtyping. This yields a naïve but correct type-checking algorithm. However, it proves to be inefficient (exponential in the size of the type). This prompts us to change the representation of type expressions and use automata techniques to considerably improve the efficiency. Indeed, the final algorithm has complexity $\mathcal{O}(n^7 d)$ where n is the size of the input and d is the maximum arity of the unions occurring in it.

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.

- Addition of parametric polymorphism (presumably in the style of $\mathrm{F}_{<:}$ [4, 5, 16]). We believe this should not present major difficulties.
- Strong normalization requires devising a notion of positive/negative occurrence in the presence of strong μ-type equality, which is known not to be obvious [2, page 515].
- A more ambitious extension is that of *dynamic patterns*, namely patterns that may be computed at run-time, PPC being the prime example of a calculus supporting this feature.

References

1 Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993. doi:10.1145/155183.155231.

2. H. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013. doi:10.1017/cbo9781139032636.
3. M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inf.*, 33(4):309–338, 1998. doi:10.3233/fi-1998-33401.
4. Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings*, volume 526 of *Lecture Notes in Computer Science*, pages 750–770. Springer, 1991. doi:10.1007/3-540-54415-1_73.
5. Dario Colazzo and Giorgio Ghelli. Subtyping recursion and parametric polymorphism in kernel Fun. *Inf. Comput.*, 198(2):71–147, 2005. doi:10.1016/j.ic.2004.11.003.
6. Roberto Di Cosmo, François Pottier, and Didier Rémy. Subtyping recursive types modulo associative commutative products. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2005. doi:10.1007/11417170_14.
7. Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980. doi:10.1145/322217.322228.
8. J. Edi and A. Viso. Prototype implementation of efficient type-checker in Scala. URL: https://github.com/juanedi/cap-typechecking.
9. J. Edi, A. Viso, and E. Bonelli. Efficient type checking for path polymorphism, 2017. arXiv preprint 1704.09026. URL: http://arxiv.org/abs/1704.09026.
10. B. Jay and D. Kesner. First-class patterns. *J. Funct. Program.*, 19(2):191–225, 2009. doi:10.1017/s0956796808007144.
11. Barry Jay. *Pattern Calculus - Computing with Functions and Structures*. Springer, 2009. doi:10.1007/978-3-540-89185-7.
12. T. Jim and J. Palsberg. Type inference in systems of recursive types with subtyping, 1999. Draft. URL: http://web.cs.ucla.edu/~palsberg/draft/jim-palsberg99.pdf.
13. Jan Willem Klop, Vincent van Oostrom, and Roel C. de Vrijer. Lambda calculus with patterns. *Theor. Comput. Sci.*, 398(1-3):16–31, 2008. doi:10.1016/j.tcs.2008.01.019.
14. D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient recursive subtyping. *Math. Struct. Comput. Sci.*, 5(1):113–125, 1995. doi:10.1017/s0960129500000657.
15. Jens Palsberg and Tian Zhao. Efficient and flexible matching of recursive types. *Inf. Comput.*, 171(2):364–387, 2001. doi:10.1006/inco.2001.3090.
16. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
17. V. van Oostrom. Lambda calculus with patterns. Technical Report IR-228, Vrije Universiteit Amsterdam, 1990. URL: http://www.phil.uu.nl/~oostrom/publication/pdf/IR-228.pdf.
18. Andrés Viso, Eduardo Bonelli, and Mauricio Ayala-Rincón. Type soundness for path polymorphism. *Electr. Notes Theor. Comput. Sci.*, 323:235–251, 2016. doi:10.1016/j.entcs.2016.06.015.
19. Jerome Vouillon. Subtyping union types. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 415–429. Springer, 2004. doi:10.1007/978-3-540-30124-0_32.
20. T. Zhao. *Type Matching and Type Inference for Object-Oriented Systems*. PhD thesis, Purdue University, 2002. URL: http://docs.lib.purdue.edu/dissertations/AAI3099873/.

Revision Notice

This is a revised version of the eponymous paper appeared in the proceedings of TYPES 2015 (LIPIcs, volume 69, `http://www.dagstuhl.de/dagpub/978-3-95977-030-9` published in March, 2019). This version fixes a character set problem that caused some symbols to be displayed incorrectly.

Dagstuhl Publishing – March 17, 2019.