Subtyping First-Class Polymorphic Components

João Costa Seco and Luís Caires

Departamento de Informática
Universidade Nova de Lisboa
{Joao.Seco, Luis.Caires}@di.fct.unl.pt

Abstract. We present a statically typed, class-based object oriented language where classes are first class polymorphic values. A main contribution of this work is the design of a type system that combines first class polymorphic values with structural equirecursive types and admits a subtyping algorithm which is arguably much simpler than existing alternatives. Our development is modular and can be easily instantiated for either a Kernel-Fun or a $F_{\leq}^{\top}$ style of subtyping discipline.

1 Introduction

When one considers the essence of programming languages and especially of programming languages with subtyping and equirecursive types the language that comes to mind is $F_{\leq}$, the extension with subtypes of the polymorphic lambda calculus introduced in [7]. When object-oriented programming is at stake, languages like the object calculus of Abadi and Cardelli [1] and Momi [15] are good examples of how to express object-oriented mechanisms. However, the subtyping relations they use to relate classes, objects and mixins stays far behind the flexible relation of $F_{\leq}$; both use invariant width subtyping relations and limited recursive subtyping relations. This expressiveness gap is due to unsoundness problems when coding the *self* reference as a generic value parameter of methods. In FJ [13], for instance, structural equivalence of types is traded by name-based equivalence, which is convenient to overcome problems with the subtyping of recursive types. In fact, structural equivalence of types, although adopted in some experimental programming languages such as OCaml and Modula3, does not seem to have had substantial impact in main-stream object-oriented languages.

Nevertheless, the increasing use of dynamic loading, late binding and mobile code in general purpose programming frameworks raises the issue of finding more flexible compatibility criteria between components. One reason is that name-based extension and subtyping as it is implemented by modern object oriented languages creates a rigid hierarchy of classes and interfaces based on their names. This implies the usage of global name spaces and, for instance, disallows the compatibility of two classes that separately combine the same set of interfaces. This problem can of course be diminished by explicitly using wrapper objects that redirect method calls and therefore make compatible two otherwise incompatible classes. But, structural equivalence would be the most natural solution to this kind of problems.

In previous work we have presented an object oriented component calculus that uses structural equivalence of types [18,19]. The calculus was ported into an experimental language that is compiled to run on the Java framework, which inherits the structural character of calculus' type relations up to a certain level. In this paper, we extract the essential aspects of the type system of the component calculus into a core class-based language that manipulates classes as values and uses second-order equirecursive types.

One of the main contributions of this paper is the presentation of a subtyping algorithm for second-order equirecursive types. Our approach is intuitive and technically much simpler to define and prove correct than previous results, such as [5]. It builds on the coinductive formulation of first-order type systems with equirecursive types of Amadio and Cardelli [2], Brandt and Henglein [3], and Gapeyev, Levin and Pierce [9,17]. The simplicity of the algorithm is directly related to the nature of the elements the algorithm manipulates which are complete judgments instead of pairs of types and to the termination conditions of the coinductive algorithm, which uses permutation-based techniques. Moreover, our development is modular, in the sense that it can be adapted to either a Kernel-Fun or a $F_{\leq}^{\top}$ subtyping discipline.

A further contribution is of this paper is the proposal of a simple composition mechanism for classes that combines the usage of structural equivalence of classes and objects with class extension mechanisms.

The remainder of the paper is structured as follows: section 2 describes the class-based language and illustrates it using a small programming example; in section 3 we define a type system for the language including the subtyping relation and its properties; in section 4 we describe the algorithm that checks the type of a language expression and enunciate its properties. Finally, in section 5 we propose a new composition mechanism for extending classes that is sound under structural subtyping of classes and objects.

2 The programming language

In this section, we define a core class-based programming language whose values are classes and objects. Both objects and classes are runtime entities in our language, the main goal is to study a type system for generic components (as classes) and objects, where the polymorphic types of classes and equirecursive types of objects are related structurally. We first introduce its syntax, which is depicted in Fig. 1. It includes constructs for objects, classes, instantiation of objects, method calls, local declarations, and recursion. Class expression combines bounded type abstraction, and value abstraction over a name denoting the object *self*. All other constructions are interpreted as shown in Fig. 2.

The evaluation relation of the language is defined by a big step semantics $e \Downarrow v$. Among these rules the evaluation of a new e expression deserves further explanation: it relies on the evaluation of the subexpression e into a class value, and closure under recursion of $self$. All other rules evaluate the corresponding

$$\begin{array}{llll}
 & \textbf{Types} & & \\
\tau ::= & X & & \text{(variable)} \\
\mid & \mathsf{Class}[X_i \leq \tau_i \ ^{i\in 1..n}]\ I & & \text{(class)} \\
\mid & I & & \text{(interface)} \\
\mid & \mathsf{Top} & & \text{(top)} \\
I ::= & \{m_i(\tau_{j_i} \ ^{j_i \in 1..n_i}) : \tau_i \ ^{i\in 1..n}\} & & \text{(interface)} \\
\mid & \mu X.I & & \text{(recursion)}
\end{array}
\qquad
\begin{array}{lll}
 & \textbf{Terms} & \\
e ::= & x & \text{(variable)} \\
\mid & v & \text{(value)} \\
\mid & \mathsf{new}\ e[\tau_i \ ^{i\in 1..n}] & \text{(instantiation)} \\
\mid & e.m(e_i \ ^{i\in 1..n}) & \text{(method call)} \\
\mid & \mathsf{let}\ x = e\ \mathsf{in}\ e & \text{(declaration)} \\
\mid & \mathsf{rec}(x:\tau)\ e & \text{(recursion)}
\end{array}$$

$$\begin{array}{lll}
 & \textbf{Values} & \\
v ::= & \{\ m_i(x_{j_i} : \tau_{j_i} \ ^{j_i\in 1..n_i}) : \tau_i = e_i \ ^{i \in 1..n}\ \} & \text{(objects)} \\
\mid & \mathsf{class}[X_i \leq \tau_i \ ^{i\in 1..n}](s)\ e & \text{(classes)}
\end{array}$$

Fig. 1. Types and terms

$$v \Downarrow v \ \text{(Value)} \qquad \frac{e_1 \Downarrow v_1 \quad e_2[x \leftarrow v_1] \Downarrow v_2}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v_2}\ \text{(Let)} \qquad \frac{e[x \leftarrow \mathsf{rec}(x:\tau)\ e] \Downarrow v}{\mathsf{rec}(x:\tau)\ e \Downarrow v}\ \text{(Fix)}$$

$$\frac{\begin{pmatrix} o = \{m_i(x_{j_i} : \tau_{j_i} \ ^{j_i \in 1..n_i}) : \tau_i = e_i \ ^{i\in 1..n}\} \\ I = \{m(x_{j_i} : \tau_{j_i} \ ^{j_i\in 1..n_i}) : \tau_i \ ^{i \in 1..n}\} \end{pmatrix} \\ e \Downarrow \mathsf{class}[X_i \leq \delta_i \ ^{i\in 1..n}](s)\ o \qquad \mathsf{rec}(s:I)\ o \Downarrow v}{\mathsf{new}\ e[\tau_i \ ^{i\in 1..n}] \Downarrow v}\ \text{(New)}$$

$$\frac{e \Downarrow \{\ \ldots, m(x_i : \tau_i \ ^{i\in 1..n}) : \tau = e_b, \ldots\ \} \\ e'_i \Downarrow v_i \ \ \forall_{i \in 1..n} \quad e_b[x_i \leftarrow v_i \ ^{i\in 1..n}] \Downarrow v}{e.m(e'_i \ ^{i\in 1..n}) \Downarrow v}\ \text{(Call)}$$

Fig. 2. Big step operational semantic rules

expressions as expected. Note that classes and objects are values and hence evaluate to themselves by means of the rule (Value).

Types already appear in the syntax of expressions, are are also defined in Fig. 1. We distinguish between two kinds of types: interface types and class types. Type variables range over types of any kind. A class type $\mathsf{Class}[X \leq \tau]\ \sigma$ is a polymorphic type corresponding to a $F_{\leq}$ bounded type quantification (cf., $\forall_{X \leq \tau}.\sigma$), but for convenience generalized to a list of type parameters. Interface types can be defined recursively, using type recursion $\mu X.I$; our separation of types in two categories τ and I is not essential, and only reflects the intended type usage of the object-oriented language.

We illustrate our language with a very simple example that uses polymorphic memory cells. Let `C` be the type defined as $\mathsf{Class}[X]\ \mu Y.\{set(X) : Y,\ get() : X\}$, and `cell` some class value of type `C`, and consider

$$\phi \vdash \diamond \ \text{(E-}\phi\text{)} \qquad \frac{\Delta \vdash \tau \ \mathsf{ok}}{\Delta, x : \tau \vdash \diamond} \ \text{(E-Var)} \qquad \frac{\Delta \vdash \tau \ \mathsf{ok}}{\Delta, X \leq \tau \vdash \diamond} \ \text{(E-TVar)}$$

$$\frac{\Delta \vdash \diamond}{\Delta \vdash X \ \mathsf{ok}} \ \text{(O-TVar)} \qquad \frac{\Delta, X \leq \mathsf{Top} \vdash I \ \mathsf{ok}}{\Delta \vdash \mu X.I \ \mathsf{ok}} \ \text{(O-Recursive)}$$

$$\frac{\Delta, X_j \leq \tau_j \ {}^{j \in 1..i-1} \vdash \tau_i \ \mathsf{ok} \quad \forall_{i \in 1..n} \quad \Delta, X_i \leq \tau_i \ {}^{i \in 1..n} \vdash I \ \mathsf{ok}}{\Delta \vdash \mathsf{Class}[X_i \leq \tau_i \ {}^{i \in 1..n}] \ I \ \mathsf{ok}} \ \text{(O-Polymorphic)}$$

$$\frac{\Delta \vdash \tau_{j_i} \ \mathsf{ok} \ \forall_{j_i \in 1..n_i} \ \forall_{i \in 1..n} \quad \Delta \vdash \tau_i \ \mathsf{ok} \ \forall_{i \in 1..n}}{\Delta \vdash \{m_i(\tau_{j_i} \ {}^{j_i \in 1..n_i}) : \tau_i \ {}^{i \in 1..n}\} \ \mathsf{ok}} \ \text{(O-Interface)}$$

Fig. 3. Well-formed types and environments

```
let d = class[Y](s){
           test(c:C, v:Y):Y =
             let o1 = (new c[Y]) in
             let o2 = o1.set(v) in o2.get()
      }
  in let n = (new d[int]).test(cell,1)
```

Class `d` defines a method `test` that accepts two arguments: a class value (a component) implementing memory cells `c` and another appropriate value `v`. The method instantiates the memory cell class, stores the value `v` in the resulting cell object, and finally retrieves the cell contents and returns it.

3 Type system

In this section we define the type system for our language. The type system has two parts, a typing system for the language expressions, and a subtyping system expressing the intended subsumption relation on types. Our presentation will focus on the latter, concentrating on the development of our approach to polymorphic recursive subtyping.

3.1 Typing Expressions

The typing of the expressions is given by the set of rules in Fig. 4, that proves judgements of the form $\Delta \vdash e : \tau$, where Δ is the typing environment declaring the types of the free value and type variables relevant for the expression e, and τ is a type. Well-formed types and environments are also defined in Fig. 3, by the judgement forms $\Delta \vdash \diamond$ and $\Delta \vdash \tau \ \mathsf{ok}$, as expected.

Most rules follow the usual pattern, we will discuss just the particularities of our presentation. Although the abstract syntax in Fig. 1 is somewhat more liberal, notice that rule (T-Class) enforces that only record expressions are accepted as a class body, consistently with our interpretation of polymorphic types

$$\frac{x : \tau \in \Delta}{\Delta \vdash x : \tau}\ \text{(T-Var)} \qquad \frac{\Delta \vdash e : \tau' \quad \Delta \vdash \tau' \leq \tau}{\Delta \vdash e : \tau}\ \text{(T-Sub)} \qquad \frac{\Delta, x : \tau \vdash e : \tau}{\Delta \vdash \mathsf{rec}(x : \tau)\ e : \tau}\ \text{(T-Fix)}$$

$$\frac{\begin{pmatrix} c = \{m_i(x_{j_i} : \tau_{j_i}\ {}^{j_i \in 1..n_i}) : \tau_i = e_i\ {}^{i \in 1..n}\} \\ I = \{m_i(\tau_{j_i}\ {}^{j_i \in 1..n_i}) : \tau_i\ {}^{i \in 1..n}\} \end{pmatrix} \\ \Delta, X_j \leq \delta_j\ {}^{j \in 1..m}, x_{j_i} : \tau_{j_i}\ {}^{j_i \in 1..n_i}, s : I \vdash e_i : \tau_i \ \ \forall_{i \in 1..n}}{\Delta \vdash \mathsf{class}[X_j \leq \delta_j\ {}^{j \in 1..m}](s)\ c : \mathsf{Class}[X_j \leq \delta_j\ {}^{j \in 1..m}]\ I}\ \text{(T-Class)}$$

$$\frac{(\ I = \{m_i(\tau_{j_i}\ {}^{j_i \in 1..n_i}) : \tau_i\ {}^{i \in 1..n}\}\ ) \quad \Delta, x_{j_i} : \tau_{j_i}\ {}^{j_i \in 1..n_i} \vdash e_i : \tau_i \ \ \forall_{i \in 1..n}}{\Delta \vdash \{\ m_i(x_{j_i} : \tau_{j_i}\ {}^{j_i \in 1..n_i}) : \tau_i = e_i\ {}^{i \in 1..n}\ \} : I}\ \text{(T-Object)}$$

$$\frac{\Delta \vdash e : \mathsf{Class}[X_i \leq \delta_i\ {}^{i \in 1..n}]\ I \quad \Delta \vdash \tau_i \leq \delta_i[X_j \leftarrow \tau_j\ {}^{j \in 1..i-1}] \ \ \forall_{i \in 1..n}}{\Delta \vdash \mathsf{new}\ e[\tau_i\ {}^{i \in 1..n}] : I[X_i \leftarrow \tau_i\ {}^{i \in 1..n}]}\ \text{(T-New)}$$

$$\frac{\Delta \vdash e : \{\ldots, m(\tau_i\ {}^{i \in 1..n}) : \tau, \ldots\} \\ \Delta \vdash e_i : \tau_i \quad \forall_{i \in 1..n}}{\Delta \vdash e.m(e_i\ {}^{i \in 1..n}) : \tau}\ \text{(T-Call)} \qquad \frac{\Delta \vdash e_1 : \tau \quad \Delta, x : \tau \vdash e_2 : \tau'}{\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau'}\ \text{(T-Let)}$$

Fig. 4. Typing rules

as polymorphic object-generating classes. The name s, whose scope is the class body, typed with the type of the instances of the class, is our notation for *self*. In the rule (T-New), the instantiation expression new is typed with a type constructed from the type declarations present in the class expression, given the proper type substitution of the type parameters, and provided that the compatibility of the type arguments with respect to the variable bounds holds.

3.2 Subtyping

Some approaches to the problem of defining a first-order subtyping relation between recursive types exist for quite a while [2,3,17].

Intuitively, the intended subsumption relation between recursive types corresponds to the usual inclusion of infinite (regular) trees, the difficulty in the polymorphic case arises due to the presence of binding occurrences of type variables on types, due to presence of type quantifiers. Usually, even for recursive types, subtyping relations have been expressed by means of inductive proof systems, where the coinduction principle appears embedded in various explicit ways [2,3]. Apart from these, the main proof rules we might expect for such a subtyping system are the ones depicted in Fig. 5. These include the usual relationships: maximality of Top, reflexivity, transitivity, width and depth record subtyping, and unfolding of recursive types. For comparing polymorphic types, we adopt a Kernel-Fun style rule, since the more general $F_{\leq}$ style subtyping is known to be undecidable even in the absence of recursion [16]. In any case, our approach applies equally well to Kernel-Fun and to variants such as $F_{\leq}^{\top}$.

$$\Delta \vdash \tau \leq \mathsf{Top} \qquad \Delta \vdash \tau \leq \tau \qquad \frac{\Delta \vdash \tau \leq \sigma \qquad X \leq \tau \in \Delta}{\Delta \vdash X \leq \sigma}$$

$$\frac{\Delta, X_i \leq \delta_i \ ^{i \in 1..n} \vdash I \leq I'}{\Delta \vdash \mathsf{Class}[X_i \leq \delta_i \ ^{i \in 1..n}]\ I \leq \mathsf{Class}[X_i \leq \delta_i \ ^{i \in 1..n}]\ I'}$$

$$\frac{n' \leq n \qquad \Delta \vdash \tau_i \leq \tau'_i \ \forall_{i \in 1..n'} \qquad \Delta \vdash \tau'_{j_i} \leq \tau_{j_i} \ \forall_{j_i \in 1..n_i} \forall_{i \in 1..n'}}{\Delta \vdash \{m_i(\tau_{j_i} \ ^{j_i \in 1..n_i}) : \tau_i \ ^{i \in 1..n}\} \leq \{m_i(\tau'_{j_i} \ ^{j_i \in 1..n_i}) : \tau'_i \ ^{i \in 1..n'}\}}$$

$$\frac{\Delta \vdash I \leq J[\alpha \leftarrow \mu\alpha.J]}{\Delta \vdash I \leq \mu\alpha.J} \qquad \frac{\Delta \vdash I[\alpha \leftarrow \mu\alpha.I] \leq J}{\Delta \vdash \mu\alpha.I \leq J}$$

Fig. 5. Subtyping inductive rules

Unfortunately, the adoption of these rules results in an incomplete type system, that does not seem to easily lead to a terminating algorithm, as remarked in [11], although a rather complex subtyping algorithm following this approach was already developed in [5]. In fact, our difficulties in getting a clear understanding of this work lead us to attempt a different approach, leading to the presentation in this paper. We then follow essentially the development of [9] for first-order types, and extend it in a natural way to polymorphic types. Therefore, we start from a coinductive definition of the subtyping relation, presented below.

Let $\mathcal{D}$ denote the set of all well-formed environments and $\mathcal{T}$ the set of all well-formed types. We also denote by $\mathcal{J}$ the set $\mathcal{D} \times \mathcal{T} \times \mathcal{T}$ of all subtyping judgements (represented as tuples).

Definition 3.1. *The* subtyping generating function *is the mapping $S \in \mathcal{P}(\mathcal{J}) \rightarrow \mathcal{P}(\mathcal{J})$ defined by:*

$$\begin{aligned}
S(\mathfrak{R}) = & \{(\Delta; \tau; \tau) \mid \tau \in \mathcal{T} \text{ and } FV(\tau) \subseteq Dom(\Delta)\} \\
\cup\ & \{(\Delta; \tau; \mathsf{Top}) \mid \tau \in \mathcal{T} \text{ and } FV(\tau) \subseteq Dom(\Delta)\} \\
\cup\ & \{(\Delta; X; \sigma) \mid (\Delta; \tau; \sigma) \in \mathfrak{R} \text{ and } X \leq \tau \in \Delta\} \\
\cup\ & \{(\Delta; \mathsf{Class}[X_i \leq \delta_i \ ^{i \in 1..n}]\ I; \mathsf{Class}[X_i \leq \delta_i \ ^{i \in 1..n}]\ I') \mid \\
& \quad (\Delta, X_j \leq \delta_j \ ^{j \in 1..n}; I; I') \in \mathfrak{R}\} \\
\cup\ & \{(\Delta; I; \mu X.J) \mid (\Delta; I; J[X \leftarrow \mu X.J]) \in \mathfrak{R}\ \} \\
\cup\ & \{(\Delta; \mu X.I; J) \mid (\Delta; I[X \leftarrow \mu X.I]; J) \in \mathfrak{R} \text{ and } J \not\equiv \mu X.J'\ \} \\
\cup\ & \{(\Delta; \{m_i(\tau_{j_i} \ ^{j_i \in 1..n_i}) : \tau_i \ ^{i \in 1..n}\}; \{m_i(\tau'_{j_i} \ ^{j_i \in 1..n'_i}) : \tau'_i \ ^{i \in 1..n'}\}) \mid \\
& \quad n' \leq n \text{ and } n'_i = n_i \ \forall_{i \in 1..n'} \text{ and} \\
& \quad (\Delta; \tau_i; \tau'_i) \in \mathfrak{R} \ \forall_{i \in 1..n'} \text{ and } (\Delta; \tau'_{j_i}; \tau_{j_i}) \in \mathfrak{R} \ \forall_{j_i \in 1..n_i} \forall_{i \in 1..n'}\}.
\end{aligned}$$

We can verify that the mapping S is monotonic. Therefore, there exists its greatest fixed point $\nu S \in \mathcal{P}(\mathcal{J})$. We then define the subtyping relation as follows

Definition 3.2. $\Delta \vdash \tau \leq \sigma \triangleq (\Delta, \tau, \sigma) \in \nu S$.

The relation thus defined enjoys the basic properties of weakening, substitution of type variables, equivariance, narrowing and transitivity which are essential to prove the type safety of the language (Theorem 3.10) and the correctness of the subtyping algorithm (Theorem 4.10). In general, these kind of results are proved by somewhat involved inductions on derivations; in our setting, due to the natural definition of subtyping as a greatest fixed point, we may handle them by quite standard coinductive proof techniques. We start by considering the weakening property in νS.

Proposition 3.3 (Weakening). *For all typing environments $\Delta, \Delta' \in \mathcal{D}$, and types $\tau, \sigma, \delta \in \mathcal{T}$, if $\Delta, \Delta' \vdash \tau \leq \sigma$, $X \notin Dom(\Delta, \Delta')$, and Delta $\vdash \delta$ ok then $\Delta, X \leq \delta, \Delta' \vdash \tau \leq \sigma$.*

Proof. Consider the following set:

$$\begin{aligned}\mathfrak{W} \triangleq \{(\Delta, X \leq \delta, \Delta'; \tau; \sigma) \mid{} & (\Delta, \Delta'; \tau; \sigma) \in \nu S \text{ and} \\ & X \notin Dom(\Delta, \Delta') \text{ and } FV(\delta) \subseteq Dom(\Delta)\}.\end{aligned}$$

By case analysis in the definition of S we prove $\mathfrak{W}$ to be S-consistent, $\mathfrak{W} \subseteq S(\mathfrak{W})$, thus by the coinduction principle we have that $\mathfrak{W} \subseteq \nu S$. $\Diamond$

We use the same coinductive technique to prove that the substitution of type variables is sound, and that the subtyping relation is closed under name permutation.

Proposition 3.4 (Substitution of type variables). *For all $\Delta, \Delta' \in \mathcal{P}(\mathcal{J})$, and $\tau, \sigma, \delta, \delta' \in \mathcal{T}$, if $\Delta, X \leq \delta, \Delta' \vdash \tau \leq \sigma$ and $\Delta \vdash \delta' \leq \delta$ then we have $\Delta, \Delta'[X \leftarrow \delta'] \vdash \tau[X \leftarrow \delta'] \leq \sigma[X \leftarrow \delta']$.*

Proposition 3.5 (Equivariance). *For all $\Delta \in \mathcal{P}(\mathcal{J})$, $\tau, \sigma \in \mathcal{T}$, if $\Delta \vdash \tau \leq \sigma$ then $\Delta[X \leftrightarrow Y] \vdash \tau[X \leftrightarrow Y] \leq \sigma[X \leftrightarrow Y]$.*

We now prove that νS is transitive by considering a combined property where transitivity is expressed together with narrowing. We start by defining narrowing of typing environments, and then closure of νS under narrowing.

Definition 3.6. *For all $\Delta, \Delta' \in \mathcal{D}$, we have that Δ is narrower than Δ' with relation to $\mathfrak{R} \in \mathcal{P}(\mathcal{J})$, written $\Delta \sqsubseteq_{\mathfrak{R}} \Delta'$, where the relation $\sqsubseteq_{\mathfrak{R}}$ is inductively defined by letting $\emptyset \sqsubseteq_{\mathfrak{R}} \emptyset$ and*

$$\Gamma, X \leq \gamma \sqsubseteq_{\mathfrak{R}} \Gamma', X \leq \gamma' \quad \text{if } \Gamma \sqsubseteq_{\mathfrak{R}} \Gamma' \text{ and } (\Delta, \gamma, \gamma') \in \mathfrak{R}.$$

Definition 3.7. *For all $n \in \mathbb{N}$ we inductively define the sets $\mathfrak{N}^n$ as follows:*

$$\begin{aligned}&\mathfrak{N}^0 \triangleq \nu S \\ &\mathfrak{N}^n \triangleq \{(\Delta, \tau, \sigma) \mid (\Gamma, \tau, \sigma) \in \mathfrak{N}^{n-1} \text{ and } \Delta \sqsubseteq_{\mathfrak{N}^{n-1}} \Gamma\}.\end{aligned}$$

We then define the transitive closure of νS, and prove that νS is closed under transitivity. Instead of a more direct definition, for technical convenience we present the transitivity relation based on chains of tuples with finite length.

Definition 3.8 (Extended Transitive Closure of νS).

$$\mathfrak{T} \triangleq \{(\Delta, \alpha_0, \alpha_n) \mid \exists_{n \in \mathbb{N}}.\exists_{\alpha_0..\alpha_n \in \mathcal{T}}.\forall_{i \in 0..n-1}.(\Delta, \alpha_i, \alpha_{i+1}) \in \mathfrak{N}^n\}$$

Notice that $\mathfrak{T}$ includes the transitive closure of νS ($n = 2$), and closure under narrowing ($n = 1$). We can then state and prove

Proposition 3.9 (Transitivity). *If $\Delta \vdash \tau \leq \sigma$ and $\Delta \vdash \sigma \leq \gamma$ then $\Delta \vdash \tau \leq \gamma$.*

Proof. We show that $\mathfrak{T} \subseteq S(\mathfrak{T})$, using an inner induction on the number of tuples and by case analysis on the last rule used on the first tuple of a chain in $\mathfrak{T}$. We then conclude $\mathfrak{T} \subseteq \nu S$ by the coinduction principle. $\Diamond$

An interesting fact about our proof is that it exposes the loss of transitivity elimination pointed out in Ghelli's inductive system [11]. In the particular case of the variable transitivity, a tuple of $\mathfrak{T}$ supported by a chain of tuples in νS is supported in $S(\mathfrak{T})$ by a longer chain of tuples. Which nevertheless leads to the conclusion that the tuple is in the greatest fixed point of S.

3.3 Type safety

We can now state and prove subject reduction for our class based programming language, that can then be used to show that well typed programs "don't go wrong" along usual lines.

Theorem 3.10 (Subject Reduction). *If $\Delta \vdash e : \tau$ and $e \Downarrow v$ then $\Delta \vdash v : \tau'$ where $\Delta \vdash \tau' \leq \tau$.*

Proof. By induction on the length of the typing derivations and by case analysis on the last rule used. We also use in several parts the fact that all valid subtyping judgments are supported in νS. $\Diamond$

As a result of this section we obtain declarative type and subtype systems whose implementability and decidability is far from being obvious (full detailed proofs for the results in this paper can be found in [20]). In the next section, we define and explain two simple algorithms that implement them.

4 Typing algorithm

We define a type checking algorithm for our type system, thus proving that it is decidable. The algorithm is composed by two procedures: a typing algorithm that given an environment and a typable expression returns its the minimal type, and a subtyping algorithm, that is called by the typing procedure to verify the subtyping relations.

4.1 Typing expressions

Typing of expressions is implemented by interpreting a set of rules bottom up: this defines a procedure that given a typing environment and a language expression returns a type. The new algorithmic rules are shown in Fig. 6, to these rules we must add (T-Var), (T-Let), (T-Fix), and (T-New), which are exactly

$$\frac{\begin{pmatrix} c = \{m_i(x_{j_i} : \tau_{j_i}{}^{j_i \in 1..n_i}) : \tau_i = e_i{}^{i \in 1..n}\} \\ I = \{m_i(\tau_{j_i}{}^{j_i \in 1..n_i}) : \tau_i{}^{i \in 1..n}\} \end{pmatrix} \\ \Delta, X_j \leq \delta_j{}^{j \in 1..m}, x_{j_i} : \tau_{j_i}{}^{j_i \in 1..n_i}, s : I \vdash_a e_i : \tau_i' \quad \Delta \vdash \tau_i' \leq \tau_i \;\; \forall_{i \in 1..n}}{\Delta \vdash_a \mathsf{class}[X_j \leq \delta_j{}^{j \in 1..m}](s)\; c : \mathsf{Class}[X_j \leq \delta_j{}^{j \in 1..m}]\; I} \text{(A-Class)}$$

$$\frac{\Delta, x_{j_i} : \tau_{j_i}{}^{j_i \in 1..n_i} \vdash e_i : \tau_i' \qquad \Delta \vdash \tau_i' \leq \tau_i \;\; \forall_{i \in 1..n}}{\Delta \vdash \{\; m_i(x_{j_i} : \tau_{j_i}{}^{j_i \in 1..n_i}) = e_i{}^{i \in 1..n}\;\} : \{m_i(\tau_{j_i}{}^{j_i \in 1..n_i}) : \tau_i{}^{i \in 1..n}\}} \text{(A-Object)}$$

$$\frac{\Delta \vdash_a e : \gamma \quad \Delta \vdash \gamma \Uparrow \gamma' \quad (\tau, \tau_i{}^{i \in 1..n}) = \mathit{lookup}(m, \gamma') \\ \Delta \vdash_a e_i : \tau_i' \qquad \Delta \vdash \tau_i' \leq \tau_i \; \forall_{i \in 1..n}}{\Delta \vdash_a e.m(e_i{}^{i \in 1..n}) : \tau} \text{(A-Call)}$$

$$\frac{X \leq \sigma \in \Delta \qquad \Delta \vdash \sigma \Uparrow \tau}{\Delta \vdash X \Uparrow \tau} \text{(X-Var)} \qquad \frac{\tau \notin \mathit{Dom}(\Delta)}{\Delta \vdash \tau \Uparrow \tau} \text{(X-Default)}$$

Fig. 6. Algorithmic typing rules

as in Fig. 4. The resulting proof system is algorithmic because in all rules the resulting types are constructed either from the expression itself or from the types resulting from typing strictly smaller subexpressions. Notice that not every rule in Fig. 4 has a corresponding rule here; the (T-Sub) rule is not used in the algorithm as it depends on an unknown type that cannot be obtained neither from the expression nor from the types of the subexpressions, we replace it by subtyping verifications in rules (A-Class), (A-Object), and (A-Call). Moreover, we use an auxiliary function *lookup* to find a method in an interface and the judgment $\Delta \vdash \tau \Uparrow \sigma$, defined by the rules (X-Var) and (X-Default), to access the structure of type variables.

4.2 Subtyping algorithm

We now present and prove correct our subtyping algorithm for deciding membership of a tuple $t \in \mathcal{J}$ in the greatest fixed point νS. The algorithm is defined by the recursive procedure shown in Fig. 7, and closely follows existing approaches for first-order equirecursive types [2,3,9]. Briefly, these algorithms progress by computing, given a pair of types to be checked for subsumption, a consistent set of pairs that includes it: by the coinduction principle, all the pairs in the set belong to the greatest fixed point. The consistent set is built by saturating the current approximation through backward rule application, and accumulating pairs of types, until a terminal case, corresponding to the application of an axiom, or an already visited pair is found.

We naturally extend those approaches building on the generating function in Definition 3.1, by defining an algorithm that manipulates judgments instead of pairs of types; this turns out to lead to a remarkably simple way of dealing with the binding information of the type variables. Notice that environments grow as a result of comparing polymorphic types, and, due to α-equivalence, the

```
$Subtyping(A, (\Delta, \tau, \sigma)) =$
    if $(\Delta, \tau, \sigma) \in^{\simeq} A$ then $A$
    else let $A_0 = A \cup \{(\Delta, \tau, \sigma)\}$ in
    if $\tau \equiv \sigma$ then $A$
    else if $\sigma \equiv \mathsf{Top}$ then $A$
    else if $\tau \equiv X$ then $Subtyping(A_0, (\Delta, \Delta(X), \sigma))$
    else if $\tau \equiv \mu X.\tau'$ then $Subtyping(A_0, (\Delta, \tau'[X \leftarrow \tau], \sigma))$
    else if $\tau \not\equiv \mu X.\tau'$ and $\sigma \equiv \mu X.\sigma'$ then $Subtyping(A_0, (\Delta, \tau, \sigma'[X \leftarrow \sigma]))$
    else if $\tau \equiv \{m_i(\tau_{j_i}{}^{j_i \in 1..n_i}) : \tau_i{}^{i \in 1..n}\}$ and $\sigma \equiv \{m_i(\tau'_{j_i}{}^{j_i \in 1..n_i}) : \tau'_i{}^{i \in 1..n'}\}$ then
            $A^{\ell}_{n_\ell}$ where $\forall_{i \in 1..n}\ (A^i_0 = Subtyping(A^{i-1}_{m_{i-1}}, (\Delta, \tau_i, \tau'_i))$ and
                                      $\forall_{j \in 1..n_i}\ \ A^i_j = Subtyping(A^i_{j-1}, (\Delta, \tau'_{j_i}, \tau_{j_i})))$
            with $A^0_{n_0} = A_0$
    else if $\tau \equiv \mathsf{Class}[X_i \leq \delta_i{}^{i \in 1..n}]\ I$ and $\sigma \equiv \mathsf{Class}[X_i \leq \delta_i{}^{i \in 1..n}]\ I'$ then
            $Subtyping(A_0, (\Delta, X_i \leq \delta_i{}^{i \in 1..n}; I; I'))$
    else $fail$
```

Fig. 7. Subtyping algorithm

greatest fixed point νS is closed under renaming (Proposition 3.5). Moreover, in our setting, the number of tuples reachable from a given tuple are finite up to such renaming and pruning of useless variables (Lemma 4.5). Therefore, our algorithm checks for membership of a tuple in the current approximation modulo a similarity relation on tuples that includes renaming, and allows us to detect cycles at the level of similarity equivalence classes, instead of expecting the exact tuple to reappear in the sequence of calls.

Definition 4.1 (Similarity). Similarity *is the binary relation* $\simeq$ *on* $\mathcal{J}$ *defined by:* $(\Delta, \tau, \sigma) \simeq (\Delta', \tau', \sigma')$*, if there are two typing environments* $\Gamma \subseteq \Delta$ *and* $\Gamma' \subseteq \Delta'$ *with* $\Gamma \vdash \tau$ ok*,* $\Gamma \vdash \sigma$ ok*, and* $\Gamma' \vdash \tau'$ ok*,* $\Gamma' \vdash \sigma'$ ok*, and a bijection* $\rho : Dom(\Gamma) \to Dom(\Gamma')$ *such that* $\rho(\Gamma) = \Gamma'$*,* $\rho(\tau) = \tau'$*,* $\rho(\sigma) = \sigma'$*.*

In the sequel we will use the following abbreviation $t \in^{\simeq} A \triangleq \exists u \in A. t \simeq u$, in particular in the first clause of the subtyping algorithm. Notice that similarity is decidable, it can be checked by matching the structure of the types, modulo bijective renaming of their free type variables, and recursively checking if the corresponding bounds are similar. All unused variables are discarded from the comparison as it goes through the structure of types and bounds of relevant type variables in the environments. For instance, the following two tuples are similar:

$$(X \leq \{m(\tau){:}\tau\}, Y \leq X; \mu Z.\{m(X){:}Z\}; X) \simeq (Z \leq \{m(\tau){:}\tau\}; \mu X.\{m(Z){:}X\}; Z)$$

Notice the redundancy of Y, and that the tuples are similar with the permutation $[X \leftrightarrow Z]$.

An important fact is that the subtyping relation νS is closed under similarity.

Definition 4.2 (Closure under similarity). *For any* $R \in \mathcal{P}(\mathcal{J})$ *we define its closure under similarity, noted* R^**, as follows:*

$$R^* \triangleq \{t' \mid t' \in \mathcal{J} \text{ and } t \in R \text{ and } t' \simeq t\}.$$

Lemma 4.3. *νS is closed under similarity ($\nu S^* = \nu S$).*

Proof. By substitution of type variables (Proposition 3.4), equivariance (Proposition 3.5) and then by weakening (Proposition 3.3). $\Diamond$

We now prove the correctness and decidability of the subtyping algorithm. We first show that the algorithm terminates on all inputs. This is done by showing that the search space of the algorithm is finite in some sense. To characterize such search space we introduce the following reachability relation:

Definition 4.4 (Reachability). *Reachability is the binary relation on $\mathcal{J}$, noted $t \gg t'$, inductively defined as follows:*

1. $(\Delta, \tau, \sigma) \gg (\Delta, \tau, \sigma)$
2. *if* $(\Delta, \tau, \sigma) \gg (\Delta', X, \sigma')$ *then* $(\Delta, \tau, \sigma) \gg (\Delta', \Delta'(X), \sigma')$
3. *if* $(\Delta, \tau, \sigma) \gg (\Delta', \mathsf{Class}[X_i \leq \delta_i\ {}^{i\in 1..n}]\ I; \mathsf{Class}[X_i \leq \delta_i\ {}^{i\in 1..n}]\ I')$
   then $(\Delta, \tau, \sigma) \gg (\Delta', X_j \leq \delta_j\ {}^{j\in 1..n}; I; I')$
4. *if* $(\Delta, \tau, \sigma) \gg (\Delta'; I; \mu X.J)$ *then* $(\Delta, \tau, \sigma) \gg (\Delta'; I; J[X \leftarrow \mu X.J])$
5. *if* $(\Delta, \tau, \sigma) \gg (\Delta'; \mu X.I; J)$ *then* $(\Delta, \tau, \sigma) \gg (\Delta'; I[X \leftarrow \mu X.I]; J)$
6. *if* $(\Delta, \tau, \sigma) \gg (\Delta'; \{m_i(\tau_{j_i}\ {}^{j_i\in 1..n_i}) : \tau_i\ {}^{i\in 1..n}\}; \{m_i(\tau'_{j_i}\ {}^{j_i\in 1..n'_i}) : \tau'_i\ {}^{i\in 1..n'}\})$
   then $(\Delta, \tau, \sigma) \gg (\Delta'; \tau_i; \tau'_i)\ \forall_{i\in 1..n'}$
   and $(\Delta, \tau, \sigma) \gg (\Delta'; \tau'_{j_i}; \tau_{j_i})\ \forall_{j_i\in 1..n_i}\forall_{i\in 1..n'}$

We let $\mathit{Reach}(t) \triangleq \{t' \mid t \gg t'\}$. We have the following

Lemma 4.5. *$\mathit{Reach}(t)_{/\simeq}$ is finite.*

Proof. We prove by induction on the reachability relation that all types occurring in reachable tuples are subexpressions of the initial tuple and that the number of relevant type variables in those tuples, both in the environments and in the type expressions, decreases with relation to the initial tuple. We then prove by contradiction that these two results support the fact that the set of reachable tuples, *Reach*(t), is finite modulo similarity. $\Diamond$

It is important to remark that the finite reachability property of Lemma 4.5 holds both for Kernel-Fun and $F^{\top}_{\leq}$ style subtyping, although in the first case the proof is slightly more involved (see [20]). The proof also enlightens why the same result cannot be extended to $F_{\leq}$.

Theorem 4.6. *For all $A \in \mathcal{P}(\mathcal{J})$, and $t \in \mathcal{J}$, Subtyping(A, t) terminates.*

Proof. By induction using a measure that represents the number of non-visited equivalence classes of reachable tuples of t. Since the algorithm always increases the visited tuples with a tuple reachable from t, the measure decreases in all cases. Hence, the algorithm terminates. $\Diamond$

To prove that our algorithm is sound and complete, it is technically convenient to follow the approach of [9] and introduce a function *gfp* that allows us to characterize νS in a form both suitable for the correctness proofs and for establishing the correspondence between the algorithm and the extensional definition of the subtyping relation. Moreover, unlike the analogous notion in [9], instead of accumulating tuples, our *gfp* function works with $\simeq$ equivalence classes.

Definition 4.7. *Let gfp be the partial function $\mathcal{P}(\mathcal{J}) \times \mathcal{J} \to \mathcal{P}(\mathcal{J})$ defined by:*

```
$gfp(A,t) =$ if $\{t\}^* \subseteq A$ then $A$
             else if $support(t)$ is undefined then undefined
             else let $\{t_1,\dots,t_n\} = support(t)$ in
               let $A_0 = A \cup \{t\}^*$ in
               let $A_1 = gfp(A_0, t_1)$ in
               $\dots$
               let $A_n = gfp(A_{n-1}, t_n)$ in $A_n$.
```

We prove next that *gfp* indeed characterizes the subtyping relation.

Lemma 4.8 (Correctness of *gfp***).** *For all $t \in \mathcal{J}$, and $A \in \mathcal{P}(\mathcal{J})$,*

1. *if $gfp(\emptyset, t) = A$ then $t \in \nu S$.*
2. *if $gfp(\emptyset, t)$ is undefined then $t \notin \nu S$.*

Proof. 1. By induction in the definition of *gfp* to prove the following more general property: for all $A, A' \in \mathcal{P}(\mathcal{J})$, and $t \in \mathcal{J}$, if $A^* \subseteq A$ and $gfp(A,t) = A'$ then $A \subseteq A', A'^* \subseteq A'$, $\{t\}^* \subseteq A'$, and $A' \subseteq S(A') \cup A$. We then conclude by considering $A = \emptyset$. 2. Also by induction on the definition of *gfp*. ♢

Knowing that *gfp* correctly checks if any tuple belongs to the subtyping relation νS, we prove that *gfp* and the subtyping algorithm *Subtyping* are equivalent. Since the algorithm terminates on all inputs (Theorem 4.6), we can show that it is sound and complete.

Lemma 4.9 (Correctness of *Subtyping***).** *For all $t \in \mathcal{J}$, $A, A' \in \mathcal{P}(\mathcal{J})$,*

1. *$Subtyping(A,t) = A'$ iff $gfp(A^*, t) = A'^*$.*
2. *$Subtyping(A,t) = fail$ iff $gfp(A^*, t)$ is undefined.*

Proof. By induction on the recursive calls of *Subtyping*. ♢

Theorem 4.10 (Correctness of *Subtyping***).** *For all $\Delta \in \mathcal{D}$, and $\tau, \sigma \in \mathcal{T}$,*

$$Subtyping(\emptyset, (\Delta, \tau, \sigma)) = A \text{ iff } \Delta \vdash \tau \leq \sigma$$

Proof. It follows directly from Lemmas 4.8 and 4.9. ♢

To summarize, we conclude that the type system defined in section 3 is decidable and that our typing and subtyping algorithms are sound and complete.

5 Composition of classes

We have presented a language that treats classes as first class values but that lacks class composition operations, so that no new class values can be actually created at runtime. In this section, we discuss an interesting and possible extension of this object language so to include class manipulation mechanisms,

$$\begin{array}{c}
\left(\begin{array}{l}
\pi = [X'_i \leftarrow \gamma_i{}^{\,i\in 1..n'}] \qquad \pi' = [X''_i \leftarrow \gamma'_i{}^{\,i\in 1..n''}] \\
I = \{m_i(\vec{\sigma}\ \pi) : \tau_i\pi^{\,i\in\mathcal{I}-\mathcal{J}},\ m_i(\vec{\sigma'}\ \pi') : \tau'_i\pi'^{\,i\in\mathcal{J}}\}\
\end{array}\right) \\
\varDelta \vdash e_1 : \mathsf{Class}[X'_i \leq \delta'_i{}^{\,i\in 1..n'}]\ \{m_i(\vec{\sigma_i}) : \tau_i{}^{\,i\in\mathcal{I}}\} \quad \varDelta \vdash \gamma_i \leq \delta'_i\ \ \forall_{i\in 1..n'} \\
\varDelta \vdash e_2 : \mathsf{Class}[X''_i \leq \delta''_i{}^{\,i\in 1..n''}]\ \{m_i(\vec{\sigma'_i}) : \tau'_i{}^{\,i\in\mathcal{J}}\} \quad \varDelta \vdash \gamma'_i \leq \delta''_i\ \ \forall_{i\in 1..n''} \\
\hline
\varDelta \vdash \mathsf{mix}[X_i \leq \delta_i{}^{\,i\in 1..n}](e_1[\gamma_i{}^{\,i\in 1..n'}] \lhd e_2[\gamma'_i{}^{\,i\in 1..n''}]) : \mathsf{Class}[X_i \leq \delta_i{}^{\,i\in 1..n}]\ I
\end{array}$$

$$\begin{array}{c}
\left(\begin{array}{l}
\pi = [s \leftarrow s_{(m_i \leftrightarrow m'_i{}^{\,i\in\mathcal{I}\cap\mathcal{J}})}]\ (m'_i \text{ fresh}) \quad \pi' = [X'_i \leftarrow \gamma_i{}^{\,i\in 1..n'}][X''_i \leftarrow \gamma'_i{}^{\,i\in 1..n''}] \\
v = \{m_i(\vec{\sigma_i}) : \tau_i = e_i\pi^{\,i\in\mathcal{I}-\mathcal{J}},\ m_i(\vec{\sigma'_i}) : \tau'_i = e'_i{}^{\,i\in\mathcal{J}},\ m'_i(\vec{\sigma_i}) : \tau_i = e_i\pi^{\,i\in\mathcal{I}\cap\mathcal{J}}\}
\end{array}\right) \\
e_1 \Downarrow \mathsf{class}[X'_i \leq \delta'_i{}^{\,i\in 1..n'}](s)\ \{m_i(\vec{\sigma_i}) : \tau_i = e_i{}^{\,i\in\mathcal{I}}\} \\
e_2 \Downarrow \mathsf{class}[X''_i \leq \delta''_i{}^{\,i\in 1..n''}](s)\ \{m_i(\vec{\sigma'_i}) : \tau'_i = e'_i{}^{\,i\in\mathcal{J}}\} \\
\hline
\mathsf{mix}[X_i \leq \delta_i{}^{\,i\in 1..n}](e_1[\gamma_i{}^{\,i\in 1..n'}] \lhd e_2[\gamma'_i{}^{\,i\in 1..n''}]) \Downarrow \mathsf{class}[X_i \leq \delta_i{}^{\,i\in 1..n}](s)\ v\pi'
\end{array}$$

$$\frac{e \Downarrow v}{e_{(m\leftrightarrow m')} \Downarrow v(m \leftrightarrow m')} \qquad \frac{\varDelta \vdash e : \tau}{\varDelta \vdash e_{(m\leftrightarrow m')} : \tau(m \leftrightarrow m')}$$

Fig. 8. Typing and evaluation for mix.

similar to inheritance or mixin application. As an alternative, we propose a general mechanism of class composition which combines two classes without any of them having to be developed with extension in mind. The composition mechanism is expressed as follows:

$$\mathsf{mix}[X_i \leq \delta_i{}^{\,i\in 1..n}](e_1[\gamma_i{}^{\,i\in 1..n'}] \lhd e_2[\gamma'_i{}^{\,i\in 1..n''}])$$

It takes two class values, e_1 and e_2, and produces a new class value, parameterized by a fresh set of type parameters, containing the methods in e_1 and e_2 and where name clashes are resolved in favor of e_2. This is apparent in the rules in Fig. 8 which must be added to the type and evaluation systems to extend the initial language. Notice that for a mix to be well typed e_1 and e_2 must denote class values, and that the arguments $\gamma_i{}^{\,i\in 1..n'}$ and $\gamma'_i{}^{\,i\in 1..n''}$ must be compatible with the bounds of each class value.

It is well known that extending a class by simply replacing the methods of one class with methods of another easily generates type inconsistencies (*e.g.* see [1,6]). To avoid this, we maintain the local coherence of the subsumed class, which in this case is e_1, by means of an explicit permutation of method names, $e_{(m\leftrightarrow m')}$. This corresponds to the run-time permutation of method names between m and m' and $\tau(m \leftrightarrow m')$ has the same meaning but for type expressions. The typing and evaluation rules for this expression are also depicted in 8.

So, whenever s, now replaced by $s_{(m_i\leftrightarrow m'_i{}^{\,i\in\mathcal{I}\cap\mathcal{J}})}$, is evaluated in the body of a method of e_1, the methods in s are switched back to the subsumed methods of e_1 that were hidden under a different name. The evaluation of such methods and possibly of other methods using the self reference passed by a method of e_1 is type preserving, and thus the core language extended with mix can be shown to enjoy a subject reduction property [20].

6 Related work and concluding remarks

It is known that the type system of $F_{\leq}$ is not decidable [12,16] and that its extension with recursive types is non-conservative [11]. Simpler versions were proposed but even so, not free from problems: system $F_{\leq}^{\top}$, proposed by Castagna and Pierce, is decidable revealed to lack the minimal typing property [4] and the approach to subtyping in Kernel Fun extended with recursive types by Colazzo and Ghelli [5] results in a fairly complex algorithm; it uses a labeling mechanism to identify cycles in derivations and stop the unfolding process. The authors show that the use of renaming of type variables results in an divergent algorithm and that this labeling technique is sound. However, the algorithm only stops unfolding a given pair of types the third time it occurs. The reasons for this fact are far from intuitive. On the contrary, our approach and algorithm are a natural extension of well known techniques for first-order types. Alan Jeffrey defines in [14] a notion of simulation for higher order types using a symbolic labeled transition systems, and uses it to define a (non decidable) subtyping relation in $\mu F_{\leq}$, there is then some connection between his approach and the general coinductive techniques we have presented here. An efficient algorithm to unify two recursive second-order types was proposed by Gauthier and Pottier in [10]. It relies on an encoding of second-order type expressions into first-order trees, and on the application of standard first-order unification algorithms for infinite trees. We have no perspective on how this may be adapted to the subtyping problem.

On the other hand, we present a subtyping algorithm for second-order systems with equirecursive types which is an uniform extension of existing work on first-order equirecursive types by Amadio and Cardelli [2], Brandt and Henglein [3]. In particular, we build on the coinductive presentations of first-order type systems with recursive types of Gapeyev, Levin, and Pierce [9,17]. Our treatment of reachability modulo a similarity relation that includes equivariance (Lemma 3.5) is inspired on notions by Gabbay and Pitts [8]. Our definition and correctness proof is, from our point of view, much simpler than the ones of the algorithms referred above. Our proofs are modular, in the sense that they can be applied to any polymorphic type system with recursive types that satisfies a certain finite reachability condition up to a notion of similarity that includes equivariance. In particular, they suggest an interesting decidable fragment of $F_{\leq}$, defined by restricting the subtype rule for $\forall X \leq \tau.\sigma$ types to just compare bounds with the same free type variables, we leave this topic for future work.

Given these results, we develop a class based language and discuss a possible extension of it that allows combination of classes with a mixin like construct, while avoiding the unsoundness problems of subsumption and class extension often found in object calculi.

We would like to thank Dario Colazzo and the reviewers for their comments on a preliminary version of this work. This work is partially supported by FCT/MCES.

References

1. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
2. Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
3. Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2–4, 1997*, volume 1210, pages 63–81. Springer-Verlag, 1997.
4. Giuseppe Castagna and Benjamin Pierce. Corrigendum: Decidable bounded quantification. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*. ACM, January 1995.
5. Dario Colazzo and Giorgio Ghelli. Subtyping recursive types in Kernel Fun. In *14th Symp. on Logic in Computer Science (LICS'99)*, pages 137–146, 1999.
6. William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 497–518. The MIT Press, Cambridge, MA, 1994.
7. Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. In *Theoretical aspects of object-oriented programming: types, semantics, and language design*, pages 247–292. MIT Press, 1994.
8. M. Gabbay and A. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2002.
9. Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. In *Proc. of the Intl. Conference on Functional Programming (ICFP)*, 2000.
10. Nadji Gauthier and François Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *Proc. of the 2004 ACM SIGPLAN Intl. Conference on Functional Programming (ICFP'04)*, pages 150–161, 2004.
11. Giorgio Ghelli. Recursive types are not conservative over $\mathrm{F}_{\leq}$. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculus and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1993.
12. Giorgio Ghelli. Divergence of $F_{\leq}$ type checking. *Theoretical Computer Science*, 139(1-2):131–162, 1995.
13. Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proc. of the 1999 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.
14. Alan Jeffrey. A symbolic labelled transition system for coinductive subtyping of $F_{\mu\leq}$. In *16th Annual IEEE Symposium on Logic in Computer Science*, June 2001.
15. Betti Venneri Lorenzo Bettini, Viviana Bono. Subtyping mobile classes and mixins. In *FOOL 10*, 2003.
16. Benjamin C. Pierce. Bounded quantification is undecidable. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 427–459. MIT Press, 1994.
17. Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
18. João Costa Seco and Luís Caires. A basic model of typed components. In *Proc. of the European Conf. on Object-Oriented Programming (ECOOP)*, 2000.
19. João Costa Seco and Luís Caires. The parametric component calculus. Technical Report UNL-DI-7-2002, FCT-UNL, 2002.
20. João Costa Seco and Luís Caires. Subtyping first class polymorphic components. Technical Report UNL-DI-1-2005, FCT-UNL, 2004.