Subtyping Recursive Types modulo Associative Commutative Products

Roberto Di Cosmo,[1] François Pottier,[2] and Didier Rémy[2]

[1] Université Paris 7 roberto@dicosmo.org
[2] INRIA Rocquencourt {Francois.Pottier, Didier.Remy}@inria.fr

Abstract. This work sets the formal bases for building tools that help retrieve classes in object-oriented libraries. In such systems, the user provides a query, formulated as a set of class interfaces. The tool returns classes in the library that can be used to implement the user's request and automatically builds the required glue code. We propose subtyping of recursive types in the presence of associative and commutative products—that is, subtyping modulo a restricted form of type isomorphisms—as a model of the relation that exists between the user's query and the tool's answers. We show that this relation is a composition of the standard subtyping relation with equality up to associativity and commutativity of products and we present an efficient decision algorithm for it. We also provide an automatic way of constructing coercions between related types.

1 Introduction

The study of type isomorphisms is concerned with identifying data types by abstracting away from irrelevant details in the syntax of types, or—in other words—irrelevant choices in the representation of data. The basic idea is quite simple: one wishes to identify two data types if data of one type can be transformed into data of the other type without loss of information. Formally speaking, τ_1 and τ_2 are said to be isomorphic if and only if there exist functions $f : \tau_1 \to \tau_2$ and $g : \tau_2 \to \tau_1$ that are mutual inverses, in the sense that they make the following diagram commute:

$$\begin{tikzcd} id_{\tau_1} \arrow[loop left] & \tau_1 \arrow[r, bend left, "f"] & \tau_2 \arrow[l, bend left, "g"] \arrow[loop right] & id_{\tau_2} \end{tikzcd}$$

This study has wide ramifications in different research fields, ranging from number theory to category theory, and from λ-calculus to proof theory [14]. In particular, it helps attack some of the problems raised by the growing complexity of today's multi-language code bases. Indeed, the vast majority of currently available search tools suffer from the fact that they only allow *textual* searches of libraries for method or function *names*, while such names are largely arbitrary. An interesting instance of this phenomenon is the ubiquity of the *fold* identifier in ML-like languages, pointed out by Rittri [20].

The key idea behind the use of type isomorphisms in information retrieval is to forget about names altogether and to rely on *types* instead. Indeed, a type provides a

(possibly partial) specification of a component. Of course, types must be equated up to type isomorphisms, so as to make queries robust against arbitrary choices on the library implementor's part. *Which* type isomorphisms to use depends on the type system, the programming language, and the observational equivalence at hand. A large variety of complete equational theories are known that axiomatize type isomorphisms in various core calculi. Probably best known is the theory of isomorphisms for Cartesian Closed Categories—the models of the simply-typed lambda calculus with products and a unit type [23, 10]:

1.	$\tau \times \tau' = \tau' \times \tau$	5.	$\tau \times 1 = \tau$
2.	$\tau \times (\tau' \times \tau'') = (\tau \times \tau') \times \tau''$	6.	$\tau \to 1 = 1$
3.	$(\tau \times \tau') \to \tau'' = \tau \to (\tau' \to \tau'')$	7.	$1 \to \tau = \tau$
4.	$\tau \to (\tau' \times \tau'') = (\tau \to \tau') \times (\tau \to \tau'')$		

The type isomorphisms-based approach can help in retrieving complex software components from large libraries of functions [13, 21, 22] or modules [25, 3] and in automatically producing bridge code between different representations of a (possibly recursive) data type in systems like Mockingbird [5, 6]. These active areas are currently attracting the attention of many researchers. Unfortunately, the general problem of characterizing isomorphic types for a full-fledged type system, including sums, products, polymorphic and recursive types—such as that underlying Mockingbird [4, 8]—is extremely complex and remains open; there are, in particular, difficulties with recursive types [1] and with sum types [7]. In view of this difficulty, Jha, Palsberg, and Zhao [19, 18] proposed to study a weak approximation of isomorphisms of recursive types, obtained by viewing products as associative and commutative, which we refer to as AC-equality. This relation may be decided in time $O(N \log N)$, where N is the sum of the sizes of the input types. (The same time bound was obtained by Downey, Sethi and Tarjan [15] for the closely related problem of *symmetric congruence closure*.) AC-equality captures a lot of the inessential syntactic details one wants to get rid of when querying a library. Jha *et al.* propose to view a collection of Java interface declarations as a collection of types, using arrow types to encode method signatures and n-ary products to encode collections of methods or method parameters. Of course, the types thus obtained are recursive, because Java interface declarations may mutually refer to one another. For instance, the Java interfaces:

```
interface I1 {          interface I2 {
  float m1 (I1 a);        I1 m3 (float a);
    int m2 (I2 a);        I2 m4 (float a);
}                       }
```

may be encoded (forgetting method names) as the mutually recursive types $I_1 = (I_1 \to \mathit{float}) \times (I_2 \to \mathit{int})$ and $I_2 = (\mathit{float} \to I_1) \times (\mathit{float} \to I_2)$. Thus, the notion of AC-equality of recursive types gives rise to a notion of equivalence between (collections of) Java interfaces.

However, AC-equality is not the right relation on types for searching libraries. As noted by Thatté [24], when querying a complex object-oriented library, the actual type of the desired class or interface may be extremely complex, because it lists *all* of (the

types of) its methods. As a result, it is not reasonable to require the query (that is, the type provided by the user) to be AC-*equal* to the actual type. Indeed, the user would have to guess the list of (the types of) *all* methods in the class. It is more realistic to allow the user to formulate a query that is only a *supertype* of the actual type, so for instance, a user looking for a collection implementation may formulate the query:

```
public interface SomeCollection {
  public void    add (Object o);
  public void    remove (Object o);
  public boolean contains (Object o);
  public int     size ();
}
```

In the Java standard library, the `Collection` interface has 15 methods. As a result, every class that implements it has at least 15 methods as well, which means that *no* match will be found for this query if types are compared up to AC-equality. The purpose of this paper is to introduce a notion of AC-*subtyping* defined so that the `Collection` interface is an AC-*subtype* of this query. Furthermore, even such a simple notion of isomorphism of recursive types can give rise to very complex conversion functions. As a result, it is quite unrealistic to expect that a user could be satisfied with a mere *true* or *false* answer. A practical search system must be able to generate code for converting between the search result and the search query, as already advocated by Thatté [24].

In this paper, we pursue Thatté's seminal work and give an efficient subtyping algorithm modulo AC for a core language with products, arrows, and recursive types. The algorithm also produces coercion code when it succeeds. We believe that when the language is extended to cover a class-based object-oriented language such as Java, our algorithm could be combined with ideas from Thatté to synthesize adapters for existing classes.

The paper is laid out as follows. §2 gives a comparison with related work and an overview of our results. In §3, we recall a few basic notions about recursive types, as well as Palsberg and Zhao's notion of equality up to associativity and commutativity of products [19]. In §4, we introduce the notion of AC-subtyping and prove that it is a composition of the usual subtyping relation with AC-equality. Then, in §5, we describe an algorithm that decides whether two types are in the subtyping relation modulo associativity and commutativity of products. We establish its correctness and assess its time complexity. In §6, we discuss how to generate code for coercion functions. Throughout the paper, we consider recursive types built out of arrows, products, and the constants $\bot$ and $\top$. In §7, we argue that this simple setting is general enough.

2 Related Work and Overview of our Results

Two main lines of work are closely related to ours. To our knowledge, Thatté is the first to have put forth a relaxed form of subtyping between recursive types as a model of object-oriented retrieval tools [24]. Without relating to Thatté's work, Palsberg *et al.* have studied efficient algorithms to solve AC-equality of recursive types [19, 18].

By comparison with Thatté's work, we have taken a more foundational approach by working directly with recursive types. We also use co-inductive techniques—which

were not yet popular at the time of Thatté's work—to provide an efficient, deterministic decision algorithm that improves over his exponential algorithm (essentially a variant of Amadio and Cardelli's original subtyping algorithm). However, some work remains to be done to specialize our results to classed-based languages and build "adapters", in Thatté's terminology, out of our coercions.

Technically, our co-inductive algorithms share a common ground with the work of Palsberg *et al.* on AC-equality [19]. Indeed, co-induction is a most natural tool for reasoning about recursive types. Unfortunately, many of the well-known algorithmic optimizations (inspired by classic foundational work on finite automata) that are applicable when dealing with equivalence relations [19, 18] break down when dealing with an ordering. This is very well explained by Jha *et al.* [18, Section 6], who describe AC-subtyping, but quickly dismiss it as not amenable to the optimizations used for AC-equality. The authors state that this relation is decidable, but make no effort to give a tight complexity bound or describe an actual decision algorithm. Yet, a naive generalization of Palsberg and Zhao's ideas [19] to the setting of AC-subtyping—as opposed to AC-equality—already leads to a decision procedure whose worst-case time complexity is $O(n^2 n'^2 d^{5/2})$ (1), where n and n' count the sub-terms of the types that are being compared and d is a bound on the arity of the products involved.

The naive procedure starts from the full relation—a graph with $O(nn')$ edges—and repeatedly removes edges that are found not to be in the AC-subtyping relation. Because it might be necessary to inspect all edges in order to remove only one of them, and because, in the worst case, all edges have to be removed, the procedure might require $O(n^2 n'^2)$ edge inspections, each of which happens to require time $O(d^{5/2})$ in the worst case.

In this paper, we improve on this naive procedure by a careful choice of the *order* in which edges must be inspected. The worst-case time complexity of our improved algorithm may be bounded by (1), which shows that it performs no worse than the naive procedure. It may also be bounded by $O(NN'd^{5/2})$ (2), where N and N' are the sizes of the types that are being compared. In practice, N and N' might be significantly less than n^2 and n'^2, respectively. Furthermore, we show that, if the types at hand are not recursive (that is, do not involve cycles), then our algorithm runs in time $O(nn'd^{5/2})$ (3). One may expect the algorithm's performance to degrade gracefully when the types at hand involve few cycles. Last, in §5, we give worst-case complexity bounds analogous to (2) and (3), but where the quantities $O(NN')$ and $O(nn')$ are replaced with the size of a certain graph. Intuition suggests that, in practice, the size of this graph might be significantly less than quadratic. For all these reasons, we expect our algorithm to perform well in practice, whereas an implementation of the naive algorithm would not be realistic—even though, in rare cases, both algorithms may require the same amount of computation.

A mild difference with Palsberg and Zhao [19] is that we allow products to be immediately nested. Indeed, our definition of AC-equality and AC-subtyping is such that flattening nested products is *not* part of equality. That is, if we write $(\tau_1 \times \ldots \times \tau_n)$ for $\Pi_{i=1}^{n} \tau_i$, then the types $(\tau_1 \times \tau_2 \times \tau_3)$ and $(\tau_1 \times (\tau_2 \times \tau_3))$ are *not* AC-related. If one wishes that these types be identified, one can preprocess the input types by flattening nested products before running our algorithm. (Of course, this is possible only in the ab-

sence of infinite products, but this restriction makes practical sense, since "flat" infinite products cannot exist in memory.) However, there are situations where we want to keep these types distinct. For example, products representing persistent database information may be kept nested, as stored on disk, while products used for passing arguments to functions may be flattened.

To sum up, we feel our work is more in line with Thatté's, in that we want to provide a formal basis for *actual* search tools, that need AC-subtyping and the automatic synthesis of the coercions, even if this means giving up the algorithmic optimizations that make deciding an equivalence relation more efficient. Still, identifying types up to AC-equality may remain useful as a preprocessing phase, in order to decrease the number of nodes in the problem that is submitted to the AC-subtyping algorithm.

3 Recursive Types

Recursive types are usually given in concrete syntax as finite systems of contractive type equations, which, according to Courcelle [12], uniquely define regular trees; or as finite terms involving μ binders [16]. The process of unfolding these finite representations gives rise to regular infinite trees.

Definition 1 (Signature). A *signature* is a mapping from *symbols*, written s, to integer *arities*. In this paper, we consider a fixed signature, which consists of a binary symbol $\rightarrow$, a n-ary symbol Π^n for every nonnegative integer n, and the constant symbols $\bot$ and $\top$. $\diamond$

Definition 2 (Path, tree, type). A *path* p is a finite sequence of integers. The empty path is written ϵ and the concatenation of the paths p and p' is written $p \cdot p'$. A *tree* is a partial function τ from paths to symbols whose domain is nonempty and prefix-closed and such that, for every path p in the domain of τ, $p \cdot i \in dom(\tau)$ holds if and only if i is comprised between 1 and the arity of the symbol $\tau(p)$, inclusive. If p is in the domain of τ, then the *subtree* of τ rooted at p, written τ/p, is the partial function $p' \mapsto \tau(p \cdot p')$. A tree is *regular* if and only if it has a finite number of distinct subtrees. (Every finite tree is thus regular.) A *recursive type* (or *type* for short) is a regular tree. We write $\mathcal{T}$ for the set of all types. We write $\bot$ (resp. $\top$) for the tree that maps ϵ to $\bot$ (resp. $\top$). We write $\tau_1 \rightarrow \tau_2$ for the tree that maps ϵ to $\rightarrow$ and whose subtrees rooted at 1 and 2 are τ_1 and τ_2, respectively. We write $\Pi_{i=1}^n \tau_i$ for the tree that maps ϵ to Π^n and whose subtree rooted at i is τ_i for every $i \in \{1, \ldots, n\}$. $\diamond$

There are many ways to present equality of recursive types, ranging from traditional definitions based on finite approximations [2] to more modern co-inductive approaches [9, 11]. Following Brandt and Henglein, we reason in terms of simulations.

Definition 3 (Equality). A binary relation $\mathcal{R} \subseteq \mathcal{T}^2$ is a $=$*-simulation* if and only if it satisfies the following implications:

$$\textsc{Eq-Top}\quad \frac{\tau \;\mathcal{R}\; \tau'}{\tau(\epsilon) = \tau'(\epsilon)} \qquad \textsc{Eq-Arrow}\quad \frac{\tau_1 \rightarrow \tau_2 \;\mathcal{R}\; \tau_1' \rightarrow \tau_2'}{\tau_1 \;\mathcal{R}\; \tau_1' \qquad \tau_2 \;\mathcal{R}\; \tau_2'} \qquad \textsc{Eq-Pi}\quad \frac{\Pi_{i=1}^n \tau_i \;\mathcal{R}\; \Pi_{i=1}^n \tau_i'}{(\tau_i \;\mathcal{R}\; \tau_i')^{i \in \{1,\ldots,n\}}}$$

Equality $=$ is the largest $=$-simulation. $\diamond$

Palsberg and Zhao [19] define equality up to associativity and commutativity of products as follows; see also Downey *et al.* [15, section 4.1]. We write Σ_n^m for the set of all injective mappings from $\{1, \ldots, m\}$ into $\{1, \ldots, n\}$. In particular, Σ_n^n is the set of all permutations of $\{1, \ldots, n\}$.

Definition 4 (AC-Equality). A binary relation $\mathcal{R} \subseteq \mathcal{T}^2$ is a $=_{AC}$*-simulation* if and only if it satisfies the following implications:

$$\textsc{Eqac-Top}\quad \frac{\tau \;\mathcal{R}\; \tau'}{\tau(\epsilon) = \tau'(\epsilon)} \qquad \textsc{Eqac-Arrow}\quad \frac{\tau_1 \to \tau_2 \;\mathcal{R}\; \tau_1' \to \tau_2'}{\tau_1 \;\mathcal{R}\; \tau_1' \qquad \tau_2 \;\mathcal{R}\; \tau_2'} \qquad \textsc{Eqac-Pi}\quad \frac{\Pi_{i=1}^n \tau_i \;\mathcal{R}\; \Pi_{i=1}^n \tau_i'}{\exists \sigma \in \Sigma_n^n \quad (\tau_{\sigma(i)} \;\mathcal{R}\; \tau_i')^{i \in \{1,\ldots,n\}}}$$

AC-Equality $=_{AC}$ is the largest $=_{AC}$-simulation. ◇

Note that a product one of whose components is itself a product is not considered AC-equal to the corresponding "flattened" product. We come back to this point in §7.

4 Subtyping and AC-Subtyping

In this section, we define subtyping of recursive types up to associativity and commutativity of products, and show that it is precisely a composition of the usual subtyping relation with equality up to associativity and commutativity of products.

Let us first define subtyping between recursive types. This requires extending the standard definition of subtyping from the case of binary products [9] to that of n-ary products.

Definition 5 (Subtyping). Let $\leq_0$ be the ordering on symbols generated by the rules:

$$\bot \leq_0 s \qquad s \leq_0 \top \qquad \to \leq_0 \to \qquad \frac{n \geq m}{\Pi^n \leq_0 \Pi^m}$$

A binary relation $\mathcal{R} \subseteq \mathcal{T}^2$ is a $\leq$*-simulation* if and only if it satisfies the following implications:

$$\textsc{Sub-Top}\quad \frac{\tau_1 \;\mathcal{R}\; \tau_2}{\tau_1(\epsilon) \leq_0 \tau_2(\epsilon)} \qquad \textsc{Sub-Arrow}\quad \frac{\tau_1 \to \tau_2 \;\mathcal{R}\; \tau_1' \to \tau_2'}{\tau_1' \;\mathcal{R}\; \tau_1 \qquad \tau_2 \;\mathcal{R}\; \tau_2'} \qquad \textsc{Sub-Pi}\quad \frac{\Pi_{i=1}^n \tau_i \;\mathcal{R}\; \Pi_{i=1}^m \tau_i'}{(\tau_i \;\mathcal{R}\; \tau_i')^{i \in \{1,\ldots,m\}}}$$

Subtyping $\leq$ is the largest $\leq$-simulation. ◇

This definition allows *depth* and *width* subtyping. Depth subtyping refers to the covariance of products. Width subtyping refers to the fact that a product with more components may be a subtype of a product with fewer components. Enabling width subtyping better suits our intended applications. Furthermore, it is possible, if desired, to introduce a distinct family of product constructors, which forbid width subtyping; see §7.

We now define subtyping of recursive types up to associativity and commutativity of products. Its definition relaxes Definition 5 by allowing the components of a product to be arbitrarily permuted. It is given in a slightly generalized style, introducing the notion of simulation *up to* a relation; this helps state the algorithm's invariant in §5.2.

Definition 6 (AC-Subtyping). Let $\mathcal{R} \subseteq \mathcal{T}^2$ and $\mathcal{R}' \subseteq \mathcal{T}^2$ be binary relations. $\mathcal{R}$ is a $\leq_{AC}$*-simulation up to* $\mathcal{R}'$ if and only if the following implications are satisfied:

$$\textsc{Subac-Top}\quad \frac{\tau_1 \ \mathcal{R} \ \tau_2}{\tau_1(\epsilon) \leq_0 \tau_2(\epsilon)} \qquad \textsc{Subac-Arrow}\quad \frac{\tau_1 \to \tau_2 \ \mathcal{R} \ \tau_1' \to \tau_2'}{\tau_1' \ (\mathcal{R} \cup \mathcal{R}') \ \tau_1 \qquad \tau_2 \ (\mathcal{R} \cup \mathcal{R}') \ \tau_2'}$$

$$\textsc{Subac-Pi}\quad \frac{\Pi_{i=1}^{n} \tau_i \ \mathcal{R} \ \Pi_{i=1}^{m} \tau_i'}{\exists \sigma \in \Sigma_n^m \quad (\tau_{\sigma(i)} \ (\mathcal{R} \cup \mathcal{R}') \ \tau_i')^{i \in \{1,\ldots,m\}}}$$

$\mathcal{R}$ is a $\leq_{AC}$*-simulation* if and only if it is a $\leq_{AC}$-simulation up to the empty relation. *AC-Subtyping* $\leq_{AC}$ is the largest $\leq_{AC}$-simulation. ◇

It is known that $=_{AC}$ is a congruence and $\leq$ is an ordering. We show that $\leq_{AC}$ is a preorder, that is, it is reflexive and transitive.

Proposition 7. $\leq_{AC}$ *is a preorder.* ◇

We argue that our definition of subtyping modulo associativity and commutativity of products is natural by establishing that it is a composition of the pre-existing relations $=_{AC}$ and $\leq$. One may hope to prove that $\leq_{AC}$ coincides with $=_{AC} \circ \leq$. However, this does not hold, because the contravariance of the arrow symbol forces $=_{AC}$ to be used on both sides of $\leq$. This is illustrated by the pair $(\Pi^1(\top) \to \Pi^2(\bot, \top),\ \Pi^2(\bot, \top) \to \Pi^1(\top))$, which is a member of $\leq_{AC}$, but not a member of $=_{AC} \circ \leq$ or of $\leq \circ =_{AC}$. As a result, $=_{AC}$ must in fact be used on both sides of $\leq$, as stated below.

Theorem 8. *The relations* $\leq_{AC}$ *and* $(=_{AC}) \circ (\leq) \circ (=_{AC})$ *coincide.* ◇

5 Deciding AC-Subtyping

Let us say that a pair of types $p = (\tau, \tau')$ is *valid* if $\tau \leq_{AC} \tau'$ holds and *invalid* otherwise. We now define an algorithm that, given a pair of types $p_0 = (\tau_0, \tau_0')$, determines whether p_0 is valid.

The algorithm's complexity is assessed as a function of the following parameters. Let T and T' be the sets of all subtrees of τ_0 and τ_0', respectively. Let n and n' be the cardinalities of these sets; they are finite. Let us view T and T' as directed graphs, where every tree is a node and there is an edge from τ to τ' labeled i if and only if τ/i is τ'. In other words, there is an edge from every tree to each of its immediate subtrees. Please note that there may be multiple edges, with distinct labels, between τ and τ'. If τ is a node in T or T', let $d(\tau)$ denote its outgoing degree, that is, the arity of the symbol $\tau(\epsilon)$. Let $u(\tau)$ denote its incoming degree, that is, the number of its predecessors in the graph T or T'. We write d for the maximum of $d(\tau)$ when τ ranges over all nodes in T and T'. Last, let N (resp. N') be the *size* of the graph T (resp. T'), where every node and every edge contributes one unit. Please note that we have: $N = \Sigma_{\tau \in T}(1 + u(\tau))$ as well as a similar identity concerning T'.

The algorithm maintains sets of pairs of nodes. We assume that elementary set operations can be performed in constant time. This is indeed possible by using an array of size $O(nn')$, or, more realistically, a hash table.

5.1 First Phase: Exploration

Specification The first phase of the algorithm consists in constructing a (finite) set U of pairs of types whose validity one must determine in order to be able to tell whether p_0 is valid. The universe U may be defined as the smallest set that contains p_0 and is closed under the following two rules:

$$\text{EXPLORE-ARROW}\quad \frac{(\tau_1 \to \tau_2, \tau_1' \to \tau_2') \in U}{(\tau_1', \tau_1) \in U \qquad (\tau_2, \tau_2') \in U} \qquad\qquad \text{EXPLORE-PI}\quad \frac{(\Pi_{i=1}^n \tau_i, \Pi_{j=1}^m \tau_j') \in U}{((\tau_i, \tau_j') \in U)^{i \in \{1,\ldots,n\},\, j \in \{1,\ldots,m\}}}$$

The set $(T \times T') \cup (T' \times T)$ contains p_0 and is closed under these rules. This ensures that U exists and has cardinality $O(nn')$.

We have explained above how to view T and T' as graphs. It is useful to view $(T \times T') \cup (T' \times T)$ as a graph as well. Let there be an (unlabeled) edge from a pair of types p to a pair of types p' if p matches the premise of EXPLORE-ARROW or EXPLORE-PI while p' matches one of its conclusions. In that case, we also say that p is a *parent* of p'. Then, the exploration phase can be viewed simply as an explicit traversal (and construction) of *part of* the graph $(T \times T') \cup (T' \times T)$, starting from the node p_0. In other words, U is the connected component of p_0 in the directed graph $(T \times T') \cup (T' \times T)$.

The number of nodes in the graph U is clearly bounded by $O(nn')$. Because U is an unlabeled graph, the number of its edges must be bounded by $O(n^2 n'^2)$. This yields $\mathit{size}(U) \leq O(n^2 n'^2)$. Furthermore, because the predecessors of a pair (τ, τ') are pairs of a predecessor of τ and a predecessor of τ', we have $u(\tau, \tau') \leq u(\tau)u(\tau')$. This yields another bound on the size of the graph U:

$$\begin{aligned}\mathit{size}(U) &= \Sigma_{(\tau,\tau') \in U}(1 + u(\tau, \tau')) \leq \Sigma_{\tau \in T,\, \tau' \in T'}(1 + u(\tau)u(\tau')) \\ &\leq (\Sigma_{\tau \in T}(1 + u(\tau)))(\Sigma_{\tau' \in T'}(1 + u(\tau'))) = NN'\end{aligned}$$

In practice, we expect both of these bounds to be pessimistic. In the particular case where the types at hand are not recursive (that is, do not involve cycles) and do not involve any products, the size of U may be bounded by $\min(N, N')$. There is a lot of slack between this optimistic bound and the worst-case bounds given above. It should be interesting to measure the size of U in real-world situations.

Implementation The graph U can be computed using a simple iterative procedure, as follows.

1. Let $U = \emptyset$ and $W = \{p_0\}$.
2. While W is nonempty, do:
   (a) Take a pair p out of W;
   (b) If $p \in U$, continue at 2;
   (c) Insert p into U;
   (d) If p is of the form $(\tau_1 \to \tau_2, \tau_1' \to \tau_2')$, then insert (τ_1', τ_1) and (τ_2, τ_2') into W;
   (e) If p is of the form $(\Pi_{i=1}^n \tau_i, \Pi_{j=1}^m \tau_j')$,
   then insert every (τ_i, τ_j'), for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$, into W.

It is clear that this procedure implements the construction of U as specified above. In step 2e, one should remove any duplicate elements from the families $(\tau_i)_{i=1}^n$ and $(\tau'_j)_{j=1}^m$ prior to iterating over them. Then, this procedure runs in time $O(size(U)d)$. It is dominated by the running time of the second phase.

5.2 Second Phase: Fixpoint Computation

The idea behind the second phase of the algorithm is to determine the greatest subset of U that is a $\leq_{AC}$-simulation, then to check whether p_0 is a member of it. In order to build this subset, we start from the full relation U, and successively remove pairs that violate SUBAC-TOP, SUBAC-ARROW or SUBAC-PI, until we reach a fixpoint. Whether a pair violates SUBAC-TOP or SUBAC-ARROW may be determined in constant time. However, in the case of SUBAC-PI, the check requires solving a matching problem in a bipartite graph, whose time complexity may be bounded by $O(d^{5/2})$, as we shall see.

A naive procedure begins by iterating once over all pairs, removing those that violate one of the rules; this takes time $O(nn'd^{5/2})$. But one such iteration may not be enough to reach the fixpoint, so the naive procedure repeats this step as many times as required. In the worst case, each step invalidates only one pair, in which case up to $O(nn')$ successive steps are required. Thus, the overall time complexity is $O(n^2n'^2d^{5/2})$. Below, we propose an enhanced approach, whose convergence is faster. Instead of blindly checking every pair at each iteration, we check only the *parents* of pairs that have just been invalidated. Downey, Sethi, and Tarjan exploit the same idea to accelerate the convergence of their congruence closure algorithm [15].

Description The universe U is now fixed. We maintain three sets W, S, and F, which form a partition of U. The set W is a *worklist* and consists of pairs whose validity remains to be determined. The set S consists of *suspended* pairs, which are conditionally valid: the algorithm maintains the invariant that S is a $\leq_{AC}$-simulation up to W. In other words, a pair S is known to be valid *provided* its (indirect) descendants in W are found to be valid as well. The set F consists of known invalid (*false*) pairs.

When a pair p is found to be invalid, it is moved to the set F and all (if any) of its parents within S are transferred to W for examination. We refer to this auxiliary procedure as *invalidating* p. The time complexity of this procedure is $O(1 + u(p))$, where $u(p)$ is the incoming degree of the pair p in the graph U (see §5.1).

The second phase of the algorithm is as follows.

1. Let $W = U$ and $S = F = \emptyset$.
2. While W is nonempty, do:
   (a) Take a pair p out of W;
   (b) If p is of the form $(\bot, \tau')$ or $(\tau, \top)$, then insert p into S;
   (c) If p is of the form $(\tau_1 \to \tau_2, \tau'_1 \to \tau'_2)$, then
   if $(\tau'_1, \tau_1) \notin F$ and $(\tau_2, \tau'_2) \notin F$ then insert p into S else invalidate p;
   (d) If p is of the form $(\Pi_{i=1}^n \tau_i, \Pi_{j=1}^m \tau'_j)$, then
   if there exists $\sigma \in \Sigma_n^m$ such that, for all $j \in \{1, \ldots, m\}$, $(\tau_{\sigma(j)}, \tau'_j) \notin F$ holds, then insert p into S else invalidate p;
   (e) If p satisfied none of the three previous tests, then invalidate p.
3. If $p_0 \notin F$, return *true*, otherwise return *false*.

Correctness Each iteration of the main loop (step 2) takes a pair p out of W and either inserts it into S or invalidates it. In either case, it is clear that (W, S, F) remains a partition of U.

Let us now check that S remains a $\leq_{AC}$-simulation up to W. If the pair p is inserted into S, then p satisfies SUBAC-TOP, and there exist pairs in $W \cup S$ (that is, outside F) whose validity is sufficient for p to satisfy SUBAC-ARROW or SUBAC-PI. So, the invariant is preserved. If, on the other hand, the pair p is invalidated, then all of its parents within S are transferred back to W, which clearly preserves the invariant as well.

Last, let us check that F remains a set of invalid pairs only. If the pair p is invalidated at step 2c, then p is invalid, for otherwise, by SUBAC-ARROW, the pairs (τ_1', τ_1) and (τ_2, τ_2') would be valid—but these pairs are members of F, a contradiction. Because p is invalid, inserting it into F preserves the invariant. If the pair p is invalidated at steps 2d or 2e, then p may be shown invalid analogously, using SUBAC-PI or SUBAC-TOP.

When the algorithm terminates, W is empty, so S is a $\leq_{AC}$-simulation, which implies that every member of S is valid. On the other hand, every member of F is invalid. We have established that the result returned in step 3 is correct, as stated below:

Theorem 9. *If the algorithm returns true, then $\tau_0 \leq_{AC} \tau_0'$ holds. If the algorithm returns false, then $\tau_0 \leq_{AC} \tau_0'$ does not hold.* $\diamond$

Termination and Complexity Invalidating a pair transfers it from W to F. Because pairs are never taken out of F, and because W and F remain disjoint, no pair is ever invalidated twice.

The initial size of W is the number of nodes in U. Furthermore, when a pair p is invalidated, the size of W increases by $u(p)$. Thus, considering that every pair is invalidated at most once, the total number of pairs that are ever taken out of W—that is, the total number of iterations of step 2—is at most

$$(\Sigma_{p \in U}\, 1) + (\Sigma_{p \in U}\, u(p)) = \Sigma_{p \in U}(1 + u(p)) \;=\; size(U)$$

Let us now estimate the cost of a single iteration of step 2. In step 2d, determining whether an appropriate σ exists is a matching problem in a bipartite graph with at most $2d$ nodes and d^2 edges. Such a problem can be solved in time $O(d^{5/2})$ using Hopcroft and Karp's algorithm [17]. The cost of invalidating a pair may be viewed as $O(1)$ if we consider that the price for transferring a parent from S to W is paid when that parent is later examined. Thus, the (amortized) cost of a single iteration of step 2 is $O(d^{5/2})$.

Combining these results, we find that the second phase of the algorithm runs in time $O(size(U)d^{5/2})$. This is more expensive that the first phase, so we may state

Theorem 10. *The algorithm runs in time $O(size(U)d^{5/2})$, which is bounded both by $O(NN'd^{5/2})$ and $O(n^2n'^2d^{5/2})$.* $\diamond$

As explained in §5.1, the size of the graph U might be significantly smaller, in practice, than either of NN' and $O(n^2n'^2)$, which is why we give the first complexity bound. The second bound shows that, in the worst case, the algorithm remains linear in each of the sizes of the input types, namely N and N', with additional overhead $O(d^{5/2})$, where d is a bound on the arity of the products involved. The third bound

shows that our improved algorithm performs no worse than the naive procedure outlined in §1 and §5.2.

For comparison, Downey *et al.*'s symmetric congruence closure algorithm [15], as well as Jha *et al.*'s decision procedure for AC-equality [18], run in time $O((N + N')\log(N + N'))$. These algorithms compute an *equivalence* relation. This opens the way to a more efficient data representation, where a relation is not stored as a set of pairs but as a partition, and simplifies the matching problem.

5.3 Further Refinements

A cheap refinement consists in modifying the first phase so that it fails as soon as it reaches a pair p that does not satisfy SUBAC-TOP, *provided* the path from p_0 to p never leaves a pair of products—that is, provided the validity of p_0 implies that of p. This helps immediately detect some failures. For this refinement to be most effective, the paths in U where immediate failure may occur should be explored first. One way of achieving this effect is simply to give higher priority to edges that leave a pair of arrows than to edges that leave a pair of products.

A more interesting refinement consists in specifying in what order pairs should be taken out of the worklist W during the second phase. It is more efficient to deal with descendants first and with ancestors last, because dealing with an ancestor too early might be wasted work—we might decide to suspend it and later be forced to transfer it back to the worklist because new information about its descendants has been made available. Of course, because types are recursive, the relation "to be a parent of" is in general only a preorder, not an ordering—that is, the graph U may exhibit cycles.

Let us remark, though, that when U *is* acyclic, it is indeed possible to process pairs in order. This ensures that, when a pair is processed, none of its parents have been processed yet, so all of them must still be in the worklist. Thus, when invalidating a pair, it is no longer necessary to iterate over its parents. In that case, the algorithm's time complexity becomes $O(nodes(U)d^{5/2})$, where $nodes(U)$ counts the nodes of the graph U, but *not* its edges, and is bounded by $O(nn')$.

It is possible to take advantage of this remark even in the presence of cycles. The first phase, upon completion, can be made to produce an explicit representation of the graph U. Determine its strongly connected components and topologically sort them. Then, *remove all edges whose endpoints do not belong to the same component*. The cost of this additional preprocessing is linear in the size of U. Now, run the second phase, one component at a time, in topological order, that is, descendants first and ancestors last. Because of the removed edges, when invalidating a pair p, only the parents of p that belong to the *same* strongly connected component are checked. This is correct because components are being processed in topological order, which ensures that the parents of p that belong to a *distinct* component must still be in the worklist.

The modified algorithm runs in time $O(size(U')d^{5/2})$, where U' is the result of pruning the graph U, that is, of keeping only the edges that participate in a cycle. Thus, its complexity may still be bounded by $O(NN'd^{5/2})$ in the worst case, but this bound gradually decreases down to $O(nn'd^{5/2})$ in the case of nonrecursive types. We conjecture that, in practice, cycles often involve only a fraction of the type structure, so this improvement may be significant.

Searching a Whole Library For our purposes, a software library is a collection of possibly mutually recursive types, which we may view as a single recursive type τ_L, some distinguished subterms of which form a set T_L. The programmer's query is a possibly recursive type τ_Q. The problem is to find all components in the library that provide (at least) the requested functionality, that is, to find every $\tau \in T_L$ such that $\tau \leq_{AC} \tau_Q$ holds.

One possibility is to run the algorithm with $p_0 = (\tau, \tau_Q)$ successively for every $\tau \in T_L$. However, this is inefficient. Let U_τ denote the universe explored by the algorithm when run with initial pair (τ, τ_Q). Then, the universes $(U_\tau)_{\tau \in T_L}$ might overlap, causing repeated work. It is more efficient to run the algorithm once with *multiple* initial pairs, that is, with the family of initial pairs $(\tau, \tau_Q)_{\tau \in T_L}$. Extending the algorithm to deal with a set of initial pairs $\{p_0, \ldots, p_{k-1}\}$ is immediate; it suffices to define the universe U as the smallest superset of $\{p_0, \ldots, p_{k-1}\}$ that is closed under EXPLORE-ARROW and EXPLORE-PI. By running the algorithm only once, we ensure that the worst-case time complexity is bounded by $O(NN'd^{5/2})$, where N is the size of the library τ_L and N' is the size of the query τ_Q.

In fact, running the algorithm once with a set of initial pairs $\{p_0, \ldots, p_{k-1}\}$ is equivalent to running it k times in succession, supplying the single initial pair p_i to the ith run, *provided* each run starts where the previous left off, that is, re-uses the sets U, S, F computed by the previous run. With this proviso, one may, without loss of efficiency, provide initial pairs to the algorithm one after the other.

This remark leads to an optimization. Imagine that T_L is organized as a graph, with an edge from τ to τ' if and only if $\tau \leq_{AC} \tau'$ holds. (This graph might be built during a preprocessing phase. We may assume that it is acyclic: if it isn't, cycles may be collapsed.) Then, pick a maximal node τ, that is, a node with no successors in the graph. Run the algorithm with initial pair (τ, τ_Q). If τ is found to be comparable with τ_Q, then, by transitivity of $\leq_{AC}$, so is *every predecessor* of τ in the graph. In that case, remove τ *and all of its predecessors* from the graph; otherwise, remove τ alone. Then, pick a maximal node in what remains of the graph, and proceed in the same manner. This approach offers the double advantage of being potentially more efficient and of providing successful answers in groups, where each group contains a distinguished maximal (w.r.t. $\leq_{AC}$) answer to the query and distinct groups contain incomparable answers. We believe that the user should find this behavior natural. The actual efficiency gain remains to be assessed.

One should point out that this optimization is but a simple way of exploiting the fact that $\leq_{AC}$ is transitive. One might wonder whether it is possible to exploit transitivity at the core of the algorithm: for instance, by directly inserting a pair into S, without examining its descendants, if it is a transitive consequence of the pairs that are members of S already. This issue is left for future research.

6 Building Coercions

We now discuss the coercions that witness the relation $\leq_{AC}$, and how to compute them from the simulation discovered by the algorithm, when it succeeds. We follow Brandt

and Henglein's presentation [9], but work directly with regular trees, instead of using the μ notation, which allows us to make "fold" and "unfold" coercions implicit.

Definition 11 (Coercions for $\leq_{AC}$). Coercions are defined by the grammar

$$c ::= \iota_\tau \mid f \mid \mathit{fix}\ f.c \mid c \to c' \mid \Pi_i^\sigma c_i \mid \mathit{abort}_\tau \mid \mathit{discard}_\tau \qquad \diamond$$

Most coercion forms are taken from Brandt and Henglein's paper, with the same typing rules [9, figure 6]. Let us recall that a typing judgment is of the form $E \vdash c : \tau \to \tau'$, where the environment E maps coercion variables f to coercion types of the form $\tau \to \tau'$. The one new coercion form is $\Pi_i^\sigma c_i$, whose typing rule is

$$\frac{\sigma \in \Sigma_n^m \qquad (E \vdash c_i : \tau_{\sigma(i)} \to \tau_i')^{i \in \{1,\ldots,m\}}}{E \vdash \Pi_i^\sigma c_i : \Pi_{i=1}^n \tau_i \to \Pi_{i=1}^m \tau_i'}$$

and whose operational meaning is $\lambda p.\Pi_{i=1}^m c_i(\pi_{\sigma(i)}(p))$. If $\tau \leq_{AC} \tau'$ holds, then the algorithm, applied to the pair (τ, τ'), produces a finite $\leq_{AC}$-simulation S that contains (τ, τ'). It is straightforward to turn S into a system of recursive equations that defines one coercion for each pair within S, including, in particular, a coercion of type $\tau \to \tau'$.

Theorem 12. *If $\tau \leq_{AC} \tau'$ holds, there exists a (closed) coercion c s. t. $\vdash c : \tau \to \tau'$.* $\diamond$

The size of the equation associated with (τ, τ') is $O(1 + d(\tau'))$, where $d(\tau')$ is the outgoing degree of the node τ' in the graph T or T'. As a result, the total size of the system of equations is bounded by

$$\begin{aligned} & O(\Sigma_{\tau \in T,\, \tau' \in T'}(1 + d(\tau')) + \Sigma_{\tau' \in T',\, \tau \in T}(1 + d(\tau))) \\ = {} & O(n(\Sigma_{\tau' \in T'}(1 + d(\tau'))) + n'(\Sigma_{\tau \in T}(1 + d(\tau)))) \\ = {} & O(nN' + n'N) \end{aligned}$$

The system can be produced in linear time with respect to its size, so the time complexity of producing code for the coercions is $O(nN' + n'N)$. (If one applies Bekič's theorem, as suggested above, then the time and space complexity increases quadratically, but there is no reason to do so in practice.)

It is worth pointing out that not all well-typed coercions have the same operational meaning, and some user interaction is, in practice, necessary to ensure that the coercion code suits the user's needs.

7 Practical Considerations

In practical applications, the language of types is usually much richer than the one considered in this paper. The grammar of types may include a set of atoms (such as *int*, *float*, etc.), equipped with a subtyping relation, and a set of parameterized type constructors. Each of these type constructors may have some contravariant and some covariant parameters, may support or forbid permutations of its parameters, and may support or forbid width subtyping.

Fortunately, it is straightforward to adapt the results of this paper to such an extended language of types. As far as atoms and atomic subtyping are concerned, it

suffices to add appropriate clauses to the definition of a $\leq_{AC}$-simulation and to the algorithms for deciding AC-subtyping and building coercions; these new clauses are variations of the existing clauses for $\bot$ and $\top$. As far as parameterized type constructors are concerned, it is enough to extend our definitions by distinguishing four kinds of products that respectively support or forbid parameter permutations and width subtyping. The rules that describe the three new (restricted) kinds of products are special cases of our current rules, since our current product constructor allows both parameter permutations and width subtyping. Then, every parameterized type constructor may be desugared into a combination of atoms, the arrow constructor (which allows encoding contravariance) and the four product constructors.

Our core language is purely functional. However, real-world languages, and object-oriented languages in particular, often have mutable data structures and a notion of object identity. Then, it is important that coercions preserve object identity. One might wish the following property to hold: the program that is linked, using adapters, to a certain library, should have the same semantics as that obtained by linking, without adapters, to a library whose method and class names have been suitably renamed. We believe that, combining our algorithms with the adapter model sketched by Thatté [24], it is possible to achieve such a property. We leave this as future work.

8 Conclusion

We have introduced a notion of subtyping of recursive types up to associativity and commutativity of products. We have justified our definition by showing that this relation is a composition of the usual subtyping relation with Palsberg and Zhao's notion of equality up to associativity and commutativity of products. We have provided an algorithm for deciding whether two types are in the relation. The algorithm's worst-case time complexity may be bounded by $O(NN'd^{5/2})$ and $O(n^2n'^2d^{5/2})$; we believe it will prove fairly efficient in practice. It is straightforward and cheap to produce coercion code when the algorithm succeeds.

We believe this paper may constitute the groundwork for *practical* search tools within libraries of object-oriented code. Indeed, as argued in §1, AC-equality alone is not flexible enough, since it does not allow looking for only a *subset* of the features provided by a library.

References

[1] Martín Abadi and Marcelo P. Fiore. Syntactic considerations on recursive types. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 242–252, July 1996.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[3] Maria-Virginia Aponte and Roberto Di Cosmo. Type isomorphisms for module signatures. In *Symposium on Programming Language Implementation and Logic Programming (PLILP)*, volume 1140 of *Lecture Notes in Computer Science*, pages 334–346. Springer Verlag, 1996.

[4] Joshua Auerbach, Charles Barton, and Mukund Raghavachari. Type isomorphisms with recursive types. Technical Report RC 21247, IBM Yorktown Heights, 1998.

[5] Joshua Auerbach and Mark C. Chu-Carrol. The Mockingbird system: a compiler-based approach to maximally interoperable distributed systems. Technical Report RC 20718, IBM Yorktown Heights, 1997.

[6] Joshua Auerbach, Mark C. Chu-Carrol, Charles Barton, and Mukund Raghavachari. Mockingbird: Flexible stub generation from pairs of declarations. Technical Report RC 21309, IBM Yorktown Heights, 1998.

[7] Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Remarks on isomorphisms in typed lambda calculi with empty and sum type. In *IEEE Symposium on Logic in Computer Science (LICS)*, July 2002.

[8] Charles M. Barton. M-types and their coercions. Technical Report RC-21615, IBM Yorktown Heights, December 1999.

[9] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33:309–338, 1998.

[10] Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

[11] Felice Cardone. A coinductive completeness proof for the equivalence of recursive types. *Theoretical Computer Science*, 275(1–2):575–587, 2002.

[12] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, March 1983.

[13] Roberto Di Cosmo. Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming*, 3(3):485–525, 1993.

[14] Roberto Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995.

[15] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[16] Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2003.

[17] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, December 1973.

[18] Somesh Jha, Jens Palsberg, and Tian Zhao. Efficient type matching. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 2303 of *Lecture Notes in Computer Science*, pages 187–204. Springer Verlag, April 2002.

[19] Jens Palsberg and Tian Zhao. Efficient and flexible matching of recursive types. *Information and Computation*, 171:364–387, 2001.

[20] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

[21] Mikael Rittri. Retrieving library functions by unifying types modulo linear isomorphism. *RAIRO Theoretical Informatics and Applications*, 27(6):523–540, 1993.

[22] Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, 1991.

[23] Sergei V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983.

[24] Satish R. Thatté. Automated synthesis of interface adapters for reusable classes. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–187, January 1994.

[25] Jeannette M. Wing, Eugene Rollins, and Amy Moormann Zaremski. Thoughts on a Larch/ML and a new application for LP. In *First International Workshop on Larch*, pages 297–312, July 1992.