Parametric Subtyping for Structural Parametric Polymorphism

HENRY DEYOUNG, Carnegie Mellon University, USA
ANDREIA MORDIDO, Universidade de Lisboa, Portugal
FRANK PFENNING, Carnegie Mellon University, USA
ANKUSH DAS, Amazon Web Services, USA

We study the interaction of structural subtyping with parametric polymorphism and recursively defined type constructors. Although structural subtyping is undecidable in this setting, we describe a notion of parametricity for type constructors and then exploit it to define *parametric subtyping*, a conceptually simple, decidable, and expressive fragment of structural subtyping that strictly generalizes nominal subtyping. We present and prove correct an effective saturation-based decision procedure for parametric subtyping, demonstrating its applicability using a variety of examples. An implementation of this decision procedure is available in the supplementary materials.

CCS Concepts: • **Theory of computation → Type theory**; **Type structures**; **Functional constructs**; • **Software and its engineering → Polymorphism**; **Recursion**.

Additional Key Words and Phrases: structural subtyping, parametric polymorphism, type constructors, saturation-based algorithms

1 INTRODUCTION

Recursive types, parametric polymorphism (also called generics), and subtyping are all essential features for modern programming languages across numerous paradigms. Recursive types describe unbounded data structures; parametric polymorphism provides type-level modularity by allowing programmers to use instantiations of $\mathsf{list}[\alpha]$ rather than separate monomorphic types for integer and boolean lists, for example; and subtyping provides flexibility in the ways that objects and terms can be used, enabling code reuse. Structural subtyping, in particular, is especially flexible and expressive and, in principle, relatively lightweight for programmers to incorporate.

This combination of features is present to varying degrees in many of today's widely used languages, such as Go, Rust, TypeScript and Java, but the combination is difficult to manage. For example, subtyping for generics in Java is known to be undecidable [Grigore 2017], so various restrictions on types' structure have been proposed, such as material-shape separation [Greenman et al. 2014; Mackay et al. 2020], and the prohibition of contravariance, unbounded expansion of types, or multiple instantiation inheritance [Kennedy and Pierce 2006], to name a few.

We contend that these restrictions are often either too limiting or too unintuitive for programmers to readily reason about. A reconstruction of the interaction between recursive types, parametric polymorphism, and structural subtyping from first principles is needed, accompanied by a clear,

Authors' addresses: Henry DeYoung, Carnegie Mellon University, Pittsburgh, USA, hdeyoung@cs.cmu.edu; Andreia Mordido, Universidade de Lisboa, Lisboa, Portugal, afmordido@ciencias.ulisboa.pt; Frank Pfenning, Carnegie Mellon University, Pittsburgh, USA, fp@cs.cmu.edu; Ankush Das, Amazon Web Services, 5000 Forbes Avenue, Cupertino, USA, daankus@amazon.com.

XXXX-XXXX/2023/10-ART
https://doi.org/XXXXXXX.XXXXXXX

relatively simple declarative characterization of subtyping. However, to the best of our knowledge, no such work has been undertaken thus far. This paper fills that gap.

As a first step, we prove that structural subtyping is undecidable in the presence of recursive types and parametric polymorphism.[1] Given this undecidability, our goal is to identify an expressive, practical fragment of structural subtyping that has three key properties:

(1) The fragment should have a relatively simple *declarative characterization*, so that the valid subtypings are readily predictable by the programmer.
(2) The fragment should be *decidable*, with an effective algorithm that performs well on the kinds of subtyping problems that arise in practice.
(3) The fragment should *generalize nominal subtyping*, preserving all subtypings that already hold under a nominal treatment of type constructors.

It is not immediately clear that such a fragment of structural subtyping should even exist, as seemingly innocent variations of the problem are already undecidable or impractical. Solomon [1978] showed that structural equality for parametric polymorphism can be reduced to equivalence of deterministic pushdown automata, but it took more than 20 additional years to establish decidability [Sénizergues 2001; Stirling 2001a,b], albeit by an intractable algorithm. As another example, even *without* recursive types, bounded quantification is undecidable [Pierce 1994].

Nevertheless, in this paper, we are able to achieve our goal: we propose a notion of parametricity for pairs of type constructors that forms the basis of a suitable fragment of structural subtyping, a fragment that we call *parametric subtyping*. Parametric pairs of constructors will map subtyping-related arguments to subtyping-related results, echoing Reynolds's characterization of parametric functions [1983]. Moreover, by exploiting parametricity, we avoid unintuitive restrictions on types' structure and can support even non-regular types [Bird and Meertens 1998; Mycroft 1984].

Because of its fundamental nature, our notion of parametric subtyping and associated decision procedure could be applied to a wide variety of languages: object-oriented systems; lazy and eager functional languages; imperative languages; mixed inductive/coinductive languages, such as call-by-push-value [Levy 2001]; session-typed languages [Caires and Pfenning 2010; Gay and Hole 2005; Honda et al. 1998; Silva et al. 2023]; and so on.[2]

In this paper, we do not address subtypings such as $\mathbf{0} \leqslant \mathbf{1}$ and $\mathbf{1} \leqslant \mathbf{0} \to \mathbf{1}$ that arise when some types are uninhabited [Ligatti et al. 2017], instead interpreting all types coinductively, which makes them all (including $\mathbf{0}$) inhabited. However, the parametric subtyping rules in this paper are sound even in languages where some types are interpreted inductively. Moreover, we want to examine the fundamental core of parametric subtyping: had we insisted on an inductive treatment of some types, parametric subtyping would be unsound for lazy functional and session-typed languages.

1.1 Overview of parametric subtyping

To provide the reader with some intuition for our notion of parametric subtyping, we will now sketch, at a high level, how parametric subtyping satisfies the three desired properties.

(1) *Relatively simple declarative characterization.* A pair of type constructors, $t[\vec{\alpha}]$ and $u[\vec{\beta}]$, will be considered parametric if the subtyping problem $t[\vec{\alpha}] \leqslant u[\vec{\beta}]$ can be reduced to finitely many subtyping problems among the arguments $\vec{\alpha}$ and $\vec{\beta}$ alone.

[1] A related result has been proven by Padovani [2019] for subtyping of context-free session types [Thiemann and Vasconcelos 2016]. Our proof adapts this result to the setting of recursively defined type constructors (which has been shown to be more general than context-free session types [Gay et al. 2022]) and identifies two other minimal undecidable fragments.
[2] Of course, when applied to a given language, there will be additional language-specific considerations, e.g., interaction with intersection types in TypeScript or type classes in Haskell. We do not claim to address such considerations here.

As an example, consider an interface for stack objects (or, from a functional perspective, a record type for stacks), parameterized by a type α of stack elements:

$$\mathsf{stack}[\alpha] \triangleq \&\{\mathsf{push}\colon \alpha \to \mathsf{stack}[\alpha],\ \mathsf{pop}\colon \mathsf{option}[\alpha \times \mathsf{stack}[\alpha]]\}$$

where $\mathsf{option}[\beta] \triangleq +\{\mathsf{none}\colon \mathbf{1},\ \mathsf{some}\colon \beta\}$. Programmers sometimes want to ensure that a stack be used according to a particular protocol. For example, when implementing a queue using a pair of stacks (as sometimes done in functional languages), the protocol in which all pushes must occur before any pops can be expressed by the types

$$\begin{aligned} \mathsf{qstack}_1[\beta] &\triangleq \&\{\mathsf{push}\colon \beta \to \mathsf{qstack}_1[\beta], \mathsf{pop}\colon \mathsf{option}[\beta \times \mathsf{pops}[\beta]]\} \\ \mathsf{pops}[\beta] &\triangleq \&\{\qquad\qquad\qquad\qquad\qquad\quad \mathsf{pop}\colon \mathsf{option}[\beta \times \mathsf{pops}[\beta]]\}\,. \end{aligned}$$

By virtue of having definitions with compatible structures, $\mathsf{stack}[-]$ is a subtype of itself, $\mathsf{qstack}_1[-]$, and $\mathsf{pops}[-]$ according to the admissible subtyping rules

$$\frac{\alpha \leqslant \beta \quad \beta \leqslant \alpha}{\mathsf{stack}[\alpha] \leqslant \mathsf{stack}[\beta]}\,, \quad \frac{\alpha \leqslant \beta \quad \beta \leqslant \alpha}{\mathsf{stack}[\alpha] \leqslant \mathsf{qstack}_1[\beta]}\,, \quad \textit{and} \quad \frac{\alpha \leqslant \beta}{\mathsf{stack}[\alpha] \leqslant \mathsf{pops}[\beta]}$$

Specifically, these are valid *parametric* subtyping rules because the premises involve only arguments, α and β. Rules such as "$t[\alpha] \leqslant u[\beta]$ if $\alpha \leqslant \mathbf{1}$" or "$t[\alpha] \leqslant u[\beta]$ if $\beta \leqslant t'[\alpha]$" would not be parametric because their premises are not subtypings among arguments alone.

Given such rules, parametric subtyping is conceptually rather straightforward: A subtyping between types, such as $\mathsf{stack}[\mathsf{stack}[\tau]] \leqslant \mathsf{pops}[\mathsf{qstack}_1[\sigma]]$, holds if it has a finite derivation from the admissible parametric subtyping rules.

In Section 3, we present an equivalent characterization of parametric subtyping that is more amenable to metatheoretic proofs. A series of examples in Section 7 demonstrates that the valid parametric subtypings are readily predictable by the programmer and expressive enough for many subtypings desired in practice.

(2) *Decidable.* In Section 5, we prove that parametric subtyping is decidable by giving a saturation algorithm that is sound and complete with respect to the declarative characterization of parametric subtyping (Theorems 5.2 and 5.4). The algorithm infers, for each pair of type constructors, the most general parametric subtyping rule, if one exists. Moreover, when no such parametric rule exists, the algorithm determines whether the cause is a fundamental violation of structural subtyping or merely a violation of parametricity. After inferring all such admissible rules, a given subtyping problem can be decided by backward proof construction of a finite derivation using the inferred rules.

We have implemented this decision procedure, and it is available in the supplementary materials (and intended for later submission as an artifact).

(3) *Generalizes nominal subtyping.* Nominal subtyping is characterized by those parametric rules that relate identical type constructors, such as the above rule for $\mathsf{stack}[\alpha] \leqslant \mathsf{stack}[\beta]$. Our notion of parametric subtyping is indeed strictly more general than nominal subtyping, in that it also admits those parametric rules that relate distinct type constructors.

The most closely related work is that on refinement types, specifically datasort refinements [Freeman and Pfenning 1991]; there are significant differences, however. First, whereas the refinement system refines a nominal type into a collection of structural sorts, we use a single-layer, fully structural system. Second, Davies [2005, Section 7.4] defines an algorithm for subsorting parameterized sort constructors that respects parametricity, but requires explicit declarations for the constructors' variance and handles only very limited cases of nested sorts. (On the other hand, he deals with general pattern matching, module boundaries, and intersections that are beyond the scope of the present work.) Third, Skalka [1997] gives an algorithm to decide the emptiness of refinement types, but does not give a subtyping algorithm and handles only regular type constructors. Last, whereas the nominal core of refinement types means that a defined type cannot later be widened into a supertype, our fully structural system has the advantage of naturally permitting widening.

In summary, the contributions of this paper are: to identify several minimal fragments for which structural subtyping is undecidable (Section 2.3); to give a simple, declarative characterization of parametric subtyping, as a fragment of structural subtyping (Section 3); to present a saturation algorithm for deciding parametric subtyping for parametric polymorphism (Section 5), as well as proofs of its soundness and completeness with respect to the declarative characterization (Theorems 5.2 and 5.4); to implement this decision procedure (Section 6); and to give, as a special case of this decision procedure, a saturation-based decision procedure for structural subtyping of monomorphic types that has several advantages over existing algorithms (Section 4).

2 STRUCTURAL SUBTYPING FOR PARAMETRIC POLYMORPHISM

In this section, we describe the syntax of types, present a declarative characterization of structural subtyping, and show that it is undecidable in the presence of recursively defined type constructors.

2.1 Syntax of types

Programmers write types in the form to which they are accustomed, such as in the type definition $\mathsf{list}[\alpha] \triangleq +\{\mathsf{nil}\colon \mathbf{1},\, \mathsf{cons}\colon \alpha \times \mathsf{list}[\alpha]\}$. However, throughout this paper, it will often be convenient to work with types in a normal form that maintains a strict distinction and alternation between *named types*, τ, and *structural types*, A. For this reason, the user-defined types will be normalized in a preliminary elaboration phase that inserts additional type constructors, in a manner reminiscent of the conversion of context-free grammars to Greibach normal form [1965] and Huet's syntax used in deciding extensional equality of total Böhm trees [1998]. Details of this elaboration are postponed to Section 6.

For types in normal form, the syntax is as follows. In addition to these syntactic categories, we use α for type constructor parameters, and x for explicitly quantified type variables.

$$
\begin{array}{rrl}
\textit{Named types} & \tau, \sigma & ::= t[\theta] \mid \alpha \mid x \\
\textit{Type substitutions} & \theta, \phi & ::= (\cdot) \mid \theta, \tau/\alpha \\
\textit{Structural types} & A, B & ::= \tau_1 \times \tau_2 \mid \mathbf{1} \mid +\{\ell\colon \tau_\ell\}_{\ell \in L} \mid \exists x.\tau \\
 & & \quad\mid \tau_1 \rightarrow \tau_2 \mid \&\{\ell\colon \tau_\ell\}_{\ell \in L} \mid \forall x.\tau \\
\textit{Signatures} & \Sigma & ::= (\cdot) \mid \Sigma, t[\vec{\alpha}] \triangleq A
\end{array}
$$

2.1.1 Structural types. The structural types, A, consist of: product types $\tau_1 \times \tau_2$ and the unit type $\mathbf{1}$; variant record types $+\{\ell\colon \tau_\ell\}_{\ell \in L}$, indexed by (possibly empty) finite sets L of alternatives ℓ; existentially quantified types $\exists x.\tau$; function types $\tau_1 \rightarrow \tau_2$; record types $\&\{\ell\colon \tau_\ell\}_{\ell \in L}$, again indexed by (possibly empty) finite sets L; and universally quantified types $\forall x.\tau$. The somewhat nonstandard feature of this syntax is that it enforces the first part of the strict alternation between structural and named types that is prescribed by the normal form: here, structural types A have only named types τ as immediate subformulas.

2.1.2 Named types and signatures. Named types τ are primarily type constructor instantiations of the form $t[\theta]$, where t is a defined type constructor[3] and θ is a type substitution. Such type

[3] *Defined* type constructors t are distinct from structural type constructors like $\rightarrow$. However, in the remainder of this paper, we will frequently drop the 'defined' qualifier for conciseness and simply use 'type constructor' to refer exclusively to defined type constructors.

constructors t are recursively defined[4] in a signature Σ. The signature provides finitely many definitions of the form $t[\vec{\alpha}] \triangleq A$, exactly one for each defined type constructor, where the structural type A may contain free occurrences of the parameters $\vec{\alpha}$ but must be otherwise closed. The substitution θ in $t[\theta]$ then serves to instantiate the type parameters $\vec{\alpha}$ used in t's definition. (In examples, we use an application-like syntax in place of substitutions, such as $t[\tau]$ instead of $t[\tau/\alpha]$.)

Notice that definitions enforce the other part of the strict alternation between structural types and named types that is prescribed by this normal form: here, type constructors $t[\vec{\alpha}]$ are defined only in terms of structural types A, not named types τ. Moreover, this ensures that all definitions are contractive [Gay and Hole 2005].

Given the shallow syntax of structural types, named types τ must also include type parameters α, so that the structural body of a definition $t[\vec{\alpha}] \triangleq A$ may indeed contain occurrences of parameters $\vec{\alpha}$. Similarly, named types τ also include type variables x bound by the $\forall$ and $\exists$ quantifiers.

2.1.3 Type substitutions. In structural subtyping, definitions $t[\vec{\alpha}] \triangleq A$ will be interpreted transparently, with $t[\theta]$ and its unfolding, $\theta(A)$, being treated indistinguishably (aside from belonging to distinct syntactic categories). Because such type definitions are closed apart from their parameters $\vec{\alpha}$, the domains of type substitutions θ consist only of type parameters α. Moreover, substitutions map these parameters to named types τ, not to structural types, so that the instantiation of a structural type, $\theta(A)$, is itself a well-formed structural type.

2.1.4 Examples. Here we present two examples to which we will repeatedly return in this paper.

Even and odd natural numbers. As a simple example of a type, the programmer could write the following type definitions to describe a unary representation of natural numbers, as well as even and odd natural numbers. (We omit [] when a defined type takes no parameters.)

$$\mathsf{nat} \triangleq +\{\mathsf{z}\colon \mathbf{1},\ \mathsf{s}\colon \mathsf{nat}\}\,, \quad \mathsf{even} \triangleq +\{\mathsf{z}\colon \mathbf{1},\ \mathsf{s}\colon \mathsf{odd}\}\,, \quad \textit{and} \quad \mathsf{odd} \triangleq +\{\mathsf{s}\colon \mathsf{even}\}$$

The elaboration phase would normalize these types by introducing an auxiliary type name, one, and revising the definitions of nat and even so that structural and named types alternate:

$$\mathsf{one} \triangleq \mathbf{1}\,, \quad \mathsf{nat} \triangleq +\{\mathsf{z}\colon \mathsf{one},\ \mathsf{s}\colon \mathsf{nat}\}\,, \quad \mathsf{even} \triangleq +\{\mathsf{z}\colon \mathsf{one},\ \mathsf{s}\colon \mathsf{odd}\}\,, \quad \textit{and} \quad \mathsf{odd} \triangleq +\{\mathsf{s}\colon \mathsf{even}\}\,.$$

To avoid the bureaucracy of having to write types in normal form, future examples given in this paper presume that types will be normalized during elaboration.

The even and odd natural numbers are, of course, subsets of the natural numbers. So, taking a sets-of-values interpretation of subtyping, we ought to have even and odd as subtypes of nat, but we ought *not* to have nat as a subtype of even and odd.

Context-free languages. As a more complex example, we consider the type of words belonging to the context-free language $\{\mathsf{L}^n\mathsf{R}^n\$ \mid n \geq 0\}$. (The terminal symbol, \$, is necessary to make the language prefix-free [Korenjak and Hopcroft 1966] and thereby represent the empty word in a typable way.) To aid intuition, we show both the context-free grammar (in Greibach normal form [1965]) for this language and the corresponding, quite parallel, type definitions.

$$\begin{array}{llll} e_0 \to \mathsf{L}\,e\,\mathit{end} \mid \$ \quad \mathit{end} \to \$ & \mathsf{e}_0 \triangleq +\{\mathsf{L}\colon \mathsf{e}[\mathsf{end}],\ \$\colon \mathsf{one}\} & & \mathsf{end} \triangleq +\{\$\colon \mathsf{one}\} \\ e \to \mathsf{L}\,e\,r \mid \mathsf{R} \qquad r \to \mathsf{R} & \mathsf{e}[\kappa] \triangleq +\{\mathsf{L}\colon \mathsf{e}[\mathsf{r}[\kappa]],\ \mathsf{R}\colon \kappa\} & \textit{where} & \mathsf{r}[\kappa] \triangleq +\{\mathsf{R}\colon \kappa\} \end{array}$$

Here, the type e_0 relies on the constructor $\mathsf{e}[\kappa]$, which describes the language $\{\mathsf{L}^n\mathsf{R}^{n+1}\kappa \mid n \geq 0\}$; that is, the parameter κ maintains a continuation to be used when the unmatched R is produced.

[4]We could have chosen to use a recursion operator μ and explicit folds and unfolds, but by using definitions, we avoid the complication of comparing μ-types for equality. We also find definitions easier to read and closer to actual practice.

$$\frac{t[\vec{\alpha}] \triangleq A \quad u[\vec{\beta}] \triangleq B \quad \theta(A) \leqslant \phi(B)}{t[\theta] \leqslant u[\phi]}\ \textsc{unf-s} \qquad \frac{}{x \leqslant x}\ \textsc{var-s}$$

$$\frac{\tau_1 \leqslant \sigma_1 \quad \tau_2 \leqslant \sigma_2}{\tau_1 \times \tau_2 \leqslant \sigma_1 \times \sigma_2} \times\text{s} \qquad \frac{}{\mathbf{1} \leqslant \mathbf{1}}\ \mathbf{1}\text{s} \qquad \frac{(L \subseteq K) \quad \forall \ell \in L\colon \tau_\ell \leqslant \sigma_\ell}{+\{\ell\colon \tau_\ell\}_{\ell \in L} \leqslant +\{k\colon \sigma_k\}_{k \in K}} +\text{s}$$

$$\frac{\sigma_1 \leqslant \tau_1 \quad \tau_2 \leqslant \sigma_2}{\tau_1 \to \tau_2 \leqslant \sigma_1 \to \sigma_2} \to\text{s} \qquad \frac{(K \subseteq L) \quad \forall k \in K\colon \tau_k \leqslant \sigma_k}{\&\{\ell\colon \tau_\ell\}_{\ell \in L} \leqslant \&\{k\colon \sigma_k\}_{k \in K}}\ \&\text{s}$$

$$\frac{(z \text{ fresh}) \quad [z/x]\tau \leqslant [z/y]\sigma}{\forall x.\tau \leqslant \forall y.\sigma}\ \forall\text{s} \qquad \frac{(z \text{ fresh}) \quad [z/x]\tau \leqslant [z/y]\sigma}{\exists x.\tau \leqslant \exists y.\sigma}\ \exists\text{s}$$

Fig. 1. Structural subtyping rules (s for 'structural'). These inference rules are interpreted coinductively.

Because the type e_0 refers to $\mathsf{e}[-]$ only after producing an initial L, the words described by e_0 are indeed a string of Ls followed by the same number of Rs (followed by \$).

In a similar way the context-free grammar (again in Greibach normal form) and the type d_0 given below describe the (\$-terminated) Dyck language of balanced delimiters, here L and R.

$$d_0 \to \mathsf{L}\, d\, d_0 \mid \$ \qquad\qquad \mathsf{d}_0 \triangleq +\{\mathsf{L}\colon \mathsf{d}[\mathsf{d}_0],\ \$\colon \mathsf{one}\}$$

$$d \to \mathsf{L}\, d\, d \mid \mathsf{R} \qquad\qquad \mathsf{d}[\kappa'] \triangleq +\{\mathsf{L}\colon \mathsf{d}[\mathsf{d}[\kappa']],\ \mathsf{R}\colon \kappa'\}$$

The type d_0 relies on the type constructor $\mathsf{d}[\kappa']$, which describes the context-free language of "nearly balanced" delimiters, in which words of balanced delimiters are followed by one additional unmatched R; once again, the type parameter κ maintains a continuation to be used when that unmatched R is produced. The type d_0 refers to $\mathsf{d}[-]$ only after producing an initial L, so the words described by d_0 are indeed balanced.

Because $\{\mathsf{L}^n\mathsf{R}^n\$ \mid n \geq 0\}$ is a subset of the \$-terminated Dyck language, we ought to have e_0 as a subtype of d_0, but *not* d_0 as a subtype of e_0.

2.2 Structural subtyping

Because our normalized types are separated into named types and structural types, structural subtyping will be given a declarative characterization in terms of derivations of two judgments: $\tau \leqslant \sigma$ for named type τ a subtype of named type σ, and $A \leqslant B$ for structural type A a subtype of structural type B. Derivations of the $\tau \leqslant \sigma$ and $A \leqslant B$ judgments will be defined coinductively. That is, these derivations may be infinitely deep (but will be finitely wide). Stated differently, subtyping's coinductive nature and underlying greatest fixed point mean that a subtyping relationship holds in the absence of a counterexample, and that absence is witnessed by a potentially infinite derivation.[5]

Returning to the first of our running examples, for even to be a subtype of nat, we must be able to construct infinite derivations of even $\leqslant$ nat. On the other hand, because nat ought *not* to be a subtype of even, there must *not* exist a derivation of nat $\leqslant$ even.

The entire set of inference rules used to construct (potentially) infinite derivations of subtyping judgments can be found in Fig. 1. These rules are interpreted coinductively and are most clearly read bottom-up, from conclusion to premises. We will now comment on a few of the rules.

2.2.1 Structural subtyping of named types. Structural subtyping treats type definitions in an entirely transparent way: when $t[\vec{\alpha}] \triangleq A$ and $u[\vec{\beta}] \triangleq B$, the type $t[\theta]$ is a subtype of $u[\phi]$ exactly when the same subtyping relationship holds for their unfoldings, $\theta(A)$ and $\phi(B)$, respectively. This is

[5]For monomorphic subtyping, merely *circular* derivations [Brotherston and Simpson 2010], which are finite representations of *regular* infinite derivations, would suffice [Lakhani et al. 2022].

expressed by the UNF-S rule. Additionally, a type variable x is considered to be a subtype of only itself, as captured in the VAR-S rule.

2.2.2 Structural subtyping of structural types. Aside from the alternation of structural and named types, the rules for structural subtyping of structural types, $A \leqslant B$, are standard [Pierce 2002]. The rules decompose the structural types into their immediate subformulas and then require certain subtyping relationships on those subformulas. For example, the +S rule for variant record types is standard (see e.g. [Gay and Hole 2005; Pierce 2002]). For $+\{\ell\colon \tau_\ell\}_{\ell \in L}$ to be a subtype of $+\{k\colon \sigma_k\}_{k \in K}$, the condition $L \subseteq K$ demands that the latter type offer at least as many alternatives as, but possibly more than, the former type, thereby accounting for width subtyping of variant record types. Moreover, by requiring that $\tau_\ell \leqslant \sigma_\ell$ holds for all alternatives ℓ shared by the two types, this rule also accounts for covariant depth subtyping of variant record types.

Subtyping for the explicit polymorphic quantifiers $\forall x.\tau$ and $\exists x.\tau$ is also standard. However, we do *not* currently support subtyping for implicit polymorphism [Odersky and Läufer 1996], in which $\forall x.\tau \leqslant [\sigma/x]\tau$ and $[\sigma/x]\tau \leqslant \exists x.\tau$ would hold for all σ. Neither do we currently support bounded quantification [Cardelli et al. 1994; Cardelli and Wegner 1985]. Because our interest is in the interaction of subtyping, parametric polymorphism, and recursion, these are outside the scope of this paper and left as future work.

2.2.3 Examples. We now return to the running examples in the context of structural subtyping.

Even and odd natural numbers. For even and odd to be subtypes of nat, we must be able to construct derivations of even $\leqslant$ nat and odd $\leqslant$ nat. Because structural subtyping derivations are (potentially) infinite, they cannot be directly written down in their entirety. A finite, constructive proof of their existence instead suffices, and a useful proof technique here is coinduction. For example, for even $\leqslant$ nat and odd $\leqslant$ nat, mutual coinduction can be used:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{1} \leqslant \mathbf{1}}\ \textsc{1s}}{\mathsf{one} \leqslant \mathsf{one}}\ \textsc{unf-s} \qquad \overline{\mathsf{odd} \leqslant \mathsf{nat}}}{+\{\mathsf{z}\colon \mathsf{one},\ \mathsf{s}\colon \mathsf{odd}\} \leqslant +\{\mathsf{z}\colon \mathsf{one},\ \mathsf{s}\colon \mathsf{nat}\}}\ \textsc{+s}}{\mathsf{even} \leqslant \mathsf{nat}}\ \textsc{unf-s} \qquad \textit{and} \qquad \dfrac{\dfrac{\overline{\mathsf{even} \leqslant \mathsf{nat}}}{+\{\mathsf{s}\colon \mathsf{even}\} \leqslant +\{\mathsf{z}\colon \mathsf{one},\ \mathsf{s}\colon \mathsf{nat}\}}\ \textsc{+s}}{\mathsf{odd} \leqslant \mathsf{nat}}\ \textsc{unf-s}$$

We use a dotted line to indicate the coinductive appeals to odd $\leqslant$ nat and even $\leqslant$ nat, which can also be thought of as admissible structural subtyping rules that can always be unfolded to the corresponding infinite derivations. Each of these appeals is guarded by the UNF-S and +S rules.

Here, the full expressive power of infinite derivations is not needed. Because the types are monomorphic, circular derivations [Brotherston and Simpson 2010], which are finite representations of *regular* infinite derivations, would suffice [Lakhani et al. 2022]: For even $\leqslant$ nat, the right-hand derivation segment could be inlined within the left-hand segment, with the inlined coinductive appeal to even $\leqslant$ nat then circling back to even $\leqslant$ nat at the "root". Then odd $\leqslant$ nat is similar.

As a negative example, we *cannot* derive nat $\leqslant$ odd because, after unfolding nat and odd with the UNF-S rule, we would need to show that $\{\mathsf{z}, \mathsf{s}\} \subseteq \{\mathsf{s}\}$, which is simply false. Similarly, we *cannot* derive nat $\leqslant$ even because, after unfolding nat and even, we would need to derive nat $\leqslant$ odd.

Context-free languages. Recall that $\{\mathsf{L}^n\mathsf{R}^n\$ \mid n \geq 0\}$ is a subset of the Dyck language and that the type e_0 should accordingly be a subtype of d_0; there ought therefore to exist a derivation of $\mathsf{e}_0 \leqslant \mathsf{d}_0$. However, direct application of coinduction is not enough to establish $\mathsf{e}[\mathsf{end}] \leqslant \mathsf{d}[\mathsf{d}_0]$ because it produces an infinite stream of subgoals, $\mathsf{e}[\mathsf{end}] \leqslant \mathsf{d}[\mathsf{d}_0], \mathsf{e}[\mathsf{r}[\mathsf{end}]] \leqslant \mathsf{d}[\mathsf{d}[\mathsf{d}_0]], \ldots$, none of which is a direct instance of any preceding one. Instead, we generalize the coinductive hypothesis, proving that $\kappa \leqslant \kappa'$ implies $\mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa']$ for all named types κ and κ'. Then, because $\mathsf{end} \leqslant \mathsf{d}_0$, derivations of $\mathsf{e}[\mathsf{end}] \leqslant \mathsf{d}[\mathsf{d}_0]$ and hence of $\mathsf{e}_0 \leqslant \mathsf{d}_0$ indeed exist.

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{1} \leqslant \mathbf{1}}\ \textsc{1s}}{+\{\$\colon \mathbf{1}\} \leqslant +\{\mathsf{L}\colon \mathsf{d}[\mathsf{d}_0],\ \$\colon \mathbf{1}\}}\ \textsc{+s}}{\dfrac{\mathsf{end} \leqslant \mathsf{d}_0}{\mathsf{e}[\mathsf{end}] \leqslant \mathsf{d}[\mathsf{d}_0]}}\ \textsc{unf-s} \qquad \dfrac{}{\mathbf{1} \leqslant \mathbf{1}}\ \textsc{1s}}{\dfrac{+\{\mathsf{L}\colon \mathsf{e}[\mathsf{end}],\ \$\colon \mathbf{1}\} \leqslant +\{\mathsf{L}\colon \mathsf{d}[\mathsf{d}_0],\ \$\colon \mathbf{1}\}}{\mathsf{e}_0 \leqslant \mathsf{d}_0}\ \textsc{unf-s}}\ \textsc{+s}
\quad \textit{and} \quad
\dfrac{\dfrac{\dfrac{\dfrac{\kappa \leqslant \kappa'}{+\{\mathsf{R}\colon \kappa\} \leqslant +\{\mathsf{L}\colon \mathsf{d}[\mathsf{d}[\kappa']],\ \mathsf{R}\colon \kappa'\}}\ \textsc{+s}}{\mathsf{r}[\kappa] \leqslant \mathsf{d}[\kappa']}\ \textsc{unf-s}}{\mathsf{e}[\mathsf{r}[\kappa]] \leqslant \mathsf{d}[\mathsf{d}[\kappa']]} \qquad \kappa \leqslant \kappa'}{\dfrac{+\{\mathsf{L}\colon \mathsf{e}[\mathsf{r}[\kappa]],\ \mathsf{R}\colon \kappa\} \leqslant +\{\mathsf{L}\colon \mathsf{d}[\mathsf{d}[\kappa']],\ \mathsf{R}\colon \kappa'\}}{\mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa']}\ \textsc{unf-s}}\ \textsc{+s}
$$

In other words, the rules marked with dotted lines are admissible and can always be unfolded to the partial derivation on the right-hand side above.

This example demonstrates why circular derivations do not suffice for subtyping of recursively defined type constructors that employ non-regular recursion: In the right-hand derivation, we cannot directly close a cycle from $\mathsf{e}[\mathsf{r}[\kappa]] \leqslant \mathsf{d}[\mathsf{d}[\kappa']]$ back to $\mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa']$. The former is indeed an instance of the latter, but the subtyping depends on having $\kappa \leqslant \kappa'$ and so we are required to show the instance $\mathsf{r}[\kappa] \leqslant \mathsf{d}[\kappa']$: we need the expressive power of *non*-regular infinite derivations. Given that $\{\mathsf{L}^n\mathsf{R}^n\$ \mid n \geq 0\}$ and the Dyck language are context-free languages, perhaps it is not surprising that the regularity of circular derivations is insufficiently expressive to establish $\mathsf{e}_0 \leqslant \mathsf{d}_0$.

2.3 Undecidability of structural subtyping

A priori, it seems possible that structural subtyping in the presence of recursively defined type constructors might be decidable. After all, structural *equality* for coinductively interpreted types is decidable [Das et al. 2022], using a reduction from trace equivalence for deterministic first-order grammars, which is itself decidable but intractable [Jančar 2021]. However, structural *subtyping* is undecidable in the presence of recursively defined type constructors, as we will now show.

Our strategy for proving its undecidability will be to give a reduction from simulation of guarded Basic Process Algebra (BPA) processes [Bergstra and Klop 1984] to structural subtyping. Because simulation of guarded BPA processes is undecidable [Groote and Hüttel 1994], this reduction will witness the undecidability of structural subtyping.

2.3.1 Background on basic process algebra. Guarded BPA processes are defined by a set of guarded equations. For our purposes, a general definition of guardedness is unimportant; what is important is that any set of guarded BPA equations can be put into the following Greibach normal form [Baeten et al. 1993]:

$$X \triangleq \sum_{\ell \in L} (\ell \cdot p'_\ell)\,, \text{ where } L \text{ is nonempty and } p, q ::= \epsilon \mid Y \cdot p\,.$$

As usual for process algebras, there is a labeled transition system to describe process behavior. When restricted to processes in Greibach normal form, the labeled transition system consists of a single rule:

$$\dfrac{X \triangleq \sum_{\ell \in L} (\ell \cdot p'_\ell) \quad (a \in L)}{X \cdot q \xrightarrow{a} p'_a \odot q} \qquad \text{(no rule for } \epsilon) \qquad \textit{where} \qquad \begin{aligned} \epsilon \odot q &= q \\ (X \cdot p) \odot q &= X \cdot (p \odot q)\,. \end{aligned}$$

The simulation (or "is-simulated-by") relation, $\lesssim$, for BPA processes is the largest relation such that whenever $p \lesssim q$ and $p \xrightarrow{a} p'$ hold, there exists a process q' for which $p' \lesssim q'$ and $q \xrightarrow{a} q'$ hold. In particular, $\epsilon \lesssim q$ holds for all processes q because ϵ cannot make any transitions.

2.3.2 Reduction of BPA simulation to structural subtyping. For each guarded BPA equation in Greibach normal form, $X \triangleq \sum_{\ell \in L} (\ell \cdot p'_\ell)$ where L is nonempty, we define a corresponding type constructor $t_X[\alpha]$ that encodes the behavior of process variable X, parametrically in a type α that describes the behavior to follow that of X:

$$t_X[\alpha] \triangleq \&\{\ell\colon p'_\ell \mathbin{⨾} \alpha\}_{\ell \in L} \quad \textit{where} \quad \epsilon \mathbin{⨾} \tau = \tau \;\; \textit{and} \;\; (X \cdot p) \mathbin{⨾} \tau = t_X[p \mathbin{⨾} \tau]\,.$$

($p \mathbin{⨾} \tau$ yields a type in normal form because p is finite and τ is a named type.) The ideas behind this encoding are twofold. First, width subtyping of $\&$ ensures that a process $Y \cdot q$ can match any transition that $X \cdot p$ can take: width subtyping ensures that the type $t_Y[\beta]$ offers at least those alternatives that the type $t_X[\alpha]$ does. Second, depth subtyping for $\&$ ensures that this simulation holds hereditarily for the processes to which $X \cdot p$ and $Y \cdot q$ transition. We can prove the following.

Theorem 2.1. *Let $t \triangleq \&\{\,\}$. Then $p \lesssim q$ if and only if $(q \mathbin{⨾} t) \leqslant (p \mathbin{⨾} t)$, for all processes p and q.*

The key properties of t necessary for the proof are: that $(q \mathbin{⨾} t) \leqslant t$, for all processes q; and that $t \leqslant (p \mathbin{⨾} t)$ implies $p = \epsilon$, for all processes p. Because simulation for BPA processes is undecidable [Groote and Hüttel 1994], we therefore have the following corollary.

Corollary 2.2. *In the presence of record types with no alternatives and recursively defined type constructors, structural subtyping is undecidable.*

Although they make for an arguably cleaner proof, record types with no alternatives are not at all essential. Even if all record types must have at least one alternative, structural subtyping is still undecidable. The encoding can be revised to include an endmarker, \$, as an additional alternative for each t_X. Let t_0 be any closed type, such as $t_0 \triangleq \&\{\$\colon t_0\}$ or $t_0 \triangleq \mathbf{1}$ (among others), and define

$$t_X[\alpha] \triangleq \&\{\ell\colon p'_\ell \mathbin{⨾} \alpha\}_{\ell \in L} \mathbin{\&} \{\$\colon t_0\}\,,$$

where $p \mathbin{⨾} \tau$ is defined as above. With the revised encoding, we can prove the following theorem.

Theorem 2.3. *Let $t \triangleq \&\{\$\colon t_0\}$. Then $p \lesssim q$ if and only if $(q \mathbin{⨾} t) \leqslant (p \mathbin{⨾} t)$, for all processes p and q.*

Corollary 2.4. *In the presence of record types and recursively defined type constructors, structural subtyping is undecidable.*

Furthermore, because we assign a coinductive interpretation to all types, virtually the same theorems hold for variant record types, with $+$ substituted for $\&$ in the definitions of $t_X[\alpha]$ – only the subtyping direction changes to "$p \lesssim q$ if and only if $(p \mathbin{⨾} t) \leqslant (q \mathbin{⨾} t)$." That is, structural subtyping remains undecidable in the presence of variant record types and recursively defined type constructors, *even if* there are no record types at all.[6]

3 PARAMETRIC SUBTYPING FOR PARAMETRIC POLYMORPHISM

In this section, we identify a fragment of structural subtyping for parametric polymorphism that has a relatively simple declarative characterization. (Section 5 will show that this fragment is also decidable.) We call this fragment *parametric subtyping*, for its basis in a notion of parametricity.

Recall the context-free languages example from Section 2.2.3 in which we were trying to prove that the type e_0, corresponding to $\{\mathsf{L}^n\mathsf{R}^n\$ \mid n \geq 0\}$, is a subtype of the type d_0, corresponding to the Dyck language. We could not use direct coinduction to prove the existence of a derivation of $\mathsf{e}_0 \leqslant \mathsf{d}_0$ because that led to an infinite stream of subgoals, $\mathsf{e}[\mathsf{end}] \leqslant \mathsf{d}[\mathsf{d}_0], \mathsf{e}[\mathsf{r}[\mathsf{end}]] \leqslant \mathsf{d}[\mathsf{d}[\mathsf{d}_0]], \mathsf{e}[\mathsf{r}[\mathsf{r}[\mathsf{end}]]] \leqslant \mathsf{d}[\mathsf{d}[\mathsf{d}[\mathsf{d}_0]]], \ldots$, none of which is an instance of any preceding one. We somehow need to quotient this space into finitely many subproblems, each of which is decidable.

[6] In a mixed inductive/coinductive setting, where variant record types would be interpreted inductively, we conjecture that structural subtyping would remain undecidable even in the purely inductive fragment (*i.e.*, without record types), and that this could be proved by reducing from BPA language inclusion [Friedman 1976; Groote and Hüttel 1994], not simulation.

The key insight behind this quotienting comes in revisiting the coinductive generalization that we used to prove $\mathsf{e}_0 \leqslant \mathsf{d}_0$: in Section 2.2.3, we instead proved that $\kappa \leqslant \kappa'$ implies $\mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa']$ for all named types κ and κ'. This statement could also be viewed as an admissible inference rule:

$$\frac{\kappa \leqslant \kappa'}{\mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa']}$$

This rule looks very much like the kind of rules around which nominal subtyping [Kennedy and Pierce 2006] is based, with a key difference: nominal subtyping requires such rules to use the same type constructor on both sides of the conclusion. Importantly for our purposes, this admissible rule is parametric, in the sense that the type constructors $\mathsf{e}[-]$ and $\mathsf{d}[-]$ map related arguments, $\kappa \leqslant \kappa'$, to related results, $\mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa']$. This echoes Reynolds's characterization of parametric functions as mapping related arguments to related results [1983].

The importance of parametricity in the coinductive generalization used in this example suggests that we ought to consider a notion of subtyping that uses parametric rules, *i.e.*, rules of the form

$$\frac{\alpha_{i_1} \leqslant \beta_{j_1} \;\cdots\; \alpha_{i_m} \leqslant \beta_{j_m} \quad \beta_{j_{m+1}} \leqslant \alpha_{i_{m+1}} \;\cdots\; \beta_{j_{m+n}} \leqslant \alpha_{i_{m+n}}}{t[\vec{\alpha}] \leqslant u[\vec{\beta}]} \quad \text{but not rules like} \quad \frac{\alpha \leqslant \mathbf{1} \quad t'[\alpha] \leqslant \beta}{t[\vec{\alpha}] \leqslant u[\vec{\beta}]},$$

as a candidate for being a decidable fragment of structural subtyping with a relatively simple declarative characterization.

3.1 Declarative characterization of parametric subtyping

The requirement that the candidate fragment use only parametric rules could already serve as a relatively simple declarative characterization. However, to develop a decision procedure and prove its correctness, it is very useful to devise an equivalent declarative characterization that is more closely aligned with the presentation of structural subtyping. Before doing so, it is helpful to see why structural subtyping, as defined in Fig. 1, violates parametricity.

Consider the type definition $\mathsf{snat}[\kappa] \triangleq +\{\mathsf{z}\colon \kappa,\ \mathsf{s}\colon \mathsf{snat}[\kappa]\}$, which generalizes the type nat via the structural subtyping $\mathsf{nat} \leqslant \mathsf{snat}[\mathbf{1}]$. (The name snat, for "serialized nat", alludes to serialized data structures, as discussed in Section 7.3.) However, the admissible rule for nat and $\mathsf{snat}[\kappa]$ would be the *non-parametric* rule "$\mathsf{nat} \leqslant \mathsf{snat}[\kappa]$ if $\mathbf{1} \leqslant \kappa$." The UNF-S rule of structural subtyping cannot detect non-parametric judgments, such as $\mathbf{1} \leqslant \kappa$ here, because unfolding eagerly applies substitutions and free type parameters do not appear: by the time that structural subtyping reaches this non-parametric judgment, it will be $\mathbf{1} \leqslant [\mathbf{1}/\kappa]\kappa = \mathbf{1}$, with the non-parametricity no longer apparent in the judgment.

Therefore, instead of eagerly applying substitutions when unfolding, we need to postpone the substitutions, applying them only after determining that they do not conceal any non-parametricity. The idea of postponing substitutions in this way is inspired by the Girard–Reynolds logical relation for parametricity [Girard 1972; Reynolds 1983]. The judgments $\tau \leqslant \sigma$ and $A \leqslant B$ are revised to postpone substitutions by pushing them onto stacks when unfolding type constructor instantiations. Substitution stacks are given by the grammar

$$\textit{Substitution stacks} \quad \Theta, \Phi ::= (\cdot) \mid \theta; \Theta$$

and we thus arrive at the judgments $\tau\langle\Theta\rangle \leqslant \sigma\langle\Phi\rangle$ and $A\langle\Theta\rangle \leqslant B\langle\Phi\rangle$ for parametric subtyping. As with structural subtyping, a parametric subtyping judgment holds if there exists a (potentially infinite) derivation of that judgment using the rules found in Fig. 2. These declarative rules are again interpreted coinductively and are most clearly read bottom-up, from conclusion to premises.

To better understand these rules, it can be helpful to imagine constructing a (potentially infinite) derivation of $t[\theta]\langle\Theta\rangle \leqslant u[\phi]\langle\Phi\rangle$, where $t[\vec{\alpha}] \triangleq A$ and $u[\vec{\beta}] \triangleq B$. That would proceed as follows.

$$\frac{t[\vec{\alpha}] \triangleq A \quad u[\vec{\beta}] \triangleq B \quad A\langle\theta;\Theta\rangle \leqslant B\langle\phi;\Phi\rangle}{t[\theta]\langle\Theta\rangle \leqslant u[\phi]\langle\Phi\rangle}\ \textsc{inst-p} \qquad \frac{\theta(\alpha)\langle\Theta\rangle \leqslant \phi(\beta)\langle\Phi\rangle}{\alpha\langle\theta;\Theta\rangle \leqslant \beta\langle\phi;\Phi\rangle}\ \textsc{param-p}$$

$$\text{(no rules for } \alpha\langle\Theta\rangle \leqslant \tau\langle\Phi\rangle \text{ and } \tau\langle\Theta\rangle \leqslant \beta\langle\Phi\rangle \text{ if } \tau \text{ is not a parameter)} \qquad \frac{}{x\langle\Theta\rangle \leqslant x\langle\Phi\rangle}\ \textsc{var-p}$$

$$\frac{\tau_1\langle\Theta\rangle \leqslant \sigma_1\langle\Phi\rangle \quad \tau_2\langle\Theta\rangle \leqslant \sigma_2\langle\Phi\rangle}{\tau_1 \times \tau_2\langle\Theta\rangle \leqslant \sigma_1 \times \sigma_2\langle\Phi\rangle}\ \times\textsc{p} \qquad \frac{}{\mathbf{1}\langle\Theta\rangle \leqslant \mathbf{1}\langle\Phi\rangle}\ \mathbf{1}\textsc{p}$$

$$\frac{(L \subseteq K) \quad \forall \ell \in L\colon \tau_\ell\langle\Theta\rangle \leqslant \sigma_\ell\langle\Phi\rangle}{+\{\ell\colon \tau_\ell\}_{\ell\in L}\langle\Theta\rangle \leqslant +\{k\colon \sigma_k\}_{k\in K}\langle\Phi\rangle}\ +\textsc{p}$$

$$\frac{\sigma_1\langle\Phi\rangle \leqslant \tau_1\langle\Theta\rangle \quad \tau_2\langle\Theta\rangle \leqslant \sigma_2\langle\Phi\rangle}{\tau_1 \to \tau_2\langle\Theta\rangle \leqslant \sigma_1 \to \sigma_2\langle\Phi\rangle}\ \to\textsc{p} \qquad \frac{(K \subseteq L) \quad \forall k \in K\colon \tau_k\langle\Theta\rangle \leqslant \sigma_k\langle\Phi\rangle}{\&\{\ell\colon \tau_\ell\}_{\ell\in L}\langle\Theta\rangle \leqslant \&\{k\colon \sigma_k\}_{k\in K}\langle\Phi\rangle}\ \&\textsc{p}$$

$$\frac{(z \text{ fresh}) \quad [z/x]\tau\langle\Theta\rangle \leqslant [z/y]\sigma\langle\Phi\rangle}{\forall x.\tau\langle\Theta\rangle \leqslant \forall y.\sigma\langle\Phi\rangle}\ \forall\textsc{p} \qquad \frac{(z \text{ fresh}) \quad [z/x]\tau\langle\Theta\rangle \leqslant [z/y]\sigma\langle\Phi\rangle}{\exists x.\tau\langle\Theta\rangle \leqslant \exists y.\sigma\langle\Phi\rangle}\ \exists\textsc{p}$$

Fig. 2. Parametric subtyping rules (p for 'parametric'). These rules are interpreted coinductively.

(1) This judgment can only be derived by the inst-p rule, which is parametric subtyping's answer to structural subtyping's unf-s rule. It unfolds $t[\theta]$ and $u[\phi]$, but it does not apply the substitutions θ and ϕ eagerly, instead postponing them by pushing them onto their respective stacks, Θ and Φ. The unfoldings, *i.e.*, structural types A and B, are then compared under the extended stacks, $(\theta;\Theta)$ and $(\phi;\Phi)$, respectively. Notice that A and B will, in general, contain free occurrences of the respective parameters $\vec{\alpha}$ and $\vec{\beta}$.
(2) Next, these structural types A and B are decomposed according to parametric subtyping's rules for structural types, such as $\times$p and $\to$p. These rules are virtually the same as structural subtyping's rules for structural types, with the only difference being that substitution stacks are threaded through, unchanged, from conclusion to premises. After decomposing A and B, there are several parametric subtyping subgoals of the form $\tau\langle\theta;\Theta\rangle \leqslant \sigma\langle\phi;\Phi\rangle$ (or, in the case of the $\to$p rule's first premise, of the form $\sigma\langle\phi;\Phi\rangle \leqslant \tau\langle\theta;\Theta\rangle$).
(3) Depending on the structure of the named types τ and σ, there are several possibilities for each such judgment:
   - If both τ and σ are type constructor instantiations, then the judgment can only be derived by the inst-p rule, returning us to step (1).
   - If τ and σ are type parameters α and β, then there is no violation of parametricity here, and the judgment can be derived by the param-p rule. The substitutions θ and ϕ are popped from their respective stacks and finally applied; the resulting subgoal is of the form $\theta(\alpha)\langle\Theta\rangle \leqslant \phi(\beta)\langle\Phi\rangle$. In types in normal form, substitutions map parameters to *named* types only (recall Section 2.1.3), so one of these three cases will again apply.
   - If either τ or σ is a type parameter and the other one is not, this judgment violates parametricity. Accordingly, there is no rule that can derive this judgment, and therefore $t[\theta]\langle\Theta\rangle \leqslant u[\phi]\langle\Phi\rangle$ does *not* hold.

The notion of parametric subtyping given in Fig. 2 is sound with respect to structural subtyping as defined in Fig. 1. The substitutions postponed in stacks Θ and Φ can instead be composed and applied eagerly, transforming instances of the inst-p rule into instances of the unf-s rule and eliminating occurrences of the param-p rule.

Theorem 3.1 (Soundness of parametric subtyping). *If $\tau\langle\Theta\rangle \leqslant \sigma\langle\Phi\rangle$, then $\Theta(\tau) \leqslant \Phi(\sigma)$. Likewise, if $A\langle\Theta\rangle \leqslant B\langle\Phi\rangle$, then $\Theta(A) \leqslant \Phi(B)$.*

Proof sketch. Using the mixed induction and coinduction proof technique described by Danielsson and Altenkirch [2010]. Specifically, the proof is by lexicographic mixed induction and coinduction, first by coinduction on the (potentially infinite) structural subtyping derivation, and then by induction on the finite substitution stack Θ. □

However, the converse does not hold: parametric subtyping is incomplete with respect to structural subtyping, as the above example involving nat and snat[**1**] demonstrates.

Theorem 3.2 (Incompleteness of parametric subtyping). *τ and σ exist such that the structural subtyping $\tau \leqslant \sigma$ holds but the parametric subtyping $\tau\langle\Theta\rangle \leqslant \sigma\langle\Phi\rangle$ does not, for any Θ and Φ.*

4 DECIDING STRUCTURAL SUBTYPING FOR MONOMORPHIC TYPES

In the next section, we will present a saturation-based decision procedure for parametric subtyping for parametric polymorphism. But parameterized type constructors involve some complications, so in this section, we will provide a gentle introduction to the algorithm by presenting a saturation-based decision procedure for structural subtying of monomorphic types – that is, recursively defined types that do not take parameters nor use the structural types $\forall x.\tau$ and $\exists x.\tau$.

Establishing the decidability of structural subtyping for monomorphic types is not a contribution of this paper. One existing decision procedure (see, *e.g.*, [Lakhani et al. 2022]) directly employs backward search for a derivation of the structural subtyping judgment $t \leqslant u$, using the subtyping rules themselves (Fig. 1). This procedure crucially depends on three properties: for monomorphic types, merely circular derivations suffice to characterize structural subtyping; circular derivations are finite; and there are finitely many subtyping problems involving named monomorphic types.

For polymorphic types, these key properties will no longer hold – which is why we will introduce a forward-inference, saturation-based procedure here. But even if one is uninterested in polymorphic types, this forward-inference procedure offers several distinct advantages over the backward-search algorithm (with one potentially mitigable shortcoming), as we will discuss below.

4.1 A forward-inference decision procedure for monomorphic structural subtyping

To devise a decision procedure for monomorphic subtyping based on forward inference, we will exploit the fact that subtyping is a safety property and return to the idea that, in keeping with the safety slogan "nothing bad ever happens," a subtyping relationship $t \leqslant u$ holds when there is no counterexample. Very roughly speaking, our algorithm proceeds as a kind of automated refutation by contradiction, assuming that a derivation of $t \leqslant u$ exists and repeatedly inverting that assumed derivation to check that no violations of subtyping occur (*i.e.*, that nothing bad happens) before reaching another subtyping problem, $t' \leqslant u'$. Because the signature contains finitely many type definitions $t \triangleq A$ and $u \triangleq B$ and there are therefore finitely many subtyping problems $t \leqslant u$, we can check each problem in this way.

More precisely, the forward-inference procedure uses three judgments: the primary judgment, $t \preccurlyeq u \Rightarrow \bot$; and two intermediate judgments, $t \preccurlyeq u \Rightarrow A \preccurlyeq B$ and $t \preccurlyeq u \Rightarrow \tau \preccurlyeq \sigma$. (Notice that we use $\preccurlyeq$ to distinguish these from the declarative $\leqslant$.) Ultimately, the judgment $t \preccurlyeq u \Rightarrow \bot$ will be inferred if and only if $t \nleqslant u$, allowing us to decide the structural subtyping $t \leqslant u$ by running forward inference to saturation and checking that $t \preccurlyeq u \Rightarrow \bot$ has *not* been inferred. (Saturation is guaranteed, as we will prove in Theorem 4.3 below.) The judgments $t \preccurlyeq u \Rightarrow A \preccurlyeq B$ and $t \preccurlyeq u \Rightarrow \tau \preccurlyeq \sigma$ are inferred if and only if $A \leqslant B$ and $\tau \leqslant \sigma$, respectively, would necessarily occur as subderivations of any derivation of $t \leqslant u$ (assuming such a derivation exists).

Alternatively, these judgments can be seen as describing necessary consequences of $t \leqslant u$. From yet another perspective, these judgments can be seen as stating those *constraints* that must hold for the structural subtyping $t \leqslant u$ to be derivable, with $\bot$ being the unsatisfiable constraint. This last

$$\frac{t \triangleq A \quad u \triangleq B}{t \leqslant u \Rightarrow A \leqslant B}\ \textsc{init-f} \qquad \frac{t \leqslant u \Rightarrow t' \leqslant u' \quad t' \leqslant u' \Rightarrow \bot}{t \leqslant u \Rightarrow \bot}\ \textsc{compose-f}_\bot$$

$$\frac{t \leqslant u \Rightarrow \tau_1 \times \tau_2 \leqslant \sigma_1 \times \sigma_2 \quad (i \in \{1,2\})}{t \leqslant u \Rightarrow \tau_i \leqslant \sigma_i}\ \times\textsc{f} \qquad (\text{no } \mathbf{1}\textsc{f} \text{ rule for } t \leqslant u \Rightarrow \mathbf{1} \leqslant \mathbf{1})$$

$$\frac{t \leqslant u \Rightarrow +\{\ell\colon \tau_\ell\}_{\ell\in L} \leqslant +\{k\colon \sigma_k\}_{k\in K} \quad (\ell \in L \subseteq K)}{t \leqslant u \Rightarrow \tau_\ell \leqslant \sigma_\ell}\ +\textsc{f}$$

$$\frac{t \leqslant u \Rightarrow \tau_1 \to \tau_2 \leqslant \sigma_1 \to \sigma_2}{t \leqslant u \Rightarrow \sigma_1 \leqslant \tau_1}\ {\to}\textsc{f}_1 \qquad \frac{t \leqslant u \Rightarrow \tau_1 \to \tau_2 \leqslant \sigma_1 \to \sigma_2}{t \leqslant u \Rightarrow \tau_2 \leqslant \sigma_2}\ {\to}\textsc{f}_2$$

$$\frac{t \leqslant u \Rightarrow \&\{\ell\colon \tau_\ell\}_{\ell\in L} \leqslant \&\{k\colon \sigma_k\}_{k\in K} \quad (k \in K \subseteq L)}{t \leqslant u \Rightarrow \tau_k \leqslant \sigma_k}\ \&\textsc{f}$$

$$\frac{t \leqslant u \Rightarrow +\{\ell\colon \tau_\ell\}_{\ell\in L} \leqslant +\{k\colon \sigma_k\}_{k\in K} \quad (L \nsubseteq K)}{t \leqslant u \Rightarrow \bot}\ {+}{+}\textsc{f}_\bot$$

$$\frac{t \leqslant u \Rightarrow \&\{\ell\colon \tau_\ell\}_{\ell\in L} \leqslant \&\{k\colon \sigma_k\}_{k\in K} \quad (K \nsubseteq L)}{t \leqslant u \Rightarrow \bot}\ \&\&\textsc{f}_\bot \qquad \frac{t \leqslant u \Rightarrow A \leqslant B \quad A \perp\!\!\!\perp B}{t \leqslant u \Rightarrow \bot}\ \textsc{mismatch-f}_\bot$$

Fig. 3. Forward inference rules for deciding structural subtyping of monomorphic types (F for 'forward'). These rules are interpreted inductively. The notation $A \perp\!\!\!\perp B$ means that A and B use distinct top-level structural type constructors, such as + and **1**.

perspective will prove particularly useful in Section 3 and the decision procedure for parametric subtyping of polymorphic types presented there.

4.1.1 Forward inference. Forward inference proceeds according to the rules found in Fig. 3. Unlike the structural subtyping rules of Fig. 1, these algorithmic rules are interpreted inductively and most clearly read top-down, from premises to conclusion. To provide some intuition for this forward-inference decision procedure, we will walk through a few of the rules in detail.

The INIT-F rule. Suppose that the premises $t \triangleq A$ and $u \triangleq B$ hold. If $t \leqslant u$ is derivable, then, by inversion, it must have been derived by applying the UNF-S structural subtyping rule to a subderivation of $A \leqslant B$. That is, $A \leqslant B$ would necessarily occur as a subderivation of $t \leqslant u$ when t and u are defined by $t \triangleq A$ and $u \triangleq B$, justifying the inference of $t \leqslant u \Rightarrow A \leqslant B$ by the INIT-F rule.

The +F and ++F$_\bot$ rules. Suppose that the shared premise $t \leqslant u \Rightarrow +\{\ell\colon \tau_\ell\}_{\ell\in L} \leqslant +\{k\colon \sigma_k\}_{k\in K}$ has already been inferred – that is, that any derivation of $t \leqslant u$ would necessarily contain a subderivation of $+\{\ell\colon \tau_\ell\}_{\ell\in L} \leqslant +\{k\colon \sigma_k\}_{k\in K}$. By inversion, this subderivation can only be formed by applying the +S structural subtyping rule with subderivations of $\tau_\ell \leqslant \sigma_\ell$, for all $\ell \in L$, and only when $L \subseteq K$. Therefore, when $L \subseteq K$, the inference of $t \leqslant u \Rightarrow \tau_\ell \leqslant \sigma_\ell$, for all $\ell \in L$, by the +F rule is justified. On the other hand, when $L \nsubseteq K$, the subtypings $+\{\ell\colon \tau_\ell\}_{\ell\in L} \leqslant +\{k\colon \sigma_k\}_{k\in K}$ and hence $t \leqslant u$ are *not* derivable, justifying the inference of $t \leqslant u \Rightarrow \bot$ by the ++F$_\bot$ rule.

The COMPOSE-F$_\bot$ rule. Suppose that the premises $t \leqslant u \Rightarrow t' \leqslant u'$ and $t' \leqslant u' \Rightarrow \bot$ have already been inferred. Thus, any derivation of $t \leqslant u$ would necessarily contain a subderivation of $t' \leqslant u'$, and moreover $t' \leqslant u'$ is *not* derivable. Therefore, $t \leqslant u$ is also not derivable, justifying the inference of $t \leqslant u \Rightarrow \bot$ by the COMPOSE-F$_\bot$ rule.

The MISMATCH-F$_\bot$ rule. Suppose that the premises $t \leqslant u \Rightarrow A \leqslant B$ and $A \perp\!\!\!\perp B$ have already been inferred, where $A \perp\!\!\!\perp B$ indicates that A and B have distinct top-level structural type constructors, such as $+\{\ell\colon \tau_\ell\}_{\ell\in L} \perp\!\!\!\perp \mathbf{1}$. Because the first premise has been inferred, any derivation of $t \leqslant u$ would necessarily contain a subderivation of $A \leqslant B$. Because $A \perp\!\!\!\perp B$, inversion shows there is no structural subtyping rule that could possibly form this subderivation. Hence $t \leqslant u$ is *not* derivable, justifying the inference of $t \leqslant u \Rightarrow \bot$ by the MISMATCH-F$_\bot$ rule.

4.1.2 Example. Returning to our running example of even and odd natural numbers, we can examine the inferences made by our algorithm. By virtue of the INIT-F rule, the following judgments, among others, will be inferred:

$$\begin{aligned}&\mathsf{even} \leqslant \mathsf{nat} \Rightarrow +\{\mathsf{z}\colon \mathsf{one},\ \mathsf{s}\colon \mathsf{odd}\} \leqslant +\{\mathsf{z}\colon \mathsf{one},\ \mathsf{s}\colon \mathsf{nat}\};\\ &\mathsf{odd} \leqslant \mathsf{nat} \Rightarrow +\{\mathsf{s}\colon \mathsf{even}\} \leqslant +\{\mathsf{z}\colon \mathsf{one},\ \mathsf{s}\colon \mathsf{nat}\};\ \textit{and}\\ &\mathsf{one} \leqslant \mathsf{one} \Rightarrow \mathbf{1} \leqslant \mathbf{1}\,.\end{aligned}$$

Because $\{\mathrm{z},\mathrm{s}\} \subseteq \{\mathrm{z},\mathrm{s}\}$ as well as $\{\mathrm{s}\} \subseteq \{\mathrm{z},\mathrm{s}\}$, the +F rule then allows us to infer

$$\begin{aligned}\mathsf{even} \leqslant \mathsf{nat} \Rightarrow \mathsf{one} \leqslant \mathsf{one}\ \ \textit{and}\ \ &\mathsf{even} \leqslant \mathsf{nat} \Rightarrow \mathsf{odd} \leqslant \mathsf{nat};\ \ \textit{as well as}\\ &\mathsf{odd} \leqslant \mathsf{nat} \Rightarrow \mathsf{even} \leqslant \mathsf{nat}\end{aligned}$$

as necessary consequences of the initial judgments about $\mathsf{even} \leqslant \mathsf{nat}$ and $\mathsf{odd} \leqslant \mathsf{nat}$. At this point, saturation has been reached: no inference deduces any judgment that has not already been inferred. Because $\mathsf{even} \leqslant \mathsf{nat} \Rightarrow \bot$ has *not* been inferred upon saturation, we may conclude that $\mathsf{even} \leqslant \mathsf{nat}$ is derivable – *i.e.*, that even is a subtype of nat. Likewise, we conclude that odd is a subtype of nat.

4.2 Correctness of the forward-inference decision procedure

The forward-inference algorithm is both sound and complete with respect to (the monomorphic fragment of) structural subtyping as defined in Fig. 1. The proof of soundness relies on a key lemma.

LEMMA 4.1. *Upon saturation:*
(1) If $t \leqslant u \Rightarrow \tau \leqslant \sigma$ and $t \leqslant u \not\Rightarrow \bot$, then $\tau \leqslant \sigma$.
(2) If $t \leqslant u \Rightarrow A \leqslant B$ and $t \leqslant u \not\Rightarrow \bot$, then $A \leqslant B$.

PROOF SKETCH. By mutual coinduction on the derivations of $\tau \leqslant \sigma$ and $A \leqslant B$. □

THEOREM 4.2 (SOUNDNESS AND COMPLETENESS). *Upon saturation, $t \leqslant u \not\Rightarrow \bot$ if and only if $t \leqslant u$.*

PROOF SKETCH. From left to right, by appealing to Lemma 4.1; from right to left, by induction on the finite derivation of $t \leqslant u \Rightarrow \bot$ to establish a (meta-)contradiction. □

We do not provide further details of these specific proofs here because this forward-inference procedure for structural subtyping of monomorphic types will be subsumed by the decision procedure for parametric subtyping of polymorphic types that will eventually be presented in Section 5.

The preceding theorem establishes that the above forward-inference algorithm is a semi-decision procedure. However, in this setting, forward inference is, in fact, guaranteed to saturate, making our algorithm a full-fledged decision procedure for structural subtyping of monomorphic types.

THEOREM 4.3 (TERMINATION). *Forward inference according to the rules of Fig. 3 always saturates.*

PROOF SKETCH. Finitely many definitions of the form $t \triangleq A$ and $u \triangleq B$ can be drawn from a given signature Σ. For each such pair of structural types A and B, there are finitely many subformulas (without unfolding type definitions). Each of the rules found in Fig. 3 infers a judgment $t \leqslant u \Rightarrow \tau \leqslant \sigma$ only if either τ and σ are subformulas of A and B, respectively, or τ and σ are subformulas of B and A, respectively (again, without unfolding definitions). Therefore, only finitely many judgments can be inferred, so forward inference must eventually saturate. □

4.3 Further remarks

With respect to a backward-search decision procedure for structural subtyping of monomorphic types (see, *e.g.*, [Lakhani et al. 2022]), the above forward-inference algorithm has two advantages. First, it is naturally incremental and compositional: If additional type definitions are introduced later in the program, inferences involving only prior definitions still hold and need not be performed again; only inferences involving the newly introduced definitions need to be performed. Second, the forward-inference algorithm can take advantage of inferences made along one branch when considering another branch.

With respect to the backward-search algorithm, our forward-inference algorithm, as formulated in Fig. 3, does have one shortcoming: it does not account for structural subtypings that arise from uninhabited types that exist when types' interpretation is inductive or mixed inductive/coinductive, such as those that appear in work by Ligatti et al. [2017] and Lakhani et al. [2022]. In this paper, we choose to work only with types that are interpreted coinductively. Because all such types are inhabited, the above forward-inference algorithm needs not account for such subtypings. We conjecture that the algorithm can be extended to inductive and mixed inductive/coinductive settings, but we leave that as future work.

5 DECIDING PARAMETRIC SUBTYPING FOR PARAMETRIC POLYMORPHISM

Leveraging the structure of the forward-inference algorithm for deciding structural subtyping of monomorphic types presented in the preceding section, we will now present a related algorithm for deciding parametric subtyping of polymorphic types.

At a high level, the algorithm uses saturating forward inference to derive the most general admissible parametric rules for each pair of defined type constructors, such as "$\mathrm{e}[\kappa] \leqslant \mathrm{d}[\kappa']$ if $\kappa \leqslant \kappa'$". Then, once these rules have been derived, a parametric subtyping problem can be decided by a second, backward proof construction phase that builds a finite derivation using the rules derived during the first phase.

5.1 Details of the decision procedure

As in the special case algorithm for monomorphic types (Section 4), there are three judgments for necessary consequences of a subtyping relationship between two types involving type constructors. However, now that type constructors may take parameters, these judgments must account for those parameters. Also, we choose to explicitly incorporate variances into the judgments for convenience. Possible variances ξ and ζ are co- and contravariance, which we write as $+$ and $-$, respectively. (Bivariance is handled as mutual co- and contravariance, and nonvariance is handled implicitly by the algorithm.) An operation, $\neg$, on variances, given by $\neg(+) = -$ and $\neg(-) = +$, is also useful.

Because the forward inference judgments will use explicit variances and because we will want to relate them to the declarative characterization of parametric subtyping, it is helpful to define the abbreviation $\tau\langle\Theta\rangle \leqslant_\xi \sigma\langle\Phi\rangle$ such that: $\tau\langle\Theta\rangle \leqslant_+ \sigma\langle\Phi\rangle$ if and only if $\tau\langle\Theta\rangle \leqslant \sigma\langle\Phi\rangle$; and $\tau\langle\Theta\rangle \leqslant_- \sigma\langle\Phi\rangle$ if and only if $\sigma\langle\Phi\rangle \leqslant \tau\langle\Theta\rangle$. The abbreviation $A\langle\Theta\rangle \leqslant_\xi B\langle\Phi\rangle$ is defined analogously.

We finally arrive at the following three judgments for forward inference. (We again use $\preccurlyeq$ for distinction.) The judgment $t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathrel{\#} \xi \Rightarrow \bot$ will be inferred if and only if there are no substitutions θ and ϕ and stacks Θ and Φ for which a derivation of $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ exists. The judgments $t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathrel{\#} \xi \Rightarrow \tau \preccurlyeq \sigma \mathrel{\#} \xi'$ and $t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathrel{\#} \xi \Rightarrow A \preccurlyeq B \mathrel{\#} \xi'$ will be inferred if and only if $\tau\langle\theta;\Theta\rangle \leqslant_{\xi'} \sigma\langle\phi;\Phi\rangle$ and $A\langle\theta;\Theta\rangle \leqslant_{\xi'} B\langle\phi;\Phi\rangle$, respectively, would necessarily occur as subderivations of any derivation of $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ (assuming such a derivation exists).

5.1.1 Phase 1: Forward inference. Forward inference proceeds according to the rules found in Fig. 4. Once again, these rules are interpreted inductively and are more clearly read top-down, from

$$\frac{t[\vec{\alpha}] \triangleq A \quad u[\vec{\beta}] \triangleq B}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow A \preccurlyeq B \mathbin{\#} \xi}\ \textsc{init-f}$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \tau_1 \times \tau_2 \preccurlyeq \sigma_1 \times \sigma_2 \mathbin{\#} \xi' \quad (i \in \{1,2\})}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \tau_i \preccurlyeq \sigma_i \mathbin{\#} \xi'}\ \times\textsc{f} \qquad (\text{no } \mathbf{1}\textsc{f} \text{ rule for } t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \mathbf{1} \preccurlyeq \mathbf{1} \mathbin{\#} \xi')$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow +\{\ell : \tau_\ell\}_{\ell \in L} \preccurlyeq +\{k : \sigma_k\}_{k \in K} \mathbin{\#} \xi' \quad (L \subseteq_{\xi'} K) \quad (\ell \in L \cap K)}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \tau_\ell \preccurlyeq \sigma_\ell \mathbin{\#} \xi'}\ +\textsc{f}$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \tau_1 \to \tau_2 \preccurlyeq \sigma_1 \to \sigma_2 \mathbin{\#} \xi'}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \tau_1 \preccurlyeq \sigma_1 \mathbin{\#} \neg\xi'}\ {\to}\textsc{f}_1 \qquad \frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \tau_1 \to \tau_2 \preccurlyeq \sigma_1 \to \sigma_2 \mathbin{\#} \xi'}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \tau_2 \preccurlyeq \sigma_2 \mathbin{\#} \xi'}\ {\to}\textsc{f}_2$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \&\{\ell : \tau_\ell\}_{\ell \in L} \preccurlyeq \&\{k : \sigma_k\}_{k \in K} \mathbin{\#} \xi' \quad (K \subseteq_{\xi'} L) \quad (k \in L \cap K)}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \tau_k \preccurlyeq \sigma_k \mathbin{\#} \xi'}\ \&\textsc{f}$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \forall x.\tau \preccurlyeq \forall y.\sigma \mathbin{\#} \xi' \quad (z \text{ fresh})}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow [z/x]\tau \preccurlyeq [z/x]\sigma \mathbin{\#} \xi'}\ \forall\textsc{f} \qquad \frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \exists x.\tau \preccurlyeq \exists y.\sigma \mathbin{\#} \xi' \quad (z \text{ fresh})}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow [z/x]\tau \preccurlyeq [z/x]\sigma \mathbin{\#} \xi'}\ \exists\textsc{f}$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow t'[\theta'] \preccurlyeq u'[\phi'] \mathbin{\#} \xi' \quad t'[\vec{\alpha}'] \preccurlyeq u'[\vec{\beta}'] \mathbin{\#} \xi' \Rightarrow \alpha' \preccurlyeq \beta' \mathbin{\#} \zeta}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \theta'(\alpha') \preccurlyeq \phi'(\beta') \mathbin{\#} \zeta}\ \textsc{compose-f}$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow t'[\theta'] \preccurlyeq u'[\phi'] \mathbin{\#} \xi' \quad t'[\vec{\alpha}'] \preccurlyeq u'[\vec{\beta}'] \mathbin{\#} \xi' \Rightarrow \bot}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \bot}\ \textsc{compose-f}_\bot$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \alpha \preccurlyeq \sigma \mathbin{\#} \xi' \quad (\sigma \neq \beta)}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \bot}\ \textsc{param-l-f}_\bot \qquad \frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \tau \preccurlyeq \beta \mathbin{\#} \xi' \quad (\tau \neq \alpha)}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \bot}\ \textsc{param-r-f}_\bot$$

$$(\text{no rule for } t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow x \preccurlyeq x \mathbin{\#} \xi')$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow x \preccurlyeq \tau \mathbin{\#} \xi' \quad (\tau \neq x)}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \bot}\ \textsc{var-l-f}_\bot \qquad \frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \tau \preccurlyeq x \mathbin{\#} \xi' \quad (\tau \neq x)}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \bot}\ \textsc{var-r-f}_\bot$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow +\{\ell : \tau_\ell\}_{\ell \in L} \preccurlyeq +\{k : \sigma_k\}_{k \in K} \mathbin{\#} \xi' \quad (L \not\subseteq_{\xi'} K)}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \bot}\ {+}{+}\textsc{f}_\bot$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \&\{\ell : \tau_\ell\}_{\ell \in L} \preccurlyeq \&\{k : \sigma_k\}_{k \in K} \mathbin{\#} \xi' \quad (K \not\subseteq_{\xi'} L)}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \bot}\ \&\&\textsc{f}_\bot$$

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow A \preccurlyeq B \mathbin{\#} \xi' \quad A \perp\!\!\!\perp B}{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \mathbin{\#} \xi \Rightarrow \bot}\ \textsc{mismatch-f}_\bot$$

Fig. 4. Forward inference rules for phase 1 of deciding parametric subtyping for parametric polymorphism (F for 'forward'). These rules are interpreted inductively. The notation $A \perp\!\!\!\perp B$ means that A and B use distinct top-level structural type constructors, such as + and $\mathbf{1}$. Also, $L \subseteq_+ K$ iff $L \subseteq K$; and $L \subseteq_- K$ iff $L \supseteq K$.

premises to conclusion. Many of the rules are carried over from the decision procedure for structural subtyping of monomorphic types that was described in Fig. 3 of Section 4, with the addition of parameters and variances. For example, the essential aspects of the $\times$F, +F, and MISMATCH-F$_\bot$ rules are unchanged from Fig. 3. We will detail a few of the other rules.

The COMPOSE-F *rule.* The most important difference between the rules of Fig. 4 and those of Fig. 3 is that it is now possible to infer judgments of the form $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \ \#\ \xi \Rightarrow \alpha \leqslant \beta \ \#\ \xi'$. These represent constraints that must hold of any instantiation $t[\theta] \leqslant u[\phi] \ \#\ \xi$ of type constructors t and u. This idea is captured in the COMPOSE-F rule: Suppose that the first premise, $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \ \#\ \xi \Rightarrow t'[\theta'] \leqslant u'[\phi'] \ \#\ \xi'$, has already been inferred – that is, that any derivation of $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ would necessarily contain a subderivation of $t'[\theta']\langle\theta;\Theta\rangle \leqslant_{\xi'} u'[\phi']\langle\phi;\Phi\rangle$. Furthermore, suppose that the premise $t'[\vec{\alpha}'] \leqslant u'[\vec{\beta}'] \ \#\ \xi' \Rightarrow \alpha' \leqslant \beta' \ \#\ \zeta$ has already been inferred – that is, that any derivation of $t'[\theta']\langle\theta;\Theta\rangle \leqslant_{\xi'} u'[\phi']\langle\phi;\Phi\rangle$ would necessarily contain a subderivation of $\alpha'\langle\theta';(\theta;\Theta)\rangle \leqslant_\zeta \beta'\langle\phi';(\phi;\Phi)\rangle$. It then follows from the PARAM-P rule and transitivity of containment that $\theta'(\alpha')\langle\theta;\Theta\rangle \leqslant_\zeta \phi'(\beta')\langle\phi;\Phi\rangle$ would necessarily occur as a subderivation of $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$, thereby justifying the inference of $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \ \#\ \xi \Rightarrow \theta'(\alpha') \leqslant \phi'(\beta') \ \#\ \zeta$.

The PARAM-L-F$_\perp$ *and* PARAM-R-F$_\perp$ *rules.* Suppose that the premise $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \ \#\ \xi \Rightarrow \alpha \leqslant \sigma \ \#\ \xi'$ of the PARAM-L-F$_\perp$ rule has already been inferred, with σ not a parameter β. That is, any derivation of $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ would necessarily contain a subderivation of $\alpha\langle\theta;\Theta\rangle \leqslant_{\xi'} \sigma\langle\phi;\Phi\rangle$. However, because σ is not a parameter, there is no rule in Fig. 2 that could have derived that subderivation. Therefore, no derivation of $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ can exist, thereby justifying the inference of $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \ \#\ \xi \Rightarrow \perp$ by the PARAM-L-F$_\perp$ rule. The PARAM-R-F$_\perp$ rule is analogous, as are the VAR-L-F$_\perp$ and VAR-R-F$_\perp$ rules.

5.1.2 Example. Returning to our running example of context-free languages, we can examine the inferences made by our algorithm to infer admissible subtyping rules for pairs of type constructors. By virtue of the INIT-F rule, the following judgments, among others, will be inferred.

$$\begin{aligned}
&(1)\ \ \mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa'] \ \#\ {+} \Rightarrow {+}\{\mathsf{L}\colon \mathsf{e}[\mathsf{r}[\kappa]],\ \mathsf{R}\colon \kappa\} \leqslant {+}\{\mathsf{L}\colon \mathsf{d}[\mathsf{d}[\kappa']],\ \mathsf{R}\colon \kappa'\} \ \#\ {+}\\
&(2)\ \ \mathsf{r}[\kappa] \leqslant \mathsf{d}[\kappa'] \ \#\ {+} \Rightarrow {+}\{\mathsf{R}\colon \kappa\} \leqslant {+}\{\mathsf{L}\colon \mathsf{d}[\mathsf{d}[\kappa']],\ \mathsf{R}\colon \kappa'\} \ \#\ {+}\\
&(3)\ \ \mathsf{e}_0 \leqslant \mathsf{d}_0 \ \#\ {+} \Rightarrow {+}\{\mathsf{L}\colon \mathsf{e}[\mathsf{end}],\ \$\colon \mathsf{one}\} \leqslant {+}\{\mathsf{L}\colon \mathsf{d}[\mathsf{d}_0],\ \$\colon \mathsf{one}\} \ \#\ {+}\\
&(4)\ \ \mathsf{end} \leqslant \mathsf{d}_0 \ \#\ {+} \Rightarrow {+}\{\$\colon \mathsf{one}\} \leqslant {+}\{\mathsf{L}\colon \mathsf{d}[\mathsf{d}_0],\ \$\colon \mathsf{one}\} \ \#\ {+}\\
&(5)\ \ \mathsf{one} \leqslant \mathsf{one} \ \#\ {+} \Rightarrow \mathbf{1} \leqslant \mathbf{1} \ \#\ {+}
\end{aligned}$$

Because $\{\mathsf{R}\} \subseteq \{\mathsf{L},\mathsf{R}\} \subseteq \{\mathsf{L},\mathsf{R}\}$ and $\{\$\} \subseteq \{\mathsf{L},\$\} \subseteq \{\mathsf{L},\$\}$, the +F rule then allows us to infer

$$\begin{aligned}
&(6)\ \ \mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa'] \ \#\ {+} \Rightarrow \mathsf{e}[\mathsf{r}[\kappa]] \leqslant \mathsf{d}[\mathsf{d}[\kappa']] \ \#\ {+} \qquad &&(9)\ \ \mathsf{e}_0 \leqslant \mathsf{d}_0 \ \#\ {+} \Rightarrow \mathsf{e}[\mathsf{end}] \leqslant \mathsf{d}[\mathsf{d}_0] \ \#\ {+}\\
&(7)\ \ \mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa'] \ \#\ {+} \Rightarrow \kappa \leqslant \kappa' \ \#\ {+} &&(10)\ \ \mathsf{e}_0 \leqslant \mathsf{d}_0 \ \#\ {+} \Rightarrow \mathsf{one} \leqslant \mathsf{one} \ \#\ {+}\\
&(8)\ \ \mathsf{r}[\kappa] \leqslant \mathsf{d}[\kappa'] \ \#\ {+} \Rightarrow \kappa \leqslant \kappa' \ \#\ {+} &&(11)\ \ \mathsf{end} \leqslant \mathsf{d}_0 \ \#\ {+} \Rightarrow \mathsf{one} \leqslant \mathsf{one} \ \#\ {+}
\end{aligned}$$

as necessary consequences of the initial judgments. The COMPOSE-F rule can be applied to (6) and (7), as well as to (9) and (7), to infer

$$\begin{aligned}
&(12)\ \ \mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa'] \ \#\ {+} \Rightarrow \mathsf{r}[\kappa] \leqslant \mathsf{d}[\kappa'] \ \#\ {+}\\
&(13)\ \ \mathsf{e}_0 \leqslant \mathsf{d}_0 \ \#\ {+} \Rightarrow \mathsf{end} \leqslant \mathsf{d}_0 \ \#\ {+}.
\end{aligned}$$

(The COMPOSE-F rule could also be applied to (12) and (8) to infer $\mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa'] \ \#\ {+} \Rightarrow \kappa \leqslant \kappa' \ \#\ {+}$, but that has already been inferred as (7).) At this point, saturation has been reached for all pairs of constructors above. Because no such pair has had $\perp$ inferred as a consequence by the time saturation occurs, admissible subtyping rules for all such pairs do exist. Collecting the respective atomic constraints, namely (7) and (8), we see that these admissible rules are

$$\frac{\kappa \leqslant \kappa' \ \#\ {+}}{\mathsf{e}[\kappa] \leqslant \mathsf{d}[\kappa'] \ \#\ {+}}, \quad \frac{\kappa \leqslant \kappa' \ \#\ {+}}{\mathsf{r}[\kappa] \leqslant \mathsf{d}[\kappa'] \ \#\ {+}}, \quad \frac{}{\mathsf{e}_0 \leqslant \mathsf{d}_0 \ \#\ {+}}, \quad \frac{}{\mathsf{end} \leqslant \mathsf{d}_0 \ \#\ {+}}, \quad \textit{and} \quad \frac{}{\mathsf{one} \leqslant \mathsf{one} \ \#\ {+}}.$$

$$\frac{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \not\Rightarrow \bot \quad \forall(t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \alpha \leqslant \beta \,\#\, \zeta) \colon \theta(\alpha) \leqslant \phi(\beta) \,\#\, \zeta}{t[\theta] \leqslant u[\phi] \,\#\, \xi}\ \textsc{compose-b} \qquad \frac{}{x \leqslant x \,\#\, \xi}\ \textsc{var-b}$$

Fig. 5. Backward proof construction rules for phase 2 of deciding parameteric subtyping for parametric polymorphism (B for 'backward'). These rules are interpreted inductively.

5.1.3 Phase 2: Backward proof construction. Having inferred the most general admissible parametric rule for each pair of type constructors, the second, backward proof construction phase begins. For this phase, we introduce a judgment $\tau \leqslant \sigma \,\#\, \xi$. Given a parametric subtyping problem $\tau \leqslant \sigma \,\#\, \xi$, the types τ and σ are examined, searching for a derivation according to the rules of Fig. 5.

Because parameters can only appear free within type definitions, neither τ nor σ can be a parameter. Also, a type variable is a subtype of only itself, so there are no rules for $x \leqslant u[\phi] \,\#\, \xi$ nor $t[\theta] \leqslant x \,\#\, \xi$. If τ is $t[\theta]$ and σ is $u[\phi]$, then the algorithm checks that $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \bot$ has *not* been inferred during the forward-inference phase. Provided that is the case, then all atomic constraints $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \alpha \leqslant \beta \,\#\, \zeta$ are gathered. Backward proof construction continues by building derivations of each $\theta(\alpha) \leqslant \phi(\beta) \,\#\, \zeta$ that corresponds to an atomic constraint on $t[\vec{\alpha}]$ and $u[\vec{\beta}]$. Backward proof construction terminates here because the types become smaller.

5.1.4 Example. Given the admissible rules for the context-free languages example that were inferred in the first phase of the algorithm, we may conclude that $\mathsf{e}_0 \leqslant \mathsf{d}_0$ holds. Moreover, backward proof construction over these admissible rules can be used to decide other subtying problems. For example, $\mathsf{e}[\mathsf{e}[\mathsf{e}_0]] \leqslant \mathsf{d}[\mathsf{d}[\mathsf{d}_0]] \,\#\, +$ can be confirmed when backward proof construction builds the following derivation.

$$\frac{\dfrac{\dfrac{}{\mathsf{e}_0 \leqslant \mathsf{d}_0 \,\#\, +}\ \textsc{compose-b}}{\mathsf{e}[\mathsf{e}_0] \leqslant \mathsf{d}[\mathsf{d}_0] \,\#\, +}\ \textsc{compose-b}}{\mathsf{e}[\mathsf{e}[\mathsf{e}_0]] \leqslant \mathsf{d}[\mathsf{d}[\mathsf{d}_0]] \,\#\, +}\ \textsc{compose-b}$$

5.2 Correctness of the decision procedure for parametric subtyping

The algorithm described above is both sound and complete with respect to the declarative characterization of parametric subtyping given in Fig. 2. We give only sketches of the proofs here; details can be found in Appendix A of the supplementary materials.

Following the pattern laid out for monomorphic types, the proof of soundness relies on the following key lemma that generalizes Lemma 4.1.

Lemma 5.1. *Given a saturated database:*

(1) If $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \tau \leqslant \sigma \,\#\, \xi'$; $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \not\Rightarrow \bot$; *and* $\alpha\langle\Theta\rangle \leqslant_\zeta \beta\langle\Phi\rangle$ *for each* $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \alpha \leqslant \beta \,\#\, \zeta$; *then* $\tau\langle\Theta\rangle \leqslant_{\xi'} \sigma\langle\Phi\rangle$.

(2) If $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow A \leqslant B \,\#\, \xi'$; $t[\vec{\alpha}] \leqslant u[\vec{\beta}]\xi \not\Rightarrow \bot$; *and* $\alpha\langle\Theta\rangle \leqslant_\zeta \beta\langle\Phi\rangle$ *for each* $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \alpha \leqslant \beta \,\#\, \zeta$; *then* $A\langle\Theta\rangle \leqslant_{\xi'} B\langle\Phi\rangle$.

Proof sketch. By mutual coinduction on the (potentially infinite) derivations of $\tau\langle\Theta\rangle \leqslant_{\xi'} \sigma\langle\Phi\rangle$ and $A\langle\Theta\rangle \leqslant_{\xi'} B\langle\Phi\rangle$. □

Soundness then follows by structural induction on the derivation using the rules of Fig. 5 that was built by backward proof construction.

Theorem 5.2 (Soundness). *If* $\tau \leqslant \sigma \,\#\, \xi$, *then* $\tau\langle\Theta\rangle \leqslant_\xi \sigma\langle\Phi\rangle$ *for all stacks* Θ *and* Φ.

Completeness also requires a lemma, but then follows by structural induction on the type τ.

Lemma 5.3.

(1) If $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ *and* $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow A \leqslant B \;\#\; \xi'$*, then* $A\langle\theta;\Theta\rangle \leqslant_{\xi'} B\langle\phi;\Phi\rangle$*.*

(2) If $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ *and* $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \tau \leqslant \sigma \;\#\; \xi'$*, then* $\tau\langle\theta;\Theta\rangle \leqslant_{\xi'} \sigma\langle\phi;\Phi\rangle$*.*

(3) If $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$*, then* $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \not\Rightarrow \bot$*.*

Proof sketch. Each part is proved as follows.

(1) Directly, noting that only the init-f rule can derive $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow A \leqslant B \;\#\; \xi'$.
(2) By induction on the finite derivation of $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \tau \leqslant \sigma \;\#\; \xi'$.
(3) We generalize the lemma to show that $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ and $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \bot$ together imply a meta-contradiction. This is proved by induction on the finite derivation of $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \bot$. □

Theorem 5.4 (Completeness). *If* $\tau\langle\Theta\rangle \leqslant_\xi \sigma\langle\Phi\rangle$*, then* $\Theta(\tau) \leqslant \Phi(\sigma) \;\#\; \xi$*. As a particular case,* $\tau\langle\cdot\rangle \leqslant_\xi \sigma\langle\cdot\rangle$ *implies* $\tau \leqslant \sigma \;\#\; \xi$*.*

We must also prove that forward inference and backward proof construction terminate.

Theorem 5.5 (Termination). *Forward inference and backward proof construction according to the rules of Figs. 4 and 5, respectively, terminate.*

Proof sketch. Finitely many definitions $t[\vec{\alpha}] \triangleq A$ and $u[\vec{\beta}] \triangleq B$ can be drawn from a given signature and combined with one of two variances. For each such triple, the structural types A and B have finitely many subformulas (without unfolding definitions). Each of the rules found in Fig. 4, including the compose-f rule, infers a judgment $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \tau \leqslant \sigma \;\#\; \xi'$ only if τ and σ are subformulas of A and B, respectively (again, without unfolding definitions). Because only finitely many judgments can be inferred, forward inference in phase 1 must terminate. Backward proof construction in phase 2 terminates because its recursion occurs at successive subformulas. □

5.3 Nonrecursive type abbreviations

Consider the type constructors $t[\alpha] \triangleq \alpha \times \mathbf{1}$ and $u[\beta] \triangleq \mathbf{1} \times \beta$. Purely structurally, $t[\mathbf{1}]$ would be a subtype of $u[\mathbf{1}]$, with both $t[\mathbf{1}]$ and $u[\mathbf{1}]$ unfolding to $\mathbf{1} \times \mathbf{1}$ because type constructors are treated transparently by the unf-s rule. However, when constructors are treated parametrically, this relationship does *not* hold: $\alpha\langle\mathbf{1}/\alpha;\Theta\rangle \leqslant \mathbf{1}\langle\mathbf{1}/\beta;\Phi\rangle$ and $\mathbf{1}\langle\mathbf{1}/\alpha;\Theta\rangle \leqslant \beta\langle\mathbf{1}/\beta;\Phi\rangle$ violate parametricity.

If we would like certain definitions to act as mere abbreviations that are (conceptually) always expanded, then our system can easily accommodate this as long as those definitions are not recursive. To integrate such nonrecursive type abbreviations into the declarative characterization of parametric subtyping, we could add the following rule and restrict the inst-p to apply only when neither type constructor is a nonrecursive abbreviation.

$$\frac{t[\vec{\alpha}] \triangleq A \quad u[\vec{\beta}] \triangleq B \quad (t \text{ or } u \text{ is nonrecursive abbrev.}) \quad \theta(A)\langle\Theta\rangle \leqslant \phi(B)\langle\Phi\rangle}{t[\theta]\langle\Theta\rangle \leqslant u[\phi]\langle\Phi\rangle}\ \textsc{unf-p}$$

Forward inference in phase 1 and backward proof construction in phase 2 of the decision procedure would similarly expand nonrecursive abbreviations. It is easy to see that both phases still terminate.

6 IMPLEMENTATION

Our implementation in Standard ML is available in the supplementary materials (and intended as a later artifact submission). It provides a syntax for defining types, deriving parametric inference rules, and checking subtyping. All examples from this paper are available in `paper.poly`. Because of mutual recursion, the implementation proceeds in two phases: first checking basic consistency and normalizing types, thereby extending the signature with additional, internal definitions. The second phase executes the saturation algorithm (Section 5) and answers queries by consulting the saturated database. There are only the following few minor points of departure from this paper.

6.1 Nonrecursive type abbreviations.

The implementation allows *type definitions* (which are always treated parametrically) but also explicit *type abbreviations* which may be parameterized but must be nonrecursive. These abbreviations are expanded structurally, as described above. Depending on the larger language context, an alternative would be to treat every nonrecursive type definition as an abbreviation.

6.2 Elaboration to normal form.

Before elaboration, every definition in the signature Σ has the form $t[\vec{\alpha}] \triangleq A$ where A is structural (which guarantees contractiveness) but may not be in normal form. We map each such definition to $t[\vec{\alpha}] \triangleq A^*$, using the auxiliary translation $A^\dagger$ in which A need not be structural. In the process, we may add further (internal) definitions to the signature. We show this translation only for the types $t[\theta]$, α, $A_1 \times A_2$, $\mathbf{1}$, and $\exists x.A$; all others are analogous.

$$
\begin{aligned}
(A_1 \times A_2)^* &= A_1^\dagger \times A_2^\dagger & \qquad (t[\theta])^\dagger &= t[\theta^\dagger] \\
(\mathbf{1})^* &= \mathbf{1} & (\alpha)^\dagger &= \alpha \\
(\exists x.A)^* &= \exists x.\ [x/\alpha]([\alpha/x]A)^\dagger, & (A)^\dagger &= t[\vec{\alpha}] \text{ for } A \text{ structural, where } \vec{\alpha} = \mathsf{free}(A) \\
\vdots\quad & \quad \text{where } \alpha \text{ fresh} & & \quad \text{and } \Sigma := \Sigma, t[\vec{\alpha}] \triangleq A^*
\end{aligned}
$$

Here, $\theta^\dagger$ is defined pointwise. Computing the free type parameters $\mathsf{free}(A)$ avoids creating internal definitions with unnecessary parameters. Also, notice that, during elaboration, quantified type variables x become parameters α that are then instantiated with the corresponding variable x after normalization. Therefore, no case for $(x)^\dagger$ is needed. We can also obtain additional sharing (and therefore faster convergence of the saturation algorithm) in the clause for $(A)^\dagger$ by reusing a definition $u[\vec{\alpha}]$ if $u[\vec{\alpha}] \triangleq A^*$ is already in the signature Σ (modulo renaming of the parameters).

6.3 Indexing in the database.

In the implementation, we combine all facts $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow \cdots$ for a given t, u, and ξ into a single entry $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow C$, where constraints C are given by

$$
\textit{Constraints} \quad C ::= \bot \mid \alpha_i \leqslant \beta_j \# \zeta \mid \top \mid C_1 \wedge C_2
$$

We keep constraints C in a normal form where no entries are repeated and any conjunction with $\bot$ is reduced to $\bot$. This facilitates efficient lookup and detection of saturation.

7 EXAMPLES

7.1 Lists

The polymorphic type of lists of elements of a type α, as well as the types of empty and nonempty lists, can be defined as follows.

$$
\mathsf{list}[\alpha] \triangleq +\{\mathsf{nil}\colon \mathbf{1}, \mathsf{cons}\colon \alpha \times \mathsf{list}[\alpha]\} \qquad \mathsf{elist} \triangleq +\{\mathsf{nil}\colon \mathbf{1}\} \;\; \textit{and} \;\; \mathsf{nelist}[\alpha] \triangleq +\{\mathsf{cons}\colon \alpha \times \mathsf{list}[\alpha]\}
$$

By running our saturation algorithm and examining the atomic constraints on $\mathsf{list}[\alpha]$ and itself, we can verify that $\mathsf{list}[\alpha]$ is covariant in α; we can similarly verify that $\mathsf{nelist}[\alpha]$ is covariant in α. Moreover, our algorithm confirms the parametric rules $\mathsf{elist} \leqslant \mathsf{list}[\alpha]$, as well as $\mathsf{nelist}[\alpha] \leqslant \mathsf{list}[\beta]$ if $\alpha \leqslant \beta$. And lists of nonempty lists of even natural numbers are, more generally, lists of lists of natural numbers, and our algorithm confirms that $\mathsf{list}[\mathsf{nelist}[\mathsf{even}]] \leqslant \mathsf{list}[\mathsf{list}[\mathsf{nat}]]$, for example.

7.2 Binary trees and spines

7.2.1 Trees. Similarly to lists, the polymorphic type of binary trees of βs can be defined as follows.

$$\mathsf{tree}[\beta] \triangleq +\{\mathsf{leaf}\colon \mathbf{1},\ \mathsf{node}\colon \beta \times (\mathsf{tree}[\beta] \times \mathsf{tree}[\beta])\}$$

Thus, a tree of βs is either leaf with the unit value, or node with a tuple of an element of type β and the left and right subtrees. Our saturation algorithm verifies that $\mathsf{tree}[\beta]$ is covariant in β.

Similar to those for lists, the types of empty and nonempty trees of βs are subtypes of $\mathsf{tree}[\beta]$:

$$\mathsf{etree} \triangleq +\{\mathsf{leaf}\colon \mathbf{1}\} \quad \textit{and} \quad \mathsf{netree}[\beta] \triangleq +\{\mathsf{node}\colon \beta \times (\mathsf{tree}[\beta] \times \mathsf{tree}[\beta])\}\,.$$

7.2.2 Spines. Because the left and right spines of a tree are essentially lists, we might at first expect to have $\mathsf{list}[\tau] \leqslant \mathsf{tree}[\sigma]$ whenever $\tau \leqslant \sigma$. However, even if we were to coordinate the label names across the two types, that subtyping relationship would *not* hold because it would require $\mathsf{list}[\alpha]\langle\tau/\alpha\rangle \leqslant (\mathsf{tree}[\beta] \times \mathsf{tree}[\beta])\langle\sigma/\beta\rangle$ – that is, it would require a (nonempty) sum type to be a subtype of a product type. Indeed, saturation yields $\mathsf{list}[\alpha] \preccurlyeq \mathsf{tree}[\beta]\ \#\ + \Rightarrow \bot$ for that reason.

Instead, we could define a type of left spines as follows; right spines would be symmetric.

$$\mathsf{spine}[\alpha] \triangleq +\{\mathsf{leaf}\colon \mathbf{1},\ \mathsf{node}\colon \alpha \times (\mathsf{spine}[\alpha] \times \mathsf{etree})\}$$

With this definition, we *do* have $\mathsf{spine}[\tau] \leqslant \mathsf{tree}[\sigma]$ when $\tau \leqslant \sigma$ because the product type $\mathsf{spine}[\alpha] \times \mathsf{etree}$ is a subtype of $\mathsf{tree}[\beta] \times \mathsf{tree}[\beta]$ under τ/α and σ/β.

7.2.3 Object-oriented lists and trees. On a related note, we could take a more object-oriented approach to lists and trees, using record types instead of eager products:

$$\begin{aligned} \mathsf{olist}[\alpha] &\triangleq \&\{\mathsf{out}\colon +\{\mathsf{none}\colon \mathbf{1},\ \mathsf{some}\colon \alpha \times \&\{\mathsf{fst}\colon \mathsf{olist}[\alpha]\}\},\ \mathsf{size}\colon \mathsf{nat}\} \\ \mathsf{otree}[\beta] &\triangleq \&\{\mathsf{out}\colon +\{\mathsf{none}\colon \mathbf{1},\ \mathsf{some}\colon \beta \times \&\{\mathsf{fst}\colon \mathsf{otree}[\beta],\ \mathsf{snd}\colon \mathsf{otree}[\beta]\}\},\ \mathsf{size}\colon \mathsf{nat}\} \end{aligned}$$

Even purely structurally, $\mathsf{olist}[\tau] \leqslant \mathsf{otree}[\sigma]$ does not hold when $\tau \leqslant \sigma$, but $\mathsf{otree}[\sigma] \leqslant \mathsf{olist}[\tau]$ does when $\sigma \leqslant \tau$ and is in the parametric fragment. This is somewhat counterintuitive, but nevertheless the correct relationship: any context that expects a list can use a tree's spine instead.

7.2.4 Perfect binary trees. Taking advantage of the support for nested types, we can adapt Bird and Meertens's prototypical example of perfect binary trees [1998].

$$\mathsf{perfect}[\alpha] \triangleq +\{\mathsf{leaf}\colon \mathbf{1},\ \mathsf{node}\colon \alpha \times \mathsf{perfect}[\alpha \times \alpha]\}$$

Our algorithm confirms that $\mathsf{perfect}[\alpha]$ is covariant in α. However, even purely structurally, $\mathsf{perfect}[\tau]$ is *not* a subtype of $\mathsf{tree}[\sigma]$ for any τ and σ. That would require $\mathsf{perfect}[\tau \times \tau]$ to be a subtype of $\mathsf{tree}[\sigma] \times \mathsf{tree}[\sigma]$, which cannot be: $\mathsf{perfect}[\tau \times \tau]$ is a variant record type, whereas $\mathsf{tree}[\sigma] \times \mathsf{tree}[\sigma]$ is a product type. Essentially, the difference amounts to one between breadth-first and depth-first representations of trees. However, the lack of a subtyping relationship does not mean that the type $\mathsf{perfect}[\alpha]$ is unusable: given the support for nested types, an explicit coercion from $\mathsf{perfect}[\alpha]$ to $\mathsf{tree}[\alpha]$ could still be written.

7.3 Serialized binary trees and spines

7.3.1 Serialized binary trees. Here we adapt an example from Thiemann and Vasconcelos [2016] and consider it in the context of subtyping: We may sometimes wish to serialize a binary tree to send it across the network or write it to a file. A type that describes serialized trees is $\mathsf{stree}[\alpha, \kappa]$, parameterized by both the type of data elements, α, and a suffix (or continuation) type, κ:

$$\mathsf{stree}[\alpha, \kappa] \triangleq +\{\mathsf{leaf}\colon \kappa,\ \mathsf{node}\colon \alpha \times \mathsf{stree}[\alpha, \mathsf{stree}[\alpha, \kappa]]\}\,.$$

According to this type, a serialized tree is a list of leaf and node labels. A leaf is followed by a suffix of type κ; a node is followed by the pair of the tree's root element of type α and the serialization of the left subtree, which itself is followed by the serialization of the right subtree.[7] This type crucially depends on nested types to express the invariant that $\mathsf{stree}[\alpha,\kappa]$ describes preorder traversals of binary trees. Our saturation algorithm verifies that $\mathsf{stree}[\alpha,\kappa]$ is covariant in both α and κ.

Although unrelated to subtyping concerns, it is interesting to observe that the above type definition can, in fact, be *derived* syntactically by repeatedly applying type isomorphisms to the definition $\mathsf{stree}[\alpha,\kappa] \triangleq \mathsf{tree}[\alpha] \times \kappa$:

$$\begin{aligned}\mathsf{stree}[\alpha,\kappa] &\triangleq \mathsf{tree}[\alpha] \times \kappa = {+}\{\mathsf{leaf}\colon \mathbf{1},\ \mathsf{node}\colon \alpha \times (\mathsf{tree}[\alpha] \times \mathsf{tree}[\alpha])\} \times \kappa \\ &\simeq {+}\{\mathsf{leaf}\colon \mathbf{1} \times \kappa,\ \mathsf{node}\colon \alpha \times (\mathsf{tree}[\alpha] \times (\mathsf{tree}[\alpha] \times \kappa))\} \\ &\simeq {+}\{\mathsf{leaf}\colon \kappa,\ \mathsf{node}\colon \alpha \times (\mathsf{tree}[\alpha] \times \mathsf{stree}[\alpha,\kappa])\} \\ &\simeq {+}\{\mathsf{leaf}\colon \kappa,\ \mathsf{node}\colon \alpha \times \mathsf{stree}[\alpha, \mathsf{stree}[\alpha,\kappa]]\}\,.\end{aligned}$$

None of the isomorphic types in this sequence are mutual subtypes, however. In particular, although the types $\mathsf{stree}[\alpha,\kappa]$ and $\mathsf{tree}[\alpha] \times \kappa$ are isomorphic, we do *not* have $\mathsf{tree}[\tau]$ as a subtype of $\mathsf{stree}[\tau,\mathbf{1}]$, nor vice versa, for any τ. Comparing the leaf branches of both types, we see that, for these parametric subtyping relationships to hold, $\mathbf{1} \leqslant \kappa$ and $\kappa \leqslant \mathbf{1}$ must hold for *all* types κ, regardless of the fact that $\mathsf{stree}[\tau,\mathbf{1}]$ ultimately instantiates κ with $\mathbf{1}$. This is simply not true when κ is, for example, either ${+}\{\,\}$ or $\mathbf{1} \times \mathbf{1}$.

Once again, the absence of subtyping relationships does not mean that the type $\mathsf{stree}[\alpha,\kappa]$ cannot be related to $\mathsf{tree}[\alpha] \times \kappa$. Given a term language with support for nested types, it would still be possible to write explicit coercions between these types to serialize and deserialize trees.

7.3.2 Serialized spines. As an extension of this example, we can also define a type constructor $\mathsf{sspine}[\alpha,\kappa]$ to describe serialized (left) spines:

$$\begin{aligned}\mathsf{sspine}[\alpha,\kappa] &\triangleq \mathsf{spine}[\alpha] \times \kappa \simeq {+}\{\mathsf{leaf}\colon \kappa,\ \mathsf{node}\colon \alpha \times \mathsf{sspine}[\alpha, \mathsf{setree}[\kappa]]\} \\ \mathsf{setree}[\kappa] &\triangleq \qquad \mathsf{etree} \times \kappa \simeq {+}\{\mathsf{leaf}\colon \kappa\}\,.\end{aligned}$$

The subtyping relationship between (left) spines and trees is preserved under serialization: we have the parametric rule $\mathsf{sspine}[\alpha,\kappa] \leqslant \mathsf{stree}[\beta,\kappa']$ if both $\alpha \leqslant \beta$ and $\kappa \leqslant \kappa'$, as our algorithm confirms. Notice that the inclusion of $\mathsf{setree}[\kappa]$ and its ${+}\{\mathsf{leaf}\colon \kappa\}$ is essential here. Had we instead used the definition $\mathsf{sspine}[\alpha,\kappa] \triangleq {+}\{\mathsf{leaf}\colon \kappa,\ \mathsf{node}\colon \alpha \times \mathsf{sspine}[\alpha,\kappa]\}$, there would be no subtyping relationship because $\mathsf{stree}[\beta,\kappa']$ would have one more $\times$ than $\mathsf{sspine}[\alpha,\kappa]$.

7.4 Total functions and generalized tries on binary trees and spines

7.4.1 Total functions on trees. We can use the following type to describe *total* functions from α trees to βs. As for serialized trees, this definition is derivable by repeatedly applying type isomorphisms to $\mathsf{tree}[\alpha] \to \beta$. (Interestingly, this type is, in a sense, dual to that of serialized trees.)

$$\begin{aligned}\mathsf{treefn}[\alpha,\beta] &\triangleq \mathsf{tree}[\alpha] \to \beta = {+}\{\mathsf{leaf}\colon \mathbf{1},\ \mathsf{node}\colon \alpha \times (\mathsf{tree}[\alpha] \times \mathsf{tree}[\alpha])\} \to \beta \\ &\simeq \&\{\mathsf{leaf}\colon \mathbf{1} \to \beta,\ \mathsf{node}\colon \alpha \times (\mathsf{tree}[\alpha] \times \mathsf{tree}[\alpha]) \to \beta\} \\ &\simeq \&\{\mathsf{leaf}\colon \beta,\ \mathsf{node}\colon \alpha \to (\mathsf{tree}[\alpha] \to (\mathsf{tree}[\alpha] \to \beta))\} \\ &\simeq \&\{\mathsf{leaf}\colon \beta,\ \mathsf{node}\colon \alpha \to (\mathsf{tree}[\alpha] \to \mathsf{treefn}[\alpha,\beta])\} \\ &\simeq \&\{\mathsf{leaf}\colon \beta,\ \mathsf{node}\colon \alpha \to \mathsf{treefn}[\alpha, \mathsf{treefn}[\alpha,\beta]]\}\end{aligned}$$

[7] Strictly speaking, this is not a true serialization due to the inclusion of a product type. However, as there is no uniform way of serializing polymorphic data, this is as near to a true serialization as is possible. Moreover, for concrete instances of tree, such as with nat data, it is possible to give true serializations that inline the serialization of their data elements: $\mathsf{snattree}[\kappa] \triangleq {+}\{\mathsf{leaf}\colon \kappa,\ \mathsf{node}\colon \mathsf{snat}[\mathsf{snattree}[\mathsf{snattree}[\kappa]]]\}$, where $\mathsf{snat}[-]$ is defined as in Section 3.1.

Thus, an object of type $\mathsf{treefn}[\alpha,\beta]$ offers two methods, leaf and node. To look up the value of a leaf, the leaf method is invoked, resulting in the associated value of type β. To look up a nonempty tree, the node method is invoked with the root element of type α, resulting in an object of type $\mathsf{treefn}[\alpha,\mathsf{treefn}[\alpha,\beta]]$. Recursively, the left subtree is looked up in this object; its associated value is an object of type $\mathsf{treefn}[\alpha,\beta]$. Then, the right subtree is looked up in this object, and the value of type β associated with the entire nonempty tree is ultimately returned.

This example makes essential use of a record type to represent the object's methods, but most importantly, nested types are crucial to expressing the higher-order nature of lookups. In the usual way, our algorithm confirms that $\mathsf{treefn}[\alpha,\beta]$ is contravariant in α and covariant in β.

7.4.2 Total functions on spines. In a similar way, we can derive a type definition for total functions on (left) spines, using the types $\mathsf{spine}[\alpha]$ and etree defined in Section 7.2.2.

$$\begin{aligned}\mathsf{spinefn}[\alpha,\beta] &\triangleq \mathsf{spine}[\alpha] \to \beta \simeq \&\{\mathsf{leaf}\colon \beta,\ \mathsf{node}\colon \alpha \to \mathsf{spinefn}[\alpha,\mathsf{etreefn}[\beta]]\}\\ \mathsf{etreefn}[\beta] &\triangleq \qquad \mathsf{etree} \to \beta \simeq \&\{\mathsf{leaf}\colon \beta\}\end{aligned}$$

Once again, the subtyping relationship between (left) spines and trees is respected: we have the rule $\mathsf{treefn}[\alpha,\beta] \leqslant \mathsf{spinefn}[\alpha',\beta']$ if both $\alpha' \leqslant \alpha$ and $\beta \leqslant \beta'$, as our algorithm confirms. (Notice that the subtyping direction is reversed because of contravariance in the underlying function types.)

7.4.3 Tries for trees and spines. In prior work [Connelly and Morris 1995; Hinze 2000; Wadsworth 1979], the trie data structure for lists and strings was generalized to represent partial (not total) functions on more complex algebraic structures, such as binary trees. A type definition for tries from keys of type $\mathsf{tree}[\alpha]$ to values of type β can be derived from $\mathsf{trie}[\alpha,\beta] \triangleq \mathsf{tree}[\alpha] \to \mathsf{option}[\beta]$, where $\mathsf{option}[\beta] \triangleq +\{\mathsf{none}\colon \mathbf{1},\ \mathsf{some}\colon \beta\}$.

$$\begin{aligned}\mathsf{trie}[\alpha,\beta] \triangleq \mathsf{tree}[\alpha] \to \mathsf{option}[\beta] &= +\{\mathsf{leaf}\colon \mathbf{1},\ \mathsf{node}\colon \alpha \times (\mathsf{tree}[\alpha] \times \mathsf{tree}[\alpha])\} \to \mathsf{option}[\beta]\\ &\simeq \&\{\mathsf{leaf}\colon \mathsf{option}[\beta],\ \mathsf{node}\colon \alpha \to (\mathsf{tree}[\alpha] \to (\mathsf{tree}[\alpha] \to \mathsf{option}[\beta]))\}\\ &\simeq \&\{\mathsf{leaf}\colon \mathsf{option}[\beta],\ \mathsf{node}\colon \alpha \to (\mathsf{tree}[\alpha] \to \mathsf{trie}[\alpha,\beta])\}\\ &\simeq \&\{\mathsf{leaf}\colon \mathsf{option}[\beta],\ \mathsf{node}\colon \alpha \to \mathsf{treefn}[\alpha,\mathsf{trie}[\alpha,\beta]]\}\end{aligned}$$

For example, tries representing sets of τ trees could be typed as $\mathsf{trie}[\tau,\mathbf{1}]$. Our algorithm verifies that $\mathsf{trie}[\alpha,\beta]$ is contravariant in α and covariant in β. The use of nested types here is consistent with Wadsworth's observation [1979]. It would also be possible to similarly define a type $\mathsf{spinetrie}[\alpha,\beta]$ of tries for (left) spines; it would be equivalent to $\mathsf{spinefn}[\alpha,\mathsf{option}[\beta]]$.

7.5 Refined stacks

The additional expressive power of nested types allows us to also define a refined type for stacks that tracks the stack's shape:

$$\mathsf{rstack}[\alpha,\kappa] \triangleq \&\{\mathsf{push}\colon \alpha \to \mathsf{rstack}[\alpha,\ \mathsf{some}[\alpha \times \mathsf{rstack}[\alpha,\kappa]]],\ \mathsf{pop}\colon \kappa\}\,,$$

where $\mathsf{some}[\alpha] \triangleq +\{\mathsf{some}\colon \alpha\}$. Here, the type parameter κ serves as a continuation to be used when popping from the stack. Pushing an element onto the stack extends this continuation to reflect the existence (but not the identity) of the newly pushed element.

But we do not have $\mathsf{rstack}[\tau,\sigma]$ as a *parametric* subtype of $\mathsf{stack}[\tau]$ for any τ, even when we are guaranteed that $\sigma \leqslant \mathsf{option}[\tau \times \mathsf{stack}[\tau]]$. Nevertheless, it is possible to prove that $\mathsf{rstack}[\tau,\sigma]$ is a *structural* subtype of $\mathsf{stack}[\tau]$ when $\sigma \leqslant \mathsf{option}[\tau \times \mathsf{stack}[\tau]]$, making this one example of how the parametric fragment is a proper fragment of structural subtyping.

7.6 Abstract types

We can take advantage of the $\forall$ and $\exists$ quantifiers to express module signatures as abstract types. Here we define two signatures for abstract lists, with constructors x and $\alpha \times x \to x$ (respectively, $\beta \times x \to x$) for the empty list and list concatenation, parameterized by the type α (respectively, β) of list elements. Both signatures include a fold function, and $\mathsf{alist}'[\beta]$ also includes a size function.

$$\begin{aligned} \mathsf{alist}[\alpha] &\triangleq \exists x.\, x \times (\alpha \times x \to x) \times \&\{\mathsf{fold}\colon \forall z.\, z \to (\alpha \times z \to z) \to x \to z\} \\ \mathsf{alist}'[\beta] &\triangleq \exists x.\, x \times (\beta \times x \to x) \times \&\{\mathsf{fold}\colon \forall z.\, z \to (\beta \times z \to z) \to x \to z,\ \mathsf{size}\colon \mathsf{nat}\} \end{aligned}$$

Our algorithm confirms the signature subtyping relationship that derives from the additional size function: $\mathsf{alist}'[\beta] \leqslant \mathsf{alist}[\alpha]$ if both $\alpha \leqslant \beta$ and $\beta \leqslant \alpha$.

8 RELATED WORK

The most closely related work has been mentioned; we now briefly overview other related work.

Subtyping recursive types. The nature and complexity of the subtyping problem vary considerably depending on whether types are interpreted *nominally* or *structurally*; for type equivalence, the change from structural to nominal interpretation in *non-regular* types leads to a decrease in complexity from doubly-exponential to linear [Mordido et al. 2023]. If recursive types are nominal, the subtyping problem is simpler but also very limited. However, even under a nominal interpretation of how types are defined, issues arise when datasort refinements are considered [Davies 2005; Dunfield and Pfenning 2004; Freeman and Pfenning 1991].

Although we have a broader setting, our interest in structural subtyping was influenced by session types [Caires and Pfenning 2010; Honda et al. 1998], where types are traditionally interpreted structurally, equirecursively, and coinductively. Subtyping in session types has been mostly treated coinductively [Gay and Hole 2005; Silva et al. 2023] and constitutes a particular case of our system.

Developments toward structural subtyping began much earlier, regardless of whether types were treated coinductively [Amadio and Cardelli 1993; Gay and Hole 2005; Hosoya et al. 1998] or in a mixed inductive/coinductive setting [Brandt and Henglein 1998; Danielsson and Altenkirch 2010; Lakhani et al. 2022; Ligatti et al. 2017]. In fact, the notion of structural subtyping dates back to 1988, introduced by Cardelli [1988], but suggested even before [Cardelli 1984, 1985; Reynolds 1985]. In this paper, we have not explored the presence of empty or full types [Lakhani et al. 2022; Ligatti et al. 2017]; we leave this analysis for future work (more details are provided in Section 4.3).

Subtyping polymorphic types. In this paper, we chose to have a foundational approach, including the features strictly necessary to handle parametric datatypes in programming languages. Bird and Meertens [1998] and Hinze [2000] noted that implementing generalized tries required the use of nested datatypes and non-regular recursion, which is our setting for this paper. Other type systems have been developed to explore *non-regular* data structures. In the context of session types, Thiemann and Vasconcelos [2016] proposed context-free session types with predicative polymorphism, extended later with impredicative polymorphism [Almeida et al. 2022]; subtyping was also explored [Silva et al. 2023]. Nested session types were proposed by Das et al. [2022] and were proved to be more expressive than context-free session types [Das et al. 2022; Gay et al. 2022].

Inspired by the structural nature of types, these works have focused on structural subtyping and equivalence relations. For session types, the subtyping problem has been shown to be undecidable [Padovani 2019; Silva et al. 2023], even though the corresponding type equality problems are decidable [Almeida et al. 2020; Das et al. 2022; Solomon 1978]. In Section 2.3, we present the result more generally for type systems with record types, explicitly identifying sets of minimal features that guarantee the undecidability of subtyping. The undecidability of the subtyping relation leads to the design of incomplete algorithms. The *parametric subtyping* relation we propose allows us to both tackle the incompleteness problem and to understand exactly to what extent the previous relations are incomplete, distinguishing cases where parametricity is not satisfied from cases where types actually exhibit distinct behaviors and are therefore not in (any) subtyping relation. Parametricity materializes the idea that types behave uniformly for all possible instantiations. This notion was first proposed by Reynolds [1983] for System F [Girard 1972], further explored by Wadler [1989] and then extended to nested types by Johann and Ghiorzi [2021]. None of these works focused on the subtyping relation. The combination of parametricity and subtyping is the main contribution of our work through type constructors that map (subtyping-)related arguments to (subtyping-)related results, in a relation that we called *parametric subtyping*.

Several works focus on mixing subtyping with (explicit) parametric polymorphism via *bounded quantification* [Cardelli and Wegner 1985]. The most standard formulation is the second-order lambda-calculus with bounded quantification, $F_{\leq}$ [Cardelli et al. 1994], but subtyping was proved to be undecidable [Pierce 1994], even without recursion. Several $F_{\leq}$ fragments have been identified as having a decidable subtyping relation [Cardelli and Wegner 1985; Castagna and Pierce 1994; Katiyar and Sankar 1992; Mackay et al. 2020], also extended with recursion [Abadi et al. 1996; Zhou et al. 2023] or higher-order polymorphism and polarized application [Steffen 1999]. System $F_{\leq}$ was the ground for many developments in OOP [Rompf and Amin 2016]. As none of the applications we want for our type system seem to require bounded polymorphism at its most fundamental core, we limit our setting to parametric polymorphism and explicit quantifiers.

Semantic vs *algorithmic subtyping definitions.* Semantic typing and subtyping are favored in non-regular structural type systems over their declarative versions. Initially, semantic relations were motivated by the set-theoretic properties of types [Castagna and Frisch 2005; Frisch et al. 2002; Lakhani et al. 2022], but for non-regular types the need for a semantic relation to model the behaviour of types was even more natural, by means of simulations and bisimulations [Das et al. 2022; Gay and Hole 2005; Silva et al. 2023].

Algorithmic approaches for *monomorphic* or *regular* (sub)typing systems, such as standard fixed-point algorithms [Gay and Hole 2005], sequent calculus [Das et al. 2022], cyclic proofs [Brotherston and Simpson 2010; Lakhani et al. 2022], step indexing [Ahmed 2004, 2006; Appel and McAllester 2001; Dreyer et al. 2009; Lakhani et al. 2022], sized types [Abel and Pientka 2016] or bouncing threads [Baelde et al. 2022], are effective. However, these mechanisms are not scalable when we navigate beyond regular types. In section 2.2.3, we illustrate the narrow scope of the above approaches and the limitations of their application to nested types. To limit the recursion depth, a recursion bound is usually used, leading to incompleteness [Das et al. 2022] or even unsoundness[8]. The undecidability of structural subtyping for the non-monomorphic fragment gives us no hope of finding a sound and complete algorithm. In this paper, we free our type system from this limitation by proposing the novel notion of *parametric subtyping*, which takes advantage of parametricity without completely abandoning structural subtyping. In an attempt to overcome the limitations of alternatives such as bouncing threads or cyclic proofs, we end up finding a sweet spot that takes advantage of saturation-based methods to perform forward-inference, often used in constraint solving [Jaffar and Lassez 1987] and unification [Huet 1976; Martelli and Montanari 1982].

9 CONCLUSION

In this paper, we presented a theory of parametricity for type constructors that forms the basis for parametric subtyping, a decidable, practical, and expressive fragment of structural subtyping for

[8]TypeScript has an *unsound* structural subtyping: https://www.typescriptlang.org/docs/handbook/type-compatibility.html

parametic polymorphism. Moreover, the saturation-based decision procedure has led to an effective implementation that performs well on a variety of practical examples.

One opportunity for future work is to extend our results to a mixed inductive/coinductive type system, as in call-by-push-value [Levy 2001], that accounts for subtypings that rely on types being uninhabited or full, such as $+\{\} \leqslant \sigma$ and $\tau \leqslant +\{\} \rightarrow \sigma$ for all τ and σ. We do already have a prototype implementation that does so, but the decision procedure's theory and accompanying correctness proofs appear to be more complicated. Other opportunities include extension to bounded quantification, integration with intersection and union types, and development of a notion of parametric subtyping for higher-order kinds.

REFERENCES

Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. 1996. An Interpretation of Objects and Object Types *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 396–409. https://doi.org/10.1145/237721.237809

Andreas Abel and Brigitte Pientka. 2016. Well-Founded Recursion with Copatterns and Sized Types. *Journal of Functional Programming* 26 (2016), e2. https://doi.org/10.1017/S0956796816000022

Amal J. Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University. http://www.ccs.neu.edu/home/amal/ahmedsthesis.pdf AAI3136691.

Amal J. Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *15th European Symposium on Programming (ESOP 2006)*, P. Sestoft (Ed.). Springer LNCS 3924, Vienna, Austria, 69–83. https://doi.org/10.1007/11693024_6

Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. 2022. Polymorphic lambda calculus with context-free session types. *Information and Computation* 289 (2022), 104948. https://doi.org/10.1016/j.ic.2022.104948

Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. 2020. Deciding the Bisimilarity of Context-Free Session Types. In *Tools and Algorithms for the Construction and Analysis of Systems*, Armin Biere and David Parker (Eds.). Springer International Publishing, Cham, 39–56.

Roberto M. Amadio and Luca Cardelli. 1993. Subtyping Recursive Types. *ACM Transactions on Programming Languages and Systems* 15, 4 (1993), 575–631.

Andrew W. Appel and David A. McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *Transactions on Programming Languages and Systems* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

David Baelde, Amina Doumane, Denis Kuperberg, and Alexis Saurin. 2022. Bouncing Threads for Circular and Non-Wellfounded Proofs: Towards Compositionality with Circular Proofs. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science* (Haifa, Israel) *(LICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 63, 13 pages. https://doi.org/10.1145/3531130.3533375

J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. 1993. Decidability of Bisimulation Equivalence for Processes Generating Context-Free Languages. *J. ACM* 40, 3 (July 1993), 653–682. https://doi.org/10.1145/174130.174141

J.A. Bergstra and J.W. Klop. 1984. Process Algebra for Synchronous Communication. *Information and Control* 60, 1 (1984), 109–137. https://doi.org/10.1016/S0019-9958(84)80025-X

Richard Bird and Lambert Meertens. 1998. Nested Datatypes. In *Mathematics of Program Construction*, Johan Jeuring (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–67.

Michael Brandt and Fritz Henglein. 1998. Coinductive Axiomatization of Recursive Type Equality and Subtyping. *Fundamenta Informaticae* 33, 4 (1998), 309–338.

James Brotherston and Alex Simpson. 2010. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation* 21, 6 (10 2010), 1177–1216. https://doi.org/10.1093/logcom/exq052 arXiv:https://academic.oup.com/logcom/article-pdf/21/6/1177/2787531/exq052.pdf

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, P.Gastin and F.Laroussinie (Eds.). Springer LNCS 6269, Paris, France, 222–236.

Luca Cardelli. 1984. A semantics of multiple inheritance. In *International Symposium on Semantics of Data Types*. Springer, 51–67.

Luca Cardelli. 1985. Amber. In *LITP Spring School on Theoretical Computer Science*. Springer, 21–47.

Luca Cardelli. 1988. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 70–79.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1994. An extension of system F with subtyping. *Information and Computation* 109, 1–2 (1994), 4–56.

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)* 17, 4 (1985), 471–523.

Giuseppe Castagna and Alain Frisch. 2005. A gentle introduction to semantic subtyping. In *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal.* ACM, 198–199. https://doi.org/10.1145/1069774.1069793

Giuseppe Castagna and Benjamin C. Pierce. 1994. Decidable Bounded Quantification. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 151–162. https://doi.org/10.1145/174675.177844

Richard H. Connelly and F. Lockwood Morris. 1995. A Generalization of the Trie Data Structure. *Mathematical Structures in Computer Science* 5, 3 (1995), 381–418. https://doi.org/10.1017/S0960129500000803

Nils Anders Danielsson and Thorsten Altenkirch. 2010. Subtyping, Declaratively. In *Mathematics of Program Construction*, Claude Bolduc, Jules Desharnais, and Béchir Ktari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 100–118. https://doi.org/10.1007/978-3-642-13321-3_8

Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. 2022. Nested Session Types. *ACM Trans. Program. Lang. Syst.* 44, 3, Article 19 (July 2022), 45 pages. https://doi.org/10.1145/3539656

Rowan Davies. 2005. *Practical Refinement-Types Checking.* Ph.D. Dissertation. Carnegie Mellon University. Available as Technical Report CMU-CS-05-110.

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical Step-Indexed Logical Relations. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA.* IEEE Computer Society, 71–80. https://doi.org/10.1109/LICS.2009.34

Jana Dunfield and Frank Pfenning. 2004. Tridirectional Typechecking. In *Conference Record of the 31st Annual Symposium on Principles of Programming Languages (POPL'04)*, X.Leroy (Ed.). ACM Press, Venice, Italy, 281–292. Extended version available as Technical Report CMU-CS-04-117, March 2004.

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation.* ACM Press, Toronto, Ontario, 268–277.

Emily P. Friedman. 1976. The Inclusion Problem for Simple Languages. *Theoretical Computer Science* 1, 4 (1976), 297–316. https://doi.org/10.1016/0304-3975(76)90074-8

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic Subtyping. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings.* IEEE Computer Society, 137–146. https://doi.org/10.1109/LICS.2002.1029823

Simon J. Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2–3 (2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

Simon J. Gay, Diogo Poças, and Vasco T. Vasconcelos. 2022. The Different Shades of Infinite Session Types. In *Foundations of Software Science and Computation Structures*, Patricia Bouyer and Lutz Schröder (Eds.). Springer International Publishing, Cham, 347–367.

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. Dissertation. Éditeur inconnu.

Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-bounded polymorphism into shape. *ACM SIGPLAN Notices* 49, 6 (2014), 89–99.

Sheila A. Greibach. 1965. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM* 12, 1 (Jan. 1965), 42–52. https://doi.org/10.1145/321250.321254

Radu Grigore. 2017. Java generics are turing complete. *ACM SIGPLAN Notices* 52, 1 (2017), 73–85.

Jan Friso Groote and Hans Hüttel. 1994. Undecidable Equivalences for Basic Process Algebra. *Information and Computation* 115, 2 (Dec. 1994), 354–371. https://doi.org/10.1006/inco.1994.1101

Ralf Hinze. 2000. Generalizing Generalized Tries. *J. Funct. Program.* 10, 4 (July 2000), 327–351. https://doi.org/10.1017/S0956796800003713

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138.

Haruo Hosoya, Benjamin C. Pierce, and David N. Turner. 1998. Datatypes and Subtyping. *Unpublished manuscript* (1998).

Gérard Huet. 1976. *Resolution d'Equations dans des Langages d'Order 1, 2., ω.* Ph.D. Dissertation. Universite de Paris VII.

Gérard Huet. 1998. Regular Böhm Trees. *Mathematical Structures in Computer Science* 8, 6 (1998), 671–680. https://doi.org/10.1017/S0960129598002643

Joxan Jaffar and J-L Lassez. 1987. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming languages.* 111–119.

Petr Jančar. 2021. Equivalence of Pushdown Automata via First-Order Grammars. *J. Comput. System Sci.* 115 (2021), 86–112. https://doi.org/10.1016/j.jcss.2020.07.004

Patricia Johann and Enrico Ghiorzi. 2021. Parametricity for Primitive Nested Types and GADTs. *Logical Methods in Computer Science* 17, 4 (2021).

Dinesh Katiyar and Sriram Sankar. 1992. Completely bounded quantification is decidable. In *ACM SIGPLAN Workshop on ML and its Applications*. Citeseer.

Andrew J Kennedy and Benjamin C Pierce. 2006. On decidability of nominal subtyping with variance. (2006).

A. J. Korenjak and J. E. Hopcroft. 1966. Simple Deterministic Languages. In *7th Annual Symposium on Switching and Automata Theory*. IEEE Computer Society, Los Alamitos, CA, USA, 36–46. https://doi.org/10.1109/SWAT.1966.22

Zeeshan Lakhani, Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. 2022. Polarized Subtyping. In *Programming Languages and Systems (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 431–461. https://doi.org/10.1007/978-3-030-99336-8_16

Paul Blain Levy. 2001. *Call-By-Push-Value*. Ph.D. Dissertation. University of London.

Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. 2017. On Subtyping-Relation Completeness, with an Application to Iso-Recursive Types. *ACM Transactions on Programming Languages and Systems* 39, 4 (March 2017), 4:1–4:36.

Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. 2020. Syntactically Restricting Bounded Polymorphism for Decidable Subtyping. In *Programming Languages and Systems: 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30–December 2, 2020, Proceedings 18*. Springer, 125–144.

Alberto Martelli and Ugo Montanari. 1982. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems* 4, 2 (1982), 258–282.

Andreia Mordido, Janek Spaderna, Peter Thiemann, and Vasco T. Vasconcelos. 2023. Parameterized Algebraic Protocols. 7, PLDI, Article 163 (June 2023), 25 pages. https://doi.org/10.1145/3591277

Alan Mycroft. 1984. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, B. Paul, M.and Robinet (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 217–228.

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. Association for Computing Machinery, New York, NY, USA, 54–67. https://doi.org/10.1145/237721.237729

Luca Padovani. 2019. Context-Free Session Type Inference. *ACM Trans. Program. Lang. Syst.* 41, 2, Article 9 (March 2019), 37 pages. https://doi.org/10.1145/3229062

Benjamin C. Pierce. 1994. Bounded Quantification Is Undecidable. *Information and Computation* 112, 1 (1994), 131–165. https://doi.org/10.1006/inco.1994.1055

Benjamin C. Pierce. 2002. *Types and Programming Languages*. The MIT Press.

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

John C Reynolds. 1985. Three approaches to type structure. In *Mathematical Foundations of Software Development: Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT) Berlin, March 25–29, 1985 Volume 1: Colloquium on Trees in Algebra and Programming (CAAP'85) 10*. Springer, 97–138.

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 624–641.

Géraud Sénizergues. 2001. L(A)=L(B)? Decidability Results from Complete Formal Systems. *Theoretical Computer Science* 251, 1 (2001), 1–166. https://doi.org/10.1016/S0304-3975(00)00285-1

Gil Silva, Andreia Mordido, and Vasco T. Vasconcelos. 2023. Subtyping Context-Free Session Types. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2023)*. To appear.

Christian Skalka. 1997. *Some Decision Problems for ML Refinement Types*. Master's thesis. Department of Philosophy, Carnegie Mellon University.

Marvin H. Solomon. 1978. Type Definitions with Parameters. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 31–38. https://doi.org/10.1145/512760.512765

Martin Steffen. 1999. *Polarized higher-order subtyping*. Ph.D. Dissertation. University of Erlangen-Nuremberg, Germany.

Colin Stirling. 2001a. Decidability of DPDA Equivalence. *Theoretical Computer Science* 255, 1 (2001), 1–31. https://doi.org/10.1016/S0304-3975(00)00389-3

Colin Stirling. 2001b. An Introduction to Decidability of DPDA Equivalence. In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, Ramesh Hariharan, V. Vinay, and Madhavan Mukund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 42–56.

Peter Thiemann and Vasco T. Vasconcelos. 2016. Context-Free Session Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 462–475. https://doi.org/10.1145/2951913.2951926

Philip Wadler. 1989. Theorems for free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. 347–359.

C. P. Wadsworth. 1979. Recursive Type Operators Which Are More Than Type Schemes. *Bulletin of the EATCS* 8 (1979), 87–88.

Litao Zhou, Yaoda Zhou, and Bruno C. d. S. Oliveira. 2023. Recursive Subtyping for All. *Proc. ACM Program. Lang.* 7, POPL, Article 48 (jan 2023), 30 pages. https://doi.org/10.1145/3571241

A PROOFS

Theorem 2.1. *Let $t \triangleq \&\{\,\}$. Then $p \lesssim q$ if and only if $(q \mathbin{⨟} t) \leqslant (p \mathbin{⨟} t)$, for all processes p and q.*

Proof. We will prove each direction separately, beginning with the left-to-right direction. Its proof is by coinduction on the (potentially infinite) derivation of $(q \mathbin{⨟} t) \leqslant (p \mathbin{⨟} t)$. We distinguish cases on the structure of p.

Case: Consider the case in which $p = X \cdot p_2$, where $X \triangleq \sum_{\ell \in L}(\ell \cdot p'_\ell)$ for some nonempty label set L and some processes $(p'_\ell)_{\ell \in L}$ and p_2. Observe that $p \xrightarrow{\ell} p'_\ell \cdot p_2$ for all $\ell \in L$. Because L is nonempty and p is simulated by q, it follows that q can make at least one transition. Therefore, $q = Y \cdot q_2$, where $Y \triangleq \sum_{k \in K}(k \cdot q'_k)$ for some nonempty K and some processes $(q'_k)_{k \in K}$ and q_2. The derivation of $(q \mathbin{⨟} t) \leqslant (p \mathbin{⨟} t)$ may therefore begin with

$$\dfrac{\dfrac{(L \subseteq K) \quad \forall \ell \in L\colon (q'_\ell \mathbin{⨟} (q_2 \mathbin{⨟} t)) \leqslant (p'_\ell \mathbin{⨟} (p_2 \mathbin{⨟} t))}{\&\{k\colon q'_k \mathbin{⨟} (q_2 \mathbin{⨟} t)\}_{k \in K} \leqslant \&\{\ell\colon p'_\ell \mathbin{⨟} (p_2 \mathbin{⨟} t)\}_{\ell \in L}}\,\&}{t_Y[q_2 \mathbin{⨟} t] \leqslant t_X[p_2 \mathbin{⨟} t]}$$

Now we must establish the premises of the above $\&$ rule.

Because p is simulated by q, it follows that $L \subseteq K$ and $p'_\ell \cdot p_2 \lesssim q'_\ell \cdot q_2$ for all $\ell \in L$. Appealing to the coinductive hypothesis, we have $((q'_\ell \cdot q_2) \mathbin{⨟} t) \leqslant ((p'_\ell \cdot p_2) \mathbin{⨟} t)$ for all $\ell \in L$. These appeals are justified because they are guarded by the ? and $\&$ rules. Because $((q'_\ell \cdot q_2) \mathbin{⨟} t) = (q'_\ell \mathbin{⨟} (q_2 \mathbin{⨟} t))$ and $((q'_\ell \cdot q_2) \mathbin{⨟} t) = (q'_\ell \mathbin{⨟} (q_2 \mathbin{⨟} t))$ for all $\ell \in L$, this completes the required derivation of $(q \mathbin{⨟} t) \leqslant (p \mathbin{⨟} t)$.

Case: Consider the case in which $p = \epsilon$. In this case, $(p \mathbin{⨟} t) = t$ and we must show that $(q \mathbin{⨟} t) \leqslant t$. If $q = Y \cdot q_2$ for some $Y \triangleq \sum_{k \in K}(k \cdot q'_k)$, nonempty K, and processes $(q'_k)_{k \in K}$ and q_2, then the following is a derivation of $(q \mathbin{⨟} t) \leqslant t$.

$$\dfrac{\dfrac{}{\&\{k\colon q'_k \mathbin{⨟} (q_2 \mathbin{⨟} t)\}_{k \in K} \leqslant \&\{\,\}}\,\&}{t_Y[q_2 \mathbin{⨟} t] \leqslant t}$$

Otherwise, if $q = \epsilon$, then the following is a derivation of $(q \mathbin{⨟} t) \leqslant t$.

$$\dfrac{\dfrac{}{\&\{\,\} \leqslant \&\{\,\}}\,\&}{t \leqslant t}$$

Next, we will prove the right-to-left direction; the proof is by coinduction on the similarity $p \lesssim q$. It suffices to show that $p \xrightarrow{a} p'$ implies $q \xrightarrow{a}\gtrsim p'$ when $(q \mathbin{⨟} t) \leqslant (p \mathbin{⨟} t)$. Assume that $p \xrightarrow{a} p'$. Because p has a transition, there must exist an equation $X \triangleq \sum_{\ell \in L}(\ell \cdot p'_\ell)$ and a process p_2 such that $p = X \cdot p_2$ and $p' = p'_a \cdot p_2$, with $a \in L$. We will distinguish cases on the structure of q.

Case: Consider the case in which $q = Y \cdot q_2$, where $Y \triangleq \sum_{k \in K}(k \cdot q'_k)$ for a nonempty label set K. Notice that $q \xrightarrow{a} q'_a \cdot q_2$. By inversion on the (potentially infinite) derivation of $(q \mathbin{⨟} t) \leqslant (p \mathbin{⨟} t)$, we have $L \subseteq K$ and $(q'_\ell \mathbin{⨟} (q_2 \mathbin{⨟} t)) \leqslant (p'_\ell \mathbin{⨟} (p_2 \mathbin{⨟} t))$ for all $\ell \in L$. In particular, $((q'_a \cdot q_2) \mathbin{⨟} t) = (q'_a \mathbin{⨟} (q_2 \mathbin{⨟} t)) \leqslant (p'_a \mathbin{⨟} (p_2 \mathbin{⨟} t)) = ((p'_a \cdot p_2) \mathbin{⨟} t)$. Appealing to the coinductive hypothesis, $q'_a \cdot q_2 \lesssim p'_a \cdot p_2$. This appeal is valid because it is guarded by ??.

Case: Consider the case in which $q = \epsilon$. We have $t \leqslant (p \mathbin{⨟} t)$. By inversion, we have $p = \epsilon$. Therefore, the result is vacuously true in this case. □

Theorem 3.1 (Soundness of parametric subtyping). *If $\tau\langle\Theta\rangle \leqslant \sigma\langle\Phi\rangle$, then $\Theta(\tau) \leqslant \Phi(\sigma)$. Likewise, if $A\langle\Theta\rangle \leqslant B\langle\Phi\rangle$, then $\Theta(A) \leqslant \Phi(B)$.*

Proof. Using the mixed induction and coinduction proof technique described by Danielsson and Altenkirch [2010]. Specifically, the proof is by lexicographic mixed induction and coinduction, first by coinduction on the (potentially infinite) structural subtyping derivation, and then by induction on the finite substitution stack Θ.

Case: Consider the case in which the parametric subtyping derivation of $\tau\langle\Theta\rangle \leqslant \sigma\langle\Phi\rangle$ begins with:

$$\frac{t[\vec{\alpha}] \triangleq A \quad u[\vec{\beta}] \triangleq B \quad A\langle\theta;\Theta\rangle \leqslant B\langle\phi;\Phi\rangle}{t[\theta]\langle\Theta\rangle \leqslant u[\phi]\langle\Phi\rangle}\ \textsc{inst-p}$$

Because $\Theta(t[\theta]) = t[\Theta \circ \theta]$ and $\Phi(u[\phi]) = u[\Phi \circ \phi]$, the structural subtyping derivation may begin with

$$\frac{(\Theta \circ \theta)(A) \leqslant (\Phi \circ \phi)(B)}{t[\Theta \circ \theta] \leqslant u[\Phi \circ \phi]}\ \textsc{unf-s}$$

To construct a derivation of $(\Theta \circ \theta)(A) \leqslant (\Phi \circ \phi)(B)$, we may appeal to the coinductive hypothesis. Despite the substitution stack becoming larger, this appeal is valid because it is guarded by the above unf-s rule.

Case: Consider the case in which the derivation of $\tau\langle\Theta\rangle \leqslant \sigma\langle\Phi\rangle$ begins with:

$$\frac{\theta(\alpha)\langle\Theta'\rangle \leqslant \phi(\beta)\langle\Phi'\rangle}{\alpha\langle\theta;\Theta'\rangle \leqslant \beta\langle\phi;\Phi'\rangle}\ \textsc{param-p}$$

Because $(\theta;\Theta')(\alpha) = \Theta'(\theta(\alpha))$ and $(\phi;\Phi)(\beta) = \Phi'(\phi(\beta))$, we must construct a derivation of the structural subtyping $\Theta'(\theta(\alpha)) \leqslant \Phi'(\phi(\beta))$. We can do so by appealing to the inductive hypothesis at the smaller substitution stack Θ'. □

Lemma 5.1. *Given a saturated database:*

(1) *If $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \tau \leqslant \sigma \,\#\, \xi'$; $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \not\Rightarrow \bot$; and $\alpha\langle\Theta\rangle \leqslant_\zeta \beta\langle\Phi\rangle$ for each $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \alpha \leqslant \beta \,\#\, \zeta$; then $\tau\langle\Theta\rangle \leqslant_{\xi'} \sigma\langle\Phi\rangle$.*

(2) *If $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow A \leqslant B \,\#\, \xi'$; $t[\vec{\alpha}] \leqslant u[\vec{\beta}]\xi \not\Rightarrow \bot$; and $\alpha\langle\Theta\rangle \leqslant_\zeta \beta\langle\Phi\rangle$ for each $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \alpha \leqslant \beta \,\#\, \zeta$; then $A\langle\Theta\rangle \leqslant_{\xi'} B\langle\Phi\rangle$.*

Proof. By mutual coinduction on the (potentially infinite) derivations of $\tau\langle\Theta\rangle \leqslant_{\xi'} \sigma\langle\Phi\rangle$ and $A\langle\Theta\rangle \leqslant_{\xi'} \sigma\langle\Phi\rangle$. We consider only the case of $\xi' = +$. The case of $\xi' = -$ is symmetric.

(1) Assume that $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \tau \leqslant \sigma \,\#\, \xi'$; $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \not\Rightarrow \bot$; and $\alpha\langle\Theta\rangle \leqslant_\zeta \beta\langle\Phi\rangle$ for each $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \alpha \leqslant \beta \,\#\, \zeta$. We will now distinguish cases on the types τ and σ.

Case: Consider the case in which $\tau = t'[\theta']$ and $\sigma = u'[\phi']$, where $t'[\vec{\alpha}'] \triangleq A'$ and $u'[\vec{\beta}'] \triangleq B'$. The derivation of $\tau\langle\Theta\rangle \leqslant \sigma\langle\Phi\rangle$ can therefore begin with

$$\frac{A'\langle\theta';\Theta\rangle \leqslant B'\langle\phi';\Phi\rangle}{t'[\theta']\langle\Theta\rangle \leqslant u'[\phi']\langle\Phi\rangle}\ \textsc{inst-p}$$

To construct a derivation of $A'\langle\theta';\Theta\rangle \leqslant B'\langle\phi';\Phi\rangle$, we will appeal to the coinductive hypothesis. This appeal is valid because it will be guarded by the above rule. To make the appeal, we need to first establish the three preconditions.

- Because $t'[\vec{\alpha}'] \triangleq A'$ and $u'[\vec{\beta}'] \triangleq B'$, it follows from the inst-f rule that the saturated database contains $t'[\vec{\alpha}'] \leqslant u'[\vec{\beta}'] \,\#\, \xi' \Rightarrow A' \leqslant B' \,\#\, \xi'$.
- Assume, for the sake of contradiction, that $t'[\vec{\alpha}'] \leqslant u'[\vec{\beta}'] \,\#\, \xi' \Rightarrow \bot$. Because $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow t'[\theta'] \leqslant u'[\phi'] \,\#\, \xi'$, it follows from the compose-f$_\bot$ rule that the saturated database would also contain $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \,\#\, \xi \Rightarrow \bot$. This contradicts our earlier assumption, so we conclude that $t'[\vec{\alpha}'] \leqslant u'[\vec{\beta}'] \,\#\, \xi' \not\Rightarrow \bot$.

- Choose an arbitrary $t'[\vec{\alpha}'] \leqslant u'[\vec{\beta}'] \# \xi' \Rightarrow \alpha' \leqslant \beta' \# \zeta'$; we must show that $\alpha'\langle\theta';\Theta\rangle \leqslant_{\zeta'} \beta'\langle\phi';\Phi\rangle$. This derivation can begin with

$$\frac{\theta'(\alpha')\langle\Theta\rangle \leqslant_{\zeta'} \phi'(\beta')\langle\Phi\rangle}{\alpha'\langle\theta';\Theta\rangle \leqslant_{\zeta'} \beta'\langle\phi';\Phi\rangle} \ \textsc{param-p}$$

  To construct a derivation of $\theta'(\alpha')\langle\Theta\rangle \leqslant_{\zeta'} \phi'(\beta')\langle\Phi\rangle$, we will again appeal to the coinductive hypothesis. This appeal is valid because it will be guarded, by this rule as well as the above rule. The second and third preconditions follow immediately from our earlier assumptions. The first precondition, that $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow \theta'(\alpha') \leqslant \phi'(\beta') \# \zeta'$, follows from the COMPOSE-F rule, given that both $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow t'[\theta'] \leqslant u'[\phi'] \# \xi'$ and $t'[\vec{\alpha}'] \leqslant u'[\vec{\beta}'] \# \xi' \Rightarrow \alpha' \leqslant \beta' \# \zeta'$.

(2) Assume that $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow A \leqslant B \# \xi'$; $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \not\Rightarrow \perp$; and $\alpha\langle\Theta\rangle \leqslant_{\zeta} \beta\langle\Phi\rangle$ for each $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow \alpha \leqslant \beta \# \zeta$. We will now distinguish cases on the types A and B.

Case: Consider the case in which $A = +\{\ell\colon \tau_\ell\}_{\ell\in L}$ and $B = +\{k\colon \sigma_k\}_{k\in K}$ with $L \subseteq K$. The derivation of $A\langle\Theta\rangle \leqslant B\langle\Phi\rangle$ can therefore begin with

$$\frac{\forall \ell \in L\colon \tau_\ell\langle\Theta\rangle \leqslant \sigma_\ell\langle\Phi\rangle}{+\{\ell\colon \tau_\ell\}_{\ell\in L}\langle\Theta\rangle \leqslant +\{k\colon \sigma_k\}_{k\in K}\langle\Phi\rangle} \ +\textsc{p}$$

To construct a derivation of $\tau_\ell\langle\Theta\rangle \leqslant \sigma_\ell\langle\Phi\rangle$ for each $\ell \in L$, we will appeal to the coinductive hypothesis. This appeal is valid because it will be guarded by the above rule. To make the appeal, we need to first establish the three preconditions. The second and third preconditions were already assumed. The first precondition, that $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow \tau_\ell \leqslant \sigma_\ell \# \xi'$, follows from the +F rule, given that $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow +\{\ell\colon \tau_\ell\}_{\ell\in L} \leqslant +\{k\colon \sigma_k\}_{k\in K} \# \xi'$. □

THEOREM 5.2 (SOUNDNESS). *If $\tau \leqslant \sigma \# \xi$, then $\tau\langle\Theta\rangle \leqslant_\xi \sigma\langle\Phi\rangle$ for all stacks Θ and Φ.*

PROOF. By structural induction on the finite derivation of $\tau \leqslant \sigma \# \xi$.

Case: Consider the case in which the derivation of $\tau \leqslant \sigma \# \xi$ begins with:

$$\frac{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \not\Rightarrow \perp \quad \forall(t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow \alpha \leqslant \beta \# \zeta)\colon \theta(\alpha) \leqslant \phi(\beta) \# \zeta}{t[\theta] \leqslant u[\phi] \# \xi} \ \textsc{compose-b}$$

where $t[\vec{\alpha}] \triangleq A$ and $u[\vec{\beta}] \triangleq B$ for some structural types A and B. Because $t[\vec{\alpha}] \triangleq A$ and $u[\vec{\beta}] \triangleq B$, it follows that the saturated database contains $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow A \leqslant B \# \xi$. By the inductive hypothesis on the subderivations and the PARAM-P rule, we have $\alpha\langle\theta;\Theta\rangle \leqslant_\zeta \beta\langle\phi;\Phi\rangle$ for each $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow \alpha \leqslant \beta \# \zeta$. Therefore, we may appeal to Lemma 5.1 to deduce that $A\langle\theta;\Theta\rangle \leqslant_\xi B\langle\phi;\Phi\rangle$. Applying the INST-P rule, $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$, as required.

Case: Consider the case in which the derivation of $\tau \leqslant \sigma \# \xi$ is:

$$\frac{}{x \leqslant x \# \xi} \ \textsc{var-b}$$

It follows immediately from the VAR-P rule that $x\langle\Theta\rangle \leqslant_\xi x\langle\Phi\rangle$. □

LEMMA 5.3.

(1) If $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ and $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow A \leqslant B \# \xi'$, then $A\langle\theta;\Theta\rangle \leqslant_{\xi'} B\langle\phi;\Phi\rangle$.

(2) If $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ and $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \Rightarrow \tau \leqslant \sigma \# \xi'$, then $\tau\langle\theta;\Theta\rangle \leqslant_{\xi'} \sigma\langle\phi;\Phi\rangle$.

(3) If $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$, then $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \# \xi \not\Rightarrow \perp$.

PROOF. Each part is proved in sequence.

(1) Assume that $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ and that $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow A \leqslant B \;\#\; \xi'$. Because $A \leqslant B \;\#\; \xi'$ is a consequence of $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi$, inversion allows us to deduce that $t[\vec{\alpha}] \triangleq A$, $u[\vec{\beta}] \triangleq B$, and $\xi' = \xi$. By inversion on the derivation of $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$, we therefore have $A\langle\theta;\Theta\rangle \leqslant_\xi B\langle\phi;\Phi\rangle$, just as obliged.

(2) By structural induction on the finite derivation of $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \tau \leqslant \sigma \;\#\; \xi'$.

Case:

$$\frac{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow t'[\theta'] \leqslant u'[\phi'] \;\#\; \xi' \quad t'[\vec{\alpha}'] \leqslant u'[\vec{\beta}'] \;\#\; \xi' \Rightarrow \alpha' \leqslant \beta' \;\#\; \zeta}{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \theta'(\alpha') \leqslant \phi'(\beta') \;\#\; \zeta}$$

We must show that $\theta'(\alpha')\langle\theta;\Theta\rangle \leqslant_\zeta \phi'(\beta')\langle\phi;\Phi\rangle$.

Appealing to the inductive hypothesis on the left-hand subderivation, we may deduce $t'[\theta']\langle\theta;\Theta\rangle \leqslant_{\xi'} u'[\phi']\langle\phi;\Phi\rangle$. With this in hand, we then appeal to the inductive hypothesis on the right-hand subderivation to deduce $\alpha'\langle\theta';(\theta;\Theta)\rangle \leqslant_\zeta \beta'\langle\phi';(\phi;\Phi)\rangle$. By inversion, we have $\theta'(\alpha')\langle\theta;\Theta\rangle \leqslant_\zeta \phi'(\beta')\langle\phi;\Phi\rangle$, just as required.

Case:

$$\frac{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \tau_1 \times \tau_2 \leqslant \sigma_1 \times \sigma_2 \;\#\; \xi'}{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \tau_1 \leqslant \sigma_1 \;\#\; \xi'}$$

We must show that $\tau_1\langle\theta;\Theta\rangle \leqslant_{\xi'} \sigma_1\langle\phi;\Phi\rangle$.

Applying part (1) of Lemma 5.3 to the subderivation, we may deduce $\tau_1 \times \tau_2\langle\theta;\Theta\rangle \leqslant_{\xi'} \sigma_1 \times \sigma_2\langle\phi;\Phi\rangle$. By inversion, $\tau_1\langle\theta;\Theta\rangle \leqslant_{\xi'} \sigma_1\langle\phi;\Phi\rangle$, as obliged.

Cases: The other cases are similar.

(3) We will prove that $t[\theta]\langle\Theta\rangle \leqslant_\xi u[\phi]\langle\Phi\rangle$ and $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \bot$ together imply a meta-contradiction, by induction on the finite derivation of $t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \bot$.

Case:

$$\frac{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow t'[\theta'] \leqslant u'[\phi'] \;\#\; \xi' \quad t'[\vec{\alpha}'] \leqslant u'[\vec{\beta}'] \;\#\; \xi' \Rightarrow \bot}{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \bot}$$

Applying part (2) of Lemma 5.3 to the left-hand subderivation, we may deduce $t'[\theta']\langle\theta;\Theta\rangle \leqslant_{\xi'} u'[\phi']\langle\phi;\Phi\rangle$. Then, appealing to the inductive hypothesis on the right-hand subderivation, we have a meta-contradiction, as required.

Case:

$$\frac{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \alpha \leqslant \sigma \;\#\; \xi' \quad (\sigma \neq \beta)}{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \bot}$$

Applying part (2) of Lemma 5.3 to the left-hand subderivation, we may deduce $\alpha\langle\theta;\Theta\rangle \leqslant_{\xi'} \sigma\langle\phi;\Phi\rangle$. Inversion of this derivation yields a meta-contradiction, as required, because there is no declarative subtyping rule that will conclude this judgment when the type σ is not a type parameter.

Case:

$$\frac{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow +\{\ell\colon \tau_\ell\}_{\ell\in L} \leqslant +\{k\colon \sigma_k\}_{k\in K} \;\#\; \xi' \quad (L \nsubseteq_{\xi'} K)}{t[\vec{\alpha}] \leqslant u[\vec{\beta}] \;\#\; \xi \Rightarrow \bot}$$

Applying part (1) Lemma 5.3 to the subderivation, we may deduce $+\{\ell\colon \tau_\ell\}_{\ell\in L}\langle\theta;\Theta\rangle \leqslant_{\xi'} +\{k\colon \sigma_k\}_{k\in K}\langle\phi;\Phi\rangle$. By inversion, we see that $L \subseteq_{\xi'} K$. This contradicts the above $L \nsubseteq_{\xi'} K$.

Cases: Each of the other cases is similar to one of the above. □

Theorem 5.4 (Completeness). *If $\tau\langle\Theta\rangle \leqslant_\xi \sigma\langle\Phi\rangle$, then $\Theta(\tau) \leqslant \Phi(\sigma) \;\#\; \xi$. As a particular case, $\tau\langle\cdot\rangle \leqslant_\xi \sigma\langle\cdot\rangle$ implies $\tau \leqslant \sigma \;\#\; \xi$.*

Proof. By structural induction on the named type τ.

Case: Consider the case in which $\tau = t[\theta]$. By inversion, the derivation of $\tau\langle\Theta\rangle \preccurlyeq \sigma\langle\Phi\rangle$ is one of $t[\theta]\langle\Theta\rangle \preccurlyeq u[\phi]\langle\Phi\rangle \# \xi$. Therefore, the finite derivation of $\tau \preccurlyeq \sigma \# \xi$ can begin with the following rule, so long as we can derive its premises.

$$\frac{t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \# \xi \not\Rightarrow \bot \quad \forall(t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \# \xi \Rightarrow \alpha \preccurlyeq \beta \# \zeta) \colon \theta(\alpha) \preccurlyeq \phi(\beta) \# \zeta}{t[\theta] \preccurlyeq u[\phi] \# \xi}$$

These premises are indeed derivable.

- The first premise is satisfied by an appeal to part (3) of Lemma 5.3.
- Let $t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \# \xi \Rightarrow \alpha \preccurlyeq \beta \# \zeta$ be an arbitrary parameter consequence of $t[\vec{\alpha}] \preccurlyeq u[\vec{\beta}] \# \xi$. Applying part (2) of Lemma 5.3, we have $\alpha\langle\theta;\Theta\rangle \preccurlyeq \beta\langle\phi;\Phi\rangle \# \zeta$. By inversion on this derivation, we may deduce that $\theta(\alpha)\langle\Theta\rangle \preccurlyeq \phi(\beta)\langle\Phi\rangle \# \zeta$. Appealing to the inductive hypothesis at a smaller type $\theta(\alpha)$, we may deduce that $\theta(\alpha) \preccurlyeq \phi(\beta) \# \zeta$.

This fills in the premise of the above rule, completing the required derivation of $\tau \preccurlyeq \sigma \# \xi$.

Case: Consider the case in which $\tau = x$. By inversion, the derivation of $\tau\langle\Theta\rangle \preccurlyeq \sigma\langle\Phi\rangle \# \xi$ is:

$$\frac{}{x\langle\Theta\rangle \preccurlyeq_{\xi} x\langle\Phi\rangle}\ \textsc{var-p}$$

The required $x \preccurlyeq x \# \xi$ follows immediately from var-b rule. □