Subtyping Recursive Types in Kernel Fun

Dario Colazzo, Giorgio Ghelli
Dipartimento d'Informatica
Corso Italia 40, Pisa, ITALY
e-mail: {ghelli, colazzo}@di.unipi.it

Abstract

The problem of defining and checking a subtype relation between recursive types was studied in [3] for a first order type system, but for second order systems, which combine subtyping and parametric polymorphism, only negative results are known [17].

This paper studies the problem of subtype checking for recursive types in system kernel Fun, a typed λ-calculus with subtyping and bounded second order polymorphism.

Along the lines of [3], we study the definition of a subtype relation over kernel Fun recursive types, and then we present a subtyping algorithm which is sound and complete with respect to this relation. We show that the natural extension of the techniques introduced in [3] to compare first order recursive types gives a non complete algorithm. We prove the completeness and correctness of a different algorithm, which also admits an efficient implementation.

Keywords: *type theory and type systems, subtyping, recursive types, kernel Fun.*

1 Introduction

1.1 Background

Recursive type definitions are supported by every typed language, since they are needed to define essential data types, such as lists and trees, and occur in many important programming patterns, such as the subject-observer pattern.

Two different approaches to recursive types have been studied in the literature. Given a recursive definition *let rec* $X = T[X]$, the strong (or *equality based*) approach makes X equal to $T[X]$ ([26, 3]), while the weak (*isomorphism based*) approach ([20, 21]) only gives the programmer a couple of functions $\mathit{fold}_{T[X]}$: $T[X] \rightarrow X$ and $\mathit{unfold}_{T[X]}$: $X \rightarrow T[X]$. The weak approach makes type and subtype checking very easy. The strong approach is easier for programmers to use, but makes subtype checking much more challenging, and is the one we study in this paper.

The combination of subtyping and recursive types has a significant practical relevance. Both notions appear in every typed object-oriented language, and are even useful for the compilation of languages which do not have a subtyping relation (as in the ML to JavaVM compilation project at Persimmon IT, where subtyping between strongly recursive types is used in the intermediate language for optimization purposes [24]).

A complete study of the problem of defining and checking a subtype relation between first-order strongly recursive types can be found in [3].

In this paper we study the integration of strong recursion in a second order type system with subtyping. To this aim we refer to system kernel Fun, an abstract version of the language Fun [9], a language which combines subtyping with parametric polymorphism, and which allows the definition of bounded quantified types, i.e. polymorphic types whose quantifier ranges over a set of subtypes of a given type. Languages of the Fun family, with their extensions, are the main foundational tool used to model object-oriented languages with expressive and strong types (see, for example, [1, 5, 13, 14, 16, 22]). Although most current languages are based on some variant of type comparison by name, the structural approach to type comparison which has been pursued by the type-theoretical community has many advantages, especially in the context of open systems [3], and it also allows comparison by name to be understood.

The main results of this paper are as follows.

- We define an algorithm to perform subtype checking in a strongly recursive extension of kernel Fun, which is the first one presented in the literature, and we prove that it is correct and complete. The algorithm is non obvious, and its correctness proof is very challenging.
- We show that the most natural algorithm for the same problem is non complete, while its obvious general-

[1]Proceedings of the 14^{th} Annual IEEE Symposium on Logic in Computer Science (LICS), Trento, Italy, 1999, ACM Press, New York, USA.

ization is non correct, even if we restrict ourselves to a limited subset of system kernel Fun.

- In the full paper [11] we also prove that our algorithm can be implemented in an efficient way.

Our algorithm is obtained by extending the first-order Amadio-Cardelli algorithm in a non-trivial way. Their algorithm is based on the idea of keeping track of the pairs of compared types which are met during the subtype-checking process, so that it can stop when the 'same' pair is met for the second time. We show that the obvious extension of their algorithm to kernel Fun fails to be complete when 'sameness' is generalized to α-equivalence (or, even worse, just equality), and it fails to be correct when sameness is generalized to a quite natural notion of 'similarity'. However, we obtain a correct and complete algorithm if we generalize sameness to similarity, but stop execution only when a similar pair is met for the *third* time.

In this work we do not study the complexity of our algorithm. We know that it may have an exponential behavior [19], and we suspect it may be made polynomial, but in this paper this will remain an open issue.

1.2 Related work

A complete study of the problem of subtyping *first-order* recursive types can be found in [3], where a natural subtype relation over recursive types is defined, and a sound and complete subtyping algorithm is presented. In [25] a more efficient subtyping algorithm is described.

In [4] another possible axiomatization of the subtype relation between first order recursive types is presented, which is equivalent to the one defined in [3]. Our axiomatization of inclusion between second order recursive types is more similar to the one in [4] than to the one in [3]. In [2], first order recursive types are studied from a syntactic perspective and the two possible approaches to recursion, *equality-based* and *isomorphism-based*, are compared. In particular the equivalence between the two approaches is proved.

Regarding the problem of subtyping recursive types in second order systems, an important (negative) result was given in [17]: any attempt to extend system $F_{\leq}$ [15] with recursive types leads to the definition of a non conservative extension of the system. This problem is strictly related to the undecidability of subtyping in $F_{\leq}$ [27, 18].

Other papers where second order recursive types are studied don't deal with the subtype checking problem in much depth. In [5], a recursive extension of system $F^{\omega}_{<:}$ [6, 7] is defined, but only as a tool to compare different models of object-oriented languages. Hence, that paper doesn't consider the algorithmical aspects of the subtyping problem, and the defined system is far less powerful than our extension of the [3] system. The extension of kernel Fun with recursive types is studied in [10] from a semantical point of view and it is proved that extending the system with recursive types is consistent.

A different strand of research deals with a notion of subtyping where an instance of a polymorphic type is a supertype of the type itself. In such a system, the subtyping problem for second order languages is undecidable even when quantification is unbounded [28]. We do not comment on papers in this family since their results and techniques cannot be applied to subtyping for languages in the Fun family.

1.3 Structure of the paper

The paper is organized as follows. In Section 2 we present a recursive version of system kernel Fun. We extend the Amadio-Cardelli first-order subtype system to system kernel Fun, adopting the style of [4], by a set of rules whose coinductive interpretation defines the subtype relation. Our presentation is also characterized by the fact that α-renaming is explicitly managed, and no variable renaming is needed to type-check judgements where no recursive type appears. In Section 3 we give a more algorithmic version of the same system, where every type occurrence is labeled, and these labels are used to rename types in a way which makes it easier to test for type 'similarity', and to reason about the properties of the proofs in this type system. In Section 4 we modify this system so as to obtain an algorithm which is complete and sound with respect to the non-algorithmic rules. The complete proofs of soundness and completeness are given in the full paper ([11]) but they cover over thirty pages and so we only briefly describe them in Section 5. In Section 6 we show that the most natural 'algorithm' to solve the subtype checking problem is not complete; some more details are reported in Appendix A. In Section 7 we show that a version of our algorithm which stops when a similar pair is met for the second time, instead of the third time, is not sound. In Section 8 we outline our conclusions and areas of future research.

2 Recursive kernel Fun

2.1 Syntax

We only present here the kernel Fun types, since the type rules for terms are not affected by the introduction of recursion; for a more complete introduction to the system see [9, 15, 12, 8]).

Types

$T ::= \top \mid t \mid X \mid \mu X.\forall t \leq T.T \mid \mu X.T \rightarrow T$

Pre-Judgements

$P ::= \Gamma \rhd \text{Env} \mid \Gamma \rhd T \text{ Type} \mid \Pi \rhd T \leq T$

Judgements

$J ::= \Gamma \vdash \text{Env} \mid \Gamma \vdash T \text{ Type} \mid \Pi \vdash T \leq T$

Bi-Environments

$\Pi ::= () \mid \Pi, (t, t') \leq (T, T') \mid \Pi, (X = T, Y = T')$

Environments

$\Gamma ::= () \mid \Gamma, t \leq T \mid \Gamma, X = T$

In our presentation, types are not considered modulo α-equivalence, because variable names play a central role in our subtyping algorithm. We require that all variables in a single type have different names, and we say that a type which satisfies this condition is an "R-Type". This condition is not restrictive with respect to the usual presentation of the system, where types are interpreted modulo α-equivalence, since every type is α-equivalent to an R-Type.

An environment defines a bound for some type variables and a body for some recursion variables, and is used to check the good formation of a type. A bi-environment gives the same information for variables appearing in the compared types T, U in a judgement $\Pi \vdash T \leq U$; moreover, an assumption $(t, u) \leq (T', U')$ also means that t inside T is equivalent to u inside U. Bi-environments are one of the tools we use to reduce the need for variable renaming during the subtype checking process (see rule ($\forall_{\leq}$), and [19]).

Pre-judgements are the input for the subtype and good formation checking process; a judgement indicates that the corresponding pre-judgement is true.

$\Gamma \vdash$ Env and $\Gamma \vdash T$ Type are the well-formation judgements, checking, essentially, that every variable is defined and that, if a variable is in the scope of another one, their names differ. Finally $\Pi \vdash T \leq U$ means that, with respect to the bi-environment Π, T is a subtype of U.

$\mu X.\forall t \leq T.U$ is a universally quantified type where the type variable t, which may occur free in U, only ranges over the subtypes of T; $\mu X.T \rightarrow U$ denotes the type of all functions from values of type T to values of type U. In both cases, an occurrence of X in T or in U recursively denotes the whole type, $\mu X.\forall t \leq T.U$, or $\mu X.T \rightarrow U$ respectively. This means for example that, in any interpretation which respects this intuition, the following equations must hold:

$$\mu X.T = [\mu X.T/X]T = [[\mu X.T/X]T/X]T...$$

We consider a grammar where only function or bounded quantified types can be the body of a recursive type; as an alternative we may just add $\mu X.T$ to kernel Fun grammar. This is mainly a stylistic choice, with no deep effect on either the power of the language or the difficulty of the problem.

To define the rules, we first need to define the following bi-environment operations. We define them on bi-environment elements. They are lifted to the whole bi-environment in the obvious element-wise way.

	$(t, t') \leq (T, T')$	$(X = T, Y = T')$
Def	(t, t')	(X, Y)
Swap	$(t', t) \leq (T', T)$	$(Y = T', X = T)$
Left	$t \leq T$	$X = T$
Right	$t' \leq T'$	$Y = T'$

The good formation rules are the standard ones, with the exception of the variable formation rule, where we adopt the following stronger variant (note that, while all variable names in a single type are different, two different types in the same judgement may define variables which have the same name, provided that (EnvForm) and (VarForm) are respected).

$$\frac{\Gamma \vdash T \text{ Type} \quad t \notin \text{def}(\Gamma)}{\Gamma, t \leq T \vdash \text{Env}} \quad \text{(EnvForm)}$$

$$\frac{\Gamma \vdash \text{Env} \quad \Gamma \vdash \Gamma(t) \text{ Type}}{\Gamma \vdash t \text{ Type}} \quad \text{(VarForm)}$$

$\Gamma(t)$ indicates the bound of t in Γ, i.e. the type T such that $t \leq T \in \Gamma$; if $\Gamma = (t_1 \leq T_1, ..., t_n \leq T_n)$ then def(Γ)=$(t_1, ..., t_n)$.

Our notion of good formation, called UniDef in [19], is stronger than the standard one, but every standard well formed judgement admits an α-equivalent UniDef judgement. The UniDef good formation is needed to be able to avoid α-renaming when the (AlgTrans$_{\leq}$) is applied, as shown in the full paper.

We now present the rules which define our subtyping relation over recursive types. We say that a relation R

$$R \subseteq \{(\Pi, T, U) : \Pi \vdash T \leq U \text{ is well formed}\}$$

is compatible with a set of rules if R contains a judgment which unifies with the conclusion of a subtyping rule if and only if R also contains the corresponding judgements which are unified with the premises of that rule.

A set of subtype rules S is usually used to define the minimum relation R which is compatible with S, i.e. the set of all judgements which admit a finite proof in the system S (inductive interpretation). In this case, the rules we present will be interpreted coinductively, which means that our subtype relation is defined as the maximum relation which is compatible with the rules.

We may also express this fact by saying that a pre-judgement holds if there exists either a finite or an infinite proof for it (see Definition 3.8 for a formal account). WF(Π, T, U) will abbreviate Left(Π) $\vdash$ T Type and Right(Π) $\vdash$ U Type. In ($\forall_{\leq}$) rule, $\Pi \vdash T = T'$ abbreviates Swap(Π) $\vdash T' \leq T$ and $\Pi \vdash T \leq T'$.

$$\frac{\mathrm{WF}(\Pi, T, \top)}{\Pi \vdash T \leq \top}\ (\top_{\leq})$$

$$\frac{(t,u) \in \mathrm{Def}(\Pi) \quad \mathrm{WF}(\Pi, t, u)}{\Pi \vdash t \leq u}\ (\mathrm{Id}_{\leq})$$

$$\frac{\begin{array}{l}(t,u) \leq (T', U') \in \Pi \\ U \neq \top \quad U \neq u \qquad \Pi \vdash T' \leq U\end{array}}{\Pi \vdash t \leq U}\ (\mathrm{AlgTrans}_{\leq})$$

$$\frac{\begin{array}{l}\Pi' = (\Pi, (X{=}\mu X.T \to U,\ Y{=}\mu Y.T' \to U')) \\ \mathrm{Swap}(\Pi') \vdash T' \leq T \quad \Pi' \vdash U \leq U'\end{array}}{\Pi \vdash \mu X.T \to U \leq \mu Y.T' \to U'}\ (\to_{\leq})$$

$$\frac{\begin{array}{l}\Pi' = (\Pi, (X{=}\mu X.\forall t \leq T.U,\ Y{=}\mu Y.\forall t' \leq T'.U')) \\ \Pi' \vdash T = T' \quad \Pi', (t,t') \leq (T, T') \vdash U \leq U'\end{array}}{\Pi \vdash \mu X.\forall t \leq T.U \leq \mu Y.\forall t' \leq T'.U'}\ (\forall_{\leq})$$

$$\frac{X = T \in \mathrm{Left}(\Pi) \quad \Pi \vdash \uparrow T \ \leq \ U}{\Pi \vdash X \ \leq \ U}\ (\mathrm{LUnf}_{\leq})$$

$$\frac{Y = U \in \mathrm{Right}(\Pi) \quad \Pi \vdash T \ \leq \uparrow U}{\Pi \vdash T \ \leq \ Y}\ (\mathrm{RUnf}_{\leq})$$

Notice that in our definition of the $(\forall_{\leq})$ rule, the characteristic kernel Fun requirement of equality of bound types is expressed in the premises by the mutual subtyping judgement $\Pi' \vdash T = T'$.

Since we are interested in solving the subtype-checking problem, we will comment on the rules with respect to their backwards reading (where the problem of proving the conclusion is reduced to the problem of proving the premises). The forwards reading of the same rules obviously makes perfect sense as well.

The symbol $\uparrow T$ denotes a renaming of both type and recursive defined variables in T. This renaming cannot, in general, be avoided and in the next section we will define a systematic way to perform this operation.

The use of bi-environments allows judgements such as $() \vdash \forall t \leq \top.t \leq \forall u \leq \top.u$ to be proved with no need to rename t and u to a common name.

3 Labeled recursive kernel Fun

In this section we introduce a labeled variant of recursive kernel Fun which we use as a bridge between the official system and the algorithm we are going to present.

3.1 Adding labels

To move towards our subtype algorithm we now define a specific variable renaming technique to be used in the unfolding rules. Informally, we interpret the repeated backward application of the subtyping rules starting from $() \rhd T \leq U$ as a descent along the infinite unfoldings of the types T and U, and we label every type occurrence in a derived pre-judgement with the path α which corresponds to that occurrence in the unfolding of T or U. In this way, every different definition of a variable X or t in the unfolding is associated with a different label α, and we can rename that variable as $X_\alpha | t_\alpha$. In this way we obtain variable unicity, and we also preserve, inside t_α, the original name t of the variable it comes from; we will call such t the "face" of the labeled variable t_α. Paths are represented by sequences in $\{0,1\}^*$, indicated by α, β.

For example, the type "$T = \mu X.\forall t \leq \top.\mu Y.t \to X$" may be labeled as follows:

$$T_l = \mu X_\alpha.\forall t_\alpha \leq \top_{\alpha.0}.\mu Y_{\alpha.1}.(t_{\alpha.1.0} \to X_{\alpha.1.1})$$

which corresponds to the following variable renaming:

$$T_r = \mu X_\alpha.\forall t_\alpha \leq \top.\mu Y_{\alpha.1}.(t_\alpha \to X_\alpha).$$

We will use the following notation to represent the result of labeling and renaming a type.

$$T_{r|l} = \mu X_\alpha.\forall t_\alpha \leq \top_{\alpha.0}.\mu Y_{\alpha.1}.(t_{\alpha|\alpha.1.0} \to X_{\alpha|\alpha.1.1}).$$

Observe that in a variable occurrence $t_{\alpha|\beta}$, t_α is the variable itself, while β is an occurrence label, which is used during the subtype checking process.

The following definitions formalize the labeling process. Hereafter, χ will indicate either a labeled variable t_α or X_α.

Definition 3.1 *For each labeled type T, Erase(T) is the type that we get by erasing each label from variables and from $\top_\beta$ occurrences inside T. For each sequence A of labeled types, Erase$^*(A)$ is the sequence obtained by applying "Erase" to each element of A.*

To label a type T we have to specify the label β of the root, and a sequence L which contains the label of each free variable in T. These labels have to refer to an occurrence α such that β is in its scope (definition 3.2, conditions 2 and 3). In the following, we call FV(T) and DV(T) the sets of free and defined variables of the type T, respectively.

Definition 3.2 $[L, \beta]$, *where L is a sequence of labeled variables $(\chi^1, \ldots, \chi^n)$, is a labeling pair for $T \in$R-Types if the following conditions hold:*

1. *variables in L have different faces:*
   $i \neq j \ \Rightarrow\ Erase(\chi^i) \neq Erase(\chi^j)$
2. *for each $t_\alpha \in L$. β is in the scope of $\forall t_\alpha$:*
   $\beta = \alpha.1.\alpha'$
3. *for each $X_\alpha \in L$. β is in the scope of μX_α :*
   $\beta = \alpha.\alpha'$
4. *every free variable in T is defined in L :*
   $FV(T) \subseteq Erase^*(L)$
5. *no variable defined in T is defined in L too :*
   $DV(T) \cap Erase^*(L) = \emptyset$

Definition 3.3 *Let T ∈R-Types, if $[L, \beta]$ is a labeling pair for T then $[L, \beta]\,(T)$ is defined as follows:*

- $[L, \beta]\,(\top) = \top_\beta$
- $[L, \beta]\,(t) = t_{\alpha|\beta} \quad where\ t_\alpha \in L$
- $[L, \beta]\,(X) = X_{\alpha|\beta} \quad where\ X_\alpha \in L$
- $[L, \beta]\,(\mu X.\forall t \leq T.U) =$ $\mu X_\beta.\forall t_\beta \leq [(L,\, X_\beta)\,,\ \beta.0]\,(T)\,.\,[(L,\, X_\beta,\, t_\beta)\,,\ \beta.1]\,(U)$
- $[L, \beta]\,(\mu X.T \to U) =$ $\mu X_\beta.\,([(L,\, X_\beta)\,,\ \beta.0]\,(T) \to [(L,\, X_\beta)\,,\ \beta.1]\,(U))$

By labeling types in R-Types according to the previous definition, we obtain a set of labeled types which we call LR-Types (*Labeled Recursive Types*):

$$\text{LR-Types} = \{T :\ \text{exists } T' \in \text{R-Types and a labeling pair } [L, \beta]\ \text{for } T' \text{ such that } T = [L, \beta]\,(T')\}$$

By combining erasing with labeling we can now define the relabeling operator which takes an LR-Type T and a label β and updates the root label of T to β, and every other variable accordingly; note that this operator renames the bound but not the free variables.

Definition 3.4 (Relabeling) *The relabeling of an LR-Type T according to label β, written $T \uparrow \beta$, is defined as: $T \uparrow \beta\ =\ [FV(T), \beta](Erase(T))$.*

Relabeling is used when a recursion variable $X_{\alpha|\beta}$ is substituted with its body $\mu X_\alpha.T$; in this case, we will guarantee the uniqueness of variables by expanding $X_{\alpha|\beta}$ to $(\mu X_\alpha.T)\!\uparrow\!\beta$.

Over LR-Types we consider the following equivalence relation.

Definition 3.5 *For each $T,\ U$ ∈LR-Types, $T \simeq U \Leftrightarrow Erase(T) = Erase(U)$.*

The relation $\simeq$ (similarity) will be used to define the stop condition of our subtype checking algorithm (Section 4). Intuitively, two types are similar if they are two residuals of the same type in the original judgement.

3.2 The labeled subtype relation

We are now ready to give a precise definition of the types and judgements of labeled recursive kernel Fun.

For brevity, $T_{X,\alpha}$ will indicate a type such as $\mu X_\alpha.\forall t_\alpha \leq T.T'$ or $\mu X_\alpha.T \to T'$, while $\diamond$ ranges over $\leq$ and $\geq$; if $\diamond$ is $\leq$ ($\geq$), then $\diamond^{-1}$ is $\geq$ (resp., $\leq$).

Types
$T ::= \top_\beta \mid t_{\alpha|\beta} \mid X_{\alpha|\beta} \mid \mu X_\alpha.\forall t_\alpha \leq T.T' \mid \mu X_\alpha.T \to T'$
Pre-Judgements
$P ::= \Gamma \rhd \text{Env} \mid \Gamma \rhd T \text{ Type} \mid \Pi \rhd T \leq T$
Judgements
$J ::= \Gamma \vdash \text{Env} \mid \Gamma \vdash T \text{ Type} \mid \Pi \vdash T \leq T$
Bi-Environments
$\Pi ::= () \mid \Pi, (t_\alpha, t'_\beta) \leq (T, T') \mid \Pi, (X_\alpha = T_{X,\alpha}, Y_\delta = T'_{Y,\delta})$
$\quad \mid \Pi, T \diamond X_{\alpha|\beta} \mid \Pi, X_{\alpha|\beta} \diamond T$
Environments
$\Gamma ::= () \mid \Gamma, t_\alpha \leq T \mid \Gamma, X_\alpha = T_{X,\alpha}$

In this variant, when a comparison $X_{\alpha|\beta} \leq U$ or $T \leq Y_{\nu|\eta}$ is met, this information is saved in the bi-environment. This is only done for uniformity with the algorithmic version, where this information is used to stop the subtype-checking process. In this abstract version, this information is not used. The Left, Right and Swap operations are now defined according to the following table.

	$(t_\alpha, u_\beta) \leq (T, U)$	$(X_\alpha{=}T, Y_\delta{=}U)$	$T \diamond U$
Def	(t_α, u_β)	(X_α, Y_δ)	
Swap	$(u_\beta, t_\alpha) \leq (U, T)$	$(Y_\delta{=}U, X_\alpha{=}T)$	$U \diamond^{-1} T$
Left	$t_\alpha \leq T$	$X_\alpha{=}T$	
Right	$u_\beta \leq U$	$Y_\delta{=}U$	

We now present the rules which define our subtyping relation over recursive types; we will call this set of rules $\Re^\infty$. These rules will be interpreted coinductively (Definition 3.8). We omit good formation rules.

Hereafter, $u_{\nu|\_}$ will indicate that the occurrence label is an arbitrary label; likewise for $\top_\_$.

$$\frac{\text{WF}\,(\Pi, T, \top_\beta)}{\Pi \vdash T \leq \top_\beta} \quad (\top_\leq)$$

$$\frac{(t_\alpha, u_\nu) \in \text{Def}(\Pi) \quad \text{WF}\left(\Pi, t_{\alpha|\beta}, u_{\nu|\eta}\right)}{\Pi \vdash t_{\alpha|\beta} \leq u_{\nu|\eta}} \quad (\text{Id}_\leq)$$

$$\frac{\begin{array}{l}(t_\alpha, u_\nu) \leq (T', U') \in \Pi \\ U \neq u_{\nu|\_} \quad U \neq \top_\_ \qquad \Pi \vdash T' \leq U\end{array}}{\Pi \vdash t_{\alpha|\beta} \leq U} \quad (\text{AlgTrans}_\leq)$$

$$\frac{\begin{array}{l}\Pi' = \Pi, (X_\alpha{=}\mu X_\alpha.\forall t_\alpha \leq T.U,\ \ Y_\nu{=}\mu Y_\nu.\forall u_\nu \leq T'.U') \\ \Pi' \vdash T{=}T' \quad \Pi', (t_\alpha, u_\nu) \leq (T, T') \vdash U \leq U'\end{array}}{\Pi \vdash \mu X_\alpha.\forall t_\alpha \leq T.U\ \leq\ \mu Y_\nu.\forall u_\nu \leq T'.U'} \quad (\forall_\leq)$$

$$\frac{\begin{array}{l}\Pi' = \Pi, (X_\alpha{=}\mu X_\alpha.T \to U,\ \ Y_\nu{=}\mu Y_\nu.T' \to U') \\ \text{Swap}(\Pi') \vdash T'\ \leq\ T \quad \Pi' \vdash U\ \leq\ U'\end{array}}{\Pi \vdash \mu X_\alpha.T \to U\ \leq\ \mu Y_\nu.T' \to U'} \quad (\to_\leq)$$

$$\frac{\begin{array}{l}X_\alpha = T \in \text{Left}\,(\Pi) \\ \Pi, X_{\alpha|\beta} \leq U \vdash T \uparrow \beta\ \leq\ U\end{array}}{\Pi \vdash X_{\alpha|\beta}\ \leq\ U} \quad (\text{LUnf}_\leq)$$

$$\frac{\begin{array}{l}Y_\nu = U \in \text{Right}(\Pi)\\ \Pi, T \leq Y_{\nu|\eta} \vdash T \ \leq \ U \uparrow \eta\end{array}}{\Pi \vdash T \ \leq \ Y_{\nu|\eta}} \quad (\text{RUnf}_{\leq})$$

Observe that unfolding rules now use the relabeling operation to rename the unfolded types.

If the pre-judgement P reduces to P' by *one or more* backward applications of $\Re^\infty$ rules, we indicate this fact with $P \twoheadrightarrow_\infty P'$.

To simplify our study, we will restrict ourselves to the pre-judgments which can be used to prove a closed judgement, (Definition 3.7).

Definition 3.6 *We call Start-J the set of pre-judgements* $() \rhd T \leq U$ *where* T, U *are closed LR-Types, and all of their variables have different faces.*

Definition 3.7 *We define Start-J*$^\infty$ *as the set of all possible subtyping pre-judgements that we obtain by reducing a pre-judgement in Start-J by backward applications of* $\Re^\infty$ *rules. Formally:*

$$\textit{Start-J}^\infty = \{P' : \exists P \in \textit{Start-J s.t. } P \twoheadrightarrow_\infty P'\}$$

The Start-J$^\infty$ pre-judgements satisfy some invariants which we prove in the full paper; in particular, they are well-formed.

We now give our definition of inclusion between recursive types; a *failure pre-judgement* is a pre-judgement which does not match the conclusion of any rule.

Definition 3.8 *For each pre-judgement* $\Pi \rhd T \leq U$ *in Start-J*$^\infty$*:*

$$\Pi \vdash_\infty T \leq U \Leftrightarrow \begin{array}{l}\nexists \textit{ a failure pre-judgement } \Pi' \rhd T' \leq U'\\ \textit{s.t. } \Pi \rhd T \leq U \twoheadrightarrow_\infty \Pi' \rhd T' \leq U'\end{array}$$

Equivalently, $\Pi \vdash_\infty T \leq U$ holds when either a finite or an infinite proof tree exists for it.

4 A subtyping algorithm

The rules presented in the previous section save the pairs $X_{\alpha|\beta} \leq U$ or $T \leq Y_{\nu|\eta}$ in the bi-environments. Thus, following [3], we can use this information to stop backwards rule application when such a pair of types is met for the second time. Due to renaming, we cannot expect exactly the same pair to be met twice. Hence the most natural idea is to stop when we meet a pair which matches an already met pair modulo α-renaming.

This algorithm is very inefficient because of the high cost of α-equivalence comparison, but, before this study, was widely accepted as the best guess for a correct and complete algorithm. We will show in the next section that this is not the case, and we consider this as an important, though negative, result. The natural algorithm is correct, but it is not complete since there exist provable judgements which, during subtype checking, produce infinitely many pairs which, though in some sense "similar", always fail to be α-equivalent to a previously met pair.

After this result, in order to define a complete subtyping algorithm, one may look for an equivalence relation which is slightly weaker than α-equivalence, and stop the algorithm when a pair is met twice modulo this weaker equivalence. Similarity (i.e. erasure equality $\simeq$) is the first candidate for this task. Unfortunately, it is too weak. The resulting algorithm is complete, i.e. it always terminates, but is not correct, as shown in Section 6.

However, the algorithm becomes complete and correct if we use similarity but, instead of stopping the second time we meet a pair, we wait until the same pair is met, modulo similarity, for the third time, as formalized below.

Definition 4.1 *We say that* $T \leq U \in^{\simeq}_{n} \Pi$ *if the bi-environment* Π *contains at least* n *pairs* $T' \leq U'$ *such that* $T \simeq T'$ *and* $U \simeq U'$*.*

The rules which define our algorithm are obtained by adding two new termination rules to the $\Re^\infty$ system, and by modifying the unfolding rules as follows.

$$\frac{X_{\alpha|\beta} \leq U \ \in^{\simeq}_{2} \Pi}{\Pi \vdash X_{\alpha|\beta} \ \leq \ U} \ (\text{LEnd}_{\leq}) \qquad \frac{T \ \leq \ Y_{\nu|\eta} \ \in^{\simeq}_{2} \Pi}{\Pi \vdash T \ \leq \ Y_{\nu|\eta}} \ (\text{REnd}_{\leq})$$

$$\frac{\begin{array}{l}X_{\alpha|\beta} \leq U \notin^{\simeq}_{2} \Pi\\ X_\alpha = T \in \text{Left}(\Pi) \quad \Pi, X_{\alpha|\beta} \leq U \vdash T \uparrow \beta \ \leq \ U\end{array}}{\Pi \vdash X_{\alpha|\beta} \ \leq \ U} \quad (\text{LUnf}^2_{\leq})$$

$$\frac{\begin{array}{l}T \leq Y_{\nu|\eta} \notin^{\simeq}_{2} \Pi\\ Y_\nu = U \in \text{Right}(\Pi) \quad \Pi, T \leq Y_{\nu|\eta} \vdash T \ \leq \ U \uparrow \eta\end{array}}{\Pi \vdash T \ \leq \ Y_{\nu|\eta}} \quad (\text{RUnf}^2_{\leq})$$

With this extension we have a new set of rules which we call $\Re^{\textit{alg-2}}$ and whose backwards application defines a subtyping algorithm for recursive types (we outline the termination proof in the next section).

5 Soundness and completeness

The correctness and completeness of our algorithm, i.e. the equivalence between $\Re^{\textit{alg-2}}$ and $\Re^\infty$, is the main result of this work. The proof is complex and very long, and is reported in the full paper [11]. We can only give a short outline here.

We first report the lemma that states a fundamental invariant which our proof exploits over and over again. *DV(B)* are the variables defined inside B. Points 1.2 and 2.2 say that, whenever χ appears in a judgement, the last definition in T of a variable with the same face as χ is the definition of χ itself. This fact, together with the fact that the comparison in the (L/REnd$_\leq$) rules is performed modulo similarity, implies that labels can be ignored altogether during subtyping checking, which makes it possible to have efficient implementations of this algorithm.

Lemma 5.1 *For each* $\Pi' \rhd T' \leq U' \in$ *Start-J*$^{alg|\infty}$*, if* $\chi \in FV(T')$ *and* $\Gamma = Left(\Pi')$ *(if* $\chi \in FV(U')$ *and* $\Gamma = Right(\Pi')$*):*

$$\begin{array}{llll} 1. & \chi = s_\alpha & \Rightarrow & 1.1 \quad \Gamma = \Gamma', s_\alpha \leq A, \Gamma'' \\ & & & 1.2 \quad s_\alpha \notin^{\simeq}_{1} Def(Left(\Gamma'')) \\ 2. & \chi = Y_\alpha & \Rightarrow & 2.1 \quad \Gamma = \Gamma', Y_\alpha = A, \Gamma'' \\ & & & 2.2 \quad Y_\alpha \notin^{\simeq}_{1} Def(Left(\Gamma'')) \end{array}$$

Theorem 5.2 (Soundness) *For each* $(\ ) \rhd T \leq U \in$*Start-J*:

$$() \vdash_{\text{alg-2}} T \leq U \implies () \vdash_\infty T \leq U$$

Proof outline. We prove soundness by describing a way to transform any $\Re^{alg\text{-}2}$ proof tree into an $\Re^\infty$ proof tree. To this end, we repeatedly choose one instance of the (L/REnd) rule, and substitute it with an instance of the unfolding rule together with its $\Re^{alg\text{-}2}$ proof tree. We thus obtain a succession of proofs whose limit is an infinite proof with no instance of the *End* rule, i.e. an $\Re^\infty$ proof. The difficult part of the proof is to prove that for every *End*-proved judgement (which we call an "end-node" of the proof tree), its unfolding (according to the corresponding unfolding rule) admits a successful $\Re^{alg\text{-}2}$ proof tree. We know that every end-node J is similar to two different unfolding-nodes J' and J'' which have been met before (i.e. towards the root of the proof tree) and have been proved by two $\Re^{alg\text{-}2}$ subproofs P' and P''. For every rule instance in P' we can find a $\simeq$-similar rule instance to use to build a proof P for J, with the only exception of the instances of the (Id$_\leq$) rule: if t_η, u_θ have been unified in the environment used in P' there is no way to be sure that two similar variables t_α, u_μ have been unified in the environment used to prove J. This is a real, deep problem. If you unfold the counterexample given in Section 7, you will actually find an end-node whose expansion admits no $\Re^{alg\text{-}2}$ proof tree, and this is exactly because a t_η, u_θ pair was unified the first time a similar judgement was met, but the similar pair t_α, u_μ was not unified later. Hence, in our proof we have to exploit the fact that the end-node J corresponds to two different similar judgements J' and J'' which have both been proved. We first prove that, if we apply to J' the same sequence of rules which transform J'' into J we obtain a fourth similar judgement J''' which has an $\Re^{alg\text{-}2}$ proof which is contained inside the original judgement. Then we consider every $t_\beta \leq u_\nu$ comparison inside the proof of J'''. All of these comparisons can be proved, either by rule (Id$_\leq$) or by a sequence of (AlgTrans$_\leq$) applications followed by a final (Id$_\leq$) application; we have to prove that the same sequence of rules can be used to prove the corresponding $t_\alpha \leq u_\mu$ comparison inside the unfolding of J. Recall that we have a rewrite path $() \rhd T \leq U \twoheadrightarrow J' \twoheadrightarrow J''' \twoheadrightarrow \Pi \rhd t_\beta \leq u_\nu$, and a path $() \rhd T \leq U \twoheadrightarrow J' \twoheadrightarrow J'' \twoheadrightarrow J \twoheadrightarrow \overline{\Pi} \rhd t_\alpha \leq u_\mu$. If $t_\beta \leq u_\nu$ has been proved thanks to (Id$_\leq$) (case a) then t_β and u_ν have been both defined, and unified, along the path from $T \leq U$ to J' ($a.1$), or from J' to J''' ($a.2$), or from J''' to $\Pi \rhd t_\beta \leq u_\nu$ ($a.3$). In case ($a.3$), we prove that t_α and u_μ are also unified in the path from J to $\overline{\Pi} \rhd t_\alpha \leq u_\mu$. Similarly, in case ($a.2$) we prove that t_α and u_μ are also unified in the path from J'' to $\overline{\Pi} \rhd t_\alpha \leq u_\mu$. In case ($a.1$) we refer to J'' too and prove that if we apply to J'' the same sequence of rules which transform J into $\overline{\Pi} \rhd t_\alpha \leq u_\mu$ we obtain $\overline{\overline{\Pi}} \rhd t_\tau \leq u_\delta$, so we have the path $() \rhd T \leq U \twoheadrightarrow J' \twoheadrightarrow J'' \twoheadrightarrow \overline{\overline{\Pi}} \rhd t_\tau \leq u_\delta$. Now, if $\overline{\overline{\Pi}} \rhd t_\tau \leq u_\delta$ is proved by (Id$_\leq$), we prove that $\overline{\Pi} \rhd t_\alpha \leq u_\mu$ is proved by (Id$_\leq$) and that $\alpha = \tau$ and $\mu = \delta$. If $t_\tau \leq u_\delta$, has been reduced by transitivity to $t'_{\tau'} \leq u_\delta$, then we reason as in case b below.

If $t_\beta \leq u_\nu$ has been reduced by transitivity to $t'_{\beta'} \leq u_\nu$ (case b) then we prove that transitivity also reduces $t_\alpha \leq u_\mu$ to $t'_{\alpha'} \leq u_\mu$. Now, if $t'_{\beta'} \leq u_\nu$ is proved by (Id$_\leq$) then we are in case a, otherwise, if transitivity is applied once again, the proof follows by an induction reasoning.□

Theorem 5.3 (Completeness) *For each* $() \rhd T \leq U \in$*Start-J:*

$$() \vdash_\infty T \leq U \implies () \vdash_{\text{alg-2}} T \leq U$$

Proof hint. We first prove that in any infinite branch of an $\Re^\infty$ proof tree there is an infinite number of applications of an unfolding rule. Every such application records a pair $X \leq T'$ such that X and T' are similar to subterms of T and U, hence at least one of these $X \leq T'$ pairs is met, modulo similarity, more than three times along that branch.□

6 Incompleteness of the α-based algorithm

For reasons of space, we only report here the simplest judgement we have found which makes the α-based algorithm diverge; this counterexample is commented on in the full paper. To make its structure easier to grasp, we avoid useless type and recursion variables, we do not write the bound of top-bounded type variables, and we use covariant pair types and a bottom type; both of them can be encoded in kernel Fun, provided that $\perp$ is never used as a bound for a type variable. Observe that the counterexample does not need bounded variables nor does it need contravariant type constructors. However, it does need non-structural subtyping (either $\bot$ or $\top$) and, most importantly, the $\mu.\forall.\mu$ nesting. The diverging pre-judgement is the following:

$$\begin{aligned}\rhd\quad & \mu.\forall.\mu X.\forall t.\,(\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z))\\ & \le\ \mu Y.\forall u.\mu K.\forall.\,((u \times \top \times K) \times Y)\end{aligned}$$

7 Unsoundness of the similarity-based algorithm

The simplest pre-judgement we have found to prove the non soundness of the algorithm which stops the first time it meets a pair again, modulo similarity, is the following one. The same comments as in the previous section apply.

$$\begin{aligned}\rhd\quad & \mu Z.\,(\forall t.\mu X.\,(X \times (t \times Z)))\\ & \le\ \mu.\forall u.\mu Y.\,((\top \times (u \times (\mu.\forall v.Y))) \times \top)\end{aligned}$$

8 Conclusions

We have studied the problem of subtyping recursive types in kernel Fun. This problem is important because the combination of subtyping, parametric polymorphism and recursion is essential in the context of strongly typed object-oriented languages.

The main result of this work is the definition of a subtyping algorithm for strongly recursive kernel Fun, which is the only one known for this class of languages. We prove it to be sound and complete and the proof is technically very challenging. We have also been able to prove, by exhibiting non trivial counterexamples, that the most natural algorithm to attack the problem is not complete, and that its first obvious relaxation is not correct.

Moreover, we have proved that our algorithm can be equivalently defined by eliminating variable renaming. This makes the algorithm very efficient in practice since, if variable renaming can be avoided, no memory allocation is needed during the execution of the subtyping algorithm, and the key step of similarity checking can be reduced to pointer equality checking. Moreover, this property may be the key to be able to adapt the efficient subtype checking algorithms which are known for first order systems ([25]).

As a consequence of this work, we feel now the need to study the more general field of "regular" trees with variables. Both our counterexamples involve infinite trees with variables which can be finitely described but which are not regular in the usual sense of the word. The problem of finding a weaker notion of regularity which is satisfied by these trees has some deep links with this research, and we have some preliminary results.

Finally, since in our presentation there is no rule which explicitly states the transitive property of the subtype relation, its transitivity is still to be proved. This is easier than proving correctness, but requires similar techniques to be used.

Acknowledgements

We gratefully thanks the anonymous referees for insightful and constructive remarks. This work has been partially supported by Esprit Working Groups 26142 - Applied Semantics, and 22552 - PASTEL.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] M. Abadi and M. P. Fiore. Syntactic considerations on recursive types. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science* [23], pages 242–252.

[3] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[4] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *Proceedings of the Typed Lambda Calculi And Application, Nancy, France*, number 1210 in LNCS, pages 63–81, Berlin, April 1997. Springer Verlag.

[5] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 1999.

[6] L. Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, Oct. 1990.

[7] L. Cardelli and G. Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, Oct. 1991.

[8] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994.

[9] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, Dec. 1985.

[10] F. Cardone. Relational semantics for recursive types and bounded quantification. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, volume 372 of *LNCS*, pages 164–178, Stresa, Italy, July 1989. Springer-Verlag.

[11] D. Colazzo and G. Ghelli. Subtyping recursive types in kernel Fun, 1998. file://ftp.di.unipi.it/pub/Papers/ghelli/recursive.ps.

[12] P. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type checking in $F_{\le}$. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.

[13] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, Mar. 1988.

[14] K. Fisher and J. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1996.

[15] G. Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Dipartimento di Informatica, Università di Pisa, March 1990. Tech. Rep. TD-6/90.

[16] G. Ghelli. Modelling features of object-oriented languages in second order functional languages with subtypes. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in LNCS, pages 311–340, Berlin, 1991. Springer Verlag.

[17] G. Ghelli. Recursive types are not conservative over $F_{\leq}$. In M. Bezen and J. Groote, editors, *Proceedings of the Typed Lambda Calculi And Application, Utrecht, The Netherlands*, number 664 in LNCS, pages 146–162, Berlin, March 1993. Springer Verlag.

[18] G. Ghelli. Divergence of $F_{\leq}$ type checking. *Theoretical Computer Science*, 139(1-2):131–162, 1995.

[19] G. Ghelli. Complexity of kernel Fun subtype checking. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 134–145, Philadelphia, Pennsylvania, 24–26 May 1996.

[20] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *LNCS*. Springer Verlag, 1979.

[21] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. The MIT Press, 1992.

[22] C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994.

[23] IEEE Computer Society Press. *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, 27–30 July 1996.

[24] A. Kennedy, 1998. personal communication.

[25] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient recursive subtyping. In *Proceedings POPL '93*, pages 419–428, 1993.

[26] G. Nelson. *Systems Programming in Modula-3*. Prentice Hall, 1991.

[27] B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.

[28] J. Tiuryn and P. Urzyczyn. The subtyping problem for second-order types is undecidable. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science* [23], pages 74–85.

Appendix A

The α-based algorithm can be defined by adopting the following stop rule, instead of our (L/REnd) rules.

$$\frac{T' \leq U' \in \Pi \quad T \simeq_\alpha T' \quad U \simeq_\alpha U' \quad \mathrm{WF}\,(\Pi, T, U)}{\Pi \vdash T \leq U} \quad (\mathrm{End}^{\alpha}_{\leq})$$

Our complete algorithm records a pair $T \leq U$ only when the unfolding rule is applied, which gives the end-rule less possibilities to be applicable, but is still enough to make it complete. We show here that the incompleteness of the α-based algorithm is not a consequence of this choice, by studying a version of the algorithm which records *every* met pair.

We prove incompleteness of the algorithm defined by $\Re^{alg\text{-}\alpha}$ rules by exhibiting a particular provable pre-judgement which makes the algorithm diverge.

To this end, we first fix some conventions we will use in the proof. Hereafter we assume that: $\forall t_\alpha.T$ indicates the type $\forall t_\alpha \leq \top.T$ where t_α may occur free in T; $\forall.T$ indicates a type $\forall t_\alpha \leq \top.T$ where t_α doesn't occur free in T; $\mu.T$ indicates a type $\mu X_\alpha.T$ where X_α doesn't occur free in T. Moreover, we will not write the variable labels (but we behave as if they were present).

The diverging pre-judgement is $() \rhd \overline{T} \leq \overline{U}$ where:

$$\begin{aligned} \overline{T} &= \mu.\forall.\mu X.\forall t.\,(\perp \times \mu Z.\forall.\mu.\forall.\,((\perp \times\, t \times X) \times Z)) \\ \overline{U} &= \mu Y.\forall u.\mu K.\forall.\,((u \times \top \times K) \times Y) \end{aligned}$$

To simplify things, we have used a type $\perp$ which is the subtype of every type, associated to the following termination rule:

$$\frac{\mathrm{WF}\,(\Pi, \perp_\beta, T)}{\Pi \vdash \perp_\beta \leq T} \quad (\perp_{\leq})$$

We also use pair types with the usual covariant rule, i.e. the following reduction rules, where Π'' is defined as in rule $(\rightarrow_{\leq})$ and where it is explicitly indicated if the reduction is to the left or right pre-judgement.

$$\Pi \vdash \mu X_\alpha.T \times U \leq \mu Y_\nu.T' \times U' \overset{(\times_{\leq})^l}{\longrightarrow} \Pi'' \vdash T \leq T'$$
$$\Pi \vdash \mu X_\alpha.T \times U \leq \mu Y_\nu.T' \times U' \overset{(\times_{\leq})^r}{\longrightarrow} \Pi'' \vdash U \leq U'$$

Pair and bottom types make our counterexample much more readable, and we can encode both of them in system kernel Fun, as soon as one never uses $\perp$ as the bound of a type variable.

If we ignore for a moment some superfluous μ and $\forall$, both the types $\overline{T}$ and $\overline{U}$ present a particular kind of nested recursion μ-$\forall$-μ : a type variable is defined between two definitions of recursion variables. Moreover, in the inner type of each of them there are occurrences of all three defined variables. We will see that in comparing these types, the outer left hand-side type (the type $\mu X.\forall t.\,(\ldots)$) will be always compared with the inner right hand-side type (the type $\mu K.\forall.\,(\ldots)$) and viceversa, and for this reason no pair of types created in the reduction will be α-equivalent to an already met pair. We believe that divergence problem arises when types with at least such a structure are compared, but we do not prove this fact.

We omit obvious reduction steps and in each reduction we only write the last element added to the bi-environment.

Moreover, in each reduction step we will implicitly assume that the current comparison is saved in the bi-environment.

Finally, we omit proof branches concerning well-formedness (as already mentioned, in [11] we prove that, for Start-J pre-judgements, well-formedness is always guaranteed).

With these conventions, using $\Re^{alg\text{-}\alpha}$ rules, we have the following reduction chain for our pre-judgement; starting from the eleventh judgement, every pair of compared types is similar to the one which has been met nine steps before. However, every time it differs in some free variables. For example, if we consider the last nine steps, all the judgements from 12 to 17 contain the free variable t which is different from the one met nine steps before. Step 18 differs from 9 due to X and K. Step 19 differs from 10 due to K. Step 20 differs from 11 due to u and Y.

1. $$() \rhd \overline{T} \leq \overline{U}$$

2. $$\xrightarrow{(\forall_\leq)} \begin{array}{l} \ldots\left(- = \overline{T},\ Y = \overline{U}\right),\ (-,\mathbf{u}) \leq (\top,\top) \rhd \\ \mu X.\forall t.\,(\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z)) \\ \leq \mu K.\forall.\,((u \times \top \times K) \times Y) \end{array}$$

3. $$\xrightarrow{(\forall_\leq)} \begin{array}{l} \ldots(X = \mu X.\forall t.\,(\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z)), \\ K = \mu K.\forall.\,((u \times \top \times K) \times Y)), \\ (\mathbf{t},-) \leq (\top,\top) \rhd \\ (\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z)) \\ \leq ((u \times \top \times K) \times Y) \end{array}$$

4. $$\xrightarrow{(\times_\leq)^r} \begin{array}{l} \ldots(- = (\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z)), \\ - = ((u \times \top \times K) \times Y)) \rhd \\ \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z) \leq Y \end{array}$$

5. $$\xrightarrow{(\mathrm{RUnf}_\leq)} \begin{array}{l} \ldots \rhd \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z) \\ \leq \mu Y.\forall u.\mu K.\forall.\,((u \times \top \times K) \times Y) \end{array}$$

6. $$\xrightarrow{(\forall_\leq)} \begin{array}{l} \ldots(Z = \forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z), \\ Y = \mu Y.\forall u.\mu K.\forall.\,((u \times \top \times K) \times Y)), \\ (-,\mathbf{u}) \leq (\top,\top) \rhd \\ \mu.\forall.\,((\bot \times\ t \times X) \times Z) \\ \leq \mu K.\forall.\,((u \times \top \times K) \times Y) \end{array}$$

7. $$\xrightarrow{(\forall_\leq)} \begin{array}{l} \ldots(- = \mu.\forall.\,((\bot \times\ t \times X) \times Z), \\ K = \mu K.\forall.\,((u \times \top \times K) \times Y)), \\ (-,-) \leq (\top,\top) \rhd \\ (\bot \times\ t \times X) \times Z \leq (u \times \top \times K) \times Y \end{array}$$

8. $$\xrightarrow{(\times_\leq)^l} \begin{array}{l} \ldots(- = (\bot \times\ t \times X) \times Z,) \\ - = (u \times \top \times K) \times Y) \rhd \\ (\bot \times\ t) \times X \leq (u \times \top) \times K \end{array}$$

9. $$\xrightarrow{(\times_\leq)^r} \begin{array}{l} \ldots(- = ((\bot \times\ t) \times X),\ - = ((u \times \top) \times K)) \rhd \\ X \leq K \end{array}$$

10. $$\xrightarrow{(\mathrm{LUnf}_\leq)} \begin{array}{l} \ldots \rhd \mu X.\forall t.\,(\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z)) \\ \leq K \end{array}$$

11. $$\xrightarrow{(\mathrm{RUnf}_\leq)} \begin{array}{l} \ldots \rhd \mu X.\forall t.\,(\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z)) \\ \leq \mu K.\forall.\,((u \times \top \times K) \times Y) \end{array}$$

12. $$\xrightarrow{(\forall_\leq)} \begin{array}{l} \ldots(X = \mu X.\forall t.\,(\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z)), \\ K = \mu K.\forall.\,((u \times \top \times K) \times Y)), \\ (\mathbf{t},-) \leq (\top,\top) \rhd \\ (\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z)) \\ \leq ((u \times \top \times K) \times Y) \end{array}$$

13. $$\xrightarrow{(\times_\leq)^r} \begin{array}{l} \ldots(- = (\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z)), \\ - = ((u \times \top \times K) \times Y)) \rhd \\ \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z) \leq Y \end{array}$$

14. $$\xrightarrow{(\mathrm{RUnf}_\leq)} \begin{array}{l} \ldots \rhd \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z) \\ \leq \mu Y.\forall u.\mu K.\forall.\,((u \times \top \times K) \times Y) \end{array}$$

15. $$\xrightarrow{(\forall_\leq)} \begin{array}{l} \ldots(Z = \forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z), \\ Y = \mu Y.\forall u.\mu K.\forall.\,((u \times \top \times K) \times Y)), \\ (-,\mathbf{u}) \leq (\top,\top) \rhd \\ \mu.\forall.\,((\bot \times\ t \times X) \times Z) \\ \leq \mu K.\forall.\,((u \times \top \times K) \times Y) \end{array}$$

16. $$\xrightarrow{(\forall_\leq)} \begin{array}{l} \ldots(- = \mu.\forall.\,((\bot \times\ t \times X) \times Z), \\ K = \mu K.\forall.\,((u \times \top \times K) \times Y)), \\ (-,-) \leq (\top,\top) \rhd \\ (\bot \times\ t \times X) \times Z \leq (u \times \top \times K) \times Y \end{array}$$

17. $$\xrightarrow{(\times_\leq)^l} \begin{array}{l} \ldots(- = (\bot \times\ t \times X) \times Z,) \\ - = (u \times \top \times K) \times Y) \rhd \\ (\bot \times\ t) \times X \leq (u \times \top) \times K \end{array}$$

18. $$\xrightarrow{(\times_\leq)^r} \begin{array}{l} \ldots(- = ((\bot \times\ t) \times X),\ - = ((u \times \top) \times K)) \rhd \\ X \leq K \end{array}$$

19. $$\xrightarrow{(\mathrm{LUnf}_\leq)} \begin{array}{l} \ldots \rhd \mu X.\forall t.\,(\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z)) \\ \leq K \end{array}$$

20. $$\xrightarrow{(\mathrm{RUnf}_\leq)} \begin{array}{l} \ldots \rhd \mu X.\forall t.\,(\bot \times \mu Z.\forall.\mu.\forall.\,((\bot \times\ t \times X) \times Z)) \\ \leq \mu K.\forall.\,((u \times \top \times K) \times Y) \end{array}$$

Remark 1 *Another kind of renaming could be used in the unfolding rules. When $X_{\alpha|\beta}$ is defined by $\mu X_\alpha.T$, our unfolding rule substitutes $X_{\alpha|\beta}$ with $(\mu X_\alpha.T)\uparrow\beta$, according with the meaning we gave to recursive types. It would also be possible, however, to substitute $X_{\alpha|\beta}$ with just the body $T\uparrow\beta$. In this way, we do not create a new X variable, since relabeling applies only to the variables which are defined* inside T. *By not renaming the outermost recursive variable, this variant has better hopes of meeting an already met pair. We believe that this variant is sound, but we did not prove this fact. However, it is still not complete. Consider our judgement. Every time we meet a pair of types which is similar to a previous one, it differs because of the free type variables, and this difference remains if we move to the variant algorithm. The only exception are the judgements in steps 18 and 19, which only contain free recursion variables. Indeed, with the variant algorithm, the X at step 9 would be the same as in step 18, while in the basic algorithm they are different variables. However, the K would still be different, since a new μK is generated every time Y is unfolded in step $9 * i + 5$. Hence, the variant algorithm is not complete.*

This alternative way of renaming corresponds the following chain of equivalences:

$$\mu X.T = \mu X.\,[T/X]\,T = \mu X.\,[T/X]\,([T/X]\,T)$$

instead of the following chain that we exploit:

$$\mu X.T = [\mu X.T\,/\,X]\,T = [[\mu X.T\,/\,X]\,T\,/\,X]\,T\ldots$$

It would be interesting to try and prove the soundness of this alternative algorithm, since it may be more convenient in some situations. We leave this as an open problem.