Towards the Type Safety of Pure Subtype Systems

Valentin Pasquale
CEA List, Université Paris-Saclay, France

Álvaro García-Pérez
CEA List, Université Paris-Saclay, France

Abstract

Hutchins' Pure Subtype Systems (PSS) offer a unified framework for types and terms, promising significant advancements in language design for features like dependent types and higher-order subtyping. However, the theory has been hampered by a critical gap: a proof of type safety has remained an open problem for over a decade. The original attempt to prove this property relied on the conjectured commutativity of two fundamental reduction relations, equivalence and subtyping. Proving transitivity elimination, however, requires this commutativity, a property that is notoriously difficult to establish for higher-order subtyping systems.

In this paper, we address this issue by introducing Machine-Based PSS (MPSS), a novel reformulation of the original system. MPSS integrates a continuation stack mechanism, reminiscent of the Krivine Abstract Machine, to keep track of arguments that are passed during function application, enabling more fine-grained reductions. This architectural change exposes crucial intermediate reduction steps that were absent in the original PSS. The primary contribution of our work is a direct proof that the equivalence and subtyping reductions in MPSS commute. This result formally establishes transitivity elimination, which is the cornerstone of the inversion lemma required for type safety. We conclude by outlining a pathway from our foundational result to a complete, type-safe system, thereby paving the way for the practical realization of PSS-based languages.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Lambda calculus, Pure subtype systems, Dependent types, Higher-order subtyping, Type safety

Digital Object Identifier 10.4230/LIPIcs.CSL.2026.37

Related Version *Full Version*: `https://arxiv.org/abs/2407.13882`

1 Introduction

Hutchins' Pure Subtype Systems (PSS for short) [19, 18] have been proposed as a novel approach to type theory that enables a number of advanced language features for extensibility and genericity. Among other benefits, PSS (hereafter in the singular) harmoniously solves both the expression problem [30, 32] and the tag-elimination problem [18]. PSS's expressivity allows one to implement extensible modules based on deep mixin composition, which is a key result for a type system [18]. By blurring the distinction between types and terms, PSS naturally combines dependent types and higher-order subtyping, which makes the approach very promising as a basis for generic, modular, and extensible programming. However, a proof of type safety for PSS is conspicuously lacking. This crucial property has only been shown to hold under the conjecture that two key reduction relations in the system, equivalence and subtyping, commute [18]. The commutativity of these two reductions is an instance of a recurrent problem in higher-order subtyping known as *transitivity elimination*. Consequently, the type safety of PSS has remained an open problem for more than a decade.

In this paper, we resolve this long-standing open problem by introducing a novel reformulation of Hutchins' PSS. Our system is based on a mechanism reminiscent of the Krivine Abstract Machine, it allows more fine-grained reductions and exposes crucial intermediate reduction steps that were absent in the original PSS. This architectural change allows for a direct and elegant proof of transitivity elimination.

Contrary to traditional type systems, PSS harmoniously mixes terms and types. Consider the natural number 3 and the type *Nat* of natural numbers. In PSS one can have a term that encodes $Nat + 3$ with the naive Church encoding of naturals, which type-checks to *Nat*. Terms can also be used instead of types: for instance the function $\lambda x \leq 3.x$, whose subtype annotation is 3 (we write $\leq$ for the subtyping relation, that is, the substitute of typing in our calculus), is a valid expression. The latter example is an instance of *bounded quantification* [26] where the term 3 is the singleton type $\{3\}$ and the subtyping relation is akin to the subset relation.

By replacing the typing relation with a more general subtyping relation, PSS greatly increases expressivity. Since terms and types belong to the same syntactic kind, PSS naturally subsumes dependent types and higher-order subtyping. This unification enables a host of advanced language features for extensibility, genericity, and efficiency, including virtual types, recursive types, deep mixin composition, feature-oriented programming, bounded quantification, and partial evaluation [19, 32, 18].

In the original work on PSS, Hutchins introduces a *declarative* system with two relations: one for equivalence (akin to β-equivalence) and another for subtyping, both of which are defined with transitive closure. The proof of type safety requires a *type preservation* property, which states that the type of a program is preserved under evaluation. This property, in turn, hinges on a critical inversion lemma: any two functions in the subtyping relation must have equivalent subtype annotations. This lemma cannot be proven directly in the declarative system due to the presence of transitivity, which introduces intermediate terms that may not be structurally related to the functions being compared. Proving type preservation thus amounts to showing that transitivity is an admissible rule in the system.

In order to prove transitivity elimination, Hutchins proposes an equivalent syntax-directed, *algorithmic* system with two different notions of reduction: an *equivalence reduction* that models β-reduction, and a *subtyping reduction* that models small-step subtyping, where subtyping reduction subsumes equivalence reduction. For short, we may write "reduction" for the β-reduction, and "promotion" for the subtyping reduction. The declarative subtyping relation is shown to be equivalent to the combination of subtyping and equivalence reduction sequences as depicted below.

$$u \leq t \quad \text{iff} \quad \text{exists } v \text{ such that} \qquad \begin{array}{ccc} t & \xrightarrow{\equiv}\!\!\!\!\rightarrow & v \\ & & \leq\uparrow \\ & & u \end{array}$$

In the algorithmic setting, transitivity can be shown to be admissible if these two notions of reduction commute. Diagrammatically, commutativity ensures that a transitive derivation which forms a "stair" of reduction and promotion steps can be flattened into a single angle.

Despite his efforts, Hutchins failed to prove that commutativity holds in his algorithmic system, which prevented him from proving transitivity elimination and type safety. The culprit of this failure is exemplified by a very elementary case (depicted below), which involves the reduction of a redex and the promotion of the formal parameter in the redex's abstraction to its annotation.

$$\begin{array}{ccc} (\lambda x \leq t.t)\, v & \xrightarrow{\equiv} & t \\ \leq\uparrow & & ?\,\vdots \\ (\lambda x \leq t.x)\, v & \xrightarrow{\equiv} & v \end{array}$$

The redex $(\lambda x \leq t.x)v$ on the bottom-left corner of the diagram is reduced (in the horizontal) to v, and promoted (in the vertical) to $(\lambda x \leq t.t)v$. It is then unclear how to complete the right edge of the diagram in a single step.

To tackle this problem, Hutchins resorted to simultaneous reduction and to the *decreasing diagrams* technique of [29], but he failed to assign depths to each reduction step so as to show that reduction decreases in the way prescribed by the decreasing diagrams technique, because β-reduction increases this depth. Hutchins himself explains this failure in detail in his PhD thesis (Section 2.7: Confluence and commutativity)[18].

We reformulate the PSS theory by providing an alternative version of the algorithmic system with a more fine-grained notion of subtyping, in which we can now prove the commutativity result. Our version, which we dub *Machine-Based PSS* (MPSS for short), uses a continuation stack mechanism, reminiscent of the Krivine Abstract Machine (KAM) [20, 1] to track operands passed to abstractions. This mechanism enables a direct proof of commutativity. Indeed, by explicitly managing operands on a stack, the subtyping relation in MPSS exposes all the intermediate terms that are needed to complete the commutativity diagram directly, without resorting to complex techniques like decreasing diagrams. For instance, in our system, the subtyping relation $(\lambda x \leq t.x)v \xrightarrow{\leq} (\lambda x \leq t.t)v$ doesn't exist, instead the variable x must first be promoted to the operand v, which later may be promoted to the annotation t. Exposing this intermediate step, essentially disallowing *premature promotion* that exists in Hutchins' system, allows us to complete the above commutativity diagram:

$$\begin{array}{ccc} (\lambda x \leq t.v)\, v & \xrightarrow{\equiv} & v \\ {\scriptstyle\leq}\uparrow & & \uparrow{\scriptstyle\leq} \\ (\lambda x \leq t.x)\, v & \xrightarrow{\equiv} & v \end{array}$$

Our detailed contributions are the following:

- We introduce MPSS, a reformulation of Hutchins' PSS that, among other changes, keeps track of the operands that have been passed at a particular scope by using a mechanism reminiscent of the continuation stack of the KAM [20, 1]. MPSS is a more fine-grained system in the sense that the stack mechanism exposes intermediate terms that were absent in Hutchins' system.
- We prove the commutativity of the subtyping and equivalence reductions of MPSS, which enables transitivity elimination and the inversion lemma. Our proof of commutativity is direct and proceeds by simple structural induction on terms.
- We sketch how a type system can be derived from our alternative system, thanks to transitivity being now admissible.

The remainder of this paper is organised as follows. Section 2 introduces MPSS. Our system is a reformulation of Hutchins' algorithmic framework equipped with a stack mechanism inspired by the Krivine Abstract Machine (KAM) to track operands, alongside two reductions for subtyping and equivalence. Section 3 proves the commutativity of the two reduction relations in MPSS. Section 4 sketches the design of a type system built upon MPSS and discusses the path to proving its type safety. Section 5 and 6 discuss related and future work, and Section 7 concludes.

All the formal proofs of the lemmas and theorems stated in the paper are collected in the appendix of the full version of this paper accessible online at `https://arxiv.org/abs/2407.13882`.

2 Machine-Based Pure Subtype Systems

Machine-Based PSS (MPSS for short) is a reformulation of PSS where the algorithmic reductions are instrumented with a stack mechanism reminiscent of the Krivine Abstract Machine (KAM) [20, 1].

MPSS is an extension of the pure lambda calculus with a constant symbol Top for the most general supertype and with subtype annotations in the formal parameters of abstractions. Assuming a countably infinite set of variables V ranged over by $x, y, \ldots$, the syntax of terms, contexts, and stacks is defined as follows:

$$\begin{array}{rrl} t, u, v, \alpha, \ldots \in \Lambda & ::= & x \mid \mathsf{Top} \mid (\lambda x \leq t.u) \mid (u\, v) \\ \Gamma & ::= & \varepsilon \mid \Gamma, x \leq t \mid \Gamma, x \equiv \alpha \\ s & ::= & \mathrm{nil} \mid \alpha :: s \end{array}$$

A *variable* $x \in \mathrm{V}$ is a term. The term Top is the *most general supertype*, such that any other term is its subtype. An *abstraction* $(\lambda x \leq t.u)$ has a *parameter* x, a *subtype annotation* t, and a *body* u. An *application* $(u\, v)$ has an *operator* u and an *operand* v. The Greek letter α is a metavariable for terms that originate as operands from the stack.

We assume the usual notions of *free* and *bound variables*, and the usual *capture-avoiding substitution* function, denoted by $u[x \backslash v]$, that replaces the free occurrences of variable x in u by v, while avoiding the capture of any bound variable in u. We consider terms that result after the renaming of bound variables to be identical. When needed, we assume that α-equivalence is applied at will to avoid clashing of free variables.

An application of the form $(\lambda x \leq t.u)v$ is a *redex*. Contraction of a redex is modelled by β-reduction, which rewrites $(\lambda x \leq t.u)v$ into $u[x \backslash v]$. Consider the following example of a redex involving the term $\lambda x \leq \mathsf{Top}.x$ for *universal identity*. Since universal identity is a subtype of Top, the self-application of universal identity, i.e. the redex $(\lambda x \leq \mathsf{Top}.x)(\lambda x \leq \mathsf{Top}.x)$, reduces to $x[x \backslash \lambda x \leq \mathsf{Top}.x]$, which is equal to universal identity by the definition of the substitution function. We assume conventional parenthesis-dropping conventions where abstractions associate to the right and applications to the left, and applications bind stronger than abstractions. We define normal forms as terms that are either Top, functions that have a subtype annotation and a body in normal form, or a variable applied to any number of normal forms.

Prevalidity of logical contexts $\boxed{\Gamma \text{ prevalid}}$

$$\frac{}{\varepsilon \text{ prevalid}}\ \textsc{Pv-Emp} \qquad \frac{\Gamma \text{ prevalid} \qquad x \notin \mathrm{dom}(\Gamma) \qquad \mathrm{fv}(t) \subseteq \mathrm{dom}(\Gamma)}{\Gamma, x \leq t \text{ prevalid}}\ \textsc{Pv-Ctx}$$

$$\frac{\Gamma \text{ prevalid} \qquad x \notin \mathrm{dom}(\Gamma) \qquad \mathrm{fv}(\alpha) \subseteq \mathrm{dom}(\Gamma)}{\Gamma, x \equiv \alpha \text{ prevalid}}\ \textsc{Pv-EqA}$$

Prevalidity of extended contexts $\boxed{\Gamma; s \text{ prevalid}}$

$$\frac{\Gamma \text{ prevalid}}{\Gamma; \mathrm{nil} \text{ prevalid}}\ \textsc{Pv-Nil} \qquad \frac{\Gamma; s \text{ prevalid} \qquad \mathrm{fv}(\alpha) \subseteq \mathrm{dom}(\Gamma)}{\Gamma; \alpha :: s \text{ prevalid}}\ \textsc{Pv-Sta}$$

Figure 1 Prevalidity in MPSS.

Our system uses *logical contexts* (*contexts* for short) Γ to store annotations from abstractions encountered within a given scope. A context Γ is a sequence of *subtype annotations* $x \leq t$ and *equivalence annotations* $x \equiv \alpha$. We write $x \lhd t$ to denote either kind of annotation, and throughout this paper the metavariable $\lhd$ designates either $\leq$ or $\equiv$. We use ε for the *empty context* and we write $\Gamma, x \lhd t$ for the operation that appends annotation $x \lhd t$ to the subtype annotations of Γ. By abuse of notation, we write Γ_1, Γ_2 for the concatenation of contexts Γ_1 and Γ_2. We say that x *has subtype* t *in* Γ (written $x \leq t \in \Gamma$) if $x \leq t$ is the rightmost annotation for x in Γ. Respectively, we say that x *is equivalent to* α *in* Γ (written $x \equiv \alpha \in \Gamma$) under similar conditions.

Equivalence reduction $\boxed{\Gamma; s \vdash u \xrightarrow{\equiv} v}$

$$\frac{\Gamma; s \text{ prevalid} \qquad x \equiv \alpha \in \Gamma \qquad \Gamma; s \vdash \alpha \xrightarrow{\equiv} \alpha'}{\Gamma; s \vdash x \xrightarrow{\equiv} \alpha'} \textsc{Me-Pro}$$

$$\frac{\Gamma; s \vdash u \xrightarrow{\equiv} u' \qquad \Gamma; \text{nil} \vdash v \xrightarrow{\equiv} v'}{\Gamma; s \vdash (\lambda x \leq t.u)\, v \xrightarrow{\equiv} u'[x \backslash v']} \textsc{Me-Bet} \qquad \frac{\Gamma; s \text{ prevalid}}{\Gamma; s \vdash \mathsf{Top} \xrightarrow{\equiv} \mathsf{Top}} \textsc{Me-Top}$$

$$\frac{\Gamma; v :: s \vdash u \xrightarrow{\equiv} u' \qquad \Gamma; \text{nil} \vdash v \xrightarrow{\equiv} v'}{\Gamma; s \vdash u\, v \xrightarrow{\equiv} u'\, v'} \textsc{Me-App} \qquad \frac{\Gamma; s \text{ prevalid}}{\Gamma; s \vdash x \xrightarrow{\equiv} x} \textsc{Me-Var}$$

$$\frac{\Gamma; \text{nil} \vdash t \xrightarrow{\equiv} t' \qquad \Gamma, x \leq t; \text{nil} \vdash u \xrightarrow{\equiv} u'}{\Gamma; \text{nil} \vdash \lambda x \leq t.u \xrightarrow{\equiv} \lambda x \leq t'.u'} \textsc{Me-Fun} \qquad \frac{\Gamma; s \text{ prevalid}}{\Gamma; s \vdash \mathsf{Top}\, u \xrightarrow{\equiv} \mathsf{Top}} \textsc{Me-TAp}$$

$$\frac{\Gamma; \text{nil} \vdash t \xrightarrow{\equiv} t' \qquad \Gamma, x \equiv \alpha; s \vdash u \xrightarrow{\equiv} u'}{\Gamma; \alpha :: s \vdash \lambda x \leq t.u \xrightarrow{\equiv} \lambda x \leq t'.u'} \textsc{Me-FOp}$$

Subtyping reduction $\boxed{\Gamma; s \vdash u \xrightarrow{\leq} v}$

$$\frac{\Gamma; s \text{ prevalid} \qquad x \leq t \in \Gamma}{\Gamma; s \vdash x \xrightarrow{\leq} t} \textsc{Ms-Pro} \qquad \frac{\Gamma; s \text{ prevalid}}{\Gamma; s \vdash u \xrightarrow{\leq} \mathsf{Top}} \textsc{Ms-Top}$$

$$\frac{\Gamma; s \text{ prevalid} \qquad \Gamma; s \vdash u \xrightarrow{\equiv} v}{\Gamma; s \vdash u \xrightarrow{\leq} v} \textsc{Ms-Equ} \qquad \frac{\Gamma; v :: s \vdash u \xrightarrow{\leq} u'}{\Gamma; s \vdash u\, v \xrightarrow{\leq} u'\, v} \textsc{Ms-App}$$

$$\frac{\Gamma, x \leq t; \text{nil} \vdash u \xrightarrow{\leq} u'}{\Gamma; \text{nil} \vdash \lambda x \leq t.u \xrightarrow{\leq} \lambda x \leq t.u'} \textsc{Ms-Fun} \qquad \frac{\Gamma, x \equiv \alpha; s \vdash u \xrightarrow{\leq} u'}{\Gamma; \alpha :: s \vdash \lambda x \leq t.u \xrightarrow{\leq} \lambda x \leq t.u'} \textsc{Ms-FOp}$$

Figure 2 Equivalence and subtyping reduction in MPSS.

In order to accommodate the stack mechanism, we introduce *extended contexts* $\Gamma; s$, which are logical contexts Γ coupled with a *continuation stack* s. Extended contexts adapt the algorithmic reduction relations to the stack mechanism of MPSS.

Figure 1 introduces the *prevalidity* condition on contexts that ensures each annotation in a context mentions a different variable, to avoid clashing of variable names. We say that x *is in the domain of* Γ (written $x \in \text{dom}(\Gamma)$) if and only if an annotation for variable x occurs

in Γ, i.e., there exists a term t (or a term α) such that $x \leq t \in \Gamma$ (or $x \equiv \alpha \in \Gamma$). The empty context ε is prevalid (Rule Pv-Emp in Figure 1). Adding a subtype annotation $x \leq t$, or an equivalence annotation $x \equiv \alpha$ to a prevalid context Γ yields a prevalid context if the free variables of t (respectively, α) are in the domain of Γ, and if x is not already in the domain of Γ (Rules Pv-Ctx and Pv-EqA). *Prevalidity for extended contexts* (also defined in Figure 1) stipulates that a prevalid context coupled with an empty stack is prevalid (Rule Pv-Nil), and that an extended context $\Gamma; \alpha :: s$ is prevalid if the free variables of α are in the domain of Γ (Rule Pv-Sta). We now present our reduction relations.

Figure 2 presents our equivalence ($\xrightarrow{\equiv}$) and subtyping ($\xrightarrow{\leq}$) reductions. We do away with a declarative system like Hutchins' system, and our MPSS resembles his algorithmic system (p. 58 of [18]) where we adopt simultaneous reduction upfront in order to simplify the commutativity proofs (resembling p. 74 of [18]), and where contexts are coupled with a stack in order to accommodate the stack mechanism. The equivalence reduction $\xrightarrow{\equiv}$ models a reflexive, small-step, simultaneous β-reduction. Rule Me-Bet performs β-reduction, while Rule Me-Pro replaces a variable by its equivalent operand from the context. Rules Me-Var and Me-Top allow the reduction relation to be reflexive. Rule Me-TAp is a technical device from Hutchins' formulation in [18], whose objective is to accommodate terms with shape $\mathsf{Top}\, u$, which may pop up in reductions, and which cannot be β-reduced by Rule Me-Bet (see p. 59 of [18] for a further discussion on this issue). Me-App propagates promotion through unapplied applications by pushing the operand onto the stack and proceeding with the operator, mimicking KAM's stack management. Rules Me-Fun and Me-FOp distinguish between unapplied and applied abstractions: Me-Fun propagates promotion by enlarging the context with the abstraction's annotation and proceeding with the body, and Me-FOp promotes the body of an applied abstraction by taking the operand from the stack and enlarging the context with an equivalence annotation.

The subtyping reduction $\xrightarrow{\leq}$ defines promotion in a stack-aware manner. Rule Ms-Pro promotes a variable to its subtype annotation, and Rule Ms-Top promotes any term to Top. Rule Ms-Equ subsumes equivalence reduction. Rules Ms-App, Ms-Fun and Ms-FOp are the counterparts of the equivalence reduction rules Me-App, Me-Fun and Me-FOp respectively. Because Hutchins' declarative system is not contravariant, its algorithmic system, which we took inspiration from, does not change the type annotation of abstractions: in his system, there is no rule $\lambda x \leq t.u \xrightarrow{\leq} \lambda x \leq t'.u$ with assumption $t' \xrightarrow{\leq} t$. The main motivation for this design choice is to avoid the undecidability of the related typechecking, as described in [25], and to avoid complicating the meta-theory even more. Our system follows the same principle, both Rules Ms-Fun and Ms-FOp do not change the type annotation, and in general there is no reduction rule to change the type annotations, except from equivalence reductions that are allowed. Handling the interaction between the stack and the context is the cornerstone to prove the commutativity result, and to avoid the premature promotion issues that plagued Hutchins' approach.

To illustrate our stack mechanism, consider the following subtyping reduction derivation (prevalidity derivations are elided for brevity):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Gamma;\, s \text{ prevalid}} \qquad \dfrac{\dfrac{\dots}{\Gamma \text{ prevalid}} \qquad \dfrac{\dots}{\Gamma; \text{nil} \vdash v \xrightarrow{\equiv} v}}{\Gamma, x \equiv v; \text{nil} \vdash x \xrightarrow{\equiv} v}\ \textsc{Me-Pro}}{\Gamma, x \equiv v; \text{nil} \vdash x \xrightarrow{\leq} v}\ \textsc{Ms-Equ}}{\Gamma; v :: \text{nil} \vdash (\lambda x \leq t.x) \xrightarrow{\leq} (\lambda x \leq t.v)}\ \textsc{Ms-FOp}}{\Gamma; \text{nil} \vdash (\lambda x \leq t.x)v \xrightarrow{\leq} (\lambda x \leq t.v)v}\ \textsc{Ms-App}$$

Rule Ms-App pushes the operand v onto the stack. Inside the abstraction, the variable x is not promoted to its type t but is instead reduced to the operand v retrieved from the context, which was placed there by Ms-FOp after popping from the stack. Keeping track of operands prevents the premature promotion that caused Hutchins' commutativity proof to fail.

As in Hutchins' PSS, we define a subtyping relation $\leq$, on which we wish to prove that transitivity is admissible in the next section, as follows, with $\longrightarrow\!\!\!\!\!\rightarrow$ being the transitive closure of $\longrightarrow$:

$$u \leq t \quad \text{iff} \quad \text{exists } v \text{ such that} \qquad \begin{array}{ccc} t & \xrightarrow{\equiv}\!\!\!\!\!\rightarrow & v \\ & & \leq\uparrow\uparrow \\ & & u \end{array}$$

3 Transitivity elimination

As explained in the introduction, proving transitivity elimination requires us to show that the equivalence ($\xrightarrow{\equiv}$) and subtyping ($\xrightarrow{\leq}$) reductions commute, which we do below in this section. In order to state the commutativity result, we first define a reduction relation on extended contexts, which captures the evolution of annotations during reduction.

Reduction of extended context $\boxed{\Gamma; s \rightarrowtail \Gamma'; s'}$

$$\frac{\Gamma; s \rightarrowtail \Gamma'; s' \qquad \Gamma; \mathrm{nil} \vdash t \xrightarrow{\equiv} t'}{\Gamma, x \lhd t; s \rightarrowtail \Gamma', x \lhd t'; s'} \text{ Ct-Ann} \qquad \frac{\Gamma; s \rightarrowtail \Gamma'; s' \qquad \Gamma; \mathrm{nil} \vdash \alpha \xrightarrow{\equiv} \alpha'}{\Gamma; \alpha :: s \rightarrowtail \Gamma'; \alpha' :: s'} \text{ Ct-Stk}$$

With this definition, we can now state our main commutativity result.

▶ **Lemma 1** ($\xrightarrow{\leq}$ and $\xrightarrow{\equiv}$ strongly commute). *Let $\Gamma; s$ be an extended context. Let t_0, t_1, and t_2 be terms. If $\Gamma; s \vdash t_0 \xrightarrow{\equiv} t_1$ and $\Gamma; s \vdash t_0 \xrightarrow{\leq} t_2$, then for any extended context $\Gamma'; s'$ such that $\Gamma; s \rightarrowtail \Gamma'; s'$, there exists a term t_3, such that $\Gamma; s \vdash t_2 \xrightarrow{\equiv} t_3$ and $\Gamma'; s' \vdash t_1 \xrightarrow{\leq} t_3$.*

Diagrammatically, the existence of the solid arrows implies the existence of the dashed arrows:

$$\begin{array}{ccc} t_2 & \dashrightarrow^{\equiv} & t_3 \\ \leq\uparrow & & \leq\uparrow \\ t_0 & \xrightarrow{\equiv} & t_1 \end{array}$$

The generalisation over all possible resulting contexts $\Gamma'; s'$ is crucial for the inductive proof. Indeed, consider the following commutation diagram, with context $\Gamma; \mathrm{nil}$:

$$\begin{array}{ccc} \lambda x \leq t.t & \dashrightarrow^{\equiv} & ? \\ \leq\uparrow & & \leq\uparrow \\ \lambda x \leq t.x & \xrightarrow{\equiv} & \lambda x \leq t'.x \end{array}$$

The only possible arrow that could exist, apart from a promotion of the whole term to Top, is a promotion of the variable x to its subtype annotation t'. Therefore, the top right term is $\lambda x \leq t'.t'$, and the diagram is completed as follows:

$$\begin{array}{ccc} \lambda x \leq t.t & \xrightarrow{\equiv} & \lambda x \leq t'.t' \\ \leq\uparrow & & \leq\uparrow \\ \lambda x \leq t.x & \xrightarrow{\equiv} & \lambda x \leq t'.x \end{array}$$

However, in the proof of this theorem, everything works by induction on the terms. Therefore, starting from the original diagram, we apply the induction hypothesis on the following diagram, with context $\Gamma, x \leq t; \text{nil}$:

$$\begin{array}{ccc} t & \overset{\equiv}{\dashrightarrow} & ? \\ {\scriptstyle\leq}\uparrow & & {\scriptstyle\leq}\uparrow \\ x & \xrightarrow{\equiv} & x \end{array}$$

And as we have seen above, we must complete this diagram with t' being the top right term, as the only possible completion of the original diagram is the term $\lambda x \leq t'.t'$. Therefore, the completed diagram by the induction hypothesis must be:

$$\begin{array}{ccc} t & \xrightarrow{\equiv} & t' \\ {\scriptstyle\leq}\uparrow & & {\scriptstyle\leq}\uparrow \\ x & \xrightarrow{\equiv} & x \end{array}$$

Hence the requirement that the context of the right arrow must be $\Gamma, x \leq t'; \text{nil}$, and therefore the formulation of the theorem with a generalisation over contexts.

Similarly, our equivalence reduction relation satisfies the diamond property, on which the result above rests. A similar generalisation over contexts is required, with two different reduced extended contexts however, as both the top and the right edge can now do variable promotions thanks to Rule Me-Pro. The pathological case is the following, with context $\Gamma; \text{nil}$:

$$\begin{array}{ccc} (\lambda x \leq t.\alpha_0\, x)\, \alpha_2 & \xrightarrow{\equiv} & (\lambda x \leq t.\alpha_1\, \alpha_2)\, \alpha_3 \\ {\scriptstyle\equiv}\uparrow & & {\scriptstyle\equiv}\uparrow \\ (\lambda x \leq t.x\, x)\, \alpha_0 & \xrightarrow{\equiv} & (\lambda x \leq t.x\, \alpha_0)\, \alpha_1 \end{array}$$

To solve this pathological case in the induction, one of the sub-induction calls is the following diagram:

$$\begin{array}{ccc} \alpha_0\, x & \xrightarrow{\equiv} & \alpha_1\, \alpha_2 \\ {\scriptstyle\equiv}\uparrow & & {\scriptstyle\equiv}\uparrow \\ x\, x & \xrightarrow{\equiv} & x\, \alpha_0 \end{array}$$

And similarly as above, the top edge is completed with extended context $\Gamma, x \equiv \alpha_2; \text{nil}$, whereas the right edge is completed with extended context $\Gamma, x \equiv \alpha_1; \text{nil}$. Both these contexts satisfy the requirement of the theorem (that is, we have $\Gamma, x \equiv \alpha_0; \text{nil} \rightarrowtail \Gamma, x \equiv \alpha_2; \text{nil}$ and $\Gamma, x \equiv \alpha_0; \text{nil} \rightarrowtail \Gamma, x \equiv \alpha_1; \text{nil}$).

▶ **Lemma 2** ($\xrightarrow{\equiv}$ has the diamond property). *Let $\Gamma_0; s_0$ be an extended context. Let t_0, t_1, and t_2 be terms. If $\Gamma_0; s_0 \vdash t_0 \xrightarrow{\equiv} t_1$ and $\Gamma_0; s_0 \vdash t_0 \xrightarrow{\equiv} t_2$, then for any extended contexts $\Gamma_1; s_1$ and $\Gamma_2; s_2$ such that $\Gamma_0; s_0 \rightarrowtail \Gamma_1; s_1$ and $\Gamma_0; s_0 \rightarrowtail \Gamma_2; s_2$, there exists a term t_3 such that $\Gamma_1; s_1 \vdash t_1 \xrightarrow{\equiv} t_3$ and $\Gamma_2; s_2 \vdash t_2 \xrightarrow{\equiv} t_3$.*

Moreover, for any variable x, if in the derivation of $\Gamma_0; s_0 \vdash t_0 \xrightarrow{\equiv} t_1$ (respectively $\Gamma_0; s_0 \vdash t_0 \xrightarrow{\equiv} t_2$) there isn't an application of the Rule Me-Pro that makes a promotion of variable x, then in the derivation $\Gamma_2; s_2 \vdash t_2 \xrightarrow{\equiv} t_3$ (respectively $\Gamma_1; s_1 \vdash t_1 \xrightarrow{\equiv} t_3$) there won't be an application of the Rule Me-Pro that makes a promotion of variable x.

Diagrammatically:

$$\begin{array}{ccc} t_2 & \overset{\equiv}{\dashrightarrow} & t_3 \\ {\scriptstyle\equiv}\uparrow & & {\scriptstyle\equiv}\uparrow \\ t_0 & \xrightarrow{\equiv} & t_1 \end{array}$$

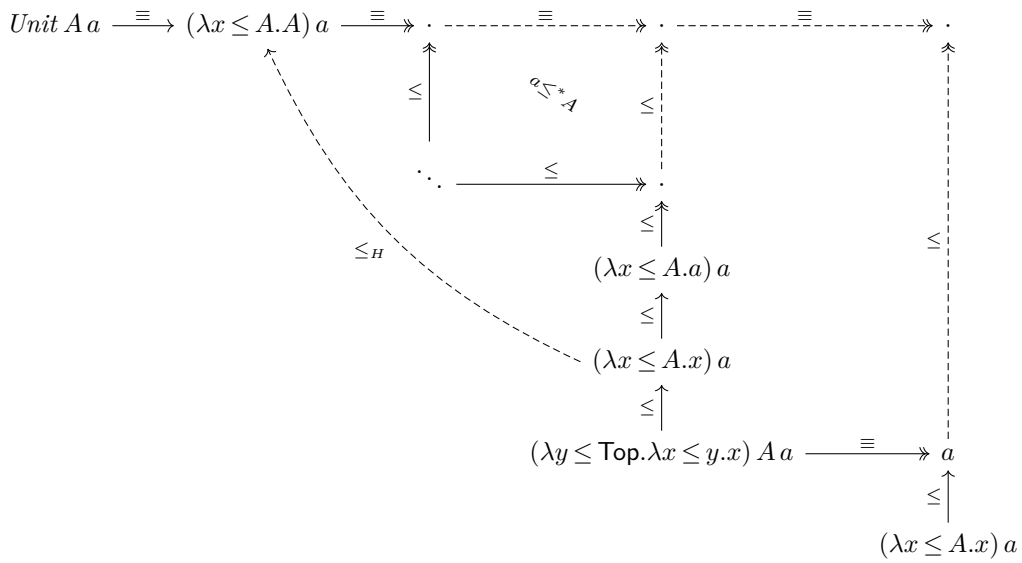

Figure 3 Derivation of $\Gamma; \mathrm{nil} \vdash (\lambda x \leq A.x)\, a \leq \mathit{Unit}\, A\, a$.

With commutativity and the diamond property established, we can prove that our system enjoys the transitivity elimination property.

▶ **Theorem 3** (Transitivity is admissible). *Let $\Gamma; s$ be an extended context. Let u and v be terms. If $\Gamma; s \vdash u \leq^* v$ then $\Gamma; s \vdash u \leq v$.*

In order to illustrate our result, and highlight the differences between PSS and MPSS, we detail below an example of an application of Theorem 3:

Example of transitivity elimination

Consider the Church encodings for type "Unit" (the type with only one inhabitant, term *Unit* below), the encoding of the constructor "single" of type Unit (term *single* below), and an arbitrary well-formed subtype A that encodes some interesting type. Let the identity over the elements of subtype A be the term id_A below, and consider an arbitrary well-formed term a of subtype A such that some derivation $\varepsilon; \mathrm{nil} \vdash a \leq^* A$ exists which may possibly be as lengthy or complicated as it may be.

$$\begin{aligned} \mathit{Unit} &= \lambda y \leq \mathsf{Top}.\lambda x \leq y.y \\ \mathit{single} &= \lambda y \leq \mathsf{Top}.\lambda x \leq y.x \\ A &= \langle \text{arbitrary subtype} \rangle \\ id_A &= \lambda x \leq A.x \\ a &= \langle \text{arbitrary subtype of } A \rangle \end{aligned}$$

Consider the following derivation:

$$\Gamma; \mathrm{nil} \vdash id_A\, a \leq \mathit{single}\, A\, a \leq^* \mathit{Unit}\, A\, a$$

Figure 3 details all the elementary steps (diagrammatically) of the above algorithmic derivation. We illustrate the proof of Theorem 3 over this diagram.

Theorem 3 ensures that the transitivity step collecting the two derivations $\Gamma; \mathrm{nil} \vdash \mathit{id}_A\, a \leq \mathit{single}\, A\, a$ and $\Gamma; \mathrm{nil} \vdash \mathit{single}\, A\, a \leq^* \mathit{Unit}\, A\, a$ can be removed, obtaining a transitivity-free derivation

$$\Gamma; \mathrm{nil} \vdash \mathit{id}_A\, a \leq \mathit{Unit}\, A\, a$$

Our constructive proof produces a transitivity-free derivation of $\Gamma; \mathrm{nil} \vdash \mathit{single}\, A\, a \leq^* \mathit{Unit}\, A\, a$ (i.e., flattens it into $\Gamma; \mathrm{nil} \vdash \mathit{single}\, A\, a \leq \mathit{Unit}\, A\, a$ witnessed by the straight vertical and horizontal paths in the middle of the diagram), and then applies the commutativity theorem to complete the upper-right corner of the diagram, obtaining the transitivity-free derivation $\Gamma; \mathrm{nil} \vdash (\lambda x \leq A.x)\, a \leq \mathit{Unit}\, A\, a$. This proof is possible in our algorithmic system thanks to the operand stack and Rule Ms-FOp which exposes the term $(\lambda x \leq A.a)\, a$, and to the fact that the derivation $\Gamma; \mathrm{nil} \vdash (\lambda x \leq A.a)\, a \leq^* (\lambda x \leq A.A)\, a$ in the middle of the diagram preserves the possibly convoluted derivation $\Gamma, x \equiv a; \mathrm{nil} \vdash a \leq^* A$ as a sub-derivation (marked with $\ddots$). Although preserving this convoluted derivation seems at first glance counter-intuitive, it gives the opportunity to apply the induction hypothesis of the Theorem 3 to the term $(\lambda x \leq A.a)\, a$, which will flatten the convoluted derivation. Contrary to ours, Hutchins' attempts at proving that transitivity is admissible reduce the complexity of the intermediate step by avoiding the convoluted derivation upfront (by directly taking the reduction $(\lambda x \leq A.x)\, a \xrightarrow{\leq} (\lambda x \leq A.A)\, a$ which promotes the formal parameter a of the redex to its annotation A).

But this premature promotion of a formal parameter to its type annotation is in fact counter-productive, which manifests in the failure of Hutchins' proof of the commutativity theorem, where the premature promotion prevents one from completing the diagram for redexes with a transitivity-free right edge.

This result was the missing cornerstone in Hutchins' original theory, and its proof here is necessary for establishing type safety. In the next section, we present what can be a type-safe theory based on our MPSS.

4 Towards a Type-Safe System

Thanks to the above commutative system, we can now sketch the design of a type-safe system. The static semantics for this system is defined by a *well-formedness* judgment, presented in Figure 4, which is the static condition for type safety. Rules Wf-PrS and Wf-PrE ensure that a variable is well-formed if and only if it occurs in a prevalid context (respectively, in a subtype or an equivalence annotation). Rule Wf-Top ensures that Top is universally well-formed. Rule Wf-Fun ensures that an abstraction is well-formed if and only if it has a well-formed annotation, and its body is well-formed in a context enlarged with the abstraction annotation. Finally, Rule Wf-App ensures that an application is well-formed if and only if it has an operator that is a well-subtype of a function, and an operand that is a well-subtype of the operator's subtype annotation.

Well-formedness depends on the *well-subtyping* relation $\leq_{\mathrm{wf}}$, which in turn subsumes *well-equivalence* $\equiv_{\mathrm{wf}}$. The rules for well-subtyping connect the static system to our dynamic reduction semantics. For example, Rule Ws-Lf2 states that v is a well-subtype of t if v promotes to some v' that is itself a well-subtype of t.

We define type safety as *progress* (well-formed terms either are in normal form or can be reduced), and *preservation* (subtyping relations between terms are preserved under reduction by the operational semantics below). We define the operational semantics as a context-based reduction semantics [13]: the evaluation $\mapsto$ is defined as the congruence closure (Rule Os-Con and evaluation contexts C) of β-reduction (Rule Os-Bet).

Term well-formedness $\boxed{\Gamma \vdash t \text{ wf}}$

$$\frac{\Gamma \text{ prevalid} \qquad x \leq t \in \Gamma}{\Gamma \vdash x \text{ wf}} \textsc{Wf-PrS} \qquad \frac{\Gamma \text{ prevalid} \qquad x \equiv \alpha \in \Gamma}{\Gamma \vdash x \text{ wf}} \textsc{Wf-PrE}$$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash \mathsf{Top} \text{ wf}} \textsc{Wf-Top} \qquad \frac{\Gamma, x \leq t \vdash u \text{ wf} \qquad \Gamma \vdash t \text{ wf}}{\Gamma \vdash \lambda x \leq t.u \text{ wf}} \textsc{Wf-Fun}$$

$$\frac{\Gamma \vdash u \leq^*_{\text{wf}} \lambda x \leq t.\mathsf{Top} \qquad \Gamma \vdash v \leq^*_{\text{wf}} t}{\Gamma \vdash u\, v \text{ wf}} \textsc{Wf-App}$$

Transitive well-subtyping $\boxed{\Gamma \vdash v \lhd^*_{\text{wf}} t}$

$$\frac{\Gamma \vdash v \text{ wf} \qquad \Gamma \vdash v \lhd_{\text{wf}} t \qquad \Gamma \vdash t \text{ wf}}{\Gamma \vdash v \lhd^*_{\text{wf}} t} \textsc{Ws-Sub}$$

$$\frac{\Gamma \vdash v \lhd^*_{\text{wf}} u \qquad \Gamma \vdash u \text{ wf} \qquad \Gamma \vdash u \lhd^*_{\text{wf}} t}{\Gamma \vdash v \lhd^*_{\text{wf}} t} \textsc{Ws-Trs}$$

Well-subtyping $\boxed{\Gamma \vdash u \lhd_{\text{wf}} t}$

$$\frac{\Gamma \text{ prevalid}}{\Gamma \vdash t \lhd_{\text{wf}} t} \textsc{Ws-Rfl} \qquad \frac{\Gamma; \text{nil} \vdash v \xrightarrow{\equiv} v' \qquad \Gamma \vdash v' \lhd_{\text{wf}} t}{\Gamma \vdash v \lhd_{\text{wf}} t} \textsc{Ws-Lf1}$$

$$\frac{\Gamma \vdash v \text{ wf} \qquad \Gamma; \text{nil} \vdash v \xrightarrow{\leq} v' \qquad \Gamma \vdash v' \text{ wf} \qquad \Gamma \vdash v' \leq_{\text{wf}} t}{\Gamma \vdash v \leq_{\text{wf}} t} \textsc{Ws-Lf2}$$

$$\frac{\Gamma \vdash v \lhd_{\text{wf}} t' \qquad \Gamma; \text{nil} \vdash t \xrightarrow{\equiv} t'}{\Gamma \vdash v \lhd_{\text{wf}} t} \textsc{Ws-Rgh}$$

Figure 4 Well-subtyping in MPSS.

Operational Semantics $\boxed{t \mapsto t'}$

$$\frac{t \mapsto t'}{\mathsf{C}[t] \mapsto \mathsf{C}[t']} \textsc{Os-Con} \qquad \frac{}{(\lambda x \leq t.u)v \mapsto u[x \backslash v]} \textsc{Os-Bet}$$

$$\mathsf{C} ::= \square \mid (\lambda x \leq \mathsf{C}.t) \mid (\lambda x \leq t.\mathsf{C}) \mid (\mathsf{C}\, t) \mid (t\, \mathsf{C})$$

Well-formedness depends on the *well-subtyping* relation defined in Figure 4. Well-subtyping ($\leq_{\text{wf}}$) subsumes *well-equivalence* ($\equiv_{\text{wf}}$). We use the meta-variable $\lhd$ and define ($\lhd_{\text{wf}}$), which captures both relations, and similarly for their transitive closure ($\lhd^*_{\text{wf}}$). By a slight abuse of language, and to simplify the text, we write "well-subtyping" to refer to the meta-notation ($\lhd_{\text{wf}}$) in the paragraph that follows.

Figure 4 introduces *transitive well-subtyping*, the straightforward transitive closure of the well-subtyping relation (Rules Ws-Sub and Ws-Trs). Next in the figure is *well-subtyping*, which is defined similarly to the diagrammatic subtyping relation $\leq$ defined in Section 2, but with extra requirements on the well-formedness of the terms involved. Rule Ws-Rfl

states that well-subtyping is reflexive. Rule Ws-Lf1 ensures that v is a well-subtype of t if and only if v' is a well-subtype of t and v reduces to v'. Rule Ws-Lf2 ensures that v is a well-subtype of t if and only if v' is a well-subtype of t and v promotes to v', but with the extra requirements that both v and v' are well-formed, whose role is explained in a minute. Rule Ws-Rgh ensures that v is a well-subtype of t if and only if v is a well-subtype of t' and t reduces to t'.

Thanks to these definitions, we may define type safety in our calculus as the following progress and preservation theorems.

▶ **Theorem 4** (Progress). *Let t be a term. For every logical context Γ, if $\Gamma \vdash t$ wf then either t is in normal form, or there exists a term t' such that $t \mapsto t'$.*

▶ **Theorem 5** (Preservation). *Let Γ be a logical context. Let t, t', and u be terms. If $\Gamma \vdash t \leq^*_{\text{wf}} u$ and $t \mapsto t'$, then $\Gamma \vdash t' \leq^*_{\text{wf}} u$.*

The proof of type preservation (Theorem 5) rests on showing that reduction preserves term well-formedness. Lemma 6 establishes this fact.

▶ **Lemma 6** (Evaluation preserves well-formedness). *Let Γ be a logical context. Let t and t' be terms such that $t \mapsto t'$ and such that $\Gamma \vdash t$ wf. We have $\Gamma \vdash t'$ wf.*

A crucial case in the proof of Lemma 6 is β-reduction, where a well-formed application $(\lambda x \leq t.u)\, \alpha$ reduces to $u[x \backslash \alpha]$. To handle this case, we need to show that well-formedness is preserved by substitution. The following lemma provides this guarantee, where the premise $\Gamma \vdash \alpha \leq^*_{\text{wf}} t$ comes from the well-formedness of the original term.

▶ **Lemma 7** (Substitution preserves well-formedness). *Let Γ and Γ' be logical contexts, and let t, u and α be terms. Let x be a variable. If $\Gamma, x \leq t, \Gamma' \vdash u$ wf, and $\Gamma \vdash \alpha \leq^*_{\text{wf}} t$, then $\Gamma, \Gamma'[x \backslash \alpha] \vdash u[x \backslash \alpha]$ wf.*

Our subtyping relation in MPSS is context-dependent, unlike in PSS where it isn't, which implies that Lemma 7 relies on the conjecture collected below, which states that the well-subtyping relation is independent of the surrounding program structure, on certain contexts named *covariant contexts* defined as follows:

$$\mathsf{Co} ::= \Box \mid (\lambda x \leq t.\mathsf{Co}) \mid (\mathsf{Co}\, t)$$

▶ **Conjecture 8** (Well-subtyping is context independent). *Let Γ be a logical context and u and t be terms such that $\Gamma \vdash u \leq^*_{\text{wf}} t$. Let Co be a covariant context such that both $\mathsf{Co}[u]$ and $\mathsf{Co}[t]$ are well-formed in Γ. We conjecture that $\Gamma \vdash \mathsf{Co}[u] \leq^*_{\text{wf}} \mathsf{Co}[t]$.*

We believe that the conjecture holds with the additional well-formedness requirements that both v and v' are well-formed in Rule Ws-Lf2 above.

All in all, type safety, which is the combination of progress (Theorem 4) and preservation (Theorem 5), holds under the assumption that Conjecture 8 holds. The proof that both these theorems hold under the assumption that Conjecture 8 holds can be found in the appendix of the full version online at `https://arxiv.org/abs/2407.13882`.

Establishing this conjecture is the final step towards a complete proof of type safety. The sketch of a complete type system provided in this section demonstrates a path forward, built upon the solid foundation of our commutativity result. Future work could either adopt this sketch and complete the proof of type safety, or develop a different type system that leverages the commutativity result established in Section 3 to establish type safety in a different way.

5 Related work

PSS naturally subsumes dependent types and higher-order subtyping, so we comment on both ideas.

There are many works related to dependent types. For instance in Cayenne [4], Idris [7] or Haskell [21], dependent types are an extension of the language, in contrast to our case where dependent types are built-in as there is no distinction between terms and types.

Other languages mainly use dependent types for theorem proving, for instance Coq [12] or Agda [6]. Our language is not strongly normalising, and our main goal is not theorem proving as we cannot use the language as a proof system, since applying the Curry-Howard correspondence to PSS in a consistent way is still an open question.

To accommodate the Curry-Howard correspondence, one possible approach is to distinguish between potentially non-terminating terms and terminating terms, either with monads or via other means [23, 24]. λ^θ is a calculus that explores this idea [9, 10, 28]. It combines proofs and programs into a single language that distinguishes two fragments, a logical fragment where every term is strongly normalising that is suited to write proofs, and a programmatic fragment that is Turing complete and lacks termination. Such systems allow so-called *freedom of speech*, which is the ability to write (terminating) proofs about non-terminating terms.

Subtyping extensions of type systems have been considered. In 1997, Chen proposed $\lambda_{C_\leq}$ [11], an extension of the calculus of constructions λ_C with subtyping. However, Chen's $\lambda_{C_\leq}$ supports neither top types nor bounded quantification as we do in PSS.

Various extensions of System F, in order to equip it with subtyping, have also been considered in the literature. Within these we find System $\mathrm{F}_\leq$ [8] and System $\mathrm{F}^\omega_\leq$ [26], which respectively add subtyping and higher-order subtyping capabilities. PSS subsumes the version of System $\mathrm{F}_\leq$ without contravariant arrow-types [18].

The origin of mixing terms and types goes back to the introduction of pure type systems [22], which generalises the seminal works on lambda calculi with types by [5]. These systems have the ability to unify types and terms under the same kind, but there is a typing relation instead of a subtyping one. Moreover, in some of these pure type systems, the type $\bot$ is inhabited. This gives rise to the well-known Girard's paradox (see [17]). In our system, there is not a primitive $\bot$ subtype, but we could encode such a subtype to be inhabited by itself, and that would be unproblematic since our aim is not to use PSS as a theorem prover.

Systems where terms can be used in certain ways as types have been considered. Aspinall studied the combination of subtyping with singleton types [3]. Singleton types are close to what we study, as in our calculus, terms can be seen as singleton types of this unique value. We have for instance, for every term t, $\varepsilon \vdash t \leq t$, which would be translated to $\vdash t : \{t\}$ in a system with singleton types.

Other attempts to unify typing and subtyping exist. A more conservative attempt that tries to unify typing and subtyping is [31]. This work has two major differences with our approach. First, in [31] the subtyping relation still tracks types (their unification of subtyping and typing is more conservative than ours). Second, in PSS functions and function (sub)types are defined with the same syntax, and therefore they are both instances of the same construct, which is not the case in [31].

Because many interesting features described in [18] are related to modules, other type systems especially related to objects have been studied. Our work shares foundational goals with the Dependent Object Types (DOT) calculus, the theoretical basis for Scala 3 [2, 27]. Both PSS and DOT aim to create powerful, unified type systems that support advanced

modularity, and provide a formal basis for solving challenges akin to the expression problem, the tag-elimination problem, and others. DOT achieves this, among other features, through path-dependent types, where the type of a member, $p.T$, depends on the term p. Because DOT allows objects to contain types as members, it provides a form of dependent typing. PSS takes a more radical approach by collapsing types and terms into a single syntactic category, where the traditional typing relation (:) is completely replaced by a universal subtyping relation ($\leq$). This design is what enables PSS's key features described in Sections 3 and 4 of [18].

6 Future work

As described in Section 4, it is left as future work to define a type system that is built upon MPSS and to prove its type safety. Another interesting area to explore is the link between our system and Hutchins' system, to determine if both well-formed notions are equivalent.

Once this system is defined, it is also left as future work to obtain an algorithm that typechecks terms in MPSS. Hutchins describes an algorithm for his system in [18], but one should adapt his algorithm to this new system, and study upon which conditions it terminates. One possible approach is to use the operational relevance of terms, for which there exist some arguments that lead them to terminate. We leave as future work to explore other practical algorithms which terminate on operationally relevant terms.

Another venue of future work is to study whether PSS and MPSS are Turing-complete. To the best of our knowledge, whether PSS is Turing-complete has not been formally proven yet. The argument in favor of Turing-completeness rests on the existence of a (well-formed) looping combinator in PSS, which would consist of a family of terms L_n such that $L_n\, f$ β-reduces to $f(L_{n+1}\, f)$ for any f. Such a looping combinator exists for System $\lambda*$ [16], which can be encoded into Hutchins' PSS thanks to the encoding of System $\lambda*$ in Hutchins' PSS, provided on page 47 of [18]. In order to establish Turing-completeness of PSS, one should now adapt the argument made by [14] in Section 2 or the argument made in [15]. There is, however, no simple known expression of a looping combinator in System $\lambda*$. The looping combinator from [16] can only be derived mechanically, and working with it directly is proven to be hard as its length is more than 40 pages (see the discussion on the logical consistency and impredicativity of System $\lambda_{\triangleleft}$ on pages 48–51 of [18]). It may therefore be difficult to conclude about Turing-completeness of either PSS or MPSS, which is left as future work.

7 Conclusion

This paper presents a variant of the PSS originally introduced by Hutchins in [18], which we name MPSS, and proves that our reformulation enjoys a key commutation property which was missing in Hutchins' system. We also sketch how a type system can be derived from MPSS, which could lead to a type-safe system. To the best of our knowledge PSS and MPSS are the first theories that unify harmoniously terms and types by using a single kind where everything is a subtype.

Meta-theories of languages with advanced features on types are notoriously difficult to work with. We believe our work could help in laying the foundations for a new approach that mixes terms and types in innovative ways.

References

1 Mads S. Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of International Conference on Principles and Practice of Declarative Programming*, pages 8–19, 2003. doi:10.1145/888251.888254.

2 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 249–272, 2016. doi:10.1007/978-3-319-30936-1_14.

3 David Aspinall. Subtyping with singleton types. In *International Workshop on Computer Science Logic*, pages 1–15. Springer, 1994. doi:10.1007/BFB0022243.

4 Lennart Augustsson. Cayenne—a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, 1998.

5 Henk Barendregt. *Lambda Calculi with Types*, volume 2. Oxford University Press, 1993.

6 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009. doi:10.1007/978-3-642-03359-9_6.

7 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.

8 Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. *Information and computation*, 109(1-2):4–56, 1994. doi:10.1006/INCO.1994.1013.

9 Chris Casinghino. *Combining proofs and programs*. University of Pennsylvania, 2014.

10 Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. *ACM SIGPLAN Notices*, 49(1):33–45, 2014. doi:10.1145/2535838.2535883.

11 Gang Chen. Subtyping calculus of construction. In *International Symposium on Mathematical Foundations of Computer Science*, pages 189–198. Springer, 1997.

12 Thierry Coquand and Gérard Huet. *The calculus of constructions*. PhD thesis, INRIA, 1986.

13 Matthias Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, 1987.

14 Herman Geuvers and Joep Verkoelen. On fixed point and looping combinators in type theory, 2009.

15 J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators, An Introduction, 2nd Edition*. Cambridge University Press, 2008.

16 Douglas J. Howe. The computational behaviour of girard's paradox. In *Proceedings of the Symposium on Logic in Computer Science*, pages 205–214. IEEE Computer Society, 1987.

17 Antonius J. C. Hurkens. A simplification of girard's paradox. In *Typed Lambda Calculi and Applications: Second International Conference on Typed Lambda Calculi and Applications, TLCA'95 Edinburgh, United Kingdom, April 10–12, 1995 Proceedings 2*, pages 266–278. Springer, 1995. doi:10.1007/BFB0014058.

18 DeLesley S. Hutchins. *Pure subtype systems: A type theory for extensible software*. The University of Edinburgh, 2009.

19 DeLesley S. Hutchins. Pure subtype systems. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 287–298. Association for Computing Machinery, 2010. doi:10.1145/1706299.1706334.

20 Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007. doi:10.1007/S10990-007-9018-9.

21 Simon Marlow et al. Haskell 2010 language report, 2010. URL: http://www.haskell.org/.

22 James McKinna and Robert Pollack. Pure type systems formalized. In *International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer, 1993. doi:10.1007/BFB0037113.

23 Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

24 Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, 11(4):511–540, 2001. doi:10.1017/S0960129501003322.

25 Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, 1994. doi:10.1006/INCO.1994.1055.

26 Benjamin C. Pierce and Martin Steffen. Higher-order subtyping. *Theoretical computer science*, 176(1-2):235–282, 1997. doi:10.1016/S0304-3975(96)00096-5.

27 Tiark Rompf and Nada Amin. Type soundness for dependent object types (dot). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 624–641, 2016. doi:10.1145/2983990.2984008.

28 Vilhelm Sjoberg. *A dependently typed language with nontermination*. University of Pennsylvania, 2015.

29 Vincent van Oostrom. Confluence by decreasing diagrams. *Theoretical computer science*, 126(2):259–280, 1994. doi:10.1016/0304-3975(92)00023-K.

30 Philip Wadler. The expression problem, 1998. Posted to Java Genericity internet mailing list. URL: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

31 Yanpeng Yang and Bruno C. d. S. Oliveira. Unifying typing and subtyping. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):47:1–47:26, 2017. doi:10.1145/3133871.

32 Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *Workshop on Foundations of Object-Oriented Languages*, 2005. URL: http://zenger.org/papers/fool05.pdf.