Noname manuscript No.
(will be inserted by the editor)

Mechanizing Metatheory without Typing Contexts

Jonghyun Park · Jeongbong Seo · Sungwoo Park · Gyesik Lee

Received: date / Accepted: date

Abstract When mechanizing the metatheory of a programming language, one usually needs many lemmas proving structural properties of typing judgments, such as permutation and weakening. Such structural lemmas are sometimes unnecessary if we eliminate typing contexts by expanding typing judgments into their original hypothetical proofs. This technique of eliminating typing contexts, which has been around since Church [4], is based on the view that entailment relations, such as typing judgments, are just syntactic tools for displaying only the hypotheses and conclusion of a hypothetical proof while hiding its internal structure.

In this paper, we apply this technique to parts 1A/2A of the PoplMark challenge [1] and experimentally evaluate its efficiency by formalizing System $\mathsf{F}_{<:}$ in the Coq proof assistant in a number of different ways. An analysis of our Coq developments shows that eliminating typing contexts produces a more significant reduction in both the number of lemmas and the count of tactics than the cofinite quantification, one of the most effective ways of simplifying the mechanization involving binders. Our experiment with System $\mathsf{F}_{<:}$ suggests three guidelines to follow when applying the technique of eliminating typing contexts.

Keywords Typing context, PoplMark, System $\mathsf{F}_{<:}$

Jonghyun Park, Jeongbong Seo, and Sungwoo Park
Pohang University of Science and Technology (POSTECH), Korea
E-mail: {parjong,baramseo,gla}@postech.ac.kr

Gyesik Lee
Hankyong National University, Korea
E-mail: gslee@hknu.ac.kr

1 Introduction

In the burgeoning field of mechanizing the metatheory of programming languages, there have been various proposals for facilitating the mechanization and reducing programming effort. The basic problem of handling α-equivalence relations and capture-avoiding substitutions has led to a number of different approaches to representing binders, such as de Bruijn indices [6], the locally nameless representation [9,2], the locally named representation [14,15,20], the nominal Isabelle package [22], and higher-order abstract syntax [17,10]. There are also a number of different approaches to quantifying variables to obtain suitable induction principles, from the standard exists-fresh quantification to the for-all quantification [14,15] to the cofinite quantification [19,2]. All these ideas result in eliminating or simplifying infrastructure lemmas which do not constitute an intrinsic part of the proof as seen in manual proofs, but are nevertheless indispensable to the machine-checked proof.

Among such infrastructure lemmas, we often find a large number of structural lemmas which prove structural properties of typing judgments. To be specific, typing judgments usually use typing contexts to maintain hypotheses during a proof, which has been around since Curry and Feys [5] and has become a de facto standard, and structural lemmas prove structural properties of these typing contexts, such as permutation and weakening. These structural lemmas are so obvious to humans that manual proofs often state them without explicit proofs or even implicitly assume them in individual proof steps. In contrast, mechanized proofs should provide complete proofs of all structural lemmas, resulting in much more verbose proof scripts than manual proofs.

All structural lemmas, however, become unnecessary once we expand typing judgments into their original hypothetical proofs. Here we take the view that entailment relations (conventionally using the symbol $\vdash$), including typing judgments, are just syntactic tools for displaying only the hypotheses and conclusion of a hypothetical proof while hiding its internal structure. That is, we interpret an entailment relation $J_1, \cdots, J_n \vdash J$ as a syntactic representation of a certain hypothetical proof deducing a judgment J from a collection of hypotheses $\overline{J_1}, \cdots, \overline{J_n}$:

$$J_1, \cdots, J_n \vdash J \quad \Longleftrightarrow \quad \dfrac{\overline{J_1} \quad \cdots \quad \overline{J_n}}{\vdots}{\scriptstyle \cdot}\\ J$$

In hypothetical proofs, all structural properties are implicit — we may use a hypothesis as many times as necessary, or may choose not to use it at all. Hence, by expanding typing judgments into hypothetical proofs, or equivalently, by eliminating typing contexts, we obtain a new form of typing judgment which retains the internal structure of its own proof and renders all structural lemmas unnecessary. This new form of typing judgment may be unwieldy for manual proofs, but is highly suitable for mechanized proofs.

Such presentations of type systems without explicit typing contexts have been around since Church [4], used one way or another in several implementations of type-checkers and proof assistants, and recently put forward by Geuvers *et al.* [8,11] as a promising way of presenting the Pure Type Systems (PTSs). In this paper, we apply this approach to parts 1A/2A of the PoplMark challenge [1] and evaluate experimentally its efficiency. First we formulate System $\mathsf{F}_{<:}$ in the traditional way using a type system with typing contexts. As the technique of eliminating typing contexts requires a syntactic distinction between bound variables (with corresponding binders) and free parameters (without corresponding binders), we syntactically distinguish between bound variables and free parameters, and use the locally named representation for binders [14,15]. Then we convert all bindings in typing contexts to annotations of parameters to derive another formulation of System $\mathsf{F}_{<:}$. For example, a typing judgment $X : T, X' : T' \vdash \lambda a{:}U.\,(X, a) : S$ is converted to $\lambda a{:}U.\,(X^{:T}, a) : S$ where $X^{:T}$ is parameter X annotated with its type T. Here we do not convert binding $X' : T'$ because X' does not appear in $\lambda a{:}U.\,(X, a)$, and do not annotate a because it is a bound variable. Finally we use the Coq proof assistant to mechanize the proof of type safety in both formulations in a number of different ways.

This paper is not about theory, but about practice. It gives empirical evidence that eliminating typing contexts can significantly reduce programming effort when mechanizing the metatheory of programming languages, provided that structural lemmas are trivially true for their typing contexts. It also suggests guidelines to follow when applying the technique of eliminating typing contexts. Moreover, the practical value of the technique is not diminished at all by previous proposals for facilitating the mechanization of metatheory, since eliminating typing contexts is completely orthogonal to their focus (such as representing binders and quantifying variables). Thus, if the goal is to formalize the metatheory of a programming language involving typing contexts and confirm its soundness, the technique of eliminating typing contexts can be particularly useful.

From the study of the two formulations of System $\mathsf{F}_{<:}$ and an analysis of our Coq developments, we obtain the following findings:

- The technique of eliminating typing contexts not only makes all structural lemmas unnecessary but also simplifies many other lemmas, both in their statements and in their proof steps.
- In the design of a type system without typing contexts, it is crucial to permit no unbound type variables in annotations of parameters and not to propagate variable substitutions into annotations of parameters.
- In principle, the new formulation of System $\mathsf{F}_{<:}$ (without typing contexts) is incomparable to the original formulation because it cannot enforce consistent annotations for the same parameter. We can, however, establish an equivalence result if we consider only those terms and types that are shown to have corresponding well-formed typing contexts in the original formulation.

- The technique of eliminating typing contexts results in a more significant reduction in both the number of lemmas and the count of tactics than the cofinite quantification of parameters, one of the most effective ways of simplifying the mechanization involving binders. In our Coq developments for System $\mathsf{F}_{<:}$, eliminating typing contexts reduces the number of lemmas by about 45% while the cofinite quantification reduces the number of lemmas by about 30%. In addition, the technique of eliminating typing contexts and the cofinite quantification are orthogonal to each other and cumulative in their effect. The technique of eliminating typing contexts is, however, desirable only if one believes its adequacy; the cost of establishing the equivalence result may exceed its gain.
- The locally named representation is practically no different from the locally nameless representation with respect to the complexity of the Coq development, except that it does not provide a canonical representation of α-equivalence classes of types and terms, and thus does not automatically provide proofs of closure under α-equivalence.
- The technique of eliminating typing contexts is applicable only to those typing contexts for which structural lemmas are trivially true as in System $\mathsf{F}_{<:}$. For example, we cannot eliminate linear typing contexts because a hypothetical proof does not enforce that a hypothesis be used exactly once.

This paper is organized as follows. Section 2 presents a formulation of System $\mathsf{F}_{<:}$ which uses typing contexts in its type system. In Section 3, we derive another formulation of System $\mathsf{F}_{<:}$ which does not use typing contexts, and explain the importance of permitting no unbound variables in annotations of parameters. We also investigate the equivalence between the two formulations. Section 4 presents the result of mechanizing the metatheory of System $\mathsf{F}_{<:}$ and an analysis of our Coq developments. Section 5 presents a case study of applying the technique of eliminating typing contexts to an extension System F in which linear typing contexts coexist. Section 6 discusses related work. In particular, it gives an in-depth comparison between our work and the Pure Type Systems without explicit contexts given by Geuvers *et al.* [8, 11]. Section 7 concludes. All Coq proof scripts from this paper are available at `http://pl.postech.ac.kr/poplmark/`.

2 System $\mathsf{F}_{<:}$ with typing contexts

This section presents a formulation of System $\mathsf{F}_{<:}$ which adopts the locally named representation for binders and uses typing contexts in its type system. We give the syntax and operational semantics and prove type safety.

2.1 Syntax and operational semantics

Since we use the locally named representation for binders, our syntax for System $\mathsf{F}_{<:}$ uses two distinct classes, variables and parameters, for each syntactic

Values:

$$\frac{}{\lambda a{:}T.\,t \ \mathsf{value}}\ \textsf{val-abs} \qquad \frac{}{\Lambda A <: T.\,t \ \mathsf{value}}\ \textsf{val-tabs}$$

Reduction rules:

$$\frac{t \longmapsto t'}{t\ u \longmapsto t'\ u}\ \textsf{red-fun} \qquad \frac{t \ \mathsf{value} \quad u \longmapsto u'}{t\ u \longmapsto t\ u'}\ \textsf{red-arg} \qquad \frac{u \ \mathsf{value}}{(\lambda a{:}T.\,t)\ u \longmapsto [a \mapsto u]t}\ \textsf{red-app}$$

$$\frac{t \longmapsto u}{t\,[\![T]\!] \longmapsto u\,[\![T]\!]}\ \textsf{red-targ} \qquad \frac{}{(\Lambda A <: T.\,t)\,[\![U]\!] \longmapsto [A \mapsto U]t}\ \textsf{red-tapp}$$

Fig. 1 Operational semantics of System $\mathsf{F}_{<:}$

category involving binders. A variable has a corresponding binder and is thus locally bound, whereas a parameter has no corresponding binder and is thus assumed to be globally bound. For types, we use type variables $A, B, \cdots$, and type parameters $X, Y, \cdots$. For terms, we use term variables $a, b, \cdots$, and term parameters $x, y, \cdots$.

System $\mathsf{F}_{<:}$ extends System F by specifying a subtyping relation in every type binder:

$$\begin{array}{lrcl} \text{type} & T, U, S & ::= & A \mid X \mid \top \mid T \to U \mid \forall A <: T.\,U \\ \text{term} & t, u & ::= & a \mid x \mid \lambda a{:}T.\,t \mid t\ u \mid \Lambda A <: T.\,t \mid t\,[\![T]\!] \end{array}$$

$\top$ is a maximum type which is a supertype of every type. $T \to U$ is a function type from T to U. $\forall A <: T.\,U$ is a universal type which assumes that type variable A in U is a subtype of T. $\lambda a{:}T.\,t$ and $t\ u$ are an abstraction and an application. $\Lambda A <: T.\,t$ is a type abstraction which assumes that type variable A in t is a subtype of T. $t\,[\![T]\!]$ is a type application.

Figure 1 shows the operational semantics of System $\mathsf{F}_{<:}$ in the call-by-value style. A value judgment $t\ \mathsf{value}$ means that term t is a value, either an abstraction or a type abstraction. A reduction judgment $t \longmapsto u$ means that term t reduces to term u in a single step. The operational semantics in Figure 1 has no rule involving parameters, so we reuse it in Section 3 where we change the syntax for parameters.

A term variable substitution $[a \mapsto u]t$ in the rule red-app assumes that u contains no unbound term or type variables. Hence the following cases do not test if u contains an unbound term variable b or an unbound type variable A:

$$\begin{array}{rcll} [a \mapsto u](\lambda b{:}T.\,t) & = & \lambda b{:}T.\,[a \mapsto u]t & \text{if } a \neq b \\ [a \mapsto u](\Lambda A <: T.\,t) & = & \Lambda A <: T.\,[a \mapsto u]t & \end{array}$$

The rule red-tapp uses a type variable substitution $[A \mapsto U]t$ on term t, whose definition in turn uses another type variable substitution $[A \mapsto U]T$ on type T. Similarly to term variable substitutions, both forms of type variable substitutions assume that U contains no unbound type variables. Hence the following cases do not test if U contains an unbound type variable B:

$$\begin{array}{rcll} [A \mapsto U](\Lambda B <: T.\,t) & = & \Lambda B <: ([A \mapsto U]T).\,[A \mapsto U]t & \text{if } A \neq B \\ [A \mapsto U](\forall B <: T.\,S) & = & \forall B <: ([A \mapsto U]T).\,[A \mapsto U]S & \text{if } A \neq B \end{array}$$

2.2 Type system

Under the locally named representation, we use a typing context to record specific properties of parameters which eventually originate from binders. In the case of System $\mathsf{F}_{<:}$, we look up a typing context to find a supertype of a given type parameter or the type of a given term parameter:

$$\text{typing context } \Gamma, \Delta ::= \cdot \mid \Gamma, X <: T \mid \Gamma, x : T$$

$X <: T$ is called a type binding which asserts that type parameter X is a subtype of T. $x : T$ is called a term binding which asserts that term parameter x has type T. We interpret a typing context as an ordered set because its well-formedness depends on the order of its bindings.

The type system of System $\mathsf{F}_{<:}$ uses four kinds of judgments. A judgment $\Gamma\ \mathsf{ok}$ formalizes the notion of well-formedness of typing contexts. A subtyping judgment $\Gamma \vdash T <: U$ means that T is a subtype of U under typing context Γ. A typing judgment $\Gamma \vdash t : T$ means that term t has type T under typing context Γ. Since the type system of System $\mathsf{F}_{<:}$ does not need to deal with ill-formed types in its inference rules, it uses a local closure judgment $\Gamma \vdash T\ \mathsf{type}^I$ for identifying well-formed types. $\Gamma \vdash T\ \mathsf{type}^I$ means that type bindings in Γ account for all type parameters in T and that I is the set of all type variables in T that have no corresponding type binders. Hence, if $\Gamma \vdash T\ \mathsf{type}^{\varnothing}$ holds, type T contains no unbound type variables and we say that T is locally closed with respect to typing context Γ.

Figure 2 shows the type system of System $\mathsf{F}_{<:}$ with typing contexts. $\mathsf{dom}_{\mathsf{T}}(\Gamma)$ returns the set of type parameters declared in type bindings in Γ; $\mathsf{dom}_{\mathsf{t}}(\Gamma)$ returns the set of term parameters declared in term bindings in Γ. $\mathsf{param}_{\mathsf{T}}(T)$ returns the set of type parameters in T:

$$\mathsf{param}_{\mathsf{T}}(T) = \begin{cases} \emptyset & \text{when } T = A \text{ or } \top \\ \{X\} & \text{when } T = X \\ \mathsf{param}_{\mathsf{T}}(U) \cup \mathsf{param}_{\mathsf{T}}(S) & \text{when } T = U \to S \text{ or } \forall A <: U.\, S \end{cases}$$

$\mathsf{param}_{\mathsf{T}}(t)$ returns the set of type parameters in t:

$$\mathsf{param}_{\mathsf{T}}(t) = \begin{cases} \emptyset & \text{when } t = a \text{ or } x \\ \mathsf{param}_{\mathsf{T}}(T) \cup \mathsf{param}_{\mathsf{T}}(t') & \text{when } t = \lambda a{:}T.\, t',\ t'\ [\![T]\!], \text{ or } \Lambda A <: T.\, t' \\ \mathsf{param}_{\mathsf{T}}(u) \cup \mathsf{param}_{\mathsf{T}}(s) & \text{when } t = u\ s \end{cases}$$

$\mathsf{param}_{\mathsf{t}}(t)$ returns the set of term parameters in term t:

$$\mathsf{param}_{\mathsf{t}}(t) = \begin{cases} \emptyset & \text{when } t = a \\ \{x\} & \text{when } t = x \\ \mathsf{param}_{\mathsf{t}}(t') & \text{when } t = \lambda a{:}T.\, t',\ t'\ [\![T]\!], \text{ or } \Lambda A <: T.\, t' \\ \mathsf{param}_{\mathsf{t}}(u) \cup \mathsf{param}_{\mathsf{t}}(s) & \text{when } t = u\ s \end{cases}$$

The subtyping and typing rules dealing with binders (rules sub-forall, typing-abs, typing-tabs) introduce fresh parameter names using the exists-fresh quantification, which is the standard approach but complicates the proof development.

Locally closed types:

$$\frac{}{\Gamma \vdash \top\ \mathsf{type}^{\varnothing}}\ \textsf{typ-top} \qquad \frac{}{\Gamma \vdash A\ \mathsf{type}^{\{A\}}}\ \textsf{typ-ltvar} \qquad \frac{X \in \mathsf{dom}_{\mathsf{T}}(\Gamma)}{\Gamma \vdash X\ \mathsf{type}^{\varnothing}}\ \textsf{typ-tpar}$$

$$\frac{\Gamma \vdash T\ \mathsf{type}^{I_1} \quad \Gamma \vdash U\ \mathsf{type}^{I_2}}{\Gamma \vdash T \to U\ \mathsf{type}^{I_1 \cup I_2}}\ \textsf{typ-fun} \qquad \frac{\Gamma \vdash T\ \mathsf{type}^{I_1} \quad \Gamma \vdash U\ \mathsf{type}^{I_2}}{\Gamma \vdash \forall A <: T.\, U\ \mathsf{type}^{I_1 \cup (I_2 - \{A\})}}\ \textsf{typ-forall}$$

Well-formed typing contexts:

$$\frac{}{\cdot\ \mathsf{ok}}\ \textsf{ok-nil} \qquad \frac{\Gamma\ \mathsf{ok} \quad X \notin \mathsf{dom}_{\mathsf{T}}(\Gamma) \quad \Gamma \vdash T\ \mathsf{type}^{\varnothing}}{\Gamma, X <: T\ \mathsf{ok}}\ \textsf{ok-tpar}$$

$$\frac{\Gamma\ \mathsf{ok} \quad x \notin \mathsf{dom}_{\mathsf{t}}(\Gamma) \quad \Gamma \vdash T\ \mathsf{type}^{\varnothing}}{\Gamma, x : T\ \mathsf{ok}}\ \textsf{ok-epar}$$

Subtyping:

$$\frac{\Gamma\ \mathsf{ok} \quad \Gamma \vdash T\ \mathsf{type}^{\varnothing}}{\Gamma \vdash T <: \top}\ \textsf{sub-top} \qquad \frac{\Gamma\ \mathsf{ok} \quad \Gamma \vdash X\ \mathsf{type}^{\varnothing}}{\Gamma \vdash X <: X}\ \textsf{sub-refl}$$

$$\frac{X <: T \in \Gamma \quad \Gamma \vdash T <: U}{\Gamma \vdash X <: U}\ \textsf{sub-tpar} \qquad \frac{\Gamma \vdash T' <: T \quad \Gamma \vdash U <: U'}{\Gamma \vdash T \to U <: T' \to U'}\ \textsf{sub-fun}$$

$$\frac{\Gamma \vdash U <: T \quad X \notin \mathsf{param}_{\mathsf{T}}(T', U') \quad \Gamma, X <: U \vdash [A \mapsto X]T' <: [B \mapsto X]U'}{\Gamma \vdash \forall A <: T.\, T' <: \forall B <: U.\, U'}\ \textsf{sub-forall}$$

Typing:

$$\frac{\Gamma\ \mathsf{ok} \quad x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsf{typing-epar}$$

$$\frac{\Gamma \vdash T\ \mathsf{type}^{\varnothing} \quad x \notin \mathsf{param}_{\mathsf{t}}(t) \quad \Gamma, x : T \vdash [a \mapsto x]t : U}{\Gamma \vdash \lambda a{:}T.\, t : T \to U}\ \textsf{typing-abs}$$

$$\frac{\Gamma \vdash t : T \to U \quad \Gamma \vdash u : T}{\Gamma \vdash t\ u : U}\ \textsf{typing-app}$$

$$\frac{\Gamma \vdash T\ \mathsf{type}^{\varnothing} \quad X \notin \mathsf{param}_{\mathsf{T}}(t, U) \quad \Gamma, X <: T \vdash [A \mapsto X]t : [B \mapsto X]U}{\Gamma \vdash \Lambda A <: T.\, t : \forall B <: T.\, U}\ \textsf{typing-tabs}$$

$$\frac{\Gamma \vdash t : \forall A <: T.\, U \quad \Gamma \vdash S <: T}{\Gamma \vdash t\,[\![S]\!] : [A \mapsto S]U}\ \textsf{typing-tapp} \qquad \frac{\Gamma \vdash t : T \quad \Gamma \vdash T <: U}{\Gamma \vdash t : U}\ \textsf{typing-sub}$$

Fig. 2 Type system of System $\mathsf{F}_{<:}$ with typing contexts

The decision is deliberate, since we wish to measure the effect of eliminating typing contexts against the effect of switching to the cofinite quantification, which is not standard but simplifies the proof development.

The subtyping rules in Figure 2 deal only with locally closed types and never with ill-formed types:

Proposition 1 *If* $\Gamma \vdash T <: U$, *then* $\Gamma\ \mathsf{ok}$, $\Gamma \vdash T\ \mathsf{type}^{\varnothing}$, *and* $\Gamma \vdash U\ \mathsf{type}^{\varnothing}$

2.3 Local closure using index sets

Our local closure judgment $\Gamma \vdash T\ \mathsf{type}^{I}$ uses a typing context Γ to check (in the rule typ-tpar) if a type parameter is okay to use in type T. It also uses an index set I to compute the set of type variables in T that are temporarily unbound (see the rules typ-ltvar and typ-forall). We prefer this simple style of

$$\frac{}{\Gamma \vdash a\ \mathsf{term}^{\varnothing,\{a\}}}\ \textsf{term-var} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x\ \mathsf{term}^{\varnothing,\varnothing}}\ \textsf{term-par} \qquad \frac{\Gamma \vdash T\ \mathsf{type}^{I} \quad \Gamma \vdash t\ \mathsf{term}^{I',i}}{\Gamma \vdash \lambda a{:}T.\, t\ \mathsf{term}^{I \cup I',i-\{a\}}}\ \textsf{term-abs}$$

$$\frac{\Gamma \vdash t\ \mathsf{term}^{I_1,i_1} \quad \Gamma \vdash u\ \mathsf{term}^{I_2,i_2}}{\Gamma \vdash t\ u\ \mathsf{term}^{I_1 \cup I_2,i_1 \cup i_2}}\ \textsf{term-app} \qquad \frac{\Gamma \vdash T\ \mathsf{type}^{I} \quad \Gamma \vdash t\ \mathsf{term}^{I',i}}{\Gamma \vdash \Lambda A <: T.\, t\ \mathsf{term}^{I \cup (I' - \{A\}),i}}\ \textsf{term-tabs}$$

$$\frac{\Gamma \vdash t\ \mathsf{term}^{I,i} \quad \Gamma \vdash T\ \mathsf{type}^{I'}}{\Gamma \vdash t\,[\![T]\!]\ \mathsf{term}^{I \cup I',i}}\ \textsf{term-tapp}$$

Fig. 3 Auxiliary rules for locally closed terms with typing contexts

local closure to a more common style which chooses a fresh type parameter when inspecting universal types and thus does not need an index set, as in:

$$\frac{\begin{array}{l}\Gamma \vdash T\ \mathsf{type} \\ X \notin \mathsf{dom}_{\mathsf{T}}(\Gamma) \qquad \Gamma, X <: T \vdash [A \mapsto X]U\ \mathsf{type} \\ X \notin \mathsf{param}_{\mathsf{T}}(U)\end{array}}{\Gamma \vdash \forall A <: T.\, U\ \mathsf{type}}\ \textsf{typ-forall}'$$

We choose this simple style of local closure because our focus is mainly on measuring the effect of eliminating typing contexts in the proof of type safety of System $\mathsf{F}_{<:}$, rather than on simplifying the proof itself. Moreover switching to the more common style or its variant using cofinite quantification as in [2] makes no practical difference (or even complicates proofs) because all proofs performing induction on local closure judgments are very short anyway. For example, our proof of type safety of System $\mathsf{F}_{<:}$ contains six such lemmas which use four or fewer lines of proof scripts in Coq and three such lemmas which use no more than 11 lines of proof scripts in Coq. The use of the more common style also means that we have to choose a specific scheme for quantifying type parameter X ("exists-fresh", "for-all", or cofinitely), which is irrelevant to the issue of local closure.

2.4 Type safety

Our proof of type safety is straightforward, except that it introduces a few auxiliary definitions such as parameter substitutions and a local closure judgment for terms (which do not appear in Figure 2). A term parameter substitution $[x \mapsto u]t$ assumes that u contains no unbound type or term variables; type parameter substitutions $[X \mapsto S]t$ (on term t) and $[X \mapsto S]T$ (on type T) assume that S contains no unbound type variables. Similarly to the local closure judgment for types, a local closure judgment $\Gamma \vdash t\ \mathsf{term}^{I,i}$ for terms uses two index sets I and i. It means that term bindings in Γ account for all term parameters in t and that I and i are the set of temporarily unbound type and term variables, respectively. Hence, if $\Gamma \vdash t\ \mathsf{term}^{\varnothing,\varnothing}$ holds, term t contains no unbound type and term variables and we say that t is locally closed with respect to typing context Γ. Figure 3 shows the rules for locally closed terms.

Proposition 2 shows that the typing rules deal only with locally closed terms and types:

Proposition 2 *If* $\Gamma \vdash t : T$*, then* Γ ok*,* $\Gamma \vdash t$ $\mathsf{term}^{\varnothing,\varnothing}$*, and* $\Gamma \vdash T$ $\mathsf{type}^{\varnothing}$*.*

2.4.1 Why do we need auxiliary definitions?

We need term parameter substitutions in various lemmas involving term parameters such as the following substitution lemma:

Lemma 1 *If* $\Gamma, x : S \vdash t : T$ *and* $\Gamma \vdash u : S$*, then* $\Gamma \vdash [x \mapsto u]t : T$*.*

To illustrate why we introduce type parameter substitutions, consider the following lemma (which we use in the proof of type preservation):

Lemma 2 *If* $\Gamma, X <: T \vdash [A \mapsto X]t : [B \mapsto X]U$ *and* $X \notin \mathsf{param}_{\mathsf{T}}(t, U)$*, then* $\Gamma \vdash S <: T$ *implies* $\Gamma \vdash [A \mapsto S]t : [B \mapsto S]U$*.*

It is difficult to directly prove $\Gamma \vdash [A \mapsto S]t : [B \mapsto S]U$, since the same term is not always written as $[A \mapsto X]t$ for a unique term t (similarly for type $[B \mapsto X]U$) and induction on the judgment $\Gamma, X <: T \vdash [A \mapsto X]t : [B \mapsto X]U$ produces too many cases. Thus, we prove another lemma which uses type parameter substitutions:

Lemma 3 *If* $\Gamma, X <: T \vdash t : U$*, then* $\Gamma \vdash S <: T$ *implies* $\Gamma \vdash [X \mapsto S]t : [X \mapsto S]U$*.*

Using Lemma 3, we prove Lemma 2 just in a few steps.

To illustrate why we need the local closure judgment for terms in the proof of type safety, consider Lemma 1. If t is an abstraction $\lambda a{:}U.\, t'$, we need to show

$$[x \mapsto u][a \mapsto y]t' = [a \mapsto y][x \mapsto u]t'$$

where $x \neq y$, and if t is a type abstraction $\Lambda A <: U.\, t'$, we need to show

$$[x \mapsto u][A \mapsto X]t' = [A \mapsto X][x \mapsto u]t'.$$

We require u to be free of unbound term variables in the first case, *i.e.*, $\Gamma \vdash u$ $\mathsf{term}^{I,\varnothing}$, and unbound type variables in the second case, *i.e.*, $\Gamma \vdash u$ $\mathsf{term}^{\varnothing,i}$. The two cases show that the proof of Lemma 1 requires $\Gamma \vdash u$ $\mathsf{term}^{\varnothing,\varnothing}$, which is implied from $\Gamma \vdash u : S$ by Proposition 2. Thus, apart from showing the property of the type system in Proposition 2, the local closure judgment for terms facilitates the proof of type safety — it would be much more cumbersome to prove Lemma 1 without recourse to the local closure judgment for terms.

2.4.2 Proof of type safety

As is well-known in the literature, the use of the exists-fresh quantification in those rules dealing with binders necessitates various renaming lemmas for type and term parameters in the proof of type safety, which in turn necessitate

induction on the size of proofs. As an example, consider the case of the rule typing-abs in the proof of weakening:

$$\frac{\cdots \quad \Gamma, x : T \vdash [a \mapsto x]t : U}{\Gamma \vdash \lambda a{:}T.\, t : T \to U}\ \textsf{typing-abs}$$

We need to prove $\Gamma, y : S \vdash \lambda a{:}T.\, t : T \to U$ for an arbitrary term parameter y, but if y happens to be identical to x, the induction hypothesis on the premise does not lead to a proof of the desired typing judgment. Hence, we first rename x to a fresh term parameter z in the premise to obtain a proof of $\Gamma, z : T \vdash [a \mapsto z]t : U$. Still we cannot apply the induction hypothesis to the new typing judgment because it does not match the original premise. Hence, we resort to induction on the size of proofs (because the new typing judgment has the same size as the original premise) to complete the proof of weakening.

Our proof of type safety follows the development in the PoplMark challenge [1]. We first show transitivity of subtyping (Theorem 1) and then show type preservation (Theorem 2) and progress (Theorem 3). The proof of type safety does not depend on closure under α-equivalence of types and terms, but the type system in Figure 2 is designed in such a way that closure under the usual α-equivalence relation holds for both types and terms.

Theorem 1 (Transitivity of subtyping)
If $\Gamma \vdash T <: U$ and $\Gamma \vdash U <: S$, then $\Gamma \vdash T <: S$.

Theorem 2 (Preservation) *If $\Gamma \vdash t : T$ and $t \longmapsto u$, then $\Gamma \vdash u : T$.*

Theorem 3 (Progress) *If $\cdot \vdash t : T$, then either t value holds or there exists u such that $t \longmapsto u$.*

3 System $\mathsf{F}_{<:}$ without typing contexts

This section presents another formulation of System $\mathsf{F}_{<:}$ which adopts the locally named representation for binders but does not use typing contexts. First, we discuss design decisions for the new formulation. Then, we present the new type system and prove type safety. Finally, we explain the importance of permitting no unbound type variables in annotations of parameters and study the relationship between the two type systems. We reuse the operational semantics given in Figure 1.

3.1 Designing the type system

The basic idea for designing the type system without typing contexts is simple: instead of maintaining a typing context with type bindings $X <: T$ and term bindings $x : T$, we explicitly annotate every type parameter with its supertype and every term parameter with its type:

$$\begin{array}{lrcl} \mathsf{type} & T & ::= & A \mid X^{<:T} \mid \cdots \\ \mathsf{term} & t & ::= & a \mid x^{:T} \mid \cdots \end{array}$$

As typing contexts are no longer necessary, we use three new forms of judgments: a local closure judgment $T\ \mathsf{type}^I$, a subtyping judgment $T <: U$, and a typing judgment $t : T$, none of which uses typing contexts. Then, we can simplify the rules in Figure 2 by eliminating typing contexts. For example, the rule typing-epar is rewritten as:

$$\frac{}{x^{:T} : T}\ \textsf{x-typing-epar}'$$

The problem of eliminating typing contexts without compromising type safety, however, is more subtle than it appears to be. For example, it is not immediately clear whether substitutions for type variables should be propagated into annotations of parameters:

$$\begin{array}{rcl} [A \mapsto S]X^{<:T} & = & X^{<:[A\mapsto S]T} \\ [A \mapsto S]x^{:T} & = & x^{:[A\mapsto S]T} \end{array} \quad \text{or} \quad \begin{array}{rcl} [A \mapsto S]X^{<:T} & = & X^{<:T} \\ [A \mapsto S]x^{:T} & = & x^{:T} \end{array}$$

The left definition is intuitively more natural because type T may contain type variable A in general. The right definition, however, also conforms to the intuition that type variable substitutions do not affect type parameters as in the previous type system ($[A \mapsto S]X = X$).

It turns out that all serious complications in the design of the new type system are due to the potential presence of unbound type variables in annotations of parameters, especially type parameters declaring subtyping relations, which affects both the meaning of the local closure judgment and the definition of type variable substitutions. If we permit unbound type variables in annotations of parameters, we obtain the following definitions:

$$\frac{T\ \mathsf{type}^I}{X^{<:T}\ \mathsf{type}^I}\ \textsf{x-typ-tpar}' \qquad \begin{array}{rcl} [A \mapsto S]X^{<:T} & = & X^{<:[A\mapsto S]T} \\ [A \mapsto S]x^{:T} & = & x^{:[A\mapsto S]T} \end{array}$$

Although these seemingly innocuous definitions are intuitively correct and also allow us to design the type system and type variable substitutions independently, they considerably complicate or even make it practically impossible to complete the proof of type safety, in particular because different kinds of substitutions fail to commute as in $[X^{<:U} \mapsto S][A \mapsto U']T$ and $[A \mapsto X^{<:T}][x^{:U} \mapsto u]t$.

Thus, we permit only well-formed types and terms by maintaining an invariant that unbound type variables never appear in annotations of parameters and, more importantly, exploit this invariant to its fullest to simplify all the definitions. We design type variable substitutions and the type system simultaneously in the following three steps:

1. **Designing the local closure judgment**
   To begin with, we reexamine the meaning of the local closure judgment $T\ \mathsf{type}^I$. As in the previous type system, it means that I is the set of type variables in T that are temporarily unbound. In addition, for each type parameter $X^{<:S}$ in T, we require $S\ \mathsf{type}^\varnothing$ to ensure that X is properly

annotated with a locally closed type S. This characterization of $T\ \mathsf{type}^I$ is recursive in that we also have to inspect all type parameters in S in the same manner, which is concisely expressed in the following rule:

$$\frac{T\ \mathsf{type}^{\varnothing}}{X^{<:T}\ \mathsf{type}^{\varnothing}}\ \textsf{x-typ-tpar}$$

Note that proving $X^{<:T}\ \mathsf{type}^{\varnothing}$ by the rule x-typ-tpar is not the same as proving $X^{<:T}\ \mathsf{type}^{\varnothing}$ by the rule x-typ-tpar′. For example, $X^{<:\forall A<:S.\,Y^{<:A}}\ \mathsf{type}^{\varnothing}$ is provable by the rule x-typ-tpar′, but not by the rule x-typ-tpar.

2. **Designing type variable substitutions**
   Next, we obtain the definition of type variable substitutions as the result of exploiting the strong hygienic condition imposed on every parameter:

$$\begin{aligned}[A \mapsto S]X^{<:T} &= X^{<:T}\\ [A \mapsto S]x^{:T} &= x^{:T}\end{aligned}$$

   Here, we do not propagate type variable substitutions into annotations of parameters which are assumed to contain no unbound type variables.

3. **Designing the type system**
   Finally, we design the type system so that it never generates a fresh parameter $X^{<:T}$ or $x^{:T}$ for which $T\ \mathsf{type}^{\varnothing}$ does not hold. For example, the last premise in the following rule generates $X^{<:T}$ and we explicitly require a proof of $T\ \mathsf{type}^{\varnothing}$ in the first premise:

$$\frac{T\ \mathsf{type}^{\varnothing} \quad \cdots \quad [A \mapsto X^{<:T}]t : [B \mapsto X^{<:T}]U}{\Lambda A<:T.\,t : \forall B<:T.\,U}\ \textsf{x-typing-tabs}$$

   In addition, whenever we use a type parameter $X^{<:T}$ and a term parameter $x^{:T}$ in the leaf of a proof of a subtyping judgment and a typing judgment, respectively, we prove $T\ \mathsf{type}^{\varnothing}$ to ensure that T is locally closed and every type parameter in it is properly annotated. For example, we use the following rules:

$$\frac{T\ \mathsf{type}^{\varnothing}}{X^{<:T} <: X^{<:T}}\ \textsf{x-sub-refl} \qquad \frac{T\ \mathsf{type}^{\varnothing}}{x^{:T} : T}\ \textsf{x-typing-epar}$$

Once we agree on these design decisions, it is straightforward to reformulate System $\mathsf{F}_{<:}$. The proof of type safety also becomes much simpler, as we will see below.

3.2 Type system

Figure 4 shows the type system of System $\mathsf{F}_{<:}$ without typing contexts. $\mathsf{param}_{\mathsf{T}}(\cdot)$ returns the set of type parameters in its argument. In particular, it recurses

Locally closed types:

$$\frac{}{\top\ \mathsf{type}^{\varnothing}}\ \textsf{x-typ-top} \qquad \frac{}{A\ \mathsf{type}^{\{A\}}}\ \textsf{x-typ-ltvar} \qquad \frac{T\ \mathsf{type}^{\varnothing}}{X^{<:T}\ \mathsf{type}^{\varnothing}}\ \textsf{x-typ-tpar}$$

$$\frac{T\ \mathsf{type}^{I_1} \quad U\ \mathsf{type}^{I_2}}{T \to U\ \mathsf{type}^{I_1 \cup I_2}}\ \textsf{x-typ-fun} \qquad \frac{T\ \mathsf{type}^{I_1} \quad U\ \mathsf{type}^{I_2}}{\forall A <: T.\, U\ \mathsf{type}^{I_1 \cup (I_2 - \{A\})}}\ \textsf{x-typ-forall}$$

Subtyping:

$$\frac{T\ \mathsf{type}^{\varnothing}}{T <: \top}\ \textsf{x-sub-top} \qquad \frac{T\ \mathsf{type}^{\varnothing}}{X^{<:T} <: X^{<:T}}\ \textsf{x-sub-refl} \qquad \frac{T <: U}{X^{<:T} <: U}\ \textsf{x-sub-tpar}$$

$$\frac{T' <: T \quad U <: U'}{T \to U <: T' \to U'}\ \textsf{x-sub-fun}$$

$$\frac{U <: T \quad X \notin \mathsf{param_T}(T, T', U, U') \quad [A \mapsto X^{<:U}]T' <: [B \mapsto X^{<:U}]U'}{\forall A <: T.\, T' <: \forall B <: U.\, U'}\ \textsf{x-sub-forall}$$

Typing:

$$\frac{T\ \mathsf{type}^{\varnothing}}{x^{:T} : T}\ \textsf{x-typing-epar} \qquad \frac{T\ \mathsf{type}^{\varnothing} \quad x \notin \mathsf{param_t}(t) \quad [a \mapsto x^{:T}]t : U}{\lambda a{:}T.\, t : T \to U}\ \textsf{x-typing-abs}$$

$$\frac{t : T \to U \quad u : T}{t\ u : U}\ \textsf{x-typing-app}$$

$$\frac{T\ \mathsf{type}^{\varnothing} \quad X \notin \mathsf{param_T}(t, U) \quad [A \mapsto X^{<:T}]t : [B \mapsto X^{<:T}]U}{\Lambda A <: T.\, t : \forall B <: T.\, U}\ \textsf{x-typing-tabs}$$

$$\frac{t : \forall A <: T.\, U \quad S <: T}{t\ [\![S]\!] : [A \mapsto S]U}\ \textsf{x-typing-tapp} \qquad \frac{t : T \quad T <: U}{t : U}\ \textsf{x-typing-sub}$$

Fig. 4 Type system of System $\mathsf{F}_{<:}$ without typing contexts

into annotations of parameters:

$$\mathsf{param_T}(T) = \begin{cases} \emptyset & \text{when } T = A \text{ or } \top \\ \{X\} \cup \mathsf{param_T}(T') & \text{when } T = X^{<:T'} \\ \mathsf{param_T}(U) \cup \mathsf{param_T}(S) & \text{when } T = U \to S \text{ or } \forall A <: U.\, S \end{cases}$$

$$\mathsf{param_T}(t) = \begin{cases} \emptyset & \text{when } t = a \\ \mathsf{param_T}(T) & \text{when } t = x^{:T} \\ \mathsf{param_T}(T) \cup \mathsf{param_T}(t') & \text{when } t = \lambda a{:}T.\, t',\ t'\ [\![T]\!], \text{ or } \Lambda A <: T.\, t' \\ \mathsf{param_T}(u) \cup \mathsf{param_T}(s) & \text{when } t = u\ s \end{cases}$$

$\mathsf{param_t}(t)$ returns the set of term parameters in term t:

$$\mathsf{param_t}(t) = \begin{cases} \emptyset & \text{when } t = a \\ \{x\} & \text{when } t = x^{:T} \\ \mathsf{param_t}(t') & \text{when } t = \lambda a{:}T.\, t',\ t'\ [\![T]\!], \text{ or } \Lambda A <: T.\, t' \\ \mathsf{param_t}(u) \cup \mathsf{param_t}(s) & \text{when } t = u\ s \end{cases}$$

The type system uses three kinds of substitutions: term variable substitution $[a \mapsto x^{:T}]t$ and type variable substitutions $[A \mapsto X^{<:T}]t$ and $[A \mapsto X^{<:T}]S$,

$$\frac{}{a\ \mathsf{term}^{\varnothing,\{a\}}}\ \textsf{x-term-var} \qquad \frac{T\ \mathsf{type}^{\varnothing}}{x^{:T}\ \mathsf{term}^{\varnothing,\varnothing}}\ \textsf{x-term-par} \qquad \frac{T\ \mathsf{type}^{I} \quad t\ \mathsf{term}^{I',i}}{\lambda a{:}T.\,t\ \mathsf{term}^{I\cup I',i-\{a\}}}\ \textsf{x-term-abs}$$

$$\frac{t\ \mathsf{term}^{I_1,i_1} \quad u\ \mathsf{term}^{I_2,i_2}}{t\ u\ \mathsf{term}^{I_1\cup I_2,i_1\cup i_2}}\ \textsf{x-term-app}$$

$$\frac{T\ \mathsf{type}^{I} \quad t\ \mathsf{term}^{I',i}}{\Lambda A <: T.\,t\ \mathsf{term}^{I\cup(I'-\{A\}),i}}\ \textsf{x-term-tabs} \qquad \frac{t\ \mathsf{term}^{I,i} \quad T\ \mathsf{type}^{I'}}{t\,[\![T]\!]\ \mathsf{term}^{I\cup I',i}}\ \textsf{x-term-tapp}$$

Fig. 5 Auxiliary rules for locally closed terms without typing contexts

where $x^{:T}$ and $X^{<:T}$ cannot contain unbound variables. Recall that we do not propagate type variable substitutions into annotations of parameters.

We obtain the rules in Figure 4 by removing typing contexts from the rules in the previous type system. We continue to use a simple style of local closure using index sets and the exists-fresh quantification to introduce fresh parameter names in the subtyping and typing rules dealing with binders (rules x-sub-forall, x-typing-abs, x-typing-tabs).

Similarly to the previous type system, the proof of type safety requires parameter substitutions and a local closure judgment for terms. Parameter substitutions are as usual, except for the following cases of type parameter substitutions:

$$\begin{aligned} [X^{<:T} \mapsto S]X^{<:T} &= S \\ [X^{<:T} \mapsto S]Y^{<:U} &= Y^{<:[X^{<:T}\mapsto S]U} \quad \text{if } X^{<:T} \neq Y^{<:U} \\ [X^{<:T} \mapsto S]x^{:U} &= x^{:[X^{<:T}\mapsto S]U} \end{aligned}$$

Note that we propagate type parameter substitutions into annotations of parameters which may contain type parameters. We use a local closure judgment $t\ \mathsf{term}^{I,i}$ for terms (without a context) which means that I and i are the set of temporarily unbound type and term variables, respectively; see Figure 5 for the rules. Note that similarly to the rule x-typ-tpar, the rule x-term-par allows only locally closed types in annotations of term parameters.

The subtyping rules in Figure 4 deal only with locally closed types and never with ill-formed types. Likewise, the typing rules deal only with locally closed terms and types. We use the following propositions extensively in the proof of type safety:

Proposition 3 *If $T <: U$, then $T\ \mathsf{type}^{\varnothing}$ and $U\ \mathsf{type}^{\varnothing}$.*

Proposition 4 *If $t : T$, then $t\ \mathsf{term}^{\varnothing,\varnothing}$ and $A\ \mathsf{type}^{\varnothing}$.*

3.3 Type safety

The proof of type safety is similar to the proof given in Section 2.4: we first show transitivity and then show type preservation and progress. The proof requires much fewer lemmas, however, because we no longer need to prove

structural properties such as permutation and weakening, which, in a certain sense, are already built into every judgment. Moreover, individual lemmas also become simpler, both in their statements and in proof steps, because we no longer need to manipulate typing contexts.

As an example of an individual lemma becoming simpler, we examine a substitution lemma for the subtyping judgment from the previous type system:

Lemma 4 *If* $\Delta, X <: S, \Gamma \vdash T <: U$ *and* $\Delta \vdash S' <: S$, *then* $\Delta, [X \mapsto S']\Gamma \vdash [X \mapsto S']T <: [X \mapsto S']U$.

Note that we use two separate regions (Δ and Γ) in the typing context and extend type parameter substitutions to typing contexts as in $[X \mapsto S']\Gamma$. Such complications disappear altogether when we discard typing contexts in the new type system:

Lemma 5 *If* $T <: U$ *and* $S' <: S$, *then* $[X^{<:S} \mapsto S']T <: [X^{<:S} \mapsto S']U$.

To illustrate that proof steps also become simpler, let us consider the case of the rule sub-tpar in Lemma 4. In this case, the proof of $\Delta, X <: S, \Gamma \vdash T <: U$ is given as

$$\frac{Y <: T \in \Delta, X <: S, \Gamma \quad \Delta, X <: S, \Gamma \vdash T <: U}{\Delta, X <: S, \Gamma \vdash Y <: U}\ \textsf{sub-tpar}$$

where $X \neq Y$, and $\Delta, [X \mapsto S']\Gamma \vdash [X \mapsto S']T <: [X \mapsto S']U$ is assumed to hold by induction hypothesis. Note that we may complete the proof if we show $Y <: [X \mapsto S']T \in \Delta, [X \mapsto S']\Gamma$:

$$\frac{\begin{array}{l} Y <: [X \mapsto S']T \in \Delta, [X \mapsto S']\Gamma \\ \Delta, [X \mapsto S']\Gamma \vdash [X \mapsto S']T <: [X \mapsto S']U \end{array}}{\Delta, [X \mapsto S']\Gamma \vdash Y <: [X \mapsto S']U}\ \textsf{sub-tpar}.$$

Thus, the proof reduces to showing $Y <: [X \mapsto S']T \in \Delta, [X \mapsto S']\Gamma$ from $Y <: T \in \Delta, X <: S, \Gamma$. Let us consider the corresponding case of the rule x-sub-tpar in Lemma 5. In this case, the proof of $Y^{<:T} <: U$ is given as follows:

$$\frac{T <: U}{Y^{<:T} <: U}\ \textsf{x-sub-tpar}$$

Note that $[X^{<:S} \mapsto S']T <: [X^{<:S} \mapsto S']U$ is assumed to hold by induction hypothesis. Then, we do not need such an analysis at all because we immediately complete the proof as

$$\frac{[X^{<:S} \mapsto S']T <: [X^{<:S} \mapsto S']U}{[X^{<:S} \mapsto S']Y^{<:T} <: [X^{<:S} \mapsto S']U}\ \textsf{x-sub-tpar}$$

where we use $Y^{<:[X^{<:S} \mapsto S']T} = [X^{<:S} \mapsto S']Y^{<:T}$. In essence, the use of typing contexts in the previous type system simplifies the representation of proofs of judgments, but at the cost of explicitly proving those structural properties that involve typing contexts. In contrast, the new type system adheres to the

most direct (but more verbose) representation of proofs of judgments, but does not incur the cost of proving such structural properties.

In fact, the effect of building structural properties into every judgment permeates the whole proof of type safety, eliminating many proof steps that require unusual tricks in the previous type system. As an example, consider a renaming lemma from the previous type system:

Lemma 6 *If* $\Gamma, x : T \vdash [a \mapsto x]t : S$ *and* $x \notin \mathsf{param_t}(t)$ *and* $y \notin \mathsf{dom_t}(\Gamma)$*, then* $\Gamma, y : T \vdash [a \mapsto y]t : S$*.*

If $t = \lambda a{:}U.\, u$, term variable substitutions $[a \mapsto x]$ and $[a \mapsto y]$ have no effect on t and we need to prove $\Gamma, y : T \vdash t : S$ from $\Gamma, x : T \vdash t : S$ where $x \notin \mathsf{param_t}(t)$ and $y \notin \mathsf{dom_t}(\Gamma)$. Ideally, we would strengthen $\Gamma, x : T \vdash t : S$ to $\Gamma \vdash t : S$ (because t does not use x) and then weaken it to $\Gamma, y : T \vdash t : S$. Unfortunately this strategy does not work because the proof of weakening itself requires the renaming lemma being proved (see Section 2.4). Hence we resort to an unusual trick that analyzes the structure of the proof of $\Gamma, x : T \vdash \lambda a{:}U.\, u : S$ and applies the induction hypothesis to the premise of the rule typing-abs twice (first to rename x to y and second to rename a fresh term parameter in the premise to another term parameter). In a corresponding renaming lemma from the new type system, this case becomes trivial:

Lemma 7 *If* $[a \mapsto x^{:T}]t : S$*, then* $[a \mapsto y^{:T}]t : S$*.*

Here letting $t = \lambda a{:}U.\, u$ makes the assumption identical to the conclusion: $t = [a \mapsto x^{:T}]t = [a \mapsto y^{:T}]t$.

Transitivity of subtyping (Theorem 4) and type safety (Theorems 5 and 6) are as expected. Proving $\mathsf{param_t}(u) \subset \mathsf{param_t}(t)$ in the preservation theorem corresponds to using the same typing context for both terms t and u in Theorem 2. The progress theorem requires $\mathsf{param_t}(t) = \varnothing$ to ensure that t contains no term parameters, whereas the typing judgment in Theorem 3 uses an empty typing context instead. As in the previous type system, the proof of type safety does not depend on closure under α-equivalence of types and terms, but closure under the usual α-equivalence relation holds for both types and terms.

Theorem 4 (Transitivity of subtyping)
If $T <: U$ *and* $U <: S$*, then* $T <: S$*.*

Theorem 5 (Preservation)
If $t : T$ *and* $t \longmapsto u$*, then* $u : T$ *and* $\mathsf{param_t}(u) \subset \mathsf{param_t}(t)$*.*

Theorem 6 (Progress) *If* $t : T$ *and* $\mathsf{param_t}(t) = \varnothing$*, then either* t value *holds or there exists* u *such that* $t \longmapsto u$*.*

3.4 Comparison with System $\mathsf{F}_{<:}$ with typing contexts

Although we have derived the new type system from the previous type system in a principled way, the relationship between the two type systems is not so

simple. For example, the new type system cannot enforce a consistent annotation across different instances of the same type parameter. Hence, despite a strong hygienic condition imposed on every parameter, a valid judgment may have no corresponding judgment in the previous type system. In the following examples, $X^{<:X^{<:T}} <: T$ and $T \to X^{<:U} <: X^{<:T} \to U$ are valid judgments, but their corresponding judgments in the previous type system use ill-formed typing contexts $\Gamma_1 = X <: X, X <: T$ and $\Gamma_2 = X <: T, X <: U$:[1]

$$\dfrac{\dfrac{\dfrac{\vdots}{T <: T}}{X^{<:T} <: T}}{X^{<:X^{<:T}} <: T} \implies \dfrac{X <: X \in \Gamma_1 \qquad \dfrac{X <: T \in \Gamma_1 \quad \Gamma_1 \vdash \dfrac{\vdots}{T <: T}}{\Gamma_1 \vdash X <: T}}{\Gamma_1 \vdash X <: T}$$

$$\dfrac{\dfrac{\dfrac{\vdots}{T <: T}}{X^{<:T} <: T} \quad \dfrac{\dfrac{\vdots}{U <: U}}{X^{<:U} <: U}}{T \to X^{<:U} <: X^{<:T} \to U} \implies \dfrac{\dfrac{X <: T \in \Gamma_2 \quad \Gamma_2 \vdash \dfrac{\vdots}{T <: T}}{\Gamma_2 \vdash X <: T} \qquad \dfrac{X <: U \in \Gamma_2 \quad \Gamma_2 \vdash \dfrac{\vdots}{U <: U}}{\Gamma_2 \vdash X <: U}}{\Gamma_2 \vdash T \to X <: X \to U}$$

Thus, in general, we cannot establish an equivalence result between the two type systems.

Fortunately, the two type systems coincide in their typing capabilities if we make an additional assumption on the new type system: we use terms and types that are shown to have corresponding well-formed typing contexts from the previous type system. To be formal, we assume Γ ok and write $\Gamma \Uparrow T$ and $\Gamma \Uparrow t$ for the result of recursively annotating parameters in T and t, respectively, according to typing context Γ. Here, annotated parameters and unannotated variables may temporarily coexist in T and t:

$$\begin{aligned}
\cdot \Uparrow T &= T \\
(\Gamma, X <: S) \Uparrow T &= \Gamma \Uparrow [X \hookrightarrow X^{<:(\Gamma \Uparrow S)}]T \\
(\Gamma, a : S) \Uparrow T &= \Gamma \Uparrow T \\
\cdot \Uparrow t &= t \\
(\Gamma, X <: S) \Uparrow t &= \Gamma \Uparrow [X \hookrightarrow X^{<:(\Gamma \Uparrow S)}]t \\
(\Gamma, x : S) \Uparrow t &= \Gamma \Uparrow [x \hookrightarrow x^{:(\Gamma \Uparrow S)}]t
\end{aligned}$$

Note that $[X \hookrightarrow X^{<:(\Gamma \Uparrow S)}]$ and $[x \hookrightarrow x^{:(\Gamma \Uparrow S)}]$ are new forms of substitutions which in effect annotate parameters. Like previous substitutions, these new forms of substitutions are not propagated into annotations of parameters:

$$\begin{aligned}
[X \hookrightarrow X^{<:T}]Y^{<:U} &= Y^{<:U} \\
[X \hookrightarrow X^{<:T}]y^{:U} &= y^{:U} \\
[x \hookrightarrow x^{:T}]y^{:U} &= y^{:U}
\end{aligned}$$

Then, we can establish an equivalence result between the two type systems subject to the additional assumption:

[1] If we adopt the view of Geuvers *et al.* [8] and treat $X^{<:T}$ and $X^{<:U}$ as different type variables (where $T \neq U$), every valid judgment produces a well-formed typing context.

Theorem 7 *Assume that T, U, and t are from the previous type system and contain no annotated parameters. Then the following relations hold:*
$\Gamma \vdash T <: U$ if and only if Γ ok and $(\Gamma \Uparrow T) <: (\Gamma \Uparrow U)$.
$\Gamma \vdash t : T$ if and only if Γ ok and $(\Gamma \Uparrow t) : (\Gamma \Uparrow T)$.

Theorem 7 states that the two type systems are incomparable in principle, but equivalent in practice because we never need to deal with ill-formed typing contexts.

Proving Theorem 7 is far from straightforward. For example, the *if* part of the second clause uses an unusual definition of the size of a proof of $t : T$ such that the rule x-typing-sub is ignored in calculating the size. That is, we consider proofs $\mathcal{D}$ and $\mathcal{E}$ shown below to be of the same size:

$$\mathcal{D} \;=\; \dfrac{\begin{array}{cc}\mathcal{E} & \vdots \\ t : U & U <: T\end{array}}{t : T}\ \textsf{x-typing-sub}$$

Theorem 7 also relies on the following lemma:

Lemma 8 *Suppose Γ ok. Then $(\Gamma \Uparrow t) : T$ implies that there exists a type T' in the previous type system such that $(\Gamma \Uparrow t) : (\Gamma \Uparrow T')$ and $(\Gamma \Uparrow T') <: T$.*

Proof By induction on the size of the proof of $(\Gamma \Uparrow t) : T$.

The statement in Lemma 8 hints that $(\Gamma \Uparrow t) : T$ does not necessarily mean $T = \Gamma \Uparrow T''$ for some type T' from the previous type system, which is, in fact, due to the rule x-typing-sub. As an example, consider the following proof of $(\Gamma \Uparrow t) : T$ where $\Gamma = X <: \top, x : X$ with $t = \lambda a : \top . x$ and $T = X^{<:\top\to\top} \to X^{<:\top}$:

$$\dfrac{\dfrac{\begin{array}{c}\vdots \\ \lambda a:\top.\, x^{:X^{<:\top}} : \top \to X^{<:\top}\end{array} \qquad \dfrac{\begin{array}{cc}\vdots & \vdots \\ X^{<:\top\to\top} <: \top & X^{<:\top} <: X^{<:\top}\end{array}}{\top \to X^{<:\top} \;<:\; X^{<:\top\to\top} \to X^{<:\top}}\ \textsf{x-sub-fun}}{\lambda a:\top.\, x^{:X^{<:\top}} : X^{<:\top\to\top} \to X^{<:\top}}\ \textsf{x-typing-sub}$$

We see that there is no type T' such that $T = \Gamma \Uparrow T'$ because type parameter X in T is inconsistently annotated with two different types $\top$ and $\top \to \top$. Our proof of Theorem 7 also uses various forms of renaming lemmas because of the use of the exists-fresh quantification.

The remaining question is: if we are to use the technique of eliminating typing contexts, should we also prove an equivalence result like Theorem 7 to justify its use? (Note that we do not need the equivalence result for the proof of type safety like Theorems 5 and 6). If we accept the view that typing judgments are just syntactic tools for representing hypothetical proofs (which is our view), the equivalence result is unnecessary. If not, we should establish the equivalence result before mechanizing the metatheory. A better approach in this case, however, may be to work directly with the original type system using typing contexts, since the proof of the equivalence result itself can be

Coq development	Representation	Typing context	Quantification	Definitions	Lemmas	Tactics
(1)	L-named	yes	exists-fresh	25 (12 + 13)	126	2579
(2)	L-named	yes	cofinite	23 (10 + 13)	89	1682
(3)	L-named	no	exists-fresh	22 (10 + 12)	72	1176
(4)	L-named	no	cofinite	19 (8 + 11)	50	813
(3')	L-nameless	no	exists-fresh	22 (10 + 12)	72	1184
(4')	L-nameless	no	cofinite	19 (8 + 11)	50	819
Theorem 7	L-named	yes/no	exists-fresh	42 (17 + 25)	141	2242
Leroy [12]	L-nameless	yes	for-all	29 (13 + 16)	119	(2128)
Aydemir *et al.* [2]	L-nameless	yes	cofinite	20 (11 + 9)	65	(747)

Fig. 6 Coq developments of the proof of type safety of System $\mathsf{F}_{<:}$ (PoplMark challenge 1A and 2A). L-named = Locally named and L-nameless = Locally nameless.

costly. For example, proving both type safety (Theorems 5 and 6) and the equivalence result (Theorem 7) in the Coq proof assistant is more costly than mechanizing the formulation in Section 2, as we will see in Section 4.

4 Experimental Results

This section presents the result of mechanizing the metatheory of System $\mathsf{F}_{<:}$. As solutions to the PoplMark challenge (parts 1A and 2A), we have formalized the proof of type safety of System $\mathsf{F}_{<:}$ in a number of different approaches in the Coq proof assistant. We analyze our Coq developments to measure the effect of eliminating typing contexts, in particular against the effect of using the cofinite quantification for dealing with binders. We also compare our Coq developments with a couple of existing solutions to the PoplMark challenge. Figure 6 summarizes the Coq developments which this section describes in detail.

4.1 Description of our Coq developments

Our reference development, (1) in Figure 6, is a direct translation of the formulation in Section 2. It adopts the locally named representation of binders, uses typing contexts in the type system, and introduces fresh parameter names using the exists-fresh quantification. This reference development uses no special techniques for simplifying the proof.

The development (2) is an adaptation of the reference development (1) and is designed to measure the effect of switching from the exists-fresh quantification to the cofinite quantification. Our main development (3) is a direct translation of the formulation in Section 3 which uses no typing contexts. We compare these two developments (2) and (3) to see how the technique of dispensing with typing contexts compares with the cofinite quantification in terms of reducing the complexity of the Coq development. The next development (4) is designed to see if dispensing with typing contexts is still effective

even when using the cofinite quantification. By measuring the combined effect of using the cofinite quantification and dispensing with typing contexts, we can also test if the two techniques are orthogonal to each other and cumulative in their effect.

The two developments, (3') and (4') in Figure 6, are adaptations of (3) and (4), respectively, which use the locally nameless representation of binders in both types and terms. We analyze these two developments to confirm that the locally named representation is practically no different from the locally nameless representation with respect to the complexity of the Coq development. The last development proves the equivalence result in Theorem 7 with the locally named representation of binders and the exists-fresh quantification.

The last column in Figure 6 lists three measurements for each Coq development: the number of definitions (counting inductive definitions and fixpoint definitions), the number of lemmas, and the number of times that the Coq script uses tactics. When counting lemmas, we ignore those libraries that do not directly address the metatheory of System $\mathsf{F}_{<:}$ (such as a library for lists of natural numbers). When counting tactics, we use only syntactic occurrences of tactics without considering the effect of using tacticals. For example, `T1;T2` and `T1|T2` are both counted as two tactics. We also do not take into consideration the semantics of tactics and treat all tactics, including even custom tactics, equal. For example, `intros`, `apply`, and `firstorder`, albeit vastly different in their functionalities, are all counted as one tactic. We exclude from the count those optional tactics (such as `clear`) that only rearrange hypotheses and make no contribution to the proof.

We structure all our Coq developments in a consistent way that permits no lemma to perform induction more than once in its proof, and according to this rule, we occasionally expand a single complex lemma into two simpler lemmas. We also use a consistent programming style across all our Coq developments — in the choice of tactics, in the frequency of using tacticals, and in the degree of exploiting automation tactics. Hence, all measurements in Figure 6 are valid metrics for the complexity of the Coq development as long as the comparison is limited to our Coq developments.

In the following analysis of measurements in Figure 6, we may use the following interpretation, which is by no means accurate, but still acceptable:

- The number of definitions roughly indicates the size of the problem at hand, namely formalizing the proof of type safety of System $\mathsf{F}_{<:}$.
- The number of lemmas indicates the difficulty in structuring the entire proof script, *i.e.*, "figuring out what to prove," as a lemma performs induction at most once in its proof.
- The count of tactics indicates the difficulty in completing individual proofs, *i.e.*, "figuring out how to prove," as we use a consistent programming style.

4.2 Analysis of our Coq developments

As already seen in the literature [2], the cofinite quantification indeed produces a significant reduction in both the number of lemmas and the count of tactics,

whether we use typing contexts or not. In the presence of typing contexts, switching to the cofinite quantification, or changing from (1) to (2), reduces the number of lemmas by about 30% and the count of tactics by about 35%. In the absence of typing contexts, switching to the cofinite quantification, or changing from (3) to (4), reduces both the number of lemmas and the count of tactics by about 30%. We attribute such reductions in both measures solely to the use of the cofinite quantification, which obviates the need for renaming lemmas, which in turn makes unnecessary those definitions and lemmas that enable induction on the size of proofs.

Eliminating typing contexts, however, produces a more significant reduction in both the number of lemmas and the count of tactics. When using the exists-fresh quantification, eliminating typing contexts, or changing from (1) to (3), reduces the number of lemmas by about 45% and the count of tactics by about 55%. This result already suggests that eliminating typing contexts is more effective than the cofinite quantification in reducing the complexity of the Coq development. Even when using the cofinite quantification, eliminating typing contexts, or changing from (2) to (4), still reduces the number of lemmas by about 45% and the count of tactics by about 50%, which is remarkable because (2) is the result of taking advantage of the cofinite quantification. Such reductions in both measures are precisely what we expect from Section 3.3: those structural lemmas proving structural properties of typing judgments are no longer necessary and individual lemmas become simpler in proof steps.

Our analysis also shows that eliminating typing contexts is to a large extent orthogonal to the cofinite quantification. For example, reduction ratios due to eliminating typing contexts are reasonably stable whether the cofinite quantification is in use or not: about 45% in the number of lemmas and about 50% in the number of tactics. This result is unsurprising because the presence or the absence of typing contexts is irrelevant to the scheme for quantifying parameters. Thus, we expect that typical Coq developments are likely to be amenable to the technique of eliminating typing contexts, resulting in much simpler proof scripts.

The analysis of our Coq developments (3') and (4') shows that the locally named representation incurs only a slightly higher cost than the locally nameless representation. There is no change in either the number of definitions or the number of lemmas, and the decrease in the count of tactics is negligible. Thus, unless a separate proof of closure under α-equivalence is a must,[2] the locally named representation is preferable to the locally nameless representation because of its closer resemblance to the nominal representation for manual proofs. If we wish to obtain a canonical representation of α-equivalence classes of types and terms for free, we can always revert to the locally nameless representation, to which the technique of eliminating typing contexts is equally applicable.

[2] One of the discoveries in [14, 15, 20] is that a separate proof of closure under α-equivalence is usually unnecessary.

Finally, the analysis of the Coq development for Theorem 7 shows that the technique of eliminating typing contexts is desirable only if one accepts the view that typing judgments (which use typing contexts) are just syntactic tools for representing hypothetical proofs. The Coq development for Theorem 7 contains 44 lemmas using tactics 673 times borrowed from the development (3), but still the total cost of the development (3) and the development for Theorem 7 slightly exceeds the cost of the development (1) alone.

4.3 Comparison with existing Coq developments

Although a fair comparison is impossible because of radical differences in programming style and methodology, we can still analyze existing solutions to the PoplMark challenge to further assess our Coq developments. The bottom of Figure 6 shows the complexity of two existing solutions which we compare with our Coq developments.

The development by Leroy [12] is based on the locally nameless representation and strives for readability in particular. Except for the use of swapping operations on names (rather than renaming operations), it does not use custom tactics or special techniques for reducing the size of the proof script. A comparison with his development suggests that our reference development (1) has a reasonable complexity and is not excessively verbose — it also uses typing contexts and the exists-fresh quantification is moderately more complex than the for-all quantification, and unsurprisingly it uses only a slightly higher number of lemmas.

The development by Aydemir *et al.* [2] is based on the locally nameless representation and strives for compactness in particular. Apart from the cofinite quantification, it makes heavy use of custom tactics specialized to the problem, resulting in a remarkably concise proof script. A comparison with their development suggests that our final developments (4) and (4') are indeed very concise — their development is already very concise, but we further reduce the number of lemmas even without using many custom tactics and just by eliminating typing contexts, which is the main claim of this paper.

5 Case study

We have seen in Section 3.1 that there are three important guidelines to follow when applying the technique of eliminating typing contexts:

1. We define local closure judgments in such a way that all annotations of parameters are required to be locally closed, as in the rule x-typ-tpar.
2. We do not propagate variable substitutions into annotations of parameters.
3. The type system never introduces fresh parameters whose annotations are not locally closed.

In order to further test the effect of eliminating typing contexts, we now carry out a case study in a different setting in which ordinary typing contexts coexist with linear typing context, to which the technique is not applicable.

For our case study, we use System F° in [13] which extends System F with linear types. Its typing judgment uses not only an unrestricted typing context which allows such structural properties as weakening and strengthening, but also a linear typing context which does not allow such structural properties at all:

$$\begin{array}{rrcl} \text{type} & T & ::= & A \mid \cdots \\ \text{term} & t & ::= & a \mid \cdots \\ \text{unrestricted typing context} & \Gamma & ::= & \cdot \mid \Gamma, A : \kappa \mid \Gamma, a : T \\ \text{linear typing context} & \Delta & ::= & \cdot \mid \Delta, a : T \end{array}$$

Here, kind κ specifies whether a given type is linear or not. Given a typing judgment $\Gamma; \Delta \vdash t : T$, we can eliminate unrestricted typing context Γ as usual, but not linear typing context Δ because a term binding $a : T$ in Δ does not specify which term variable a in t should be annotated with type T. After eliminating unrestricted typing contexts, we end up with a typing judgment $\Delta \vdash t : T$ using the following definitions:

$$\begin{array}{rrcl} \text{type} & T & ::= & A \mid X^\kappa \mid \cdots \\ \text{term} & t & ::= & a \mid x^{:T} \mid \cdots \\ \text{linear typing context} & \Delta & ::= & \cdot \mid \Delta, x^{:T} \end{array}$$

We test the effect of eliminating (unrestricted) typing contexts by rewriting the Coq development provided by Mazurak *et al.* [13]. Their Coq development proves type safety of System F° by using the locally nameless representation for binders and the cofinite quantification for both local closure judgments and typing judgments. It exploits the library from Aydemir *et al.* [2] and is already very compact. We rewrite all the definitions and lemmas according to the three guidelines given above. The first guideline, however, is not particularly relevant because a type parameter is annotated just with a kind, not with another type. When rewriting their Coq development, we do not change the style of local closure and continue to use the cofinite quantification (and no index sets).

The following table summarizes the two Coq developments:

Coq development	Definitions	Lemmas	Tactics
Mazurak *et al.* [13]	25 (14 + 11)	107	2195
Our development	21 (11 + 10)	86	1724

We eliminate four definitions about unrestricted typing contexts and local closure judgments for types. We eliminate a total of 24 lemmas and add three new lemmas. All structural lemmas proving structural properties of unrestricted typing contexts become unnecessary in the new development. The only interesting lemma introduced in the new development proves that a type is either non-linear or linear from the beginning. (In System F°, a non-linear type can be converted to a linear type because of the subtyping relation between kinds.)

We observe that the reduction in both the number of lemmas and the count of tactics (about 20%) is not so significant as in Section 4.2. This result is,

however, unsurprising: unlike the case of System $\mathsf{F}_{<:}$, we cannot completely eliminate all typing contexts and need to keep those structural lemmas manipulating linear typing contexts. We also learn that although only term parameters are annotated with types (unlike System $\mathsf{F}_{<:}$), propagating variable substitutions into annotations of parameters still considerably complicates the proof of type safety, which suggests the importance of the second guideline when eliminating typing contexts. On the whole, the three guidelines allow us to rewrite the previous Coq development with no serious challenge and we conclude that the technique of eliminating typing contexts works well for System F°.

6 Related work

Previous research on mechanizing the metatheory of programming languages usually focuses on the problem of representing binders to deal with α-equivalence relations and capture-avoiding substitutions. There are two categories of approaches, first-order and higher-order, depending on the way that the metalanguage handles binders and variables from the object language.

In first-order approaches, we encode variables in the object language with names or numbers in the metalanguage. For example, we can represent all variables with de Bruijn indices [6] which determine unique representations of types or terms involving binders (by counting the number of binders lying between each variable and its corresponding binder) and thus provide a canonical representation of α-equivalence classes. The locally nameless representation [9, 2] is a popular scheme adopted in several solutions to the PoplMark challenge. It syntactically separates bound variables (with corresponding binders) and free parameters (without corresponding binders), thereby resolving the difficulty of implementing capture-avoiding substitutions; it represents bound variables with de Bruijn indices, thereby giving a canonical representation of α-equivalence classes. The locally named representation [14,15,20] also syntactically separates bound variables and free parameters, but represents both bound variables and free parameters with names without using de Bruijn indices. Although it does not provide a canonical representation of α-equivalence classes, formal definitions under the locally named representation are much more readable than under the locally nameless representation. At the other end of the spectrum lies the nominal representation which represents both free variables and bound variables with a single kind of names as in manual "pencil and paper" proofs. The nominal Isabelle package [22], based on the idea of nominal logic [19], supports the nominal representation without requiring the user to define α-equivalence relations. As it requires a syntactic distinction between variables (without annotations) and parameters (with annotations), our technique of eliminating typing contexts can be combined only with the locally nameless or named representation.

In higher-order approaches based on higher-order abstract syntax [17,10], we encode binders in the object language using functions in the metalan-

guage. Since the metalanguage equates all α-equivalent functions and already implements capture-avoiding substitutions, we need no additional machinery for α-equivalence relations and capture-avoiding substitutions, which is the main advantage over first-order approaches. Another less well-known advantage is that structural properties of typing judgments can be built into the higher-order encoding of typing contexts if typing contexts in the object language are converted to regular hypotheses in the metalanguage, such as blocks in Twelf [18], for which all structural properties are usually implicit. As we have seen in Section 4, eliminating proofs of structural properties results in a significant reduction in the complexity of the mechanization of metatheory, which is another reason that mechanization in higher-order approaches can be much simpler (with no additional cost for structural properties as well as α-equivalence relations and capture-avoiding substitutions) than in first-order approaches. Note that this feature of higher-order approaches does *not* automatically translate to the technique of eliminating typing contexts proposed in this paper, since object languages in higher-order approaches may still assume typing contexts.

An orthogonal issue in first-order approaches, which is irrelevant to the technique of eliminating typing contexts, is how to quantify variables to obtain suitable induction principles. The standard approach is the exists-fresh quantification which quantifies a bound variable over a single fresh name or number not included in a specific finite set. The dual approach is the for-all quantification [14,15] which quantifies a bound variable over all fresh names or numbers excluding a specific finite set. A good compromise between the two approaches is the cofinite quantification [19,2] which quantifies a bound variable over all fresh names or numbers excluding an unknown finite set, which can be determined later in the proof. All these approaches turn out to be equivalent, but the cofinite quantification is gaining popularity in the programming language community because it eliminates typical renaming lemmas (but not all renaming lemmas). Recent work by Garrigue [7], Montagu [16], and Rossberg *et al.* [21] reports the strength of cofinite quantification as well as its pitfalls.

The idea of eliminating typing contexts already appears in the solution to part 1A of the PoplMark challenge by Charguéraud [3], which uses de Bruijn indices for both free variables and bound variables. In order to maintain a uniform representation of variables, his solution annotates every variable with a type, but ignores annotations for bound variables. Unlike in our solution, the potential presence of bound variables in annotations of free variables does not complicate the mechanization because a uniform representation of variables allows a common operation to be defined for both free variables and bound variables. As such, his proof script is remarkably short, even with a proof of the equivalence with the traditional formulation.

The closest to our work is the Pure Type Systems (PTSs) without explicit contexts given by Geuvers *et al.* [8] and the Coq development in [11]. The key idea in their work is the same: variables and parameters are distinguished and every parameter is annotated with its type in order to eliminate typing

contexts. There are, however, differences in the motivation. Their motivation is to investigate a new LCF-style architecture for proof assistants which does not maintain global environments, whereas we use the technique of eliminating typing contexts in order to reduce programming effort in mechanizing the metatheory of programming languages. The results are also different. The main theorem in their work proves the correspondence between the two type systems (with contexts and without contexts) similarly to Theorem 7, whereas the main result in our work is empirical evidence of the effect of eliminating typing contexts together with the three guidelines given in Section 3.1. In the end, these results should converge to the same conclusion while substantiating each other, if we take the view that entailment relations, such as typing judgments in type theory, are just syntactic tools for representing hypothetical proofs concisely.

7 Conclusion

We have experimented with the technique of eliminating typing contexts from typing judgments for mechanizing the metatheory of programming languages. Simple as it is, the technique produces a significant reduction in the complexity of the mechanization by making all structural lemmas unnecessary and simplifying many other lemmas, both in their statements and in their proof steps. We also present a case study of applying the technique to an extension System F in which linear typing contexts coexist.

Acknowledgments

The authors are grateful to Steve Zdancewic for providing the Coq proof script for System F°, and to Xavier Leroy and the anonymous reviewers for their helpful comments. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2012-0000472) and Mid-career Researcher Program through NRF funded by the MEST (2010-0022061). Gyesik Lee is supported by the 2012 Overseas Benchmarking Program of the Planning Office of Hankyong National University.

References

1. Brian Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In Joe Hurd and Thomas F. Melham, editors, *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, pages 50–65. Springer, 2005.

2. Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 3–15. ACM, 2008.
3. Arthur Charguéraud. `http://www.charglueraud.org/research/2006/poplmark/`, 2006.
4. Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.
5. Haskell B. Curry and Robert Feys. *Combinatory Logic*. North-Holland, 1958.
6. N. G. de Bruijn. Lambda calculus notation with nameless dummies. A tool for automatic formula manipulation with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
7. Jacques Garrigue. A certified implementation of ML with structural polymorphism. In *Proceedings of the 8th Asian conference on Programming Languages and Systems*, APLAS'10, pages 360–375. Springer-Verlag, 2010.
8. Herman Geuvers, Robbert Krebbers, James McKinna, and Freek Wiedijk. Pure type systems without explicit contexts. In *Proceedings of the 5th International Workshop on Logical Frameworks and Meta-languages (LFMTP)*, pages 53–67, 2010.
9. Andrew D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 413–425. Springer-Verlag, 1994.
10. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40:143–184, January 1993.
11. Robbert Krebbers. A formalization of Γ_∞ in Coq. `http://robbertkrebbers.nl/research/gammainf/`, 2010.
12. Xavier Leroy. A locally nameless solution to the POPLmark challenge. Research report 6098, INRIA, January 2007.
13. Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F°. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 77–88. ACM, 2010.
14. James McKinna and Robert Pollack. Pure type systems formalized. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer-Verlag, 1993.
15. James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automed Reasoning*, 23:373–409, 1999.
16. Benoît Montagu. Experience report: Mechanizing Core F-zip using the locally nameless approach (extended abstract). In *5th ACM SIGPLAN Workshop on Mechanizing Metatheory*, 2010.
17. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208. ACM, 1988.
18. Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer-Verlag LNAI, 1999.
19. Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.
20. Randy Pollack, Masahiko Sato, and Wilmer Ricciotti. A canonical locally named representation of binding. *Journal of Automated Reasoning*, 49(2):185–207, 2012.
21. Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 89–102. ACM, 2010.
22. Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40:327–356, May 2008.