Mechanized Metatheory for the Masses: The PoplMark Challenge

Brian E. Aydemir[1], Aaron Bohannon[1], Matthew Fairbairn[2], J. Nathan Foster[1], Benjamin C. Pierce[1], Peter Sewell[2], Dimitrios Vytiniotis[1], Geoffrey Washburn[1], Stephanie Weirich[1], and Steve Zdancewic[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Computer Laboratory, University of Cambridge

Subversion Revision: 171
Document generated on: May 11, 2005 at 15:53

Abstract. How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?
We propose an initial set of benchmarks for measuring progress in this area. Based on the metatheory of System $F_{<:}$, a typed lambda-calculus with second-order polymorphism, subtyping, and records, these benchmarks embody many aspects of programming languages that are challenging to formalize: variable binding at both the term and type levels, syntactic forms with variable numbers of components (including binders), and proofs demanding complex induction principles. We hope that these benchmarks will help clarify the current state of the art, provide a basis for comparing competing technologies, and motivate further research.

1 Introduction

Many proofs about programming languages are long, straightforward, and tedious, with just a few interesting cases. Their complexity arises from the management of many details rather than from deep conceptual difficulties; yet small mistakes or overlooked cases can invalidate large amounts of work. These effects are amplified as languages scale: it becomes hard to keep definitions and proofs consistent, to reuse work, and to ensure tight relationships between theory and implementations. Automated proof assistants offer the hope of significantly easing these problems. However, despite much encouraging progress in recent years and the availability of several mature tools (ACL2 [26], Coq [3], HOL [18], HOL Light [21], Isabelle [35], Lego [28], NuPRL [7], PVS [40], Twelf [42], etc.), their use is still not commonplace.

We believe that the time is ripe to join the efforts of the two communities, bringing developers of automated proof assistants together with a large pool of eager potential clients—programming language designers and researchers. In particular, we would like to answer two questions:

1. What is the current state of the art in formalizing language metatheory and semantics? What can be recommended as best practices for groups (typically not proof-assistant experts) embarking on formalizing language definitions, either small- or large-scale?
2. What improvements are needed to make the use of tool support commonplace? What can each community contribute?

Over the past several months, we have surveyed the landscape of proof assistants, language representation strategies, and related tools. Collectively, we have applied automated theorem proving technology to a number of problems, including proving transitivity of the algorithmic subtype relation in Kernel$F_{<:}$ [6, 5, 9], proving type soundness of Featherweight Java [24], proving type soundness of variants of the simply typed λ-calculus and $F_{<:}$, and a substantial formalization of the behavior of TCP, UDP, and the Sockets API. We have carried out these case studies using a variety of object-language representation strategies, proof techniques, and proving environments. We have also experimented with lightweight tools designed to make it easier to define and typeset both formal and informal mathematics. Although experts in programming language theory, we are relative outsiders with respect to computer-aided proof.

Our conclusion from these experiments is that the relevant technology has developed *almost* to the point where it can be widely used by language researchers. We seek to push it over the threshold, making the use of proof tools common practice in programming language research—mechanized metatheory for the masses.

Tool support for formal reasoning about programming languages would be useful at many levels:

1. *Machine-checked metatheory.* These are the classic problems: type preservation and soundness theorems, unique decomposition properties for operational semantics, proofs of equivalence between algorithmic and declarative variants of type systems, etc. At present such results are typically proved by hand for small to medium-sized calculi, and are not proved at all for full language definitions. We envision a future in which the papers in conferences such as *Principles of Programming Languages (POPL)* and the *International Conference on Functional Programming (ICFP)* are routinely accompanied by mechanically checkable proofs of the theorems they claim.
2. *Use of definitions as oracles for testing and animation.* When developing a language implementation together with a formal definition one would like to use the definition as an oracle for testing. This requires tools that can decide typing and evaluation relationships, and they might differ from the tools used for (1) or be embedded in the same proof assistant. In some cases one could use a definition directly as a prototype.
3. *Support for engineering large-scale definitions.* As we move to full language definitions—on the scale of Standard ML [30] or larger—pragmatic "software engineering" issues become increasingly important, as do the potential benefits of tool support. For large definitions, the need for elegant and concise

notation becomes pressing, as witnessed by the care taken by present-day researchers using informal mathematics. Even lightweight tool support, without full mechanized proof, could be very useful in this domain, e.g., for sort checking and typesetting of definitions and of informal proofs, automatically instantiating definitions, performing substitutions, etc.

Large scale formalization of languages is already within reach of current technology. For examples, see the work on proofs of correctness of the Damas-Milner type inference algorithm for ML [12, 33], semantics for C [37], semantics for Standard ML [46, 48, 20], and semantics and proofs of correctness for substantial subsets of Java [36, 27, 34]. Some other significant existing applications of mechanized metatheory include Foundational Proof Carrying Code [1] and Typed Assembly Langugages [8]. Inspired by these successes, we seek to make mechanized metatheory more accessible to programming languages researchers.

We hope to stimulate progress by providing a framework for comparing alternative technologies. We issue here an initial set of challenge problems, dubbed the PoplMark Challenge, chosen to exercise some aspects of programming languages that are known to be difficult to formalize: variable binding at both term and type levels, syntactic forms with variable numbers of components (including binders), and proofs demanding complex induction principles. Such challenge problems have been used in the past within the theorem proving community to focus attention on specific areas and to evaluate the relative merits of different tools; these have ranged in scale from benchmark suites and small problems [45, 19, 11, 23, 15, 32] up to the grand challenges of Floyd, Hoare, and Moore [13, 22, 31]. We hope that our challenge will have a similarly stimulating effect.

Our problems are drawn from the basic metatheory of a call-by-value variant of System $F_{<:}$ [5, 9], enriched with records, record subtyping, and record patterns. We provide an informal definition of its type system and operational semantics and outline proofs of some of its metatheory in Appendix A. This language is of moderate scale—significantly more complex than simply typed lambda-calculus or "mini ML," but much smaller than a full-blown programming language—to keep the work involved in attempting the challenges manageable. (Our challenges therefore explicitly address only points 1 and 2 above; we regard the pragmatic issues of point 3 as equally critical, but it is not yet clear to us how to formulate a useful challenge problem at this larger scale.) As these challenges are met, we anticipate that the set of problems will be extended to emphasize other language features and proof techniques.

We have begun collecting and publicizing solutions to these challenge problems and information related to mechanized metatheory on our web site.[3] In the longer run, we hope that this site and accompanying e-mail discussion list[4] will serve as a forum for promoting and advancing the current best practices in proof assistant technology and making this technology available to the broader programming languages community and beyond. We encourage researchers to

[3] `http://www.cis.upenn.edu/proj/plclub/mmm/`

[4] `poplmark@lists.seas.upenn.edu`

try out the PoplMark Challenge using their favorite tools and send us their solutions.

In the next section, we discuss in more detail our reasons for selecting this specific set of challenge problems. Section 3 describes the problems themselves, and Section 4 sketches some avenues for further development of the challenge problem set.

2 Design of the Challenge

This section motivates our choice of challenge problems and discusses the evaluation criteria for proposed solutions to the challenges. Since variable binding is a central aspect of the challenges, we briefly discuss relevant techniques and sketch some of our own experience in this area.

2.1 Problem Selection

The goal of the PoplMark Challenge is to provide a small, well-defined set of problems that capture many of the most critical issues in formalizing programing language metatheory. By its nature, such a benchmark will not be able to reflect *all* important issues—it is not practical to require challenge participants to formalize a large-scale language, for example. Instead, the PoplMark problems concentrate on a few important features:

– *Binding.* Most programming languages have some form of binding in their syntax and require a treatment of α-equivalence and capture-avoiding substitution in their semantics. To adequately represent many languages, the representation strategy must support multiple kinds of binders (e.g. term and type), constructs introducing multiple binders over the same scope, and potentially unbounded lists of binders (e.g. for record patterns).
– *Complex inductions.* Programming language definitions often involve complex, mutually recursive definitions. Structural induction over such objects, mutual induction, and induction on heights or pairs of derivations are all commonplace in metatheory.
– *Experimentation.* Proofs about programming languages are just one aspect of formalization; for some applications, experimenting with formalized language designs is equally interesting. It should be easy for the language designer to execute typechecking algorithms, generate sample program behaviors, and—most importantly—test real language implementations against the formalized definitions.
– *Component reuse.* To further facilitate experimentation with and sharing of language designs, the infrastructure should support some way of reusing prior definitions and parts of proofs.

We have carefully constructed the PoplMark Challenge to stress these features; a theorem-proving infrastructure that addresses the whole challenge should be applicable across a wide spectrum of programming language theory. While we believe that the features above are essential, our challenge does not address many other interesting and tricky-to-formalize constructs and reasoning principles. We discuss possible extensions to the challenge in Section 4.

2.2 Evaluation Criteria

A solution to the PoplMark Challenge will consist of appropriate software tools, a language representation strategy, and a demonstration that this infrastructure is sufficient to formalize the problems described in Section 3. Appendix A presents reasonably detailed informal proofs of the challenge properties. Solutions to the challenge should follow the overall structure of these proofs, though we expect that details will vary from prover to prover and across term representations. In all cases, there must be an argument for why the formalization is equivalent to the presentation in Section 3—i.e., an adequacy theorem—which should be as simple as possible.

The primary metric of success (beyond correctness, of course) is that a solution should give us confidence of future success of other formalizations carried out using similar techniques. In particular, this implies that:

– *The technology should impose reasonable overheads.* We accept that there is a cost to formalization, and our goal is *not* to be able to prove things more easily than by hand (although that would certainly be welcome). We are willing to spend more time and effort to use the proof infrastructure, but the overhead of doing so must not be prohibitive. (For example, as we discuss below, our experience is that explicit de Bruijn-indexed representations of variable binding structure fail this test.)
– *The technology should be transparent.* The representation strategy and proof assistant syntax should not depart too radically from the usual conventions familiar to the technical audience, and the content of the theorems themselves should be apparent to someone not deeply familiar with the theorem proving technology used or the representation strategy chosen.
– *The technology should have a reasonable cost of entry.* The infrastructure should be usable (after, say, one semester of training) by someone who is knowledgeable about programming language theory but not an expert in theorem prover technology.

2.3 Representing Binders

The problem of representing and reasoning about inductively-defined structures with binders is central to the PoplMark challenges. Representing binders has been recognized as crucial by the theorem proving community, and many different solutions to this problem have been proposed. In our (still limited) experience, none emerge as clear winners. In this section we briefly summarize the main approaches and, where applicable, describe our own experiments using them. Our survey is far from complete and we refrain from drawing any hard conclusions, to give the proponents of each method a chance to try their hand at meeting the challenge.

A first-order, named approach very similar in flavor to standard informal presentations was used by Vestergaard and Brotherston to formalize some metatheory of untyped λ-calculus [49, 50]. Their representation requires that each binder initially be assigned a unique name—one aspect of the so-called Barendregt convention [2].

Another popular concrete representation is de Bruijn's nameless representation [10]. De Bruijn indices are easy to understand and support the full range of induction principles needed to reason over terms. In our experience, however, de Bruijn representations have two major flaws. First, the *statements* of theorems require complicated clauses involving "shifted" terms and contexts. These extra clauses make it difficult to see the correspondence between informal and formal versions of the same theorem—there is no question of simply typesetting the formal statement and pasting it into a paper. Second, while the notational clutter is manageable for "toy" examples of the size of the simply-typed lambda calculus, we have found it becomes quite a heavy burden even for fairly small languages like $F_{<:}$.

In their formalization of properties of pure type systems, McKinna and Pollack use a hybrid approach that combines the above two representation strategies. In this approach, free variables are ordinary names while bound variables are represented using de Bruijn indices [29].

A radically different approach to representing terms with binders is higher-order abstract syntax (HOAS) [41]. In HOAS representations, binders in the meta-language are used to represent binders in the object language. Our experience with HOAS encodings (mainly as realized in Twelf) is that they provide a conveniently high level of abstraction, encapsulating much of the complexity of reasoning about binders. However, the strengths and limitations of the approach are not yet clearly understood, and it can sometimes require significant ingenuity to encode particular language features or proof ideas in this style.

Gordon and Melham propose a way to axiomatize inductive reasoning over untyped lambda-terms [17] and suggest that other inductive structures with binding can be encoded by setting up a correspondence with the untyped lambda terms. Norrish has pursued this direction [38, 39], but observes that these axioms are cumbersome to use without some assistance from the theorem-proving tool. In particular, the axioms use universal quantification in inductive hypotheses where in informal practice "some/any" quantification is used. He has developed a library of lemmas about a system of permutations on top of the axioms that aids reasoning significantly.

Several recent approaches to binding take the concept of "swapping" as a primitive, and use it to build a nominal logic. Gabbay and Pitts proposed a method of reasoning about binders based upon a set theory extended with an intrinsic notion of permutation [14]. Pitts followed this up by proposing a new "nominal" logic based upon the idea of permutations [44]. More recent work by Urban proposes methods based on the same intuitions but carried out within a conventional logic [47]. Our own preliminary experiments with Urban's methods have been encouraging.

3 The Challenge

Our challenge problems are taken from the basic metatheory of System $F_{<:}$. This system is formed by enriching the types and terms of System F with a subtype relation, refining universal quantifiers to carry subtyping constraints, and adding records, record subtyping, and record patterns. Our presentation is based on Pierce's *Types and Programming Languages* [43]; other good sources for background information are the seminal papers by Cardelli et al [5] and Curien and Ghelli [9].

The challenge comprises three distinct parts. The first deals just with the type language of $F_{<:}$; the second considers terms, evaluation, and type soundness. Each of these is further subdivided into two parts, starting with definitions and properties for *pure* $F_{<:}$ and then asking that the same properties be proved for $F_{<:}$ enriched with records and patterns. This partitioning allows the development to start small, but also—and more importantly—focuses attention on issues of reuse: How much of the first sub-part can be re-used verbatim in the second sub-part? The third problem asks that useful algorithms be extracted from the earlier formal definitions and used to "animate" some simple properties.

Challenge 1A: Transitivity of Subtyping

The first part of this challenge problem deals purely with the type language of $F_{<:}$. The syntax for this language is defined by the following grammar and inference rules. Although the grammar is simple—it has only four syntactic forms—some of its key properties require fairly sophisticated reasoning.

Syntax:

`T` ::=		*types*
	`X`	*type variable*
	`Top`	*maximum type*
	`T`$\to$`T`	*type of functions*
	$\forall$`X<:T.T`	*universal type*
Γ ::=		*type environments*
	$\emptyset$	*empty type env.*
	Γ, `X<:T`	*type variable binding*

In $\forall$`X<:`$\mathtt{T}_1$.$\mathtt{T}_2$, the variable `X` is a binding occurrence with scope $\mathtt{T}_2$ (`X` is not bound in $\mathtt{T}_1$). In Γ, `X<:T`, the `X` must not be in the domain of Γ, and the free variables of `T` must all be in the domain of Γ.

Following standard practice, issues such as the use of α-conversion, capture avoidance during substitution, etc. are left implicit in what follows. There are several ways in which these issues can be formalized: we might take Γ as a concrete structure (such as an association list of named variables and types) but quotient types and terms up to alpha equivalence, or we could take entire judgments up to alpha equivalence. We might axiomatize the well-formedness of typing environments and types using auxiliary $\vdash \Gamma$ *ok* and $\Gamma \vdash \mathtt{T}$ *ok* judgments. And so on. We leave these decisions to the individual formalization.

It is acceptable to make small changes to the rules below to reflect these decisions, such as adding well-formedness premises. Changing the presentation of the rules to a notationally different but "obviously equivalent" style such as HOAS is also acceptable, but there must be a clear argument that it is really equivalent. Also, whatever formalization is chosen should make clear that we are only dealing with *well-scoped* terms. For example, it should not be possible to derive `X <: Top` in the empty typing environment.

The subtyping relation captures the intuition "if `S` is a subtype of `T` (written `S <: T`) then an instance of `S` may safely be used wherever an instance of `T` is expected." It is defined as the least relation closed under the following rules.

Subtyping $\boxed{\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}}$

$$\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{Top} \qquad \text{(SA-Top)}$$

$$\Gamma \vdash \mathtt{X} \mathrel{<:} \mathtt{X} \qquad \text{(SA-Refl-TVar)}$$

$$\frac{\mathtt{X{<:}U} \in \Gamma \qquad \Gamma \vdash \mathtt{U} \mathrel{<:} \mathtt{T}}{\Gamma \vdash \mathtt{X} \mathrel{<:} \mathtt{T}} \qquad \text{(SA-Trans-TVar)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 \mathrel{<:} \mathtt{S}_1 \qquad \Gamma \vdash \mathtt{S}_2 \mathrel{<:} \mathtt{T}_2}{\Gamma \vdash \mathtt{S}_1 {\rightarrow} \mathtt{S}_2 \mathrel{<:} \mathtt{T}_1 {\rightarrow} \mathtt{T}_2} \qquad \text{(SA-Arrow)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 \mathrel{<:} \mathtt{S}_1 \qquad \Gamma, \mathtt{X{<:}T}_1 \vdash \mathtt{S}_2 \mathrel{<:} \mathtt{T}_2}{\Gamma \vdash \forall \mathtt{X{<:}S}_1\mathtt{.S}_2 \mathrel{<:} \forall \mathtt{X{<:}T}_1\mathtt{.T}_2} \qquad \text{(SA-All)}$$

These rules present an *algorithmic* version of the subtyping relation. In contrast to the more familiar *declarative* presentation, these rules are syntax-directed, as might be found in the implementation of a type checker; the algorithmic rules are also somewhat easier to reason with, having, for example, an obvious inversion property (Lemma A.12). The declarative rules differ from these by explicitly stating that subtyping is reflexive and transitive. However, reflexivity and transitivity also turn out to be derivable properties in the algorithmic system. A straightforward induction shows that the algorithmic rules are reflexive. The first challenge is to show that that they are also transitive.

3.1 Lemma [Transitivity of Algorithmic Subtyping]: If $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{Q}$ and $\Gamma \vdash \mathtt{Q} \mathrel{<:} \mathtt{T}$, then $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$. □

The difficulty here lies in the reasoning needed to prove this lemma. Transitivity must be proven simultaneously with another property, called *narrowing*, by an inductive argument with case analyses on the final rules used in the given derivations. Full details of this proof appear in Appendix A.

3.2 LEMMA [NARROWING]: If Γ, $\texttt{X<:Q}$, $\Delta \vdash \texttt{M} <: \texttt{N}$ and $\Gamma \vdash \texttt{P} <: \texttt{Q}$ then Γ, $\texttt{X<:P}$, $\Delta \vdash \texttt{M} <: \texttt{N}$. □

Challenge 1B: Transitivity of Subtyping with Records

We now extend this challenge by enriching the type language with record types. The new syntax and subtyping rule for record types are shown below. Implicit in the syntax is the condition that the labels $\{\texttt{l}_i\ ^{i \in 1..n}\}$ appearing in a record type $\{\texttt{l}_i : \texttt{T}_i\ ^{i \in 1..n}\}$ are pairwise distinct.

New syntactic forms:

T	::=	...	*types*
		$\{\texttt{l}_i\texttt{:}\texttt{T}_i\ ^{i \in 1..n}\}$	*type of records*

New subtyping rules $\boxed{\Gamma \vdash \texttt{S} <: \texttt{T}}$

$$\frac{\{\texttt{l}_i\ ^{i \in 1..n}\} \subseteq \{\texttt{k}_j\ ^{j \in 1..m}\} \qquad \text{if } \texttt{k}_j = \texttt{l}_i\text{, then} \quad \Gamma \vdash \texttt{S}_j <: \texttt{T}_i}{\Gamma \vdash \{\texttt{k}_j\texttt{:}\texttt{S}_j\ ^{j \in 1..m}\} <: \{\texttt{l}_i\texttt{:}\texttt{T}_i\ ^{i \in 1..n}\}} \qquad \text{(SA-RCD)}$$

Although it has been shown that records can actually be encoded in pure $F_{<:}$ [4, 16], dealing with them directly is a worthwhile task since, unlike other syntactic forms, record types have an arbitrary (finite) number of fields. Also, the informal proof for Challenge 1A extends to record types by only adding the appropriate cases. A formal proof should reflect this.

Challenge 2A: Type Safety for Pure $F_{<:}$

The next challenge considers the type soundness of pure $F_{<:}$ (without record types, for the moment). Below, we complete the definition of $F_{<:}$ by describing the syntax of terms, values, and typing environments with term binders and giving inference rules for the typing relation and a small-step operational semantics.

As usual in informal presentations, we elide the formal definition of substitution and simply assume that the substitutions of a type P for X in T (denoted $[\texttt{X} \mapsto \texttt{P}]\texttt{T}$) and of a term q for x in t (denoted $[\texttt{x} \mapsto \texttt{q}]\texttt{t}$) are capture-avoiding.

Syntax:

t	::=		*terms*
		x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
		λX<:T.t	*type abstraction*
		t [T]	*type application*
v	::=		*values*
		λx:T.t	*abstraction value*
		λX<:T.t	*type abstraction value*

Γ ::=		*type environments*
	$\emptyset$	*empty type env.*
	$\Gamma, \mathtt{x}\mathtt{:}\mathtt{T}$	*term variable binding*
	$\Gamma, \mathtt{X}\mathtt{<:}\mathtt{T}$	*type variable binding*

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}\mathtt{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x}\mathtt{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}\mathtt{:}\mathtt{T}_1\mathtt{.}\mathtt{t}_2 : \mathtt{T}_1 \rightarrow \mathtt{T}_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \rightarrow \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

$$\frac{\Gamma, \mathtt{X}\mathtt{<:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{X}\mathtt{<:}\mathtt{T}_1\mathtt{.}\mathtt{t}_2 : \forall\mathtt{X}\mathtt{<:}\mathtt{T}_1\mathtt{.}\mathtt{T}_2} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall\mathtt{X}\mathtt{<:}\mathtt{T}_{11}\mathtt{.}\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{T}_2 <: \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}} \qquad \text{(T-TAPP)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \qquad \text{(T-SUB)}$$

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$(\lambda\mathtt{x}\mathtt{:}\mathtt{T}_{11}\mathtt{.}\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-APPABS)}$$

$$(\lambda\mathtt{X}\mathtt{<:}\mathtt{T}_{11}\mathtt{.}\mathtt{t}_{12})\ [\mathtt{T}_2] \longrightarrow [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{t}_{12} \qquad \text{(E-TAPPTABS)}$$

Evaluation contexts:

E ::=		*evaluation contexts*
	$[-]$	*hole*
	$E\ \mathtt{t}$	*app fun*
	$\mathtt{v}\ E$	*app arg*
	$E\ [\mathtt{T}]$	*type fun*

Evaluation in context:

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{E[\mathtt{t}_1] \longrightarrow E[\mathtt{t}_1']} \qquad \text{(E-CTX)}$$

The evaluation relation is presented in two parts: the rules E-AppAbs and E-TappTabs capture the immediate reduction rules of the language, while E-Ctx permits reduction under an arbitrary *evaluation context* E. For $F_{<:}$, one would have an equally clear definition and slightly simpler proofs using explicit closure rules for the evaluation relation. We use evaluation contexts with an eye to larger languages and languages with non-local control operators such as exceptions, for which (in informal mathematics) they are an important tool for reducing notational clutter in definitions.[5] Evaluation contexts are also particularly interesting from the point of view of formalization when they include binders, though unfortunately there are no examples of this in call-by-value $F_{<:}$.

Type soundness is usually proven in the style popularized by Wright and Felleisen [51], in terms of *preservation* and *progress* theorems. Challenge 2A is to prove these properties for pure $F_{<:}$.

3.3 Theorem [Preservation]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathtt{T}$. □

3.4 Theorem [Progress]: If $\mathtt{t}$ is a closed, well-typed $F_{<:}$ term (i.e., if $\vdash \mathtt{t} : \mathtt{T}$ for some $\mathtt{T}$), then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$. □

Unlike the proof of transitivity of subtyping, the inductive arguments required here are straightforward. However, variable binding becomes a more significant issue, since this language includes binding of both type and term variables. Several lemmas relating to both kinds of binding must also be shown, in particular lemmas about type and term substitutions. These lemmas, in turn, require reasoning about permuting, weakening, and strengthening typing environments.

Challenge 2B: Type Safety with Records and Pattern Matching

The next challenge is to extend the preservation and progress results to cover records and pattern matching. The new syntax and rules for this language appear below. As for record types, the labels $\{\mathtt{l}_i\ ^{i\in 1..n}\}$ appearing in a record $\{\mathtt{l}_i = \mathtt{t}_i\ ^{i\in 1..n}\}$ are assumed to be pairwise distinct. Similarly, the variable patterns appearing in a pattern are assumed to bind pairwise distinct variables.

New syntactic forms:

t	::=	...	*terms*
		$\{\mathtt{l}_i\mathtt{=}\mathtt{t}_i\ ^{i\in 1..n}\}$	*record*
		t.l	*projection*

[5] This design choice has generated a robust debate on the PoplMark discussion list as to whether evaluation contexts *must* be used in order for a solution to count as valid, or whether an "obviously equivalent" presentation such as an evaluation relation with additional congruence rules is acceptable. We prefer evaluation contexts for the reasons we have given, but the consensus of the community appears to be that one should accept solutions in other styles. However, a good solution must be formulated in a style that provides similar clarity as the language scales.

		`let p=t in t`	*pattern binding*
`p`	::=		*patterns*
		`x:T`	*variable pattern*
		$\{\mathtt{l}_i\mathtt{=}\mathtt{p}_i\ ^{i\in 1..n}\}$	*record pattern*
`v`	::=	...	*values*
		$\{\mathtt{l}_i\mathtt{=}\mathtt{v}_i\ ^{i\in 1..n}\}$	*record value*

New typing rules $\boxed{\Gamma\vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma\vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \vdash \mathtt{p} : \mathtt{T}_1 \Rightarrow \Delta \qquad \Gamma,\Delta\vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma\vdash \mathtt{let}\ \mathtt{p}\mathtt{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \tag{T-Let}$$

$$\frac{\text{for each } i \quad \Gamma\vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma\vdash \{\mathtt{l}_i\mathtt{=}\mathtt{t}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i\mathtt{:}\mathtt{T}_i\ ^{i\in 1..n}\}} \tag{T-Rcd}$$

$$\frac{\Gamma\vdash \mathtt{t}_1 : \{\mathtt{l}_i\mathtt{:}\mathtt{T}_i\ ^{i\in 1..n}\}}{\Gamma\vdash \mathtt{t}_1\mathtt{.l}_j : \mathtt{T}_j} \tag{T-Proj}$$

Pattern typing rules:

$$\vdash (\mathtt{x}\mathtt{:}\mathtt{T}) : \mathtt{T} \Rightarrow \mathtt{x} : \mathtt{T} \tag{P-Var}$$

$$\frac{\text{for each } i \quad \vdash \mathtt{p}_i : \mathtt{T}_i \Rightarrow \Delta_i}{\vdash \{\mathtt{l}_i\mathtt{=}\mathtt{p}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i\mathtt{:}\mathtt{T}_i\ ^{i\in 1..n}\} \Rightarrow \Delta_n,\ \ldots,\ \Delta_1} \tag{P-Rcd}$$

New evaluation rules $\boxed{\mathtt{t}\longrightarrow \mathtt{t}'}$

$$\mathtt{let}\ \mathtt{p}\mathtt{=}\mathtt{v}_1\ \mathtt{in}\ \mathtt{t}_2 \longrightarrow \mathit{match}(\mathtt{p},\, \mathtt{v}_1)\mathtt{t}_2 \tag{E-LetV}$$

$$\{\mathtt{l}_i\mathtt{=}\mathtt{v}_i\ ^{i\in 1..n}\}\mathtt{.l}_j \longrightarrow \mathtt{v}_j \tag{E-ProjRcd}$$

New evaluation contexts:

E	::=	...	*evaluation contexts*
		$E\mathtt{.l}$	*projection*
		$\{\mathtt{l}_i\mathtt{=}\mathtt{v}_i\ ^{i\in 1..j-1}\mathtt{,}\mathtt{l}_j\mathtt{=}E\mathtt{,}\mathtt{l}_k\mathtt{=}\mathtt{t}_k\ ^{k\in j+1..n}\}$	*record*
		$\mathtt{let}\ \mathtt{p}\mathtt{=}E\ \mathtt{in}\ \mathtt{t}_2$	*let binding*

Matching rules:

$$\mathit{match}(\mathtt{x}\mathtt{:}\mathtt{T},\, \mathtt{v}) = [\mathtt{x}\mapsto \mathtt{v}] \tag{M-Var}$$

$$\frac{\{\mathtt{l}_i\ ^{i\in 1..n}\} \subseteq \{\mathtt{k}_j\ ^{j\in 1..m}\} \qquad \text{if } \mathtt{l}_i = \mathtt{k}_j,\ \text{then } \mathit{match}(\mathtt{p}_i,\, \mathtt{v}_j) = \sigma_i}{\mathit{match}(\{\mathtt{l}_i\mathtt{=}\mathtt{p}_i\ ^{i\in 1..n}\},\, \{\mathtt{k}_j\mathtt{=}\mathtt{v}_j\ ^{j\in 1..m}\}) \quad = \quad \sigma_n \circ \cdots \circ \sigma_1} \tag{M-Rcd}$$

Compared to the language of Challenge 2A, the `let` construct is a fundamentally new binding form, since patterns may bind an arbitrary (finite) number of term variables.

Challenge 3: Testing and Animating with Respect to the Semantics

Given a complete formal definition of a language, there are at least two interesting ways in which it can be used (as opposed to being reasoned about). When implementing the language, it should be possible to use the formal definition as an oracle for *testing* the implementation—checking that it does conform to the definition by running test cases in the implementation and confirming formally that the outcome is as prescribed. Secondly, one would like to construct a prototype implementation from the definition and use it for *animating* the language, i.e., exploring the language's properties on particular examples. In both cases, this should be done without any unverified (and thus error-prone) manual translation of definitions.

Our final challenge is to provide an implementation of this functionality, specifically for the following three tasks (using the language of Challenge 2B):

1. Given $F_{<:}$ terms $\mathtt{t}$ and $\mathtt{t}'$, decide whether $\mathtt{t} \longrightarrow \mathtt{t}'$.
2. Given $F_{<:}$ terms $\mathtt{t}$ and $\mathtt{t}'$, decide whether $\mathtt{t} \longrightarrow^{*} \mathtt{t}' \not\longrightarrow$, where $\longrightarrow^{*}$ is the reflexive-transitive closure of $\longrightarrow$.
3. Given an $F_{<:}$ term $\mathtt{t}$, find a term $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow \mathtt{t}'$.

The first two subtasks are useful for testing language implementations, while the last is useful for animating the definition. For all three subtasks, the system(s) should accept syntax that is "reasonably close" to that of informal (ASCII) mathematical notation, though it maybe necessary to translate between the syntaxes of a formal environment and an implementation. We will provide an implementation of an interpreter for $F_{<:}$ with records and patterns at the challenge's website in order to make this challenge concrete, together with a graded sequence of example terms. To make a rough performance comparison possible, solutions should indicate execution times for these terms.

A solution to this challenge might make use of decision procedures and tactics of a proof assistant or might extract stand-alone code. In general, it may be necessary to combine theorems (e.g. that a rule-based but algorithmic definition of typing coincides with a declarative definition) and proof search (e.g. deciding particular instances of the algorithmic definition).

4 Beyond the Challenge

The PoplMark Challenge is not meant to be exhaustive: other aspects of programming language theory raise formalization difficulties that are interestingly different from the problems we have proposed—to name a few: more complex binding constructs such as mutually recursive definitions, logical relations proofs, coinductive simulation arguments, undecidability results, and linear handling of type environments. As time goes on, we will issue a small number of further challenges highlighting the most important of these issues; suggestions from the community would be welcome. However, we believe that a technology that provides a good solution to the PoplMark challenge as we have formulated it here will be sufficient to attract eager adopters in the programming languages community, beginning with the authors.

So what are you waiting for? It's time to bring mechanized metatheory to the masses!

Acknowledgments

A number of people have joined us in preliminary discussions of these challenge problems, including Andrew Appel, Karl Crary, Frank Pfenning, Bob Harper, Hugo Herbelin, Jason Hickey, Michael Norrish, Andrew Pitts, Randy Pollack, Carsten Schürmann, Phil Wadler, and Dan Wang. We are especially grateful to Randy Pollack for helping guide our earliest efforts at formalization and to Randy Pollack, Michael Norrish, and Carsten Schürmann for their own work on the challenge problems. Discussions on the PoplMark mailing list have greatly deepened our understanding of the current state of the art.

Sewell and Fairbairn acknowledge support from a Royal Society University Research Fellowship and EPSRC grant GR/T11715.

References

1. Andrew W. Appel. Foundational proof-carrying code. In *IEEE Symposium on Logic in Computer Science (LICS), Boston, Massachusetts*, pages 247–258, June 2001.
2. Henk P. Barendregt. *The Lambda Calculus*. North Holland, revised edition, 1984.
3. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*, volume XXV of *EATCS Texts in Theoretical Computer Science*. Springer-Verlag, 2004.
4. Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC/Compaq Systems Research Center, January 1992. Also in C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994.
5. 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. Summary in TACS '91 (Sendai, Japan, pp. 750–770).
6. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
7. Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, James F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Paul Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

8. Karl Crary. Toward a foundational typed assembly language. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, pages 198–212, January 2003.
9. Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994.
10. 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(5):381–392, 1972.
11. Louise A. Dennis. Inductive challenge problems, 2000. `http://www.cs.nott.ac.uk/~lad/research/challenges`.
12. Catherine Dubois and Valerie Menissier-Morain. Certification of a type inference tool for ML: Damas-Milner within Coq. *Journal of Automated Reasoning*, 23(3-4):319–346, 1999.
13. Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.
14. Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In *14th Symposium on Logic in Computer Science*, pages 214–224, 1999.
15. I. P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical Report APES-09-1999, APES, 1999. Available from `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).
16. Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Università di Pisa, 1990. Technical report TD–6/90, Dipartimento di Informatica, Università di Pisa.
17. Andrew D. Gordon and Tom Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer-Verlag, 1996.
18. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
19. Ian Green. The dream corpus of inductive conjectures, 1999. `http://dream.dai.ed.ac.uk/dc/lib.html`.
20. Elsa Gunter and Savi Maharaj. Studying the ML module system in HOL. *The Computer Journal: Special Issue on Theorem Proving in Higher Order Logics*, 38(2):142–151, 1995.
21. John Harrison. HOL Light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996. See also `http://www.cl.cam.ac.uk/users/jrh/hol-light/index.html`.
22. Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
23. Holger Hoos and Thomas Stuetzle. SATLIB. `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/`.

24. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.
25. J. J. Joyce and C.-J. H. Seger, editors. *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications (HUG'93)*, volume 780 of *Lecture Notes in Computer Science*, Vancouver, B.C., Canada, August 1993. Springer-Verlag, 1994.
26. Matt Kaufmann, J Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
27. Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.
28. Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.
29. James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4), November 1999.
30. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997.
31. J Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, Lisbon, Portugal*, volume 2757 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2002.
32. J Strother Moore and George Porter. The apprentice challenge. *ACM Trans. Program. Lang. Syst.*, 24(3):193–216, 2002.
33. Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.
34. Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.
35. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant For Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
36. Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe—definitely. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 161–170, New York, NY, USA, 1998. ACM Press.
37. Michael Norrish. *Formalising C in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.
38. Michael Norrish. Mechanising Hankin and Barendregt using the Gordon-Melham axioms. In *MERLIN '03: Proceedings Of The 2003 Workshop On Mechanized Reasoning About Languages With Variable Binding*, pages 1–7. ACM Press, 2003.
39. Michael Norrish. Recursive function definition for types with binders. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004*, volume 3223 of *Lecture Notes In Computer Science*, pages 241–256. Springer-Verlag, 2004.
40. Sam Owre, Sreeranga Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In *International Conference on Computer Aided Verification (CAV), New*

Brunswick, New Jersey, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, July 1996.
41. Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.
42. Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Automated Deduction, CADE 16: 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer-Verlag, 1999.
43. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
44. Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.
45. Geoff Sutcliffe and Christian Suttner. The TPTP problem library. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
46. Donald Syme. Reasoning with the formal definition of Standard ML in HOL. In Joyce and Seger [25], pages 43–60.
47. Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. Accepted at CADE-20 in Tallinn. See `http://www.mathematik.uni-muenchen.de/~urban/nominal/`.
48. Myra VanInwegen and Elsa Gunter. HOL-ML. In Joyce and Seger [25], pages 61–74.
49. René Vestergaard and James Brotherston. The mechanisation of Barendregt-style equational proofs (the residual perspective). In *Mechanized Reasoning about Languages with Variable Binding (MERLIN)*, volume 58 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.
50. René Vestergaard and James Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names. *Information and Computation*, 183(2):212 – 244, 2003. Special edition with selected papers from RTA01.
51. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

A Paper Proofs

For the sake of concreteness and to standardize proof techniques (and thus make solutions easier to compare), here are paper proofs of the properties mentioned in the challenges.

A technical note for type experts: There are two reasonable ways of defining the subtyping relation of $F_{<:}$, differing in their formulation of the rule for comparing bounded quantifiers (rule SA-ALL): a more tractable but less flexible version called the *kernel* rule, and a more expressive but technically somewhat problematic *full* subtyping rule. We choose the full variant here as its metatheory is more interesting.

Algorithmic Subtyping (Challenges 1A and 1B)

We give the proof for the full subtyping relation including records. The proof for the pure system is obtained by simply deleting the cases involving records.

A.1 LEMMA [REFLEXIVITY]: $\Gamma \vdash \mathtt{T} \mathrel{\mathtt{<:}} \mathtt{T}$ is provable for every Γ and $\mathtt{T}$. □

Proof: By induction on the structure of $\mathtt{T}$. □

A.2 LEMMA [PERMUTATION AND WEAKENING]:

1. Suppose that Δ is a well-formed permutation of Γ—that is, Δ has the same bindings as Γ, and their ordering in Δ preserves the scopes of type variables from Γ, in the sense that, if one binding in Γ introduces a type variable that is mentioned in another binding further to the right, then these bindings appear in the same order in Δ.
   Now, if $\Gamma \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$, then $\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$ and $dom(\Delta) \cap dom(\Gamma) = \emptyset$, then $\Gamma, \Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$. □

Proof: Routine inductions. Part (1) is used in the SA-ALL case of part (2). □

A.3 LEMMA [TRANSITIVITY AND NARROWING]:

1. If $\Gamma \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{Q}$ and $\Gamma \vdash \mathtt{Q} \mathrel{\mathtt{<:}} \mathtt{T}$, then $\Gamma \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$.
2. If $\Gamma, \mathtt{X<:Q}, \Delta \vdash \mathtt{M} \mathrel{\mathtt{<:}} \mathtt{N}$ and $\Gamma \vdash \mathtt{P} \mathrel{\mathtt{<:}} \mathtt{Q}$ then $\Gamma, \mathtt{X<:P}, \Delta \vdash \mathtt{M} \mathrel{\mathtt{<:}} \mathtt{N}$. □

Proof: The two parts are proved simultaneously, by induction on the structure of $\mathtt{Q}$. At each stage of the induction, the argument for part (2) assumes that part (1) has been established already for the $\mathtt{Q}$ in question; part (1) uses part (2) only for strictly smaller $\mathtt{Q}$s.

1. We proceed by an inner induction on the structure of $\Gamma \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{Q}$, with a case analysis on the final rules of this derivation and of $\Gamma \vdash \mathtt{Q} \mathrel{\mathtt{<:}} \mathtt{T}$.
   If the right-hand derivation is an instance of SA-TOP, then we are done, since $\Gamma \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{Top}$ by SA-TOP. If the left-hand derivation is an instance of SA-TOP, then $\mathtt{Q} = \mathtt{Top}$, and, inspecting the algorithmic rules, we see

that the right-hand derivation must also be an instance of SA-Top. If either derivation is an instance of SA-Refl-TVar, then we are again done, since the other derivation is exactly the desired result.

If the left-hand derivation ends with an instance of SA-Trans-TVar, then we have $\mathtt{S} = \mathtt{Y}$ with $\mathtt{Y}\texttt{<:}\mathtt{U} \in \Gamma$ and a subderivation of $\Gamma \vdash \mathtt{U} \texttt{ <: } \mathtt{Q}$. By the inner induction hypothesis, $\Gamma \vdash \mathtt{U} \texttt{ <: } \mathtt{T}$, and, by SA-Trans-TVar again, $\Gamma \vdash \mathtt{Y} \texttt{ <: } \mathtt{T}$, as required.

If the left-hand derivation ends with an instance of SA-Arrow, SA-All, or SA-Rcd, then, since we have already considered the case where the right-hand derivation ends with SA-Top, it must end with the same rule as the left. If this rule is SA-Arrow, then we have $\mathtt{S} = \mathtt{S}_1 \rightarrow \mathtt{S}_2$, $\mathtt{Q} = \mathtt{Q}_1 \rightarrow \mathtt{Q}_2$, and $\mathtt{T} = \mathtt{T}_1 \rightarrow \mathtt{T}_2$, with subderivations $\Gamma \vdash \mathtt{Q}_1 \texttt{ <: } \mathtt{S}_1$, $\Gamma \vdash \mathtt{S}_2 \texttt{ <: } \mathtt{Q}_2$, $\Gamma \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{Q}_1$, and $\Gamma \vdash \mathtt{Q}_2 \texttt{ <: } \mathtt{T}_2$. We apply part (1) of the outer induction hypothesis twice (noting that $\mathtt{Q}_1$ and $\mathtt{Q}_2$ are both immediate subterms of $\mathtt{Q}$) to obtain $\Gamma \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{S}_1$ and $\Gamma \vdash \mathtt{S}_2 \texttt{ <: } \mathtt{T}_2$, and then use SA-Arrow to obtain $\Gamma \vdash \mathtt{S}_1 \rightarrow \mathtt{S}_2 \texttt{ <: } \mathtt{T}_1 \rightarrow \mathtt{T}_2$.

In the case where the two derivations end with SA-All, we have $\mathtt{S} = \forall \mathtt{X}\texttt{<:}\mathtt{S}_1\texttt{.}\mathtt{S}_2$, $\mathtt{Q} = \forall \mathtt{X}\texttt{<:}\mathtt{Q}_1\texttt{.}\mathtt{Q}_2$, and $\mathtt{T} = \forall \mathtt{X}\texttt{<:}\mathtt{T}_1\texttt{.}\mathtt{T}_2$, with

$$\begin{array}{ll} \Gamma \vdash \mathtt{Q}_1 \texttt{ <: } \mathtt{S}_1 & \qquad \Gamma, \mathtt{X}\texttt{<:}\mathtt{Q}_1 \vdash \mathtt{S}_2 \texttt{ <: } \mathtt{Q}_2 \\ \Gamma \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{Q}_1 & \qquad \Gamma, \mathtt{X}\texttt{<:}\mathtt{T}_1 \vdash \mathtt{Q}_2 \texttt{ <: } \mathtt{T}_2 \end{array}$$

as subderivations. By part (1) of the outer induction hypothesis ($\mathtt{Q}_1$ being an immediate subterm $\mathtt{Q}$), we can combine the two subderivations for the bounds to obtain $\Gamma \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{S}_1$. For the bodies, we need to work a little harder, since the two contexts do not quite agree. We first use part (2) of the outer induction hypothesis (noting again that $\mathtt{Q}_1$ is an immediate subterm of $\mathtt{Q}$) to narrow the bound of $\mathtt{X}$ in the derivation of $\Gamma, \mathtt{X}\texttt{<:}\mathtt{Q}_1 \vdash \mathtt{S}_2 \texttt{ <: } \mathtt{Q}_2$, obtaining $\Gamma, \mathtt{X}\texttt{<:}\mathtt{T}_1 \vdash \mathtt{S}_2 \texttt{ <: } \mathtt{Q}_2$. Now part (1) of the outer induction hypothesis applies ($\mathtt{Q}_2$ being an immediate subterm of $\mathtt{Q}$), yielding $\Gamma, \mathtt{X}\texttt{<:}\mathtt{T}_1 \vdash \mathtt{S}_2 \texttt{ <: } \mathtt{T}_2$. Finally, by SA-All, $\Gamma \vdash \forall \mathtt{X}\texttt{<:}\mathtt{S}_1\texttt{.}\mathtt{S}_2 \texttt{ <: } \forall \mathtt{X}\texttt{<:}\mathtt{T}_1\texttt{.}\mathtt{T}_2$.

Finally, if the two derivations both end with SA-Rcd, then we have $\mathtt{S} = \{\mathtt{l}_i : \mathtt{S}_i\ ^{i\in 1..n}\}$, $\mathtt{Q} = \{\mathtt{h}_j : \mathtt{Q}_j\ ^{j\in 1..m}\}$, and $\mathtt{T} = \{\mathtt{g}_k : \mathtt{T}_k\ ^{k\in 1..p}\}$, with

$$\begin{array}{l} \{\mathtt{g}_k\ ^{k\in 1..p}\} \subseteq \{\mathtt{h}_j\ ^{j\in 1..m}\} \subseteq \{\mathtt{l}_i\ ^{i\in 1..n}\} \\ \text{if } \mathtt{l}_i = \mathtt{h}_j, \text{ then } \Gamma \vdash \mathtt{S}_i \texttt{ <: } \mathtt{Q}_j \\ \text{if } \mathtt{h}_j = \mathtt{g}_k, \text{ then } \Gamma \vdash \mathtt{Q}_j \texttt{ <: } \mathtt{T}_k \end{array}$$

as premises. If $\mathtt{l}_i = \mathtt{g}_k$, then there is an $\mathtt{h}_j$ such that $\mathtt{h}_j = \mathtt{g}_k = \mathtt{l}_i$. Thus, $\Gamma \vdash \mathtt{S}_i \texttt{ <: } \mathtt{Q}_j$ and $\Gamma \vdash \mathtt{Q}_j \texttt{ <: } \mathtt{T}_k$. Observing that $\mathtt{Q}_j$ is an immediate subterm of $\mathtt{Q}$, we apply the outer induction hypothesis to obtain $\Gamma \vdash \mathtt{S}_i \texttt{ <: } \mathtt{T}_k$. By SA-Rcd, $\Gamma \vdash \{\mathtt{l}_i : \mathtt{S}_i\ ^{i\in 1..n}\} \texttt{ <: } \{\mathtt{g}_k : \mathtt{T}_k\ ^{k\in 1..p}\}$.

2. We proceed by an inner induction on the structure of the derivation of $\Gamma, \mathtt{X}\texttt{<:}\mathtt{Q}, \Delta \vdash \mathtt{M} \texttt{ <: } \mathtt{N}$. Most of the cases proceed by straightforward use of the inner induction hypothesis. The interesting case is SA-Trans-TVar with $\mathtt{M} = \mathtt{X}$ and we have $\Gamma, \mathtt{X}\texttt{<:}\mathtt{Q}, \Delta \vdash \mathtt{Q} \texttt{ <: } \mathtt{N}$ as a subderivation. Applying the inner induction hypothesis to this subderivation yields $\Gamma, \mathtt{X}\texttt{<:}\mathtt{P}, \Delta \vdash \mathtt{Q} \texttt{ <: } \mathtt{N}$.

Also, applying weakening (Lemma A.2, part 2) to $\Gamma \vdash$ P <: Q yields Γ, X<:P, $\Delta \vdash$ P <: Q. Now, by part (1) of the outer induction hypothesis (with the same Q), we have Γ, X<:P, $\Delta \vdash$ P <: N. Rule SA-Trans-TVar yields Γ, X<:P, $\Delta \vdash$ X <: N, as required. □

Type Safety (Challenges 2A and 2B): Lemmas

Again, we give the proof of type safety for the whole system including records and pattern matching. We begin with some preliminary technical facts about the typing and subtype relations.

A.4 Lemma [Permutation for Typing]: Suppose that Δ is a well-formed permutation of Γ. If $\Gamma \vdash$ t : T, then $\Delta \vdash$ t : T. □

Proof: By straightforward induction on derivations. □

A.5 Lemma [Weakening for Subtyping and Typing]:

1. If $\Gamma \vdash$ S <: T and Γ, x:U is well-formed, then Γ, x:U $\vdash$ S <: T.
2. If $\Gamma \vdash$ S <: T and Γ, X<:U is well-formed, then Γ, X<:U $\vdash$ S <: T.
3. If $\Gamma \vdash$ t : T and Γ, x:U is well-formed, then Γ, x:U $\vdash$ t : T.
4. If $\Gamma \vdash$ t : T and Γ, X<:U is well-formed, then Γ, X<:U $\vdash$ t : T. □
5. If $\Gamma \vdash$ S <: T and Γ, Δ is well-formed, then Γ, $\Delta \vdash$ S <: T.
6. If $\Gamma \vdash$ t : T and Γ, Δ is well-formed, then Γ, $\Delta \vdash$ t : T.

Proof: The proofs for parts (1) and (2) proceed by straightforward induction on derivations. The SA-All cases require permutation on subtyping derivations (Lemma A.2, part 1). The proofs for parts (3) and (4) also proceed by straightforward induction on derivations, using parts (1) and (2) in the T-Sub and T-TApp cases. The T-Abs, T-TAbs, and T-Let cases require permutation on typing derivations (Lemma A.4). Parts (5) and (6) follow by induction on the number of bindings in Δ, using the first four parts. □

A.6 Lemma [Strengthening]: If Γ, x:Q, $\Delta \vdash$ S <: T, then Γ, $\Delta \vdash$ S <: T. □

Proof: Typing assumptions play no role in subtype derivations. □

The proof of type preservation relies on several lemmas relating substitution with the typing and subtype relations. First, we state an analog for the typing relation of the narrowing lemma for subtyping (Lemma A.3, part 2).

A.7 Lemma [Narrowing for the Typing Relation]: If Γ, X<:Q, $\Delta \vdash$ t : T and $\Gamma \vdash$ P <: Q, then Γ, X<:P, $\Delta \vdash$ t : T. □

Proof: Straightforward induction, using Lemma A.3(2) for the T-Sub case. □

Next, we have the usual lemma relating substitution and the typing relation.

A.8 Lemma [Substitution preserves typing]: If Γ, $\mathtt{x{:}Q}$, $\Delta \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{q} : \mathtt{Q}$, then $\Gamma, \Delta \vdash [\mathtt{x} \mapsto \mathtt{q}]\mathtt{t} : \mathtt{T}$. □

Proof: Induction on a derivation of Γ, $\mathtt{x{:}Q}$, $\Delta \vdash \mathtt{t} : \mathtt{T}$, using the properties above. In particular, we use Lemma A.6 in the T-TApp and T-Sub cases. □

Since we may substitute types for type variables during reduction, we also need a lemma relating type substitution and typing. The proof of this lemma (specifically, the T-Sub case) depends on a new lemma relating substitution and subtyping.

A.9 Definition: We write $[\mathtt{X} \mapsto \mathtt{S}]\Gamma$ for the context obtained by substituting $\mathtt{S}$ for $\mathtt{X}$ in the right-hand sides of all of the bindings in Γ. □

A.10 Lemma [Type substitution preserves subtyping]: If Γ, $\mathtt{X{<:}Q}$, $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$, then $\Gamma, [\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{S} <: [\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}$. □

Note that we need to substitute for $\mathtt{X}$ only in the part of the environment that *follows* the binding of $\mathtt{X}$, since our conventions about scoping require that the types to the left of the binding of $\mathtt{X}$ do not contain $\mathtt{X}$.

Proof: By induction on a derivation of Γ, $\mathtt{X{<:}Q}$, $\Delta \vdash \mathtt{S} <: \mathtt{T}$. The only interesting cases are the following:

Case SA-Trans-TVar*:*

$$\mathtt{S} = \mathtt{Y}$$
$$\mathtt{Y{<:}U} \in (\Gamma, \mathtt{X{<:}Q}, \Delta)$$
$$\Gamma, \mathtt{X{<:}Q}, \Delta \vdash \mathtt{U} <: \mathtt{T}$$

There are two subcases to consider. If $\mathtt{Y} \neq \mathtt{X}$, then the result follows from considering cases on whether $\mathtt{Y{<:}U} \in \Gamma$ or $\mathtt{Y{<:}U} \in \Delta$, the induction hypothesis, and SA-Trans-TVar. (Note that $[\mathtt{X} \mapsto \mathtt{P}]\mathtt{U} = \mathtt{U}$ if $\mathtt{Y{<:}U} \in \Gamma$.)

On the other hand, if $\mathtt{Y} = \mathtt{X}$, then we have $\mathtt{U} = \mathtt{Q}$. By the induction hypothesis, $\Gamma, [\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{Q} <: [\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}$. Note that $[\mathtt{X} \mapsto \mathtt{P}]\mathtt{Q} = \mathtt{Q}$. We then have $\Gamma \vdash \mathtt{P} <: [\mathtt{X} \mapsto \mathtt{P}]\mathtt{Q}$ since $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$. By weakening, $\Gamma, [\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash \mathtt{P} <: [\mathtt{X} \mapsto \mathtt{P}]\mathtt{Q}$, and by transitivity, $\Gamma, [\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash \mathtt{P} <: [\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}$. Therefore, $\Gamma, [\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{X} <: [\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}$.

Case SA-All*:*

$$\mathtt{S} = \forall \mathtt{Z{<:}R_1.S_2} \qquad \mathtt{T} = \forall \mathtt{Z{<:}U_1.T_2}$$
$$\Gamma, \mathtt{X{<:}Q}, \Delta \vdash \mathtt{U_1} <: \mathtt{R_1}$$
$$\Gamma, \mathtt{X{<:}Q}, \Delta, \mathtt{Z{<:}U_1} \vdash \mathtt{S_2} <: \mathtt{T_2}$$

By the induction hypothesis,

$\Gamma, [\mathtt{X} \mapsto \mathtt{P}]\Delta, \mathtt{Z}{<:}[\mathtt{X} \mapsto \mathtt{P}]\mathtt{U_1} \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{S_2} <: [\mathtt{X} \mapsto \mathtt{P}]\mathtt{T_2}$, and
$\Gamma, [\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{U_1} <: [\mathtt{X} \mapsto \mathtt{P}]\mathtt{R_1}$.

By SA-All,

$\Gamma, [\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash \forall \mathtt{Z}{<:}[\mathtt{X} \mapsto \mathtt{P}]\mathtt{R_1}.[\mathtt{X} \mapsto \mathtt{P}]\mathtt{S_2} <: \forall \mathtt{Z}{<:}[\mathtt{X} \mapsto \mathtt{P}]\mathtt{U_1}.[\mathtt{X} \mapsto \mathtt{P}]\mathtt{T_2}$, i.e.,
$\Gamma, [\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}](\forall \mathtt{Z{<:}R_1.S_2}) <: [\mathtt{X} \mapsto \mathtt{P}](\forall \mathtt{Z{<:}U_1.T_2})$. □

A similar lemma relates type substitution and typing.

A.11 LEMMA [TYPE SUBSTITUTION PRESERVES TYPING]: If Γ, $\mathtt{X}$<:$\mathtt{Q}$, $\Delta \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{P}$ <: $\mathtt{Q}$, then Γ, $[\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{t} : [\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}$. □

Proof: By induction on a derivation of Γ, $\mathtt{X}$<:$\mathtt{Q}$, $\Delta \vdash \mathtt{t} : \mathtt{T}$. We give only the interesting cases.

Case T-TAPP: $\mathtt{t} = \mathtt{t}_1\ [\mathtt{T}_2]$ $\quad \Gamma, \mathtt{X}$<:$\mathtt{Q}, \Delta \vdash \mathtt{t}_1 : \forall \mathtt{Z}$<:$\mathtt{T}_{11}.\mathtt{T}_{12}$
$\mathtt{T} = [\mathtt{Z} \mapsto \mathtt{T}_2]\mathtt{T}_{12}$ $\quad \Gamma, \mathtt{X}$<:$\mathtt{Q}, \Delta \vdash \mathtt{T}_2$ <: $\mathtt{T}_{11}$

By the induction hypothesis, Γ, $[\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{t}_1 : [\mathtt{X} \mapsto \mathtt{P}](\forall \mathtt{Z}$<:$\mathtt{T}_{11}.\mathtt{T}_{12})$, i.e., Γ, $[\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{t}_1 : \forall \mathtt{Z}$<:$[\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}_{11}.[\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}_{12}$. By the preservation of subtyping under substitution (Lemma A.10), we have Γ, $[\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}_2$ <: $[\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}_{11}$. By T-TAPP,

Γ, $[\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{t}_1\ [[\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}_2] : [\mathtt{Z} \mapsto [\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}_2]([\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}_{12})$, i.e.
Γ, $[\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}](\mathtt{t}_1\ [\mathtt{T}_2]) : [\mathtt{X} \mapsto \mathtt{P}]([\mathtt{Z} \mapsto \mathtt{T}_2]\mathtt{T}_{12})$.

Case T-SUB: $\Gamma, \mathtt{X}$<:$\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{S}$ $\quad \Gamma, \mathtt{X}$<:$\mathtt{Q}, \Delta \vdash \mathtt{S}$ <: $\mathtt{T}$

By the induction hypothesis, Γ, $[\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{t} : [\mathtt{X} \mapsto \mathtt{P}]\mathtt{S}$. By the preservation of subtyping under substitution (Lemma A.10), we have Γ, $[\mathtt{X} \mapsto \mathtt{P}]\Delta \vdash [\mathtt{X} \mapsto \mathtt{P}]\mathtt{S}$ <: $[\mathtt{X} \mapsto \mathtt{P}]\mathtt{T}$, and the result follows by T-SUB. □

Next, we establish some simple structural properties of subtyping.

A.12 LEMMA [INVERSION OF SUBTYPING (RIGHT TO LEFT)]:

1. If $\Gamma \vdash \mathtt{S}$ <: $\mathtt{X}$, then $\mathtt{S}$ is a type variable.
2. If $\Gamma \vdash \mathtt{S}$ <: $\mathtt{T}_1 \to \mathtt{T}_2$, then either $\mathtt{S}$ is a type variable or else $\mathtt{S}$ has the form $\mathtt{S}_1 \to \mathtt{S}_2$, with $\Gamma \vdash \mathtt{T}_1$ <: $\mathtt{S}_1$ and $\Gamma \vdash \mathtt{S}_2$ <: $\mathtt{T}_2$.
3. If $\Gamma \vdash \mathtt{S}$ <: $\forall \mathtt{X}$<:$\mathtt{U}_1.\mathtt{T}_2$, then either $\mathtt{S}$ is a type variable or else $\mathtt{S}$ has the form $\forall \mathtt{X}$<:$\mathtt{Q}_1.\mathtt{S}_2$ with $\Gamma, \mathtt{X}$<:$\mathtt{U}_1 \vdash \mathtt{S}_2$ <: $\mathtt{T}_2$ and $\Gamma \vdash \mathtt{U}_1$ <: $\mathtt{Q}_1$.
4. If $\Gamma \vdash \mathtt{S}$ <: $\{\mathtt{l}_i : \mathtt{T}_i\ ^{i \in 1..n}\}$, then either $\mathtt{S}$ is a type variable or else $\mathtt{S}$ has the form $\{\mathtt{k}_j : \mathtt{Q}_j\ ^{j \in 1..m}\}$ with $\{\mathtt{l}_i\ ^{i \in 1..n}\} \subseteq \{\mathtt{k}_j\ ^{j \in 1..m}\}$ and where $\mathtt{l}_i = \mathtt{k}_j$ implies $\Gamma \vdash \mathtt{Q}_j$ <: $\mathtt{T}_i$ for every i and j. □

Proof: Each part is immediate from the definition of the subtyping relation. □

Lemma A.12 is used, in turn, to establish one straightforward structural property of the typing relation that is needed in the critical cases of the type preservation proof.

A.13 LEMMA:

1. If $\Gamma \vdash \lambda \mathtt{x}$:$\mathtt{S}_1.\mathtt{s}_2 : \mathtt{T}$ and $\Gamma \vdash \mathtt{T}$ <: $\mathtt{U}_1 \to \mathtt{U}_2$, then $\Gamma \vdash \mathtt{U}_1$ <: $\mathtt{S}_1$ and there is some $\mathtt{S}_2$ such that $\Gamma, \mathtt{x}$:$\mathtt{S}_1 \vdash \mathtt{s}_2 : \mathtt{S}_2$ and $\Gamma \vdash \mathtt{S}_2$ <: $\mathtt{U}_2$.
2. If $\Gamma \vdash \lambda \mathtt{X}$<:$\mathtt{S}_1.\mathtt{s}_2 : \mathtt{T}$ and $\Gamma \vdash \mathtt{T}$ <: $\forall \mathtt{X}$<:$\mathtt{U}_1.\mathtt{U}_2$, then $\Gamma \vdash \mathtt{U}_1$ <: $\mathtt{S}_1$ and there is some $\mathtt{S}_2$ such that $\Gamma, \mathtt{X}$<:$\mathtt{U}_1 \vdash \mathtt{s}_2 : \mathtt{S}_2$ and $\Gamma, \mathtt{X}$<:$\mathtt{U}_1 \vdash \mathtt{S}_2$ <: $\mathtt{U}_2$.
3. If $\Gamma \vdash \{\mathtt{l}_i = \mathtt{t}_i\ ^{i \in 1..n}\} : \mathtt{S}$ and $\Gamma \vdash \mathtt{S}$ <: $\{\mathtt{k}_j : \mathtt{T}_j\ ^{j \in 1..m}\}$, then $\{\mathtt{k}_j\ ^{j \in 1..m}\} \subseteq \{\mathtt{l}_i\ ^{i \in 1..n}\}$ and $\mathtt{l}_i = \mathtt{k}_j$ implies $\Gamma \vdash \mathtt{t}_i : \mathtt{T}_j$ for every i and j. □

Proof: Straightforward induction on typing derivations, using Lemma A.12 for the induction case (rule T-SUB). □

Type Safety (Challenges 2A and 2B): Progress

The progress theorem for $F_{<:}$ is relatively straightforward. We begin by recording a canonical forms property telling us the possible shapes of closed values of arrow, record, and quantifier types.

A.14 Lemma [Canonical Forms]:

1. If $\mathtt{v}$ is a closed value of type $\mathtt{T}_1 \to \mathtt{T}_2$, then $\mathtt{v}$ has the form $\lambda\mathtt{x}{:}\mathtt{S}_1.\mathtt{t}_2$.
2. If $\mathtt{v}$ is a closed value of type $\{\mathtt{l}_i : \mathtt{T}_i \ ^{i\in 1..n}\}$, then $\mathtt{v}$ has the form $\{\mathtt{k}_j = \mathtt{v}_j \ ^{j\in 1..m}\}$ with $\{\mathtt{l}_i \ ^{i\in 1..n}\} \subseteq \{\mathtt{k}_j \ ^{j\in 1..m}\}$.
3. If $\mathtt{v}$ is a closed value of type $\forall\mathtt{X}{<:}\mathtt{T}_1.\mathtt{T}_2$, then $\mathtt{v}$ has the form $\lambda\mathtt{X}{<:}\mathtt{S}_1.\mathtt{t}_2$. □

Proof: All parts proceed by induction on typing derivations; we give the argument only for the third part. (The others are similar.) By inspection of the typing rules, it is clear that the final rule in a derivation of $\vdash \mathtt{v} : \forall\mathtt{X}{<:}\mathtt{T}_1.\mathtt{T}_2$ must be either T-TAbs or T-Sub. If it is T-TAbs, then the desired result is immediate. So suppose the last rule is T-Sub. From the premises of this rule, we have $\vdash \mathtt{v} : \mathtt{S}$ and $\vdash \mathtt{S} <: \forall\mathtt{X}{<:}\mathtt{T}_1.\mathtt{T}_2$. From the inversion lemma (A.12), we know that $\mathtt{S}$ has the form $\forall\mathtt{X}{<:}\mathtt{Q}_1.\mathtt{Q}_2$ ($\mathtt{S}$ can not be a type variable since the typing environment is empty). The result now follows from the induction hypothesis. □

We now observe that any non-value term can be decomposed into an evaluation context and a subterm which can take a step.

A.15 Lemma: If $\vdash \mathtt{t} : \mathtt{T}$, then either $\mathtt{t}$ is a value or there exists an evaluation context E and term $\mathtt{t}_0$ such that $\mathtt{t} = E[\mathtt{t}_0]$ and $\mathtt{t}_0 \longrightarrow \mathtt{t}_0'$. □

Proof: By induction on a derivation of $\vdash \mathtt{t} : \mathtt{T}$. We give only the interesting cases.

Case T-App*:* $\quad \mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2 \qquad \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \vdash \mathtt{t}_2 : \mathtt{T}_{11}$

By the induction hypothesis applied to $\mathtt{t}_1$, we have two subcases to consider.

Subcase: $\quad \mathtt{t}_1$ is a value

We now apply the induction hypothesis to $\mathtt{t}_2$. If $\mathtt{t}_2$ is a value, then by the canonical forms lemma (A.14), $\mathtt{t}_1$ must have the form $\lambda\mathtt{x}{:}\mathtt{S}_1.\mathtt{t}_3$. We then have $\mathtt{t} \longrightarrow [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{t}_3$ by E-AppAbs. Thus, we can take $E = [-]$ and $\mathtt{t}_0 = \mathtt{t}$. Otherwise, $\mathtt{t}_2 = E'[\mathtt{t}_3]$ with $\mathtt{t}_3 \longrightarrow \mathtt{t}_3'$. Thus, we can take $E = \mathtt{t}_1\ E'$ and $\mathtt{t}_0 = \mathtt{t}_3$.

Subcase: $\quad \mathtt{t}_1 = E'[\mathtt{t}_3] \qquad \mathtt{t}_3 \longrightarrow \mathtt{t}_3'$

In this case, take $E = E'\ \ \mathtt{t}_2$ and $\mathtt{t}_0 = \mathtt{t}_3$.

Case T-TApp*:* $\quad \mathtt{t} = \mathtt{t}_1\ \ [\mathtt{T}_2]$

This case is similar to the T-App case.

Case T-LET: $\quad$ $\mathtt{t} = \mathtt{let\ p{=}t_1\ in\ t_2}$ $\qquad \vdash \mathtt{t_1 : T_1}$

By the induction hypothesis, we have two cases to consider.

Subcase: $\quad$ $\mathtt{t_1}$ is a value

Then $\mathtt{t} \longrightarrow \mathit{match}(\mathtt{p},\, \mathtt{t_1})\mathtt{t_2}$ by E-LETV. Take $E = [-]$ and $\mathtt{t_0} = \mathtt{t}$.

Subcase: $\quad$ $\mathtt{t_1} = E'[\mathtt{t_3}]$ $\qquad \mathtt{t_3} \longrightarrow \mathtt{t_3'}$

In this case, take $E = \mathtt{let\ p{=}}E'\ \mathtt{in\ t_3}$ and $\mathtt{t_0} = \mathtt{t_3}$.

Case T-PROJ: $\quad$ $\mathtt{t} = \mathtt{t_1.l_j}$ $\qquad \vdash \mathtt{t_1 : \{l_i : T_i}\ ^{i\in 1..n}\mathtt{\}}$

By the induction hypothesis, we have two cases to consider.

Subcase: $\quad$ $\mathtt{t_1}$ is a value

By the canonical forms lemma (A.14), $\mathtt{t_1}$ must have the form $\mathtt{\{h_k = v_k}\ ^{k\in 1..m}\mathtt{\}}$ with $\{\mathtt{l_i}\ ^{i\in 1..n}\} \subseteq \{\mathtt{h_k}\ ^{k\in 1..m}\}$. By E-PROJRCD, there is a $\mathtt{t'}$ such that $\mathtt{t} \longrightarrow \mathtt{t'}$. Thus, we can take $E = [-]$ and $\mathtt{t_0} = \mathtt{t}$ in this case.

Subcase: $\quad$ $\mathtt{t_1} = E'[\mathtt{t_2}]$ $\qquad \mathtt{t_2} \longrightarrow \mathtt{t_2'}$

In this case, take $E = E'.\mathtt{l_j}$ and $\mathtt{t_0} = \mathtt{t_2}$.

Case T-RCD: $\quad$ $\mathtt{t} = \mathtt{\{l_i = t_i}\ ^{i\in 1..n}\mathtt{\}}$ $\qquad \mathtt{T} = \mathtt{\{l_i : T_i}\ ^{i\in 1..n}\mathtt{\}}$
for each i $\quad \vdash \mathtt{t_i : T_i}$

If every $\mathtt{t_i}$ is a value, then we're done. Otherwise, by the induction hypothesis, there is a least j such that $\mathtt{t_j} = E'[\mathtt{s_0}]$ and $\mathtt{s_0} \longrightarrow \mathtt{s_0'}$. In this case, we can take $E = \mathtt{\{l_i = t_i}\ ^{i\in 1..j-1}, \mathtt{l_j} = E', \mathtt{l_k = t_k}^{k\in j+1..n}\mathtt{\}}$ and $\mathtt{t_0} = \mathtt{s_0}$. □

The proof of progress is now straightforward.

A.16 THEOREM [PROGRESS]: If $\mathtt{t}$ is a closed, well-typed $F_{<:}$ term (i.e., if $\vdash \mathtt{t : T}$ for some $\mathtt{T}$), then either $\mathtt{t}$ is a value or else there is some $\mathtt{t'}$ with $\mathtt{t} \longrightarrow \mathtt{t'}$. □

Proof: By Lemma A.15, either $\mathtt{t}$ is a value or else there is an evaluation context E and term $\mathtt{t_0}$ such that $\mathtt{t} = E[\mathtt{t_0}]$ and $\mathtt{t_0} \longrightarrow \mathtt{t_0'}$. In the latter case, take $\mathtt{t'} = E[\mathtt{t_0'}]$ and observe that $\mathtt{t} \longrightarrow \mathtt{t'}$ by E-CTX. □

Type Safety (Challenges 2A and 2B): Preservation

We begin the proof of preservation by proving a substitution lemma for pattern matching on records.

A.17 LEMMA [MATCHED PATTERNS PRESERVE TYPING]: Suppose $\vdash \mathtt{p : T_1} \Rightarrow \Delta$, that $\Gamma \vdash \mathtt{v_0 : T_1}$, and that $\Gamma, \Delta \vdash \mathtt{t_2 : T_2}$. Then $\Gamma \vdash \mathit{match}(\mathtt{p},\, \mathtt{v_0})\mathtt{t_2 : T_2}$. □

Proof: By induction on the derivation of $\vdash \mathtt{p : T_1} \Rightarrow \Delta$.

Case P-VAR: $\quad$ $\mathtt{p} = \mathtt{x{:}T_1}$ $\qquad \Delta = \mathtt{x : T_1}$

Then $\mathit{match}(\mathtt{x{:}T_1},\, \mathtt{v_0}) = [\mathtt{x} \mapsto \mathtt{v_0}]$, and the result follows from Lemma A.8.

Case P-RCD: $\quad \texttt{p} = \{\texttt{l}_i = \texttt{p}_i\ ^{i\in 1..n}\} \qquad \texttt{T}_1 = \{\texttt{l}_i : \texttt{S}_i\ ^{i\in 1..n}\}$
for each $i \quad \vdash \texttt{p}_i : \texttt{S}_i \Rightarrow \Delta_i$
$\Delta = \Delta_n,\ \ldots,\ \Delta_1$

By Lemma A.13, $\texttt{v}_0$ has the form $\{\texttt{g}_j = \texttt{v}_j\ ^{j\in 1..m}\}$, with $\{\texttt{l}_i\ ^{i\in 1..n}\} \subseteq \{\texttt{g}_j\ ^{j\in 1..m}\}$ and $\texttt{g}_j = \texttt{l}_i$ implies $\Gamma \vdash \texttt{v}_j : \texttt{S}_i$ for every i and j. By definition, we then have $\mathit{match}(\texttt{p}, \texttt{v}_0) = (\sigma_n \circ \cdots \circ \sigma_1)$ where $\texttt{l}_i = \texttt{g}_j$ implies $\mathit{match}(\texttt{p}_i, \texttt{v}_j) = \sigma_i$ for every i and j. Starting from $\Gamma, \Delta \vdash \texttt{t}_2 : \texttt{T}_2$, we iteratively apply the induction hypothesis with $\texttt{p}_1, \texttt{p}_2, \ldots, \texttt{p}_n$, to obtain $\Gamma \vdash \mathit{match}(\texttt{p}, \texttt{v}_1)\texttt{t}_2 : \texttt{T}_2$. More formally, the result follows by induction on n, applying weakening (Lemma A.5) to the judgments $\Gamma \vdash \texttt{v}_j : \texttt{S}_i$ as needed. $\square$

The following lemma relates evaluation contexts and the typing relation.

A.18 LEMMA:

1. If $\Gamma \vdash E[\texttt{t}] : \texttt{T}$ and if for all $\texttt{T}_0$, $\Gamma \vdash \texttt{t} : \texttt{T}_0$ implies $\Gamma \vdash \texttt{t}' : \texttt{T}_0$, then $\Gamma \vdash E[\texttt{t}'] : \texttt{T}$.
2. If $\Gamma \vdash E[\texttt{t}] : \texttt{T}$, then $\Gamma \vdash \texttt{t} : \texttt{T}_0$ for some $\texttt{T}_0$. $\square$

Proof: Both parts are proven by induction on the structure of evaluation contexts. In each case, we consider the last rule used in the derivation of $\Gamma \vdash E[\texttt{t}] : \texttt{T}$ and apply the induction hypothesis. $\square$

We now prove that immediate reduction preserves the types of terms.

A.19 LEMMA: If $\Gamma \vdash \texttt{t} : \texttt{T}$ and $\texttt{t} \longrightarrow \texttt{t}'$, where the derivation of $\texttt{t} \longrightarrow \texttt{t}'$ ends in a rule other than E-CTX, then $\Gamma \vdash \texttt{t}' : \texttt{T}$. $\square$

Proof: By induction on a derivation of $\Gamma \vdash \texttt{t} : \texttt{T}$. We give only the interesting cases.

Case T-APP: $\quad \texttt{t} = \texttt{t}_1\ \texttt{t}_2 \qquad \Gamma \vdash \texttt{t}_1 : \texttt{T}_{11}\to\texttt{T}_{12} \qquad \texttt{T} = \texttt{T}_{12} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{T}_{11}$

The derivation of $\texttt{t} \longrightarrow \texttt{t}'$ must end in E-APPABS. Thus, $\texttt{t}_1 = \lambda\texttt{x}:\texttt{U}_{11}.\texttt{u}_{12}$ and $\texttt{t}' = [\texttt{x} \mapsto \texttt{t}_2]\texttt{u}_{12}$. By Lemma A.13, $\Gamma,\ \texttt{x}:\texttt{U}_{11} \vdash \texttt{u}_{12} : \texttt{U}_{12}$ for some $\texttt{U}_{12}$ with $\Gamma \vdash \texttt{T}_{11}$ <: $\texttt{U}_{11}$ and $\Gamma \vdash \texttt{U}_{12}$ <: $\texttt{T}_{12}$. By narrowing (Lemma A.7) and the preservation of typing under substitution (Lemma A.8), $\Gamma \vdash [\texttt{x} \mapsto \texttt{t}_2]\texttt{u}_{12} : \texttt{U}_{12}$, from which we obtain $\Gamma \vdash [\texttt{x} \mapsto \texttt{t}_2]\texttt{u}_{12} : \texttt{T}_{12}$ by T-SUB.

Case T-PROJ: $\quad \texttt{t} = \texttt{t}_1.\texttt{l}_j \qquad \texttt{T} = \texttt{T}_j \qquad \Gamma \vdash \texttt{t}_1 : \{\texttt{l}_i : \texttt{T}_i\ ^{i\in 1..n}\}$

The derivation of $\texttt{t} \longrightarrow \texttt{t}'$ must end in E-PROJRCD. Thus, $\texttt{t}_1$ is a record value $\{\texttt{h}_k = \texttt{v}_k\ ^{k\in 1..m}\}$. By Lemma A.13, we have $\{\texttt{l}_i\ ^{i\in 1..n}\} \subseteq \{\texttt{h}_k\ ^{k\in 1..m}\}$ and $\texttt{l}_i = \texttt{h}_k$ implies $\Gamma \vdash \texttt{v}_k : \texttt{T}_i$ for every i and j. By E-PROJRCD, we have $\texttt{t}' = \texttt{v}_k$ where $\texttt{h}_k = \texttt{l}_j$. Thus, $\Gamma \vdash \texttt{t}' : \texttt{T}_j$ as required.

Case T-TAPP: $\quad \texttt{t} = \texttt{t}_1\ [\texttt{T}_2] \qquad \Gamma \vdash \texttt{t} : \forall\texttt{X}$<:$\texttt{T}_{11}.\texttt{T}_{12}$
$\texttt{T} = [\texttt{X} \mapsto \texttt{T}_2]\texttt{T}_{12} \qquad \Gamma \vdash \texttt{T}_2$ <: $\texttt{T}_{11}$

The derivation of $\texttt{t} \longrightarrow \texttt{t}'$ must end in E-TAPPTABS. Thus, $\texttt{t}_1 = \lambda\texttt{X}$<:$\texttt{U}_{11}.\texttt{u}_{12}$ and $\texttt{t}' = [\texttt{X} \mapsto \texttt{T}_2]\texttt{u}_{12}$. By Lemma A.13, $\Gamma \vdash \texttt{T}_{11}$ <: $\texttt{U}_{11}$ and $\Gamma, \texttt{X}$<:$\texttt{T}_{11} \vdash \texttt{u}_{12} : \texttt{U}_{12}$ for some $\texttt{U}_{12}$ with $\Gamma, \texttt{X}$<:$\texttt{T}_{11} \vdash \texttt{U}_{12}$ <: $\texttt{T}_{12}$. By the preservation of typing under substitution (Lemma A.11), $\Gamma \vdash [\texttt{X} \mapsto \texttt{T}_2]\texttt{u}_{12} : [\texttt{X} \mapsto \texttt{T}_2]\texttt{U}_{12}$, from which $\Gamma \vdash [\texttt{X} \mapsto \texttt{T}_2]\texttt{u}_{12} : [\texttt{X} \mapsto \texttt{T}_2]\texttt{T}_{12}$ follows by Lemma A.10 and T-SUB.

Case T-Let: $\quad \mathtt{t} = \mathtt{let\ p{=}t_1\ in\ t_2}$

$$\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \vdash \mathtt{p} : \mathtt{T}_1 \Rightarrow \Delta \qquad \Gamma, \Delta \vdash \mathtt{t}_2 : \mathtt{T}$$

The derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$ must end in E-LetV. Thus, $\mathtt{t}_1$ is a value and $\mathtt{t} \longrightarrow$ *match*$(\mathtt{p}, \mathtt{t}_1)\mathtt{t}_2$. The result then follows by Lemma A.17. □

Finally, we prove the main preservation theorem.

A.20 Theorem [Preservation]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathtt{T}$. □

Proof: By induction on a derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$. Suppose the derivation ends in E-Ctx. Then, $\mathtt{t} = E[\mathtt{t}_0]$, $\mathtt{t}_0 \longrightarrow \mathtt{t}_0'$, and $\mathtt{t}' = E[\mathtt{t}_0']$. By Lemma A.18, $\Gamma \vdash \mathtt{t}_0 : \mathtt{T}_0$ for some $\mathtt{T}_0$. By the induction hypothesis, for any $\mathtt{T}_0$ such that $\Gamma \vdash \mathtt{t}_0 : \mathtt{T}_0$, we have $\Gamma \vdash \mathtt{t}_0' : \mathtt{T}_0$. The result then follows by Lemma A.18. In all other cases, the result follows by Lemma A.19. □