Type Soundness Proofs with Definitional Interpreters

Nada Amin* Tiark Rompf †

*EPFL: {first.last}@epfl.ch
†Purdue University: {first}@purdue.edu

Abstract

While type soundness proofs are taught in every graduate PL class, the gap between realistic languages and what is accessible to formal proofs is large. In the case of Scala, it has been shown that its formal model, the Dependent Object Types (DOT) calculus, cannot simultaneously support key metatheoretic properties such as environment narrowing and subtyping transitivity, which are usually required for a type soundness proof. Moreover, Scala and many other realistic languages lack a general substitution property.

The first contribution of this paper is to demonstrate how type soundness proofs for advanced, polymorphic, type systems can be carried out with an operational semantics based on high-level, definitional interpreters, implemented in Coq. We present the first mechanized soundness proofs in this style for System $F_{<:}$ and several extensions, including mutable references. Our proofs use only straightforward induction, which is significant, as the combination of big-step semantics, mutable references, and polymorphism is commonly believed to require coinductive proof techniques.

The second main contribution of this paper is to show how DOT-like calculi emerge from straightforward generalizations of the operational aspects of $F_{<:}$, exposing a rich design space of calculi with path-dependent types inbetween System F and DOT, which we dub the System D Square.

By working directly on the target language, definitional interpreters can focus the design space and expose the invariants that actually matter at runtime. Looking at such runtime invariants is an exciting new avenue for type system design.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Definitional interpreters, type soundness, dependent object types, DOT, Scala

1. Introduction

The main contribution of this paper is to demonstrate how type soundness for advanced, polymorphic, type systems can be proved with respect to an operational semantics based on high-level, definitional interpreters, implemented directly in a total functional language like Coq. While this has been done before for very simple, monomorphic, type systems [49, 17], we are the first to demonstrate that, with some additional machinery, this approach scales to realistic, polymorphic type systems that include subtyping, abstract types, types with binders, mutable references, exceptions, and certain forms of dependent types.

Our motivation is twofold. The first is intellectual: It is commonly believed that proofs based on big-step semantics (of which definitional interpreters are a proper sub-category) are difficult or unsatisfactory. One criticism is that big-step semantics do not distinguish errors from nontermination. Thus, if taken as the basis of a type soundness proof, what is shown is only preservation, but not progress. Hence, the result is significantly weaker than a comparable small-step proof. Another commonly held belief is that advanced features such as mutable references require coinduction or other non-standard proof techniques in big-step.

With this paper, we present convincing evidence that, contrary to this view, definitional interpreters have no inherent drawbacks to small-step semantics for deterministic languages: making evaluation functional, instead of relational, and parameterizing over a step counter to make evaluation total, enables precise distinction between timeouts, errors, and normal values, leading to strong soundness results. Established techniques like monads [36] can be used to abstract over these case distinctions and keep the interpreter implementation elegant [54]. Furthermore, auxiliary invariants such as store typings can be used in the same way as in small-step proofs to eliminate the need for non-standard proof techniques, and gradually extend proofs for simple languages with multiple advanced features, without any disruptive changes to the proof structure.

Our second motivation is pragmatic: While type soundness proofs are taught in every graduate PL class, the gap between realistic languages and what is accessible to formal proofs is large. In the case of Scala, it has been shown that its formal model, the Dependent Object Types (DOT) calculus, cannot simultaneously support key metatheoretic properties such as environment narrowing and subtyping transitivity, which are usually required for a type soundness proof. Moreover, Scala and many other realistic languages lack a general substitution property. Thus, applying the Wright & Felleisen method [56] requires ingenuity to extend the language syntax and type system with auxiliary constructs to support subject reduction. It also requires at least an informal argument of adequacy, showing that the syntactic theory faithfully models the intended language semantics.

In the case of Scala, developing a sound formal model that captures the essence of its type system was an open problem for more than a decade. Over the years, a handful of talented postdocs and students tried many variations of syntactic theories, but ultimately failed to find one they could prove sound. The situation changed when looking at definitional interpreters. The split between the static and the runtime world exposed the key invariants that had to be maintained, identified problems with previous proof attempts that were not apparent in small-step, and finally lead to the first soundness proof for DOT. Moreover, starting from the runtime invariants has lead to a simpler and more regular calculus. The resulting proof is easily translated to small-step, and has been presented in detail elsewhere. While the resulting syntactic theory looks pleasant and blindingly obvious in hindsight, nobody had thought of this particular variant before.

The lesson we draw from this is again two-fold: First, by working directly on the target language (instead of an augmented version) definitional interpreters can constrain the design space in a good way and expose the invariants that actually matter at runtime. Second, looking at such runtime invariants is an exciting new avenue for type system design. In this paper, we illustrate how DOT-like calculi emerge as generalizations of the static typing rules to fit the operational typing aspects of the $F_{<:}$ based system—in some cases almost like removing artificial restrictions. We expose a rich design space of calculi with path-dependent types inbetween System F and DOT, which we dub the System D Square. By this, we put Scala and DOT on a firm theoretical foundation grounded in existing, well-studied, type systems, alluding to Wadler's point that "good languages are discovered, not invented" [55].
We make the following contributions:

- We discuss limitations of the syntactic approach to soundness and review a proof strategy based on high-level definitional interpreters (Section 2).
- We demonstrate that this proof strategy scales to advanced polymorphic type systems, presenting the first soundness proof for $F_{<:}$ in this style (Section 3).
- We further show that extensions such as mutable references and exceptions can be added gradually and without much difficulty, requiring only straightforward induction, and effectively separating failures from divergence (Section 4).
- We now take the opposite direction and investigate how the internal invariants in the $F_{<:}$ proof yield new static type systems. Enabling lower-bounded quantification (System $F_{<:>}$) leads to user-definable subtyping theories. Unifying term and type variables (System D) leads to path-dependent types (Section 5).
- We discuss how the combination of these two extensions (System $D_{<:>}$) can be identified as the core of Scala and DOT, which we have rediscovered from first principles. We thus put DOT on a firm theoretical grounding, by showing how it emerges as a generalization of the static typing rules of $F_{<:}$ to its runtime typing aspects (Section 6).

Our mechanized proofs are available from: `popl17.namin.net`

2. Definitional Interpreters for Type Soundness

Today, the dominant method for proving soundness of a type system is the syntactic approach of Wright and Felleisen [56]. Its key components are the progress and preservation lemmas with respect to a small-step operational semantics based on term rewriting. While this syntactic approach has a lot of benefits, as described in great detail in the original 1994 paper [56], there are also some drawbacks. An important one is that reduction semantics often pose a question of adequacy: realistic language implementations do not proceed by rewriting, so if the aim is to model an existing language, at least an informal argument needs to be made that the given reduction relation faithfully implements the intended semantics. Furthermore, few realistic languages actually enjoy the subject reduction property. If simple substitution does not hold, the syntactic approach is more difficult to apply and requires stepping into richer languages in ways that are often non-obvious. Again, care must be taken that these richer languages are self-contained and match the original intention.

To motivate a substitution-free semantics, we present three examples next where naive substitution does not preserve types:

Example 1: Return statements Consider a simple program in a language with `return` statements:

```
def fun(c) = if (c) return x; y
fun(true)
```

Taking a straightforward small-step execution strategy, this program will reduce to:

```
→ if (true) return x; y
```

But now the `return` has become unbound. We need to augment the language and reduce to an auxiliary construct like this:

```
→ scope { if (true) return x; y }
```

This means that we need to work with a richer language than we had originally intended, with additional syntax, typing, and reduction rules like the following:

```
scope E[ return v ] → v      scope v → v
```

Example 2: Private members As another example, consider an object-oriented language with access modifiers.

```
class Foo { private val data = 1;  def foo(x) = x * this.data }
```

Starting with a term

```
val a = new Foo; a.foo(7) / S
```

where `S` denotes a store, small-step reduction steps may lead to:

```
→ l0.foo(7)          / S, (l0 -> Foo(data=1))
→ x * l0.data        / S, (l0 -> Foo(data=1))
```

But now there is a reference to private field `data` outside the scope of class `Foo`.

We need a special rule to ignore access modifiers for 'runtime' objects in the store, versus other expresssions that happen to have type `Foo`. We still want to disallow `a.data` if `a` is a normal variable reference or some other expression.

Example 3: Method overloading Looking at a realistic language, many type preservation issues are documented in the context of Java, which were discussed at length on the Types mailing list [53], back in the time when Java's type system was an object of study.

Most of these examples relate to static method overloading, and to Java's conditional expressions `c ? a : b`, which require `a` and `b` to be in a subtype relationship because Java does not have least upper bounds. It is worth noting that these counterexamples to preservation are not actual type safety violations.

2.1 Alternative Semantic Models

So what can we do if our object of study does not naturally fit a rewriting and substitution model of execution? Of course one option is to make it fit (perhaps with force), but an easier path may be to pick a different semantic model. Before the syntactic approach, denotational semantics [48] and Kahn's natural semantics (or 'big-step' semantics) [30] were the tools of the trade.

Big-step semantics in particular has the benefit of being more 'high-level', in the sense of being closer to actual language implementations. Environment-based formulations are as natural as substitution-based ones, and have the additional advantage of working with unmodified syntax of the target language. The downside of big-step semantics for soundness proofs is that failure cases and nontermination are not easily distinguished. This often requires tediously enumerating all possible failure cases, which may cause a blow-up in the required rules and proof cases. Moreover, in the history of big-step proofs, advanced language features such as recursive references have required specific proof techniques (e.g. coinduction) [51] which made it hard to compose proofs for different language features. In general, polymorphic type systems pose difficulties for substitution-free semantics, because type variables need to be related across different contexts.

But the circumstances have changed since 1994. Today, most formal work is done in proof assistants such as Coq, and no longer with pencil and paper. This means that we can use software implementation techniques like monads (which, ironically were developed in the context of denotational semantics [36]) to handle the complexity of failure cases [54]. Moreover, using simple but clever inductive techniques such as step counters we can avoid the need for coinduction and other complicated techniques in many cases.

In the following, we present our approach to type soundness proofs with definitional interpreters in the style of Reynolds [44]: high-level evaluators implemented in a (total) functional language. As we will see, in a functional system such as Coq or Agda, we can implement such evaluators quite naturally.

2.2 Simply Typed Lambda Calculus: Siek's 3 Easy Lemmas

We build our exposition on Siek's type safety proof for a dialect of simply typed lambda calculus (STLC) [49], which in turn takes inspiration from Ernst, Ostermann and Cook's semantics in their formalization of virtual classes [23].

The starting point is a fairly standard definitional interpreter for STLC, shown in Figure 1 together with the STLC syntax and typing rules. We opt to show the interpreter in actual Coq syntax, but stick to formal notation for the language definition and typing rules. The interpreter consists of three functions: one for primitive operations (which we elide), one for variable lookups, and one main evaluation function `eval`, which ties everything together. Instead of working exclusively on terms, as a reduction semantics would do, the interpreter maps terms to a separate domain of values `v`. Values include primitives, and closures, which pair a term with an environment.

Notions of Type Soundness What does it mean for a language to be type safe? We follow Wright and Felleisen [56] in viewing a static type system as a filter that selects well-typed programs from a larger universe of untyped programs. In their definition of type soundness, a partial function `evalp` defines the semantics of untyped programs, returning `Error` if the evaluation encounters a type error, or any other answer for a well-typed result. We assume here that the result in this case will be `Val v`, for some value `v`. For evaluations that do not terminate, `evalp` is undefined.

The simplest soundness property states that well-typed programs do not go wrong.

Definition 1 (Weak soundness)**.**

$$\frac{\emptyset \vdash e : T}{\text{evalp } e \neq \text{Error}}$$

A stronger soundness property states that if the evaluation terminates, the result value must have the same type as the program expression, assuming that values are classified by types as well.

Definition 2 (Strong soundness)**.**

$$\frac{\emptyset \vdash e : T \qquad \text{evalp } e = r}{r = \text{Val } v \qquad v : T}$$

In our case, assigning types to values is achieved by the rules in the lower half of Figure 1.

Partiality Fuel To reason about the behavior of our interpreter, and to implement it in Coq in the first place, we have to get a handle on potential nontermination, and make the interpreter a total function. Again we follow Siek [49] by first making all error cases explicit by wrapping the result of each operation in an `option` data type with alternatives `Val v` and `Error`. This leaves us with possible nontermination. We parameterize the interpreter over a step index or 'fuel value' `n`, which bounds the amount of work the interpreter is allowed to do. If it runs out of fuel, the interpreter returns `Timeout`, otherwise `Done r`, where `r` is the option type introduced above.

It is convenient to treat this type of answers as a (layered) monad and write the interpreter in monadic `do` notation (as done in Figure 1). The `FUEL` operation in the first line desugars to a simple non-zero check:

```
match n with
  | z⇒TIMEOUT
  | S n1⇒...
end
```

Syntax

$$\begin{array}{lcl} T & ::= & B \mid T \to T \\ t & ::= & c \mid x \mid \lambda x : T.t \mid t\ t \\ v & ::= & c \mid \langle H, \lambda x : T.t \rangle \\ r & ::= & \text{Timeout} \mid \text{Done}\ (\text{Error} \mid \text{Val}\ v) \\ \Gamma & ::= & \emptyset \mid \Gamma, x : T \\ H & ::= & \emptyset \mid H, x : v \end{array}$$

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash c : B$$

$$\frac{\Gamma \ni x : T}{\Gamma \vdash x : T}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2\ ,\ t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

Consistent environments $\boxed{\Gamma \vDash H}$

$$\emptyset \vDash \emptyset$$

$$\frac{\Gamma \vDash H \qquad v : T}{\Gamma, x : T \vDash H, x : v}$$

Value type assignment $\boxed{v : T}$

$$c : B$$

$$\frac{\Gamma \vDash H \qquad \Gamma, x : T_1 \vdash t : T_2}{\langle H, \lambda x : T_1.t \rangle : T_1 \to T_2}$$

Definitional Interpreter

```
(* Coq data types and auxiliary functions elided *)
Fixpoint eval(n: nat)(env: venv)(t: tm){struct n}:
option (option vl) :=
  DO n1 ⇐ FUEL n;                              (* totality    *)
  match t with
    | tcst c     ⇒DONE VAL (vcst c)            (* constant    *)
    | tvar x     ⇒DONE (lookup x env)          (* variable    *)
    | tabs x ey  ⇒DONE VAL (vabs env x ey)     (* lambda      *)
    | tapp ef ex ⇒                             (* application *)
      DO vf ⇐ eval n1 env ef;
        DO vx ⇐  eval n1 env ex;
        match vf with
          | (vabs env2 x ey)⇒
            eval n1 ((x,vx)::env2) ey
          | _⇒ERROR
        end
  end.
```

Figure 1. STLC: Syntax and Semantics

There are other ways to define monads that encode partiality (e.g. a coinductively defined partiality monad [17]), but this simple method has the benefit of enabling easy inductive proofs about all executions of the interpreter, by performing a simple induction over n. If a fact is proved for all executions of length n, for all n, then it must hold for all finite executions. Specifically, infinite executions are by definition not 'stuck'.

Proof Structure For the type safety proof, the 'three easy lemmas' [49] are as follows. There is one lemma per function of the interpreter.

Lemma 1 (Primitives)**.** *Well-typed primitive operations are not stuck and preserve types.*

We are omitting primitive operations for simplicity, so we skip this lemma.

Lemma 2 (Lookup)**.** *Well-typed environment lookups are not stuck and preserve types.*

$$\frac{\Gamma \vDash H \qquad \text{lookup } x\ \Gamma = \text{Some } T}{\text{lookup } x\ H = \text{Val } v \qquad v : T}$$

Proof. By structural induction over the environment and case distinction on whether the lookup succeeds. □

Lemma 3 (Eval)**.** *For all n, if the interpreter returns a result that is not a timeout, then the result is a value (i.e. not stuck) and it is well-typed.*

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vDash H \qquad \text{eval } n\ H\ e = \text{Done } r}{r = \text{Val } v \qquad v : T}$$

Proof. By induction on n, and case analysis on the term e. □

It is easy to see that this lemma corresponds to the strong soundness notion (Definition 2). In fact, we can define a partial function evalp e to probe eval $n\ \emptyset\ e$ for all $n = 0, 1, 2, ...$ and return the first non-timeout result, if one exists. Restricting to the empty environment then yields *exactly* Wright and Felleisen's statement of strong soundness [56]:

Theorem 1 (Soundness of STLC)**.**

$$\frac{\emptyset \vdash e : T \qquad \text{evalp } e = r}{r = \text{Val } v \qquad v : T}$$

Using classical reasoning we can conclude that either evaluation diverges (i.e. it times out for all n), or there exists an n for which the result is well-typed.

3. Type Soundness for System $F_{<:}$

We now turn our attention to System $F_{<:}$ [13], moving beyond type systems that have been previously formalized with definitional interpreters. We pick $F_{<:}$ because it combines a range of interesting features, because its type soundness is well-studied, in particular through the POPLmark challenge [10], and because $F_{<:}$ can serve as a basis for extensions that lead up to formalizations of key Scala features (see Sections 5 and 6).

The syntax and static typing rules of $F_{<:}$ are defined in Figure 2. In addition to STLC, we have type abstraction and type application, and subtyping with upper bounds. The calculus is more expressive than STLC and more interesting from a formalization perspective, in particular because it contains type variables. These are bound in the environment, which means that we need to consider types in relation to the environment they were defined in.

3.1 Operational Semantics: The Definitional Interpreter

What would be a suitable runtime semantics for passing type arguments to type abstractions? The usual small-step semantics uses substitution to eliminate type arguments (Figure 3):

$$(\Lambda Y <: T_1.t)[T_2] \longrightarrow [T_2/Y]t$$

We could do the same in our definitional interpreter, which we extend from Figure 1 to add a new case for type application:

```
(* ... *)
    | ttapp ef T⇒
      DO vf ⇐ eval n1 env ef;
        match vf with
          | (vtabs env2 x ey)⇒
              (* ... first attempt ... *)
              eval n1 env2 (substitute ey x T)
          | _⇒ERROR end
(* ... *)
```

Syntax

$$\begin{array}{lll} X & ::= & Y \mid Z \\ T & ::= & X \mid \top \mid T \to T \mid \forall Z <: T.T^Z \\ t & ::= & x \mid \lambda x : T.t \mid \Lambda Y <: T.t \mid t\ t \mid t\ [T] \\ \Gamma & ::= & \emptyset \mid \Gamma, x : T \mid \Gamma, X <: T \end{array}$$

Subtyping $\boxed{\Gamma \vdash S <: U}$

$$\Gamma \vdash S <: \top$$

$$\Gamma \vdash X <: X$$

$$\frac{\Gamma \ni X <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}$$

$$\frac{\Gamma \vdash S_2 <: S_1\ ,\ T_1 <: T_2}{\Gamma \vdash (S_1 \to T_1) <: (S_2 \to T_2)}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, Z <: S_2 \vdash T_1^Z <: T_2^Z}{\Gamma \vdash (\forall Z <: S_1.T_1^Z) <: (\forall Z <: S_2.T_2^Z)}$$

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3}$$

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \ni x : T}{\Gamma \vdash x : T}$$

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x : S.t : S \to T}$$

$$\frac{\Gamma \vdash t_1 : S \to T\ ,\ t_2 : T}{\Gamma \vdash t_1\ t_2 : T}$$

$$\frac{\Gamma, Y <: S \vdash t : T^Y}{\Gamma \vdash \Lambda Y <: S.t : \forall Z <: S.T^Z}$$

$$\frac{\Gamma \vdash t_1 : \forall Z <: U.T^Z\ ,\ T_2 <: U}{\Gamma \vdash t_1[T_2] : T^{T_2}}$$

$$\frac{\Gamma \vdash t : S\ ,\ S <: T}{\Gamma \vdash t : T}$$

Figure 2. $F_{<:}$: syntax and static semantics

Value Syntax

$$v \quad ::= \quad \lambda x : T.t \mid \Lambda Y <: T.t$$

Reduction $\boxed{t \to t'}$

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad \frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2} \qquad \frac{t_1 \to t_1'}{t_1[T_2] \to t_1'[T_2]}$$

$$(\Lambda Y <: T_1.t_1)[T_2] \to [T_2/Y]t_1$$

$$(\lambda x : T_1.t_1)\ v_2 \to [v_2/x]t_1$$

Figure 3. $F_{<:}$: small-step semantics (call-by-value)

But then, the interpreter would have to modify program terms at runtime. This would be odd for an interpreter, which is meant to be simple, and it would conflict with our goal of a *substitution-free* semantics, to circumvent difficulties in formal reasoning due to substitution in small-step semantics (see Section 2).

Types in Environments A better idea, more consistent with an environment-passing interpreter, is to put the type argument into the runtime environment as well:

```
(* ... second attempt ... *)
eval n1 ((x,vty T)::env2) ey
```

However, this leads to a problem: the type `T` may refer to other type variables that are bound in the current environment at the call site (`env`), while evaluation switches to the environment at definition site (`env2`) of the type abstraction called. We could potentially resolve all the references, and substitute their occurrences in the type, but this will no longer work if types can be recursive. In any case, we wouldn't want to do such type resolution and manipulation in the interpreter, i.e. during evaluation. So instead, we pass the caller environment along with the type.

```
(* ... final attempt ... *)
eval n1 ((x,vty env T)::env2) ey
```

In effect, type arguments $\langle H, T\rangle$ (written `vty env T` in the code above) become very similar to function closures, in that they close over their defining environment. Intuitively, this makes a lot of sense, and is consistent with the overall workings of our interpreter.

Semantic Equivalence As a reassurance that our definitional interpreter faithfully implements the well-known small-step call-by-value semantics of $F_{<:}$ (Figure 3), we prove a semantic equivalence theorem.

We first define a correspondence relation $\sim$ between closure values of the interpreter and closed lambda terms and type abstractions, such that the closure environment is recursively substituted into the term. This makes the nature of the runtime environment as delayed substitution explicit. The $\sim$ relation is then lifted to interpreter results r, such that `Done (Val v)` corresponds to a term value `v`, `Done Error` to any stuck term (irreducible non-value), and `Timeout` to any term.

Theorem 2 (Semantic Equivalence)**.** *If $e \to^* e'$, then $\exists n.eval\ n\ \emptyset\ e \sim e'$, and if eval $n\ \emptyset\ e = r$, then $\exists e'.e \to^* e'$ with $r \sim e'$.*

Proof. The first direction is by induction over the $\to^*$ derivation, the second direction is by induction over n. □

3.2 Runtime Invariants

Having defined a suitable interpreter, the next step is to identify and model runtime invariants including value typing that will enable our soundness proof. For $F_{<:}$, the key novel features compared to STLC are subtyping and abstract types.

Runtime Subtyping Since we model runtime type arguments as closure, we need to account for each defining environment when comparing types at runtime. In our approach, on each side, the runtime subtyping judgement takes not only a type but also its defining environment:

$$J \vdash H_1\ T_1 <: H_2\ T_2$$

The judgement pairs each type T with a corresponding runtime environment H. Each runtime environment H maps a term variable x to a value v or a type variable Y to a "type" closure, which pairs a (runtime) environment with a type. In the Coq code above, we use the `vty` marker to distinguish between term variables and type variables in the mapping. We will explain the role of the J environments shortly, in the section on abstract types below.

The runtime typing rules are shown in Figure 4. Note that the rules labeled 'concrete type variables' are entirely structural:

Syntax

$$\begin{array}{lll} v & ::= & \langle H, \lambda x : T.t\rangle \mid \langle H, \Lambda Y <: T.t\rangle \\ H & ::= & \emptyset \mid H, x : v \mid H, Y = \langle H, T\rangle \\ J & ::= & \emptyset \mid J, Z <: \langle H, T\rangle \end{array}$$

Runtime subtyping $\boxed{J \vdash H_1\ T_1 <: H_2\ T_2}$

$$\frac{}{J \vdash H_1\ T <: H_2\ \top}$$

$$\frac{J \vdash H_2\ S_2 <: H_1\ S_1 \qquad J \vdash H_1\ T_1 <: H_2\ T_2}{J \vdash H_1\ (S_1 \to T_1)\ <:\ H_2\ (S_2 \to T_2)}$$

$$\frac{J \vdash H_2\ S_2 <: H_1\ S_1 \qquad J, Z <: \langle H_2, S_2\rangle \vdash H_1\ T_1^Z\ <:\ H_2\ T_2^Z}{J \vdash H_1\ (\forall Z <: S_1.T_1^Z)\ <:\ H_2\ (\forall Z <: S_2.T_2^Z)}$$

Abstract type variables

$$\frac{}{J \vdash H_1\ Z <: H_2\ Z}$$

$$\frac{J \ni Z <: \langle H, U\rangle \qquad J \vdash H\ U <: H_2\ T}{J \vdash H_1\ Z <: H_2\ T}$$

Concrete type variables

$$\frac{H_1 \ni Y_1 = \langle H, T\rangle \qquad H_2 \ni Y_2 = \langle H, T\rangle}{J \vdash H_1\ Y_1 <: H_2\ Y_2}$$

$$\frac{H_1 \ni Y = \langle H, U\rangle \qquad J \vdash H\ U <: H_2\ T}{J \vdash H_1\ Y <: H_2\ T}$$

$$\frac{J \vdash H_1\ T <: H\ S \qquad H_2 \ni Y = \langle H, S\rangle}{J \vdash H_1\ T <: H_2\ Y}$$

Transitivity

$$\frac{J \vdash H_1\ T_1 <: H_2\ T_2 \qquad H_2\ T_2 <: H_3\ T_3}{J \vdash H_1\ T_1 <: H_3\ T_3}$$

Consistent environments $\boxed{\Gamma \vDash H\ J}$

$$\emptyset \vDash \emptyset\ \emptyset$$

$$\frac{\Gamma \vDash H\ \emptyset \qquad H \vdash v : T}{\Gamma, x : T \vDash (H, x : v)\ \emptyset}$$

$$\frac{\Gamma \vDash H\ \emptyset \qquad \emptyset \vdash H_1\ T_1 <: H\ T}{\Gamma, Y <: T \vDash (H, Y = \langle H_1, T_1\rangle)\ \emptyset}$$

$$\frac{\Gamma \vDash H\ J}{\Gamma, Z <: T \vDash H\ (J, Z <: \langle H, T\rangle)}$$

Value type assignment $\boxed{H \vdash v : T}$

$$\frac{\Gamma \vDash H\ \emptyset \qquad \Gamma, x : S \vdash t : T}{H \vdash \langle H, \lambda x : S.t\rangle : S \to T}$$

$$\frac{\Gamma \vDash H\ \emptyset \qquad \Gamma, Y <: S \vdash t : T^Y}{H \vdash \langle H, \Lambda Y <: S.t\rangle : \forall Z <: S.T^Z}$$

$$\frac{H_1 \vdash v : T_1 \qquad \emptyset \vdash H_1\ T_1 <: H_2\ T_2}{H_2 \vdash v : T_2}$$

Figure 4. $F_{<:}$: runtime typing

different type variables Y_1 and Y_2 are treated equal if they map to the same $H\ T$ pair. In contrast to the surface syntax of $F_{<:}$, there is not only a rule for $Y <: T$ but also a symmetric one for $T <: Y$, i.e. with a type variable on the right hand side. This symmetry is necessary to model runtime type equality through subtyping, which gives us a handle on those cases where a small-step semantics would rely on type substitution.

Another way to look at this is that the runtime subtyping relation removes abstraction barriers (nominal variables, only one-sided comparison with other types) that were put in place by the static subtyping relation.

We further note in passing that formal reasoning involving subtyping transitivity becomes more difficult in the runtime subtyping compared to the static $F_{<:}$ subtyping, because of type variables in the middle of a chain $T_1 <: Y <: T_2$, which should contract to $T_1 <: T_2$. We will get back to this question and related ones in Section 3.3 and specifically Lemma 11.

Abstract Types So far we have seen how to handle types that correspond to existing type objects. We now turn to subtyping for $\forall$ types. In the static subtyping relation, this rule introduces a new type binding: in the premise that compares the result types, the context is extended with a binding for the quantified type variable.

How can we support this rule at runtime? We cannot quite use only the facilities discussed so far, because this would require us to 'invent' new hypothetical objects $\langle H, T\rangle$, which are not actually created during execution, and insert them into another runtime environment. Furthermore, the premise subtyping the result types should hold not just *for some* hypothetical object, but *for all* hypothetical objects that satisfy the bound on the type variable. Semantically, we do not have an exact type instantiation, only an upper bound, and we should be careful about the distinction.

Our solution is rather simple: we split the runtime environment into abstract (J) and concrete (H) parts. While the environments H, as discussed above, map type variables to concrete type values created at runtime, we use a shared environment J, that maps type variables to *hypothetical* objects: $\langle H, T\rangle$ pairs that may or may not correspond to an actual object created at runtime. For clarity, we use two disjoint alphabets for type variable names: Y for variables bound in terms and Z for variables bound in types. We use X when referring to either kind. The concrete environment (H) is indexed by variables bound in terms (Y) and extended during evaluation or typing. The abstract environment (J) is indexed by variables bound in types (Z) and extended during subtyping.

Implementation-wise, this approach fits quite well with a locally nameless representation of binders [14] that already distinguishes between free and bound identifiers.

The runtime subtyping rules for abstract type variables in Figure 4 correspond more or less directly to their counterparts in the static subtype relation (Figure 2), modulo addition of the two runtime environments. In particular, like in static subtyping but unlike for concrete type variables in runtime subtyping, there is no subtyping rule for abstract type variables on the right.

3.3 Metatheory: Soundness Proof

How do we adapt the soundness proof from Section 2.2 to work with this form of runtime subtyping? We need to show that the runtime rules are consistent with the static rules (Lemma 5) and, due to the possibility of subsumption, the main proof can no longer just rely on case analysis of value typing derivations. Hence, we require proper canonical forms lemmas (Lemmas 6,7,8,9).

Relating Static and Runtime Subtyping First, the runtime subtyping should be consistent with the static subtyping.

Lemma 4 (Static to Runtime Subtyping)**.** *Static subtyping implies runtime subtyping in well-formed consistent environments.*

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vDash H\ J}{J \vdash H\ T_1 <: H\ T_2}$$

Proof. By straightforward structural induction over the static subtyping derivation. Static rules on Z variables map to corresponding abstract runtime rules. Static rules on Y variables map to corresponding concrete runtime rules. □

Second, for the cases where $F_{<:}$ type assignment relies on substitution in types – specifically, in the result of a type application – we need to replace a hypothetical binding with an actual value.

Lemma 5 (Abstract to Concrete Substitution for Subtyping)**.**

$$\frac{Z <: \langle H, T\rangle, \emptyset \vdash H_1\ T_1^Z <: H_2\ T_2^Z}{\emptyset \vdash H_1, Y_1 = \langle H, T\rangle\ T_1^{Y_1} <: H_2, Y_2 = \langle H, T\rangle\ T_2^{Y_2}}$$

Proof. By induction and case analysis, first strengthening to allow more type variables in abstract context on the right of type variable Z, to be substituted. □

We actually prove a slighty more general lemma that incorporates the option of weakening, i.e. not extending H_i if Z does not occur in T_i, for each side $i = 1, 2$.

Inversion of Value Typing (Canonical Forms) Due to the presence of the subsumption rule, the main proof can no longer just rely on case analysis of the typing derivation, but we need proper inversion lemmas for term and type abstractions.

Lemma 6 (Inversion of Value Typing for Term Abstraction)**.**

$$\frac{\begin{array}{c} H \vdash v : S_2 \to T_2 \\ \hline v = \langle H_c, \lambda x : S_1.t\rangle \qquad \Gamma_c \vDash H_c\ \emptyset \\ \Gamma_c, x : S_1 \vdash t : T_1 \qquad \emptyset \vdash H_c\ (S_1 \to T_1) <: H\ (S_2 \to T_2) \end{array}}{}$$

Lemma 7 (Inversion of Value Typing for Type Abstraction)**.**

$$\frac{\begin{array}{c} H \vdash v : \forall Z <: S_2.T_2^Z \\ \hline v = \langle H_c, \Lambda Y <: S_1.t\rangle \qquad \Gamma_c \vDash H_c\ \emptyset \\ \Gamma_c, Y <: S_1 \vdash t : T_1^Y \\ \emptyset \vdash H_c\ (\forall Z <: S_1.T_1^Z)\ <:\ H\ (\forall Z <: S_2.T_2^Z) \end{array}}{}$$

We further need to invert the resulting subtyping derivations, so we need additional inversion lemmas for function and $\forall$ types.

Lemma 8 (Inversion of Runtime Subtyping for Function Types)**.**

$$\frac{\emptyset \vdash H_1\ (S_1 \to T_1) <: H_2\ (S_2 \to T_2)}{\emptyset \vdash H_2\ S_2 <: H_1\ S_1\ ,\ H_1\ T_1 <: H_2\ T_2}$$

Lemma 9 (Inversion of Runtime Subtyping for $\forall$ Types)**.**

$$\frac{\emptyset \vdash H_1\ (\forall Z <: S_1.T_1^Z)\ <:\ H_2\ (\forall Z <: S_2.T_2^Z)}{\begin{array}{c} \emptyset \vdash H_2\ S_2 <: H_1\ S_1 \\ \emptyset, Z <: \langle H_2, S_2\rangle \vdash H_1\ T_1^Z\ <:\ H_2\ T_2^Z \end{array}}$$

The inversion lemmas we need here depend in a crucial way on transitivity and narrowing properties of the subtyping relation (similar to small-step proofs for $F_{<:}$ [10]).

Transitivity Pushback and Cut Elimination For the static subtyping relation of $F_{<:}$, transitivity can be proved as a lemma, together with narrowing, in a mutual induction on the size of the middle type in a chain $T_1 <: T_2 <: T_3$ (see e.g. the POPLmark challenge documentation [10]).

Unfortunately, for the runtime subtyping version, the same proof strategy fails, because runtime subtyping may involve a type variable as the middle type: $T_1 <: Y <: T_3$. This setting is very similar to issues that arise with path-dependent types in Scala and DOT, but here it surprisingly arises already when just looking at the runtime semantics of $F_{<:}$. Since proving transitivity becomes

much harder, we adopt a strategy from previous DOT developments [7]: admit transitivity as an axiom, but prove a 'pushback' lemma that allows to push uses of the axiom further up into a subtyping derivation, so that the top level becomes invertible. We denote this as *precise* subtyping $T_1 <:! T_2$.

Definition 3 (Precise Subtyping)**.** *If $J \vdash H_1\ T_1 <: H_2\ T_2$ and the derivation does not end in the transitivity rule, then we say that $J \vdash H_1\ T_1 <:! H_2\ T_2$.*

Such a strategy is reminiscent of cut elimination in natural deduction, and in fact, the possibility of cut elimination strategies is already mentioned in Cardelli's original $F_{<:}$ paper [13].

Lemma 10 (Narrowing)**.**

$$\frac{J_1 \vdash H_1\ T_1 <: H_2\ T_2 \qquad J_2 \vdash H_3\ T_3 <: H_4\ T_4 \qquad J_1 = J_2(Z \to \langle H_1\ T_1\rangle) \qquad J_2 \ni Z <: \langle H_2\ T_2\rangle}{J_1 \vdash H_3\ T_3 <: H_4\ T_4}$$

Proof. By structural induction, and using the transitivity axiom. □

Lemma 11 (Transitivity Pushback)**.**

$$\frac{\emptyset \vdash H_1\ T_1 <: H_2\ T_2}{\emptyset \vdash H_1\ T_1 <:! H_2\ T_2}$$

Proof. By structural induction on the derivation using narrowing (Lemma 10) in the case for $\forall$ types. □

This completes the proofs for the inversion Lemmas 6,7,8,9.

Inversion of subtyping is only required in a concrete runtime context, without abstract component ($J = \emptyset$). Therefore, transitivity pushback is only required then. Transitivity pushback requires narrowing, but only for abstract bindings (those in J, never in H). Narrowing requires these bindings to be potentially imprecise, so that the transitivity axiom can be used to inject a step to a smaller type without recursing into the derivation. In summary, we need both (actual, non-axiom) transitivity and narrowing, but not at the same time. This is a major advantage over a purely static setting where these properties can become much more entangled, with no obvious way to break cycles. Though transitivity pushback can be shown to hold in a mixed concrete/abstract runtime context for $F_{<:}$, the insight that it is only required in a concrete-only runtime context is crucial for extensions, in which subtyping can collapse in unrealizable contexts (see Section 5.1).

Finishing the Soundness Proof We now have everything in place to complete the soundness proof.

Theorem 3 (Soundness of $F_{<:}$)**.** *For all n, if the interpreter returns a result that is not a timeout, then the result is a value (i.e. not stuck), and it is well-typed.*

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vDash H \qquad \text{eval}\ n\ H\ e = \text{Done}\ r}{r = \text{Val}\ v \qquad H \vdash v : T}$$

Proof. By induction over n. The interesting case is the one for type application. We use the inversion lemma for type abstractions (Lemma 7) and then invert the resulting subtyping relation on $\forall$ types to relate the actual type at the call site with the expected type at the definition site of the type abstraction. In order to do this, we invoke pushback (Lemma 11) once and obtain an invertible subtyping on $\forall$ types. But inversion then gives us evidence for the return type in a mixed, concrete/abstract environment, with J containing the binding for the quantified type variable. Since the abstract component J is non-empty, we cannot apply transititivy pushback again directly. So we first apply the substitution lemma (Lemma 5) to replace the abstract variable reference with a concrete reference to the actual type in a runtime environment H. After that, the abstract component is gone ($J = \emptyset$) and we can use pushback again to further invert the inner derivations. □

4. Standard Type System Extensions

In this section we consider two type system extensions that are relevant for practical languages: mutable references and exceptions. While the functionality of these extensions are standard, they are thought to require different proof methods or pose other significant challenges in big-step style. Here we show that our definitional interpreter approach scales gracefully, and that straightforward inductive proofs are sufficient.

4.1 Mutable References

We extend the syntax to support ML-style mutable references:

$$\begin{array}{lll} T & ::= & \cdots \mid \text{Ref}\ T \\ t & ::= & \cdots \mid \text{ref}\ t \mid\ !t \mid t := t \\ v & ::= & \cdots \mid \text{loc}\ i \end{array}$$

The extension of the syntax and static typing rules is standard, with a new syntactic category of store locations. The evaluator is threaded with a runtime store ρ, and reading or writing to a location accesses the store. How do we assign a runtime type to a store location? The key difficulty is that store bindings may be recursive, which has lead Tofte to discover coinduction as a proof technique for type soundness [51]. We sidestep this issue by assigning types to values (in particular store locations) with respect to a *store typing* $\mathcal{S}$ instead of the store itself. Store typings consist of $H\ T$ pairs, which can be related through the usual runtime subtyping judgements. The value type assignment judgement now takes the form $\mathcal{S}\ H \vdash v : T$ and since subtyping depends on value type assignment, it is parameterized by the store typing as well: $\mathcal{S}\ J \vdash H_1\ T_1 <: H_2\ T_2$. The type assignment rule for store locations simply looks up the correct type from the store typing:

$$\frac{\mathcal{S}(i) = \langle H, T\rangle}{\mathcal{S}\ H \vdash \text{loc}\ i : \text{Ref}\ T}$$

When new bindings are added to the store, they are assigned the type and environment from their creation site in the store typing. When accessing the store, bindings in the store typing are always preserved, i.e. store typings are invariant under reads and updates.

Objects in the store ρ must conform to the store typing $\mathcal{S}$ at all times: $\mathcal{S} \vDash \rho$. With that, an update only has to provide a subtype of the type in the store typing, and it will not change the type of that slot. So if an update creates a cycle in the store, this does not introduce circularity in the store typing.

A canonical forms lemma states that if a value v has type Ref T, v must be a store location with type T in the store typing.

Lemma 12 (Inversion of Value Typing for Store Location)**.**

$$\frac{\mathcal{S}\ H \vdash v : \text{Ref}\ T}{v = \text{loc}\ i \qquad \mathcal{S}(i) = \langle H_i, T_i\rangle}{\mathcal{S}\ \emptyset \vdash H_i\ T_i <: H\ T\ ,\ H\ T <: H_i\ T_i}$$

The main soundness statement is modified to guarantee that the interpreter threads a store that corresponds to an extension of the initial store typing, regardless of whether it terminates or not. Thus, the store typing is required to grow monotonically, while the values in the store may change arbitrarily within the constraints given by the store typing. Furthermore, as before, if the interpreter terminates, then the result is a value (i.e. not stuck), and it is well-typed.

Theorem 4 (Soundness of $F_{<:}$ with Mutable References)**.**

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vDash \mathcal{S}\ H \qquad \mathcal{S} \vDash \rho \qquad \text{eval}\ n\ \rho\ H\ e = (\rho', r)}{\mathcal{S}' = \mathcal{S} \ldots \qquad \mathcal{S}' \vDash \rho' \qquad r = \text{Timeout} \ \vee\ (r = \text{Done Val}\ v\ \wedge\ \mathcal{S}'\ H \vdash v : T)}$$

We believe that the ease with which we were able to add mutable references is a further point in favor of definitional interpreters. Back in 1991, the fact that different proof techniques were thought to be required for references in big-step style was a major criticism by Wright and Felleisen and a problem their syntactic approach sought to address [56]. While there is precedent for a simply-inductive big-step soundness proof of polymorphic references [27], this earlier proof omits explicit wrong and nonterminating cases (see followup note [28]) and thus suffers from the usual criticism towards big-step arguments of proving only preservation but not progress, and furthermore, of giving no guarantees for nonterminating evaluation (while, like in small step, we still ensure the invariant on store typing).

Finally, note that the store typing is just another invariant of runtime typing. We do not need to expose store typing in the static typing of terms, since locations are not surface terms.

4.2 Exceptions

While strong soundness promises to guarantee the absence of all runtime errors in well-typed programs, realistic languages need to support a well-defined set of benign runtime failures. These commonly include division-by-zero or file-not-found conditions, which are hard to check with static type systems.

We extend the language with support for exceptions, which abort the current flow of execution and potentially resume at an enclosing exception handler:

$$\begin{array}{rcl} t & ::= & \cdots \mid \text{raise} \mid \text{try}\ t\ \text{catch}\ t \\ r & ::= & \cdots \mid \text{Done Raise} \end{array}$$

To the term syntax, we add a facility to raise an exception, as well as try/catch blocks. The set of possible interpreter results now includes the option of returning a (benign) exception, in addition to values, timeouts, and actual errors, which we still seek to prevent.

The interpreter itself requires very little changes, as it already supports abortive behavior due to timeout and error conditions. We can modify the monadic `DO v ⇐ eval` notation to abstract over exceptions in exactly the same way. But in addition to the bind operator `DO`, which can be viewed as targeting the innermost level of a layered monad, we also need another operator `DOE`, which extracts either `Val v` or `Raise`, conceptually one level up in the layered monad. With that, the only modification to the interpreter is adding new cases for `raise` and `catch`:

```
(* ... *)
    | traise       ⇒ DONE RAISE
    | tcatch et ec ⇒
      DOE rt ⇐ eval n1 env et;
        match vf with
          | VAL v ⇒DONE VAL v
          | RAISE ⇒eval n1 env ec
        end
(* ... *)
```

If the 'try' part of a try/catch block raises an exception, control resumes at the 'catch' block.

The soundness statement changes as follows to include the possibility of runtime exceptions:

Theorem 5 (Soundness of $F_{<:}$ with Exceptions)**.** *For all n, if the interpreter returns a result that is not a timeout, then the result is either an exception or a well-typed value (but it is not stuck)*

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vDash H \qquad \text{eval}\ n\ H\ e = \text{Done}\ r}{r = \text{Raise} \ \vee\ (r = \text{Val}\ v\ \wedge\ H \vdash v : T)}$$

In conclusion, actual errors are excluded, but benign exceptions are allowed. The proof requires no additional lemmas, just handling the two new syntactic cases in the main proof.

The key take-away is that while distinguishing between nontermination, actual errors, and benign errors is a big problem with relational big-step semantics, a total functional interpreter deals with all these outcomes uniformly and eliminates this entire class of problems by design.

5. Novel Type System Extensions

In the preceding sections we have studied the ingredients of type soundess proofs with definitional interpreters, the application of this approach to known and well-studied type system such as $F_{<:}$, as well as to standard extensions. In addition to the static typing of terms on the surface, the approach involves defining runtime invariants such as typing on values. For the latter, we have some leeway: runtime typing needs to be strong enough for the inductive cases in the type safety proof, but also can be more relaxed than the surface static typing.

Up to now, we always started from given static type systems, and we tried to fit the runtime invariants to support a proof. In this section, we take the opposite route. Starting from the interpreter semantics and its runtime invariants, we derive novel and interesting static type systems.

5.1 System F, $F_{<:}$, and $F_{<:>}$

The fact that runtime typing can be more relaxed than the surface static typing is particularly striking for subtyping. Depending on how closely the static typing tracks the runtime typing, we can model a spectrum of polymorphic λ-calculi.

System F At one end of the spectrum, we can just not expose subtyping in the static semantics – by restricting the surface rules to System F. In this case, the runtime rules are more powerful than actually needed. They can be tightened by replacing the runtime subtyping relation with a runtime type equality relation. We leave this as an exercise to the reader.

System $F_{<:>}$ At the other end of the spectrum, we can explore what emerges from propelling more of the runtime typing to the surface. Abstract types in $F_{<:}$ can only be bounded from above. But internally, runtime subtyping already needs to support symmetric rules, for (concrete) type variables on either side. We can easily expose that facility on the static level as well, extending upper-bounded quantification to "translucent" quantification, which is both lower- and upper-bounded.

On the left, Figure 5 summarizes the changes in syntax and static semantics for System $F_{<:>}$, showing the progression from System F to $F_{<:>}$ as this dimension exposes more subtyping in quantification over types. In particular, we generalize the upper-bounded type variables of $F_{<:}$ from $X <: U$ to $X : S..U$, where the type S is the lower bound and the type U is the upper bound – changing the syntax of type abstraction, $\forall$ type, and binding of type variables in the static environment Γ. We also add a bottom type $\bot$ to recover the usual upper-bounded quantification of $F_{<:}$ by setting the lower bound to $\bot$.

User-Defined Subtyping Relations Now, a key question is what constraints should be put on lower and upper bounds at their declaration site in type abstractions. If we do not enforce "good bounds", i.e. do not require $S <: U$ for bounds $S..U$, then we can use this facility to enable fully user-defined subtyping relations between abstract types. Here is an example: in a context

$$\Gamma = A : \bot..\top,\ B : \bot..\top,\ C : \bot..\top$$

where A, B, C are fully abstract, we can add new bindings via

$$\Lambda U : A..C.\ \Lambda V : A..B.t$$

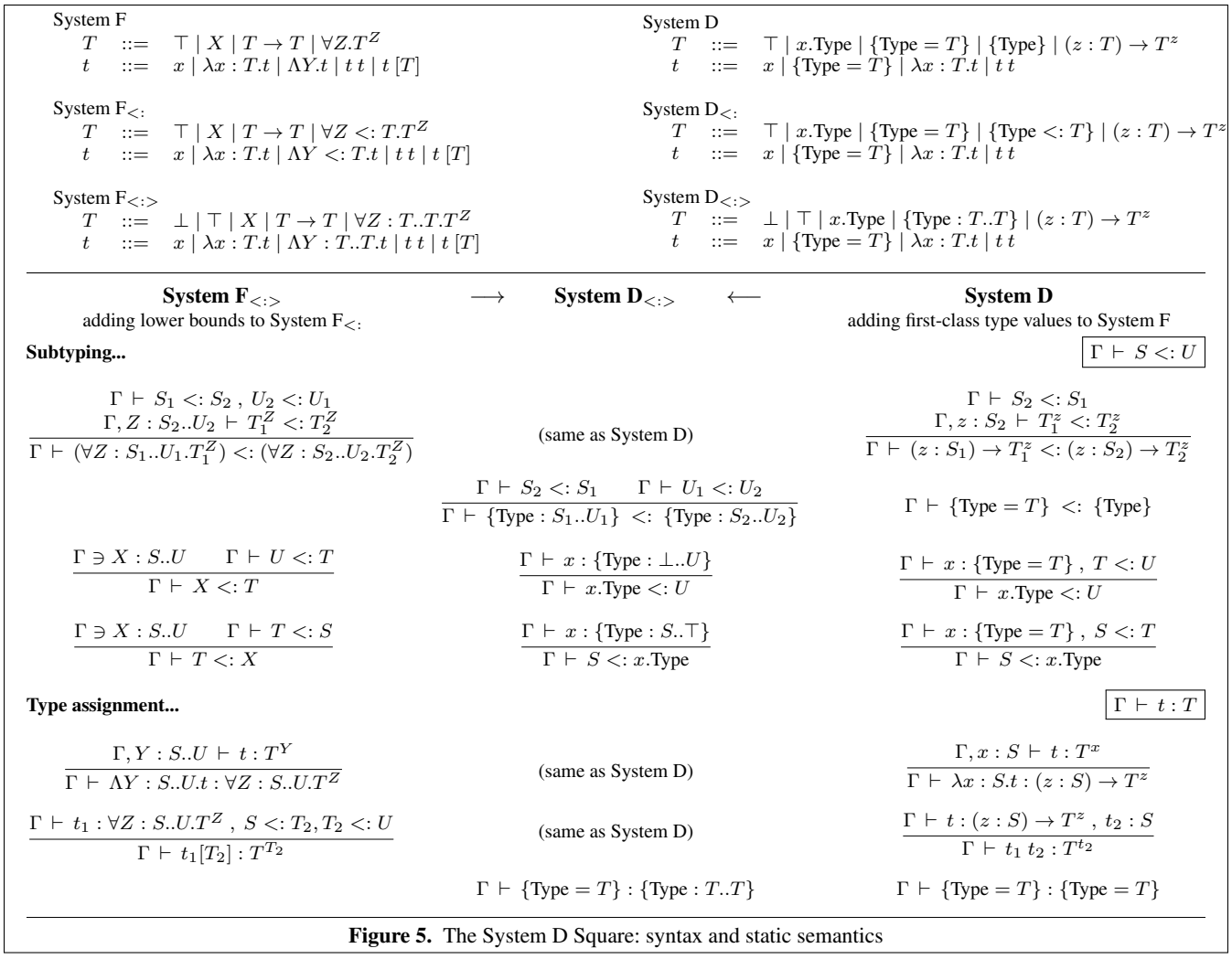

Figure 5. The System D Square: syntax and static semantics

such that in the body t of this abstraction, A is a subtype of both B and C via transitivity, while B and C remain incomparable. With only upper-bounded quantification as in $F_{<:}$, we could not achieve the same flexibility.

But of course this absence of constraints on bounds also means that user-defined subtyping relations might be contradictory. Under a type abstraction with bad bounds, for example $X : \top..\bot$, we can effectively collapse the subtyping relation via transitivity chains on X and subsumption. So why does this work at all?

Soundness Interestingly, we do not need to enforce "good bounds" for soundness. The key is that we do check that the instantiated type is in between the lower bound and upper bound during a *type application*. Therefore, any code path that was type checked using bad bounds will never be executed. The proof strategy from Section 3 already only needs canonical forms and subtyping inversion in an empty abstract context, so bad bounds do not pose any particular difficulty. In fact, the whole proof setup for $F_{<:}$ and the runtime invariants suggest at no point that checking bounds would be necessary, so the unconstrained system comes out as a natural choice.

In runtime subtyping (not shown in Figure 5), we do not need to change the concrete variable rules, while the other changes (on abstract type variable and $\forall$ type) are analogous to static subtyping. For example, bindings in the runtime environment J change from $Z <: \langle H, U\rangle$ to $Z : \langle H, S..U\rangle$.

It is instructive to compare this with a small-step proof strategy. Since there is no a priori distinction between type assignment time and runtime, the unconstrained System $F_{<:>}$ would not fall out naturally as an extension of $F_{<:}$'s soundness proof, which makes key use of transitivity and narrowing lemmas that need to work in arbitrary contexts. Of course the small-step proof can be adapted to follow a transitivity pushback strategy in empty contexts, similar to Section 3, but such a strategy would require a radical departure from the small-step proof whereas it is self-suggesting with a definitional interpreter approach.

More generally, with the definitional interpreter approach, we can first generalize runtime typing, and then find a way to incorporate a consistent set of changes back into static typing.

Language Design Trade-Offs We have seen that we gain expressiveness by not checking bounds at declarations sites, but what do we lose? First, a type checker cannot decide whether user-defined subtyping relations make sense. Hence, errors might only manifest when trying (and failing) to instantiate the type abstractions. Second, while $F_{<:}$ permits arbitrary beta-reduction, $F_{<:>}$ is more restricted. For example, $F_{<:>}$ cannot support a normal-order reduction strategy which would require reductions under type lambdas, to transform a term into head normal form. With conflicting bounds, such as `String` <: `Int`, a term like $3 +$ `"d'oh"` would be well-typed but stuck.

5.2 System D, $D_{<:}$, and $D_{<:>}$

We now consider an extension that starts with relaxing the interpreter semantics itself. We observe that type arguments, implemented as $\langle H, T\rangle$ pairs, are already treated de-facto as first-class values at runtime. So why not let them loose and make their status explicit?

Refactoring the Interpreter In the interpreter, we do not strictly need the distinction between term and type abstraction. Closures for term abstraction `vabs` and closures for type abstraction `vtabs` below hold the same data:

```
(* ... *)
    | tabs x ey  ⇒DONE VAL (vabs env x ey)   (* lambda      *)
    | ttabs x ey ⇒DONE VAL (vtabs env x ey)  (* type lambda *)
    | tapp ef T⇒                             (* application *)
      DO vf ⇐ eval n1 env ef;
        DO vx ⇐  eval n1 env ex; match vf with
          | (vabs env2 x ey)⇒ eval n1 ((x,vx)::env2) ey
          | _⇒ERROR end
    | ttapp ef T⇒                                (* type app   *)
      DO vf ⇐ eval n1 env ef; match vf with
          | (vtabs env2 x ey)⇒ eval n1 ((x,vty env T)::env2) ey
          | _⇒ERROR end
(* ... *)
```

Thus, we can do with only one data type, and settle for `vabs`. To also fuse term and type application, we need to make the evaluation of the argument uniform, regardless of whether it is a type or a term. We thus introduce a new syntactic form $\{\text{Type} = T\}$, for a term that holds a type (`ttyp T` below) and evaluate it to a `vty env T`, which gains status as first-class value:

```
(* ... *)
    | tapp ef T⇒
      DO vf ⇐ eval n1 env ef;
        DO vx ⇐  eval n1 env ex; match vf with
          | (vabs env2 x ey)⇒ eval n1 ((x,vx)::env2) ey
          | _⇒ERROR end
    | ttyp T⇒(vty env T)
(* ... *)
```

First-Class Type Values and Path-Dependent Types What happens at the type level? In System $D_{<:>}$, the term $\{\text{Type} = T\}$ introduces a corresponding type $\{\text{Type} : T..T\}$. Now, since we unify term and type abstraction at the term level, we also unify function type and universal type into one dependent function type: $(x : S) \to T^x$. How can T depend on the term variable x? Instead of a universal type $\forall X : S..U.T^X$, we write $(x : \{\text{Type} : S..U\}) \to T^x$, where x is a regular term variable. For occurrences of type variable X in T, we need to be able to select the type within the term variable x – eliminating a term holding a type at the type level: $x.\text{Type}$. We have re-discovered path-dependent types, like in Scala and DOT but with a unique, global label Type.

On the right and center, Figure 5 summarizes the changes in syntax and static semantics for System D and System $D_{<:>}$, showing the progression in the System D Square along this second dimension: by adding first-class types, System D, $D_{<:}$, and $D_{<:>}$ unify the term and type abstractions of System F, $F_{<:}$, and $F_{<:>}$. Note that even System D requires a bit of subtyping (even though System F does not) to account for matching a concrete type value with an abstract type value type: $\{\text{Type} = T\} <: \{\text{Type}\}$, which we motivate next.

"D-ing F": Encoding of System F, $F_{<:}$, $F_{<:>}$ The modified interpreter semantics trivially generalizes evaluation of System F and its variants, but we still need to show how that the static typing of System D, $D_{<:}$, $D_{<:>}$ generalizes System F, $F_{<:}$, and $F_{<:>}$.

To establish this encoding, we replace references to type variables X with path-dependent types $x.\text{Type}$. Type abstractions become term lambdas with a type-value argument, universal types become dependent function types, and type application becomes dependent function application with a concrete type value:

$$\begin{array}{rcl} \Lambda X.t^X & \rightsquigarrow & \lambda x : \{\text{Type}\}.t^{x.\text{Type}} \\ \forall X.T^X & \rightsquigarrow & (x : \{\text{Type}\}) \to T^{x.\text{Type}} \\ t\,[T] & \rightsquigarrow & t\,\{\text{Type} = T\} \end{array}$$

But how do we actually type check a type application? Let us assume that f is the polymorphic identity function, and we apply it to type T. Then we would like the following to be an admissible type assignment:

$$\frac{f : (x : \{\text{Type}\}) \to (z : x.\text{Type}) \to x.\text{Type}}{f\,\{\text{Type} = T\} : (z : T) \to T}$$

In most dependently typed systems there is a notion of reduction or normalization on the type level. Based on our definitional interpreter construction, we observe that we can just as well use subtyping. For this application to type check using standard dependent function types, we need to establish $T <: x.\text{Type} <: T$ and $\{\text{Type} = T\} <: \{\text{Type}\}$, using the rules on the right in Figure 5.

It is easy to show that System D encodes System F, but not vice versa. For example, the following function does not have a System F equivalent: $\lambda x : \{\text{Type}\}.x$

Runtime Typing For System $D_{<:>}$, we also present modified runtime typing rules in Figure 6. Type objects $\langle H, T\rangle$ are now first-class values, just like closures. We no longer need two kinds of bindings in H environments, which now bind variables to values. For J environments, we still consider only abstract structures, that is, only type values, now.

Delta in Meta-Theory (Type Soundness) Most of the changes to the soundness proof are rather minor. However, one piece requires

Syntax

$$\begin{array}{rcl} v & ::= & \langle H, \lambda x : T.t\rangle \mid \langle H, T\rangle \\ H & ::= & \emptyset \mid H, x : v \\ J & ::= & \emptyset \mid J, z : \langle H, T\rangle \end{array}$$

Runtime Subtyping... $\boxed{J \vdash H_1\ T_1 <: H_2\ T_2}$

$$\frac{J \vdash H_2\ S_2 <: H_1\ S_1 \qquad J \vdash H_1\ U_1 <: H_2\ U_2}{J \vdash H_1\ \{\text{Type} : S_1..U_1\}\ <:\ H_2\ \{\text{Type} : S_2..U_2\}}$$

$$\frac{J(z) = \langle H, T\rangle \qquad J \vdash H\ T <: H_2\ \{\text{Type} : \bot..U\}}{J \vdash H_1\ z.\text{Type} <: H_2\ U}$$

$$\frac{J(z) = \langle H, T\rangle \qquad J \vdash H\ T <: H_1\ \{\text{Type} : S..\top\}}{J \vdash H_1 S <: H_2\ z.\text{Type}}$$

$$\frac{H_1(x) = v \quad H \vdash v : T \qquad J \vdash H\ T <: H_2\ \{\text{Type} : \bot..U\}}{J \vdash H_1\ x.\text{Type} <: H_2\ U}$$

$$\frac{H_2(x) = v \quad H \vdash v : T \qquad J \vdash H\ T <: H_2\ \{\text{Type} : S..\top\}}{J \vdash H_1\ S <: H_2\ x.\text{Type}}$$

...

Value type assignment $\boxed{H \vdash v : T}$

$$\frac{\Gamma \vDash H\ \emptyset \qquad \Gamma \vdash \{\text{Type} = T\} : \{\text{Type} : S..U\}}{H \vdash \langle H, T\rangle : \{\text{Type} : S..U\}}$$

$$\frac{\Gamma \vDash H\ \emptyset \qquad \Gamma, x : S \vdash t : T^x}{H \vdash \langle H, \lambda x : S.t\rangle : (z : S) \to T^z}$$

Figure 6. $D_{<:>}$: runtime typing (excerpt)

further attention: the previous transitivity pushback proof relied crucially on being able to relate types across type variables:

$$H_1\ T_1 <!\ H\ Y <!\ H_3\ T_3$$

Inversion of this derivation would yield another chain

$$H \ni Y = \langle H_2, T_2\rangle \qquad H_1\ T_1 <:\ H_2\ T_2 <:\ H_3\ T_3,$$

which, using an appropriate induction strategy, can be further collapsed into $H_1 T_1 <!\ H_3 T_3$. But now the situation is more complicated: inversion of $H_1\ T_1 <!\ H\ x.\text{Type}\ <!\ H_3\ T_3$ yields

$$H(x) = v \qquad H_2 \vdash v : T_2$$

$$H_2\ T_2 <:\ H_1\ \{\text{Type} : T_1..\top\} \qquad H_2\ T_2 <:\ H_3\ \{\text{Type} : \bot..T_3\},$$

but there is no immediate way to relate T_1 and T_3! We would first have to invert the subtyping relations with T_2, but this is not possible because these relations are imprecise and may use transitivity. Recall that they have to be, because they may need to be narrowed—but wait! Narrowing is only required for abstract types, and we only need inversion and transitivity pushback for fully concrete contexts. So, while the imprecise subtyping judgement is required for bounds initially in the presence of abstract types, we can replace it with the precise version once we move to a fully concrete context.

This idea leads to a solution involving another conversion layer. We define an auxiliary relation $T_1\ <<:\ T_2$, which is just like $T_1\ <:\ T_2$, but with precise lookups, e.g. for a concrete variable on the left:

$$\frac{H_1(x) = \langle H, T\rangle \qquad \emptyset \vdash H\ T <<: H_2\ U}{\emptyset \vdash H_1\ x.\text{Type} <<! H_2\ U}$$

For this relation, pushback and inversion work as before, but narrowing is not supported. To make sure we do not need narrowing inductively for subtyping with precise lookup, we delegate to usual subtyping $<:$ with imprecise lookup and built-in transitivity in the body of the dependent function rule:

$$\frac{\begin{array}{c} J \vdash H_2\ S_2 <: H_1\ S_1 \\ J, z : \langle H_2, S_2\rangle \vdash H_1\ T_1^z <: H_2\ T_2^z \end{array}}{J \vdash H_1\ (x : S_1) \to T_1^x\ <<!\ H_2\ (x : S_2) \to T_2^x}$$

In this new relation, we can again remove top-level uses of the transitivity axiom. A derivation $T_1\ <:\ T_2$ can be converted into $T_1\ <<:\ T_2$ and then further into $T_1\ <<!\ T_2$. With that, we can again perform all the necessary inversions required for the soundness proof.

6. Scaling from F to DOT

The DOT (Dependent Object Types) calculus [6, 7, 5, 45] has been proposed as a new type-theoretic foundation for blending functional and object-oriented programming, as well as ML modules, based on path-dependent types. Historically, DOT was designed as a formal model of Scala [6], based on various rewriting semantics, but very little progress was made towards a soundness proof. The shift to big-step semantics has been instrumental in focusing the requirements, leading to the first soundness results, grounding the theory into known territory, and ultimately resulting in a much cleaner calculus [45]. In this section, we sketch how the techniques of this paper apply to DOT, showing the insights gained from looking at runtime invariants, and from exploiting abstractions becoming transparent at runtime.

From $D_{<:>}$ to DOT DOT consolidates all type and function values into *objects*, which contain type and method members with distinct labels. Object members can refer to the object itself and its other members through a recursive self reference:

$$\begin{array}{rcl} t & ::= & x \mid t.m(t) \mid \{x \Rightarrow \bar{d}\} \\ d & ::= & L = T \mid m(x : T) = t \end{array}$$

We could also add mutable fields based on the handling of references (Section 4.1) but we disregard this option for simplicity.

At the type level, DOT adds intersection and union types, as well as recursive self types, which are similar to equi-recursive types but quantify over a term instead of a type. Individual labeled methods and types remain separate at the type level:

$$\begin{array}{rcl} T & ::= & \bot \mid \top \mid T \wedge T \mid T \vee T \mid x.L \\ & & \{L : T..T\} \mid m(z : T) : T^z \mid \{z \Rightarrow T\} \end{array}$$

An object creation term introduces a recursive self type intersecting the member types. We present a few key examples of DOT's expressiveness next.

Objects and First-Class Modules Consider the type of an object containing a type A and a method f that returns an A:

$$\{c \Rightarrow (A : \bot..\top) \wedge (f(u : \top) : c.A)\}$$

We can create a package object or module that provides a type alias C for this type and a way to create objects of such a type:

$$\begin{array}{l} \{p \Rightarrow C = \{c \Rightarrow (A : \bot..\top) \wedge (f(u : \top) : c.A)\}; \\ \quad m(t : \{A : \bot..\top\}) = \{\_ \Rightarrow m(a : t.A) = \{c \Rightarrow \\ \quad A = t.A; f(u : \top) = a\}\}\} \end{array}$$

Nominality through Ascription We can give the package object a type which is transparent for the type member C, using the shorthand $= T$ for $: T..T$:

$$\begin{array}{l} \{p \Rightarrow (C = \{c \Rightarrow (A : \bot..\top) \wedge (f(u : \top) : c.A)\}) \wedge \\ \quad (m(t : \{A : \bot..\top\}) : (m(a : t.A) : p.C \wedge \{A = t.A\}))\} \end{array}$$

However, then, given a package object p, anyone could structurally create a type $p.C$ even by-passing the method m from the package – for example, this term would type-check as type $p.C$:

$$\{c \Rightarrow A = \top; f(u : \top) = u\}$$

We can ascribe a more abstract type to a package object. By making the lower bound of the type member C abstract, the type $p.C$ becomes nominal from the outside enabling the package to fully control the creation of objects of types $p.C$:

$$\begin{array}{l} \{p \Rightarrow (C : \bot..\{c \Rightarrow (A : \bot..\top) \wedge (f(u : \top) : c.A)\}) \wedge \\ \quad (m(t : \{A : \bot..\top\}) : (m(a : t.A) : p.C \wedge \{A = t.A\}))\} \end{array}$$

Records and Refinements as Intersections The package could also provide a refinement of type $p.C$, for example one with additional methods. By closing over a recursive type, one can concisely refer to type members from the refined type.

$$\{c \Rightarrow p.C \wedge (g(a : c.A) : p.C)\}$$

6.1 Historical Challenges and Fresh Perspective

Extending the $D_{<:>}$ soundness proof to DOT poses only few challenges. The main source of complication are recursive self types, which create mutual dependencies between various previously independent considerations, e.g., between the additional layer with precise lookup (see meta-theory in Section 5.2) and abstract to concrete substitution for subtyping (see Lemma 5). These complications are resolved with clever induction metrics and contractiveness restrictions as presented elsewhere [45]. Since the details do not yield further insights into proof techniques with definitional interpreters, we do not repeat them here. Instead, we present the historical challenges in proving 'invented' versions of DOT sound and discuss their 'discovered' solutions thanks to the bottom-up exploration with definitional interpreters.

Substitution – translating back to small-step For path-dependent types, substitution should preserve syntactic validity of paths. In our approach, we do not need substitution of terms, and we only need substitution of a concrete variable for an abstract one in types,

so this syntactic preservation is straightforward. What happens when we translate the proof back to small-step? For call-by-value, we need substitution in terms of a value for a variable. So we can either allow values in paths, i.e. relax the syntax from $x.L$ to $(x \mid v).L$, or we can put all values in a store, so that variables are only substituted with other variables, which are bound in the store. In this case, subtyping takes the form $\mathcal{S}\ \Gamma \vdash T_1 <: T_2$ to track the store $\mathcal{S}$. Both options lead to sound and clean syntactic theories [45], which look blindingly obvious in hindsight, but have not been discovered directly despite focused efforts of several person-years.

Abstraction vs Preservation – exploiting the runtime Branding is an identity function which changes the type, usually from something concrete, e.g. $\top$, to something abstract, e.g. $z.L$. This is valid e.g. if $z : \{L : \top..\top\}$. However, we might only know a less precise type $z : \{L : \bot..\top\}$. In that case, even though we can brand a term $x : \top$ as $x : z.L$ via $z.\mathsf{brand}(x) \rightarrow x$, we cannot establish $x : z.L$ directly given only the more abstract information. In our approach, thanks to the distinction between static and runtime typing, we can effectively access the most precise information at run-time, thus keeping evaluation and preservation in sync instead of in tension. In small-step, this strategy translates to also bringing down abstraction barriers selectively, i.e. keeping more precise information about values or store locations.

Transitivity vs Narrowing – pushback in empty abstract context In previous DOT developments, narrowing and subtyping transitivity were a major headache. In fact, a key step was to show that in the presence of intersection types, both statements cannot hold at the same time [7] in full generality. With the approach of extending $F_{<:}$ towards DOT in this paper, we discovered that we only need to pushback or invert subtyping transitivity in an empty abstract context, and so we can tolerate lattice collapses in non-empty abstract contexts. We have already taken a lenient strategy with respect to "bad bounds" in Section 5.1, and therefore adding intersection and union types poses no particular difficulty. However, the distinction between concrete and abstract context is unusual and essential to the strategy here.

Monotonicity Matters – embrace subsumption Finally, in the historical invented as opposed to discovered model of DOT [6], the type system had many more judgements: not just typing and subtyping, but also expansion, membership and well-formedness. In the discovered model, well-formedness is so lenient that a judgement is not needed, a type merely needs to follow the syntax and have well-bound variables. For membership, the discovered model relies on subtyping, comparing to the member type. Expansion was used for membership, but also for collecting all member requirements when creating an object (while this remains structural in the discovered model). Because some uses of expansion needed to be precise (e.g. for object creation or for ensuring good bounds), subsumption had to be controlled. However, this was untenable at the end, because such a strategy broke several monotonicity properties. By having a lenient strategy towards bad bounds and embracing subsumption, the design not only finally proved sound but became more principled, powerful, and less tied to Scala.

7. Related Work

Semantics and Proof Techniques There is a vast body of work on soundness and proof techniques. The most relevant here is Wright and Felleisen's syntactic approach [56], Plotkin's structural operational semantics [43], Kahn's Natural Semantics [30], and Reynold's Definitional Interpreters [44]. We build our proof technique on Siek's Three Easy Lemmas [49]. Other work that discusses potential drawbacks of term rewriting techniques includes Midtgaard's survey of interpreter implementations [35], Leroy and Grall's coinductive natural semantics [33] and Danielsson's semantics based on definitional interpreters with coinductively defined partiality monads [17]. Coinduction also was a key enabler for Tofte's big-step soundness proof of core ML [51]. In our setting, we get away with purely inductive proofs, thanks to numeric step indexes or depth bounds, even for mutable references. We believe that ours is the first purely inductive big-step strong soundness proof in the presence of mutable state. As discussed earlier (Section 4.1), Harper proves preservation of polymorphic references in big-step using only induction [27], though not strong soundness as we do. The use of step counters in natural semantics to distinguish between divergence and errors goes back to at least Gunter and Rémy's partial proof semantics [26] and has recently been advocated in the context of compiler verification [41]. Step counters also play a key role in proofs based on logical relations [8, 3, 4]. Our runtime environment construction bears some resemblance to Visser's name graphs [38] and also to Flatt's bindings as sets of scopes [24]. Big-step evaluators can be mechanically transformed into equivalent small-step semantics following the techniques of Danvy et al. [18, 19, 2].

Small-step techniques are well known to scale to full implementations of well-behaved but realistic languages such as ML. A notable result is the mechanized soundness result by Lee, Crary and Harper [31] for an internal language that can serve as elaboration target for all of Standard ML. A much earlier partially successful attempt at mechanizing ML was by van Inwegen [52]. To the best of our knowledge, ML is the only widely used language whose metatheory has been mechanized to a similar degree.

Scala Foundations Much work has been done on grounding Scala's type system in theory. Early efforts included νObj [40], Featherweight Scala [16] and Scalina [37], all of them more complex than what is studied here. None of them lead to a mechanized soundness result, and due to their inherent complexity, not much insight was gained why soundness was so hard to prove. DOT [6] was proposed as a simpler core calculus, focusing on path-dependent types but disregarding classes, mixin linearization and similar questions. The original DOT formulation [6] had actual preservation issues because lookup was required to be precise. This prevented narrowing, as explained in Section 6.1.

The μDOT calculus [7] is the first calculus in the line with a mechanized soundness result, (in Twelf, based on total big-step semantics), but the calculus is much simpler than what is studied in this paper, and there is no connection to systems like $F_{<:}$. Most importantly, μDOT lacks bottom, intersections and type refinement. Amin et al. [7] describe in great detail why adding these features causes trouble. Because of its simplicity, μDOT supports both narrowing and transitivity with precise lookup. The soundness proof for μDOT was also with respect to big-step semantics. However, the semantics had no concept of distinct runtime type assignment and would thus not be able to encode $F_{<:}$ and much less full DOT.

Soundness for full DOT has been established more recently [45, 5]. The proofs are presented with respect to small-step semantics, and without connecting DOT to well-studied calculi such as $F_{<:}$.

ML Module Systems 1ML [46] unifies the ML module and core languages through an elaboration to System F_ω based on earlier such work [47]. Compared to DOT, the formalism treats recursive modules in a less general way and it only models fully abstract vs fully concrete types, not bounded abstract types. Although an implementation is provided, there is no mechanized proof. In good ML tradition, 1ML supports Hindler-Milner style type inference, with only small restrictions. Path-dependent types in ML modules go back at least to SML [34], with foundational work on translucent signatures by Harper and Lillibridge [29] and Leroy [32]. MixML [20] drops the stratification requirement and enables modules as first-class values.

Other Related Languages Other calculi related to path-dependent types include the family polymorphism of Ernst [21], Virtual

Classes [23, 22, 39, 25], and ownership type systems like Tribe [15, 12]. Like System D, pure type systems [11] unify term and type abstraction. Extensions of System $F_{<:}$ related to DOT include intersection types and bounded polymorphism [42] and higher-order subtyping [50, 1]. Subtyping has also been combined with dependent types albeit without polymorphism [9], motivated by applications in the context of logical frameworks.

8. Conclusions

We presented type soundness proofs with definitional interpreters, reviewing a proof strategy on STLC, scaling it to System $F_{<:}$, and leading all the way to DOT via the System D Square. For this approach, one defines static typing on terms and runtime typing on values, ensuring that static typing approximates runtime typing. For existing systems, the approach thus focuses the creative search for a soundness proof in devising an appropriate runtime typing on values. For novel systems or extensions, the approach suggests first working out the runtime typing and gradually exposing it into the static typing. We have seen how DOT emerges from System $F_{<:}$ through relatively gentle extensions in the System D Square. This naturally raises the question what other interesting type systems can arise by devising suitable static rules from the runtime ones given by the interpreter and environment structures of interesting languages. We believe this is an exciting new research angle.

Acknowledgments

For insightful discussions, we thank Amal Ahmed, Jonathan Aldrich, Karl Crary, Derek Dreyer, Sebastian Erdweg, Erik Ernst, Matthias Felleisen, Ronald Garcia, Paolo Giarrusso, Samuel Grütter, Robert Harper, Scott Kilpatrick, Grzegorz Kossakowski, Alexander Kuklev, Viktor Kuncak, Ondřej Lhoták, Donna Malayeri, Adriaan Moors, Martin Odersky, Alex Potanin, Jon Pretty, Didier Rémy, Lukas Rytz, Miles Sabin, Ilya Sergey, Jeremy Siek, Sandro Stucki, Josh Suereth, Ross Tate, Eelco Visser, Philip Wadler, Geoffrey Washburn, and Jason Zaugg. We also thank many anonymous reviewers for their thoughtful comments. This research was supported by NSF through awards 1553471 and 1564207.

References

[1] A. Abel. Polarised subtyping for sized types. *Mathematical Structures in Computer Science*, 18:797–822, 10 2008.

[2] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, 2003.

[3] A. J. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

[4] A. J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

[5] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. The essence of dependent object types. In *WadlerFest*, A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, 2016.

[6] N. Amin, A. Moors, and M. Odersky. Dependent object types. In *FOOL*, 2012.

[7] N. Amin, T. Rompf, and M. Odersky. Foundations of path-dependent types. In *OOPSLA*, 2014.

[8] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

[9] D. Aspinall and A. Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1):273–309, 2001.

[10] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark Challenge. In *TPHOLs*, 2005.

[11] H. P. Barendregt. Handbook of logic in computer science. chapter Lambda Calculi with Types. Oxford University Press, 1992.

[12] N. R. Cameron, J. Noble, and T. Wrigstad. Tribal ownership. In *OOPSLA*, 2010.

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

[14] A. Charguéraud. The locally nameless representation. *J. Autom. Reasoning*, 49(3):363–408, 2012.

[15] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: a simple virtual class calculus. In *AOSD*, 2007.

[16] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for Scala type checking. In *MFCS*, 2006.

[17] N. A. Danielsson. Operational semantics using the partiality monad. In *ICFP*, 2012.

[18] O. Danvy and J. Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *J. Comput. Syst. Sci.*, 76(5):302–323, 2010.

[19] O. Danvy, K. Millikin, J. Munk, and I. Zerny. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *Theor. Comput. Sci.*, 435:21–42, 2012.

[20] D. Dreyer and A. Rossberg. Mixin' up the ML module system. In *ICFP*, 2008.

[21] E. Ernst. Family polymorphism. In *ECOOP*, 2001.

[22] E. Ernst. Higher-order hierarchies. In *ECOOP*, 2003.

[23] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL*, 2006.

[24] M. Flatt. Binding as sets of scopes. In *POPL*, 2016.

[25] V. Gasiunas, M. Mezini, and K. Ostermann. Dependent classes. In *OOPSLA*, 2007.

[26] C. A. Gunter and D. Rémy. A proof-theoretic assesment of runtime type errors. Technical Report Technical Memo 11261-921230-43TM, AT&T Bell Laboratories, 1993.

[27] R. Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201 – 206, 1994.

[28] R. Harper. A simplified account of polymorphic references – followup. `https://www.cs.cmu.edu/~rwh/papers/refs/ipl-followup.pdf`, 1995.

[29] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, 1994.

[30] G. Kahn. Natural semantics. In *STACS*, 1987.

[31] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of standard ML. In *POPL*, pages 173–184. ACM, 2007.

[32] X. Leroy. Manifest types, modules and separate compilation. In *POPL*, 1994.

[33] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.

[34] D. Macqueen. Using dependent types to express modular structure. In *POPL*, 1986.

[35] J. Midtgaard, N. Ramsey, and B. Larsen. Engineering definitional interpreters. In *PPDP*, 2013.

[36] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[37] A. Moors, F. Piessens, and M. Odersky. Safe type-level abstraction in Scala. In *FOOL*, 2008.

[38] P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In *ESOP*, 2015.

[39] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA*, 2004.

[40] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP*, 2003.

[41] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan. Functional big-step semantics. In *ESOP*, 2016.

[42] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991.

[43] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[44] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[45] T. Rompf and N. Amin. Type soundness for dependent object types. In *OOPSLA*, 2016.

[46] A. Rossberg. 1ML - core and modules united (f-ing first-class modules). In *ICFP*, 2015.

[47] A. Rossberg, C. V. Russo, and D. Dreyer. F-ing modules. *J. Funct. Program.*, 24(5):529–607, 2014.

[48] D. S. Scott. Domains for denotational semantics. In *Automata, Languages and Programming*, 1982.

[49] J. Siek. Type safety in three easy lemmas. `http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html`, 2013.

[50] M. Steffen. *Polarized higher-order subtyping*. PhD thesis, University of Erlangen-Nuremberg, 1997.

[51] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, 1988.

[52] M. VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, May 1996.

[53] D. von Oheimb. Re: Subject reduction fails in java. `http://www.seas.upenn.edu/~sweirich/types/archive/1997-98/msg00452.html`, 1998.

[54] P. Wadler. The essence of functional programming. In *POPL*, pages 1–14. ACM Press, 1992.

[55] P. Wadler. Propositions as types. Presentation at Strange Loop, 2015.

[56] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

A. Mechanization in Coq

We outline the correspondence between the formalism on paper and its implementation in Coq (`popl17.namin.net`). The Coq package contains individual files for each of the type system variants described in this paper (`fsub.v`, `fsubsup.v`, etc).

A.1 Model

A.1.1 Syntax

The syntax in the paper maps to Coq definitions as follows:

`ty`	$S, T, U ::=$	Type	
`TTop`	$\top$		top type
`TBot`	$\bot$		bottom type
`TMem` S U	Type : $S..U$		type member
`TFun` S U	$(x : S) \to U^x$		(dependent) function type
`TSel` X	x.Type		type selection
`TBind` T	$\{z \Rightarrow T^z\}$		recursive self type
`TAnd` T T	$T \wedge T$		intersection type
`TOr` T T	$T \vee T$		union type
`tm`	$t, u ::=$	Term	
`tvar` b x	x		variable reference
`ttyp` T	Type T	type value	
`tabs` T t	$\lambda x : T.t$		function abstraction
`ttabs` T t	$\Lambda X <: T.t$		type function abstraction
`tapp` t t	$t\ t$		function invocation
`ttapp` t T	$t\ [T]$		type function invocation
`dms`	$\overline{d}$		

For representing variable names in relation to an environment, we use a reverse de Bruijn convention, so that the name is invariant under environment extension. An environment is a list of right-hand sides. The older the binding, the more to the right, the smaller its number name. The name is uniquely determined by the position in the list as the length of the tail (see `indexr`).

In addition, for types, we use a locally-nameless de Bruijn convention for variables under dependent types so that it's easy to substitute binders without variable capture. A variable x bound in T^x by a dependent function type $(x : S) \to T^x$ (or type abstraction for $F_{<:}$) is represented by `TVarB` i where *i* is the de Bruijn level, i.e. the number of other binders in scope in between a bound variable occurrence and its binder.

A.1.2 Type System Judgements

`stp` Γ S U n	$\Gamma \vdash S <: U$	Subtyping
`has_type` Γ t T	$\Gamma \vdash t : T$	Typing
`stp2` b *H1 T1 H2 T2 J n*	$J \vdash H_1\ T_1\ <:\ H_2\ T_2$	Runtime Subtyping (b: transitivity, n: size)
`val_type` H v T	$H \vdash v : T$	Runtime Value Typing

The argument *n* in the runtime subtyping judgement denotes the *size* of the derivation. The boolean flag *b* denotes whether transitivity can be used at the top level of the derivation.

As we mention in Section 2, we omit routine well-formedness checks from the rules on paper for readability. In Coq, these correspond to `closed` conditions, which ensure that all the variables in a type are well-bound for the given environment and binding structure. The relation `closed` $k\ |J|\ |H|\ T$ ensures that `T` is well-bound in a context H, abstract environment J and under at most $\leq k$ binders.

A.2 Soundness Proofs

The Coq package contains the following source files:

1. `stlc.v` — Simply-Typed Lambda Calculus
2. `fsub.v` — $F_{<:}$ type soundness
3. `fsub_equiv.v` — F<: equivalence with Small-Step
4. `fsub_mut.v` — F<: with Mutable References
5. `fsub_exn.v` — F<: with Exceptions
6. `fsubsup.v` — F<:> from the System D Square
7. `dsubsup.v` — D<:> from the System D Square
8. `dot.v` — DOT type soundness

These correspond to individuals Definitions, Lemmas, and Theorems in the paper as outlined below.

A.2.1 Figures and Definitions

1. (STLC: syntax and semantics) — file `stlc.v`
2. ($F_{<:}$: syntax and static semantics) — file `fsub.v`
3. ($F_{<:}$: small-step semantics (call-by-value)) — file `fsub_equiv.v`

4. ($F_{<:}$: runtime typing) — file `fsub.v`
5. (The System D Square: syntax and static semantics) — files `fsubsub.v`, `dsubsub.v`
6. ($D_{<:>}$: runtime typing (excerpt)) — file `dsubsub.v`

A.2.2 Lemmas

1. (Primitives) — not used, since we do not model primitive operations
2. (Lookup) — corresponds to `Lemma indexr_safe_ex`
3. (Eval) — corresponds to `Theorem full_safety`
4. (Static to Runtime Subtyping) — corresponds to `Lemma stp_to_stp2`
5. (Abstract to Concrete Substitution for Subtyping) — corresponds to `Lemma stp2_substitute`
6. (Inversion of Value Typing for Term Abstraction) — corresponds to `Lemma invert_abs`
7. (Inversion of Value Typing for Type Abstraction) — corresponds to `Lemma invert_tabs`
8. (Inversion of Runtime Subtyping for Function Types) — corresponds to Coq's `Inversion` after eliminating transitivity via `Lemma stpd2_upgrade`
9. (Inversion of Runtime Subtyping for ∀ Types) — same as above
10. (Narrowing) — corresponds to `Lemma stpd2_narrow`
11. (Transitivity Pushback) — corresponds to `Lemma stpd2_untrans`
12. (Inversion of Value Typing for Store Location) — corresponds to `Lemma invert_loc` in file `fsub_mut.v`

A.2.3 Theorems

1. (Soundness of STLC) — corresponds to `Theorem full_safety` in file `stlc.v`
2. (Semantic Equivalence) — file `fsub_equiv.v`: in Coq, we first prove equivalence between big-step environment and substitution-based evaluators (`Theorem big_env_subst`) and then equivalence of big-step substitution with small-step evaluation (`big_to_small, small_to_big`). We plan to make this clearer in the paper.
3. (Soundness of F<:) — corresponds to `Theorem full_safety` in file `fsub.v`:
4. (Soundness of F<: with Mutable References) — corresponds to `Theorem full_safety` in file `fsub_mut.v`
5. (Soundness of F<: with Exceptions) — corresponds to `Theorem full_safety` in file `fsub_exn.v`