Master's Thesis

# Mechanized Type Soundness Proofs using Definitional Interpreters

Hannes Saffrich

University of Freiburg
Department of Computer Science
Programming Languages Chair

September 1, 2019

# Motivation

- A type system of a statically typed language has two purposes:
  - it rules out certain classes of ill-formed programs, allowing implementations to avoid unnecessary runtime checks without risking undefined behavior; and
  - it classifies the well-formed programs by certain aspects of their runtime behavior, allowing the programmer to rule out programs that exhibit well-defined but unintended behavior.
- For both purposes it is crucial that well-typed programs only exhibit the runtime behavior expected of their types.
- Languages that have this property are called *type sound*.
- Proving type soundness can be difficult
- Proof structure strongly depends on the features of the programming language and how the runtime behavior is formalized.

# Overview

- Part I: Introduction
  - Type Soundness
  - Small-Step Semantics
  - Big-Step Semantics
  - Definitional Interpreters
- Part II: A Proof in Detail
  - Type Soundness of the Simply Typed Lambda Calculus
- Part III: Extensions
  - Mutable References
  - Substructural Types
  - Subtyping
  - Parametric Polymorphism
  - Bounded Quantification
- Part IV: Conclusion

# Part I
## Introduction

# Type Soundness

- Semantics given by partial evaluation function

$$\text{eval} : \text{Programs} \rightarrow \text{Answers} \cup \{\text{wrong}\}$$

  - returns wrong for erroneous programs ("type errors");
  - undefined for non-terminating programs.
- Typing relation given by $\triangleright e : t$
- Two forms of Type Soundness

$$\textsc{WeakSoundness}\ \frac{\triangleright e : t}{\text{eval}(e) \neq \text{wrong}} \qquad \textsc{StrongSoundness}\ \frac{\triangleright e : t \quad \text{eval}(e) = v}{v \in V^t}$$

- Concise formulation of type soundness, but assumes that the semantics is specified as eval function.

# Small-Step Semantics

- Used in most soundness formalizations today
- Closely related to Term Rewrite Systems
- Defined as binary relation $\hookrightarrow$ between programs, and a notion of when a program is considered a value.
- $e_1 \hookrightarrow e_2$ denotes, that $e_1$ can be evaluated in a single step to $e_2$.
- Evaluation of a program corresponds to repeated step-evaluation

$$e_1 \hookrightarrow e_2 \hookrightarrow e_3 \hookrightarrow \ldots$$

- The evaluation is considered
  - non-terminating, if the chain is infinite;
  - successfull, if the chain stops at some $e_n$, such that $e_n$ is a value; and
  - erroneous, otherwise.

# Small-Step Semantics & Type Soundness

- Standard approach for soundness proofs with small-step semantics introduced by Wright and Felleisen in 1994
- Based on two lemmas

Preservation

$$\frac{\triangleright e_1 : t \qquad e_1 \hookrightarrow e_2}{\triangleright e_2 : t}$$

Progress

$$\frac{\triangleright e_1 : t}{\text{IsValue}\, e_1 \lor \exists e_2 . e_1 \hookrightarrow e_2}$$

- Syntactic Soundness

$$\frac{\triangleright e_1 : t}{e_1 \Uparrow \;\lor\; \exists e_2.\; e_1 \hookrightarrow^* e_2 \;\land\; \text{IsValue}\, e_2 \;\land\; \triangleright e_2 : t}$$

- Proofs are “lengthy but simple, requiring only basic inductive techniques”.

# Big-Step Semantics

- Defined as binary relation $\Downarrow$ between programs and their values
- $e \Downarrow v$ denotes, that expression $e$ successfully evaluates to value $v$.
- The relation is undefined for both non-termination and errors
- Two ideas for describing type soundness

$$\frac{e : t}{\exists \text{ v} \qquad e \Downarrow v \qquad v : t} \qquad\qquad \frac{e : t \qquad e \Downarrow v}{v : t}$$

- Left theorem is too strong: it forces any typed program to terminate.
- Right theorem is too weak: only states soundness for terminating programs.

# Definitional Interpreters

- The problem with big-step semantics is, that to state a type soundness theorem of the right strength, we need to distinguish between non-termination and errors.
- We could extend the big-step semantics, such that it
  - remains undefined for non-terminating programs;
  - relates erroneous programs to a special error value;

  and state type soundness as

$$\frac{\mathsf{e} : \mathsf{t} \qquad \mathsf{e} \Downarrow \mathsf{mv}}{\exists\, \mathsf{v} \qquad \mathsf{mv} = \mathsf{noerr}\ \mathsf{v} \qquad \mathsf{v} : \mathsf{t}}$$

- But this would require auxilliary rules for each regular rule, just to propagate errors through subexpressions.
- Instead, we formulate the big-step semantics not as a relation, but as a definitional interpreter function.
- This allows us to hide the error propagation behind an error monad of the host language.

# Definitional Interpreters

- As the host languages are intended for theorem proving, they have to be total, so we cannot directly state the definitional interpreter for a language that isn't total itself.
- A simple workaround is to reformulate the partial interpreter, such that it trys to evaluate the program in some number of steps, and returns a special timeout value, if the number was insufficient.
- The definitional interpreter can then be defined as

  $$\mathsf{eval} \ : \ \mathbb{N} \to \mathsf{Exp} \to \mathsf{CanTimeout}\ (\mathsf{CanErr}\ \mathsf{Val})$$

  and returns either
  - timeout if the number of steps n was too small;
  - done error if the evaluation caused a type error; and
  - done (noerr v) if the evaluation succeeded with value v.
- The type soundness theorem of the right strength is then stated as

$$\frac{\mathsf{e}\ :\ \mathsf{t} \qquad \mathsf{eval}\ \ \mathsf{n}\ \mathsf{e} = \mathsf{done}\ \mathsf{mv}}{\exists\ \mathsf{v} \qquad \mathsf{mv} = \mathsf{noerr}\ \mathsf{v} \qquad \mathsf{v}\ :\ \mathsf{t}}$$

# Part II
Simply Typed Lambda Calculus

# Simply Typed Lambda Calculus

## Syntax

- Single base type: t_void
- Variables represented as DeBruijn Levels
- No type annotations in abstractions

```
Inductive Typ : Type :=
  | t_void : Typ
  | t_arr  : Typ → Typ → Typ.

Inductive Exp : Type :=
  | e_var : ℕ → Exp
  | e_app : Exp → Exp → Exp
  | e_abs : Exp → Exp.
```

# Simply Typed Lambda Calculus

## Type System

```
Definition TypEnv := List Typ.

Inductive ExpTyp : TypEnv → Exp → Typ → Prop :=
| et_var :
    ∀ x te t,
    indexr x te = some t →
    ExpTyp te (e_var x) t
| et_app :
    ∀ te e1 e2 t1 t2,
    ExpTyp te e1 (t_arr t1 t2) →
    ExpTyp te e2 t1 →
    ExpTyp te (e_app e1 e2) t2
| et_abs :
    ∀ te e t1 t2,
    ExpTyp (t1 :: te) e t2 →
    ExpTyp te (e_abs e) (t_arr t1 t2).
```

# Simply Typed Lambda Calculus

## Semantics

```
Definition ValEnv := List Val.

Inductive Val :=
| v_abs (ve : ValEnv) (e : Exp).

Fixpoint eval (n : ℕ) (ve : ValEnv) (e : Exp) : CanTimeout (CanErr Val) :=
  match n with
  | 0 ⇒ timeout
  | S n ⇒
      match e with
      | e_var x ⇒ done (indexr x ve)
      | e_abs e ⇒ done (noerr (v_abs ve e))
      | e_app e1 e2 ⇒
          ' v_abs ve1' e1' ← eval n ve e1;
          ' v2 ← eval n ve e2;
          eval n (v2 :: ve1') e1'
      end
  end.
```

# Simply Typed Lambda Calculus

## Type Soundness

```
Definition WfEnv : ValEnv → TypEnv → Prop :=
  Forall2 ValTyp.

Inductive ValTyp : Val → Typ → Prop :=
| vt_abs :
    ∀ ve te e t1 t2,
    WfEnv ve te →
    ExpTyp (t1 :: te) e t2 →
    ValTyp (v_abs ve e) (t_arr t1 t2).
```

**Theorem** (Type Soundness).

$$\frac{\text{ExpTyp te e t} \qquad \text{eval n ve e} = \text{done mv} \qquad \text{WfEnv ve te}}{\exists\ \text{v},\ \text{mv} = \text{noerr v} \wedge \text{ValTyp v t}}$$

# Simply Typed Lambda Calculus

## Type Soundness Proof

**Theorem** (Type Soundness)**.**

$$\frac{\text{ExpTyp te e t} \qquad \text{eval n ve e} = \text{done mv} \qquad \text{WfEnv ve te}}{\exists\, \text{v},\ \text{mv} = \text{noerr v} \wedge \text{ValTyp v t}}$$

*Proof.* Induction over n:

- **Case** 0**.** By definition of eval, the assumption

  $$\text{eval } 0 \text{ ve e} = \text{done mv}$$

  reduces to

  $$\text{timeout} = \text{done mv}$$

  so we can discard this case by contradiction.
- **Case** n+1**.** [. . . ]

# Simply Typed Lambda Calculus

## Type Soundness Proof

**Theorem** (Type Soundness).

$$\frac{\text{ExpTyp te e t} \qquad \text{eval n ve e} = \text{done mv} \qquad \text{WfEnv ve te}}{\exists\, \text{v},\ \text{mv} = \text{noerr v} \wedge \text{ValTyp v t}}$$

*Proof.* Induction over n:

- **Case** n+1**.** Case analysis on ExpTyp te e t:
  - **Case** et_var**.** By definition of et_var, we have some x such that

    $$\text{e} = \text{e\_var x} \qquad\qquad \text{indexr x te} = \text{noerr t}.$$

    By definition of eval, the assumption

    $$\text{eval }(\text{n} + 1)\text{ ve }(\text{e\_var x}) = \text{done mv}$$

    reduces to

    $$\text{done }(\text{indexr x ve}) = \text{done mv}$$

    Thus, by substituting indexr x ve for mv, we are left to prove

    $$\frac{\text{WfEnv ve te} \qquad \text{indexr x te} = \text{noerr t}}{\exists\, \text{v},\ \text{indexr x ve} = \text{noerr v} \wedge \text{ValTyp v t}}$$

# Simply Typed Lambda Calculus

## Type Soundness Proof

**Theorem** (Type Soundness).

$$\frac{\text{ExpTyp te e t} \qquad \text{eval n ve e} = \text{done mv} \qquad \text{WfEnv ve te}}{\exists\, \text{v},\ \text{mv} = \text{noerr v} \wedge \text{ValTyp v t}}$$

*Proof.* Induction over n:

- **Case** n+1**.** Case analysis on ExpTyp te e t:
  - **Case** et_abs**.** By definition of et_abs, we have some e', t1, t2 with

    $$\text{e} = \text{e\_abs e'} \qquad \text{t} = \text{t\_arr t1 t2} \qquad \text{ExpTyp (t1 :: te) e' t2}$$

    By definition of eval, the assumption

    $$\text{eval (n + 1) ve (e\_abs e')} = \text{done mv}$$

    reduces to

    $$\text{done (noerr (v\_abs ve e'))} = \text{done mv}$$

    Thus, by substituting for mv, we are left to prove

    $$\exists\, \text{v},\ \text{v\_abs ve e'} = \text{v} \wedge \text{ValTyp v (t\_arr t1 t2)}$$

    so we choose v = v_abs ve e' and construct the value typing from our assumptions:

    $$\frac{\text{WfEnv ve te} \qquad \text{ExpTyp (t1 :: te) e' t2}}{\text{ValTyp (v\_abs ve e') (t\_arr t1 t2)}}\ \text{VT\_\_ABS}$$

# Simply Typed Lambda Calculus

## Type Soundness Proof

**Theorem** (Type Soundness).

$$\frac{\text{ExpTyp te e t} \qquad \text{eval n ve e} = \text{done mv} \qquad \text{WfEnv ve te}}{\exists\ \text{v},\ \text{mv} = \text{noerr v} \land \text{ValTyp v t}}$$

*Proof.* Induction over n:

- **Case** n+1. Case analysis on ExpTyp te e t:
  - **Case** et_app. By definition of et_app, we have some e1, e2, t1, t2 such that

    $$\text{e} = \text{e\_app e1 e2} \qquad \text{t} = \text{t2} \qquad \text{ExpTyp te e1 (t\_arr t1 t2)} \qquad \text{ExpTyp te e2 t1}$$

    By definition of eval, the assumption

    $$\text{eval } (\text{n} + 1) \text{ ve (e\_app e1 e2)} = \text{done mv}$$

    reduces to

    ```
    ' v_abs ve' e1' ← eval n ve e1;
    ' v2 ← eval n ve e2;
    eval n (v2 :: ve') e1'          = done mv
    ```

    Next, we observe that there must be some mv1 and mv2 such that

    $$\text{eval n ve e1} = \text{done mv1} \qquad\qquad \text{eval n ve e2} = \text{done mv2}$$

# Simply Typed Lambda Calculus

## Type Soundness Proof

**Theorem** (Type Soundness).

$$\frac{\text{ExpTyp te e t} \qquad \text{eval n ve e} = \text{done mv} \qquad \text{WfEnv ve te}}{\exists\ \text{v},\ \text{mv} = \text{noerr v} \wedge \text{ValTyp v t}}$$

*Proof.* Induction over n:

- **Case** n+1. Case analysis on ExpTyp te e t:
  - **Case** et_app.

    [...]

    We are now equipped to apply our induction hypothesis to the evaluation of both subexpressions:

    $$\frac{\text{eval n ve e1} = \text{done mv1} \qquad \text{ExpTyp te e1 (t\_arr t1 t2)} \qquad \text{WfEnv ve te}}{\exists\ \text{v1},\ \text{mv1} = \text{noerr v1} \wedge \text{ValTyp v1 (t\_arr t1 t2)}}$$

    $$\frac{\text{eval n ve e2} = \text{done mv2} \qquad \text{ExpTyp te e2 t1} \qquad \text{WfEnv ve te}}{\exists\ \text{v2},\ \text{mv2} = \text{noerr v2} \wedge \text{ValTyp v2 t1}}$$

    By inversion of the value typing ValTyp v1 (t_arr t1 t2), we find some te', ve', e1' such that

    $$\text{v1} = \text{v\_abs ve' e1'} \qquad \text{ExpTyp (t1 :: te') e1' t2} \qquad \text{WfEnv ve' te'}$$

# Simply Typed Lambda Calculus

## Type Soundness Proof

**Theorem** (Type Soundness).

$$\frac{\text{ExpTyp te e t} \qquad \text{eval n ve e} = \text{done mv} \qquad \text{WfEnv ve te}}{\exists\ \text{v},\ \text{mv} = \text{noerr v} \wedge \text{ValTyp v t}}$$

*Proof.* Induction over n:

- **Case** n+1**.** Case analysis on ExpTyp te e t:
  - **Case** et_app**.**
    [...]
    By substituting for mv1, mv2, and v1, we now know

    $$\text{eval n ve e1} = \text{done (noerr (v\_abs ve' e1'))}$$
    $$\text{eval n ve e2} = \text{done (noerr v2)}$$

    so the monadic sequencing in

    ```
    ' v_abs ve' e1' ← eval n ve e1;
    ' v2 ← eval n ve e2;
    eval n (v2 :: ve') e1'          = done mv
    ```

    reduces to

    ```
    eval n (v2 :: ve') e1' = done mv
    ```

# Simply Typed Lambda Calculus

## Type Soundness Proof

**Theorem** (Type Soundness).

$$\frac{\text{ExpTyp te e t} \qquad \text{eval n ve e} = \text{done mv} \qquad \text{WfEnv ve te}}{\exists\ \text{v},\ \text{mv} = \text{noerr v} \wedge \text{ValTyp v t}}$$

*Proof.* Induction over n:

- **Case** n+1. Case analysis on ExpTyp te e t:
  - **Case** et_app.
    [. . .]
    To conclude the proof, we want to apply the induction hypothesis again

    $$\frac{\text{ExpTyp (t1 :: te') e1' t2} \qquad \begin{array}{c}\text{eval n (v2 :: ve') e1'} = \text{done mv} \\ \text{WfEnv (v2 :: ve') (t1 :: te')}\end{array}}{\exists\ \text{v},\ \text{mv} = \text{noerr v} \wedge \text{ValTyp v t}}\ \text{IH}$$

    but we are still missing the well-formedness of the extended environment. We derive this last missing piece by

    $$\frac{\text{WfEnv ve' te'} \qquad \text{ValTyp v2 t1}}{\text{WfEnv (v2 :: ve') (t1 :: te')}}\ \text{FA2\_CONS}$$

# Part III
## Extensions

# Extensions

- In the context of this thesis, the following systems have been formalized and proved type sound using the Coq proof assistant:
  - Simply Typed Lambda Calculus (STLC)
  - STLC with Mutable References
  - STLC with Substructural Types
  - STLC with Subtyping
  - STLC with Parametric Polymorphism (System F)
  - STLC with Bounded Quantification (System $F_{<:}$)

# Mutable References

- Three new forms of expression
  - e_ref e creates a reference with the value of e;
  - e_get e returns the value of a reference expression e; and
  - e_set e1 e2 points the reference of e1 to the value of e2.
- Two new forms of values
  - v_unit is the unit value returned by e_set; and
  - v_loc n is a location value resulting from the n-th use of e_ref.
- To relate a location with its value, value and type stores are required, analogously to value and type environments for variables.
- As value environments may contain locations, the well-formedness of environments is now also parametrized with the type store.

# Mutable References

- Type Soundness is formalized as

```
∀ n e te ve vs ts mv t,
eval n ve vs e = done mv →
ExpTyp te e t →
WfStore vs ts →
WfEnv ve te ts →
∃ v vs' ts' ,
  mv = noerr (v, vs' ) ∧
  WfStore vs' ts' ∧
  SubStore ts ts' ∧
  ValTyp ts' v t.
```

- Core lemmas are
  - wfenv_substore and vt_substore, stating that wellformed-environments WfEnv ve te ts and value typings ValTyp ts v t are preserved, if the type store ts is replaced by a larger type store; and
  - wfstore_extend, stating that a well-formed store WfStore vs ts can be extended by a value typing ValTyp ts v t to WfStore (v :: vs) (t :: ts).

# Substructural Types

- Substructural type systems impose restrictions on how often variables are allowed to be used.
- We extend the STLC with substructural types, such that both unrestricted (arbitrary many uses) and affine (at most 1 use) lambda abstractions are possible.
- Syntax is changed such that lambda abstractions and arrow types are annotated by their multiplicity, which is affine or unrestricted.
- Semantics and type soundness statement remain unchanged

# Substructural Types

- Two changes to the type system:
  - when typing applications e_app e1 e2, then it is no longer correct to simply propagate the type environment to both sub-expressions, as this would allow both e1 and e2 to make use of the same variable that might be affine.
  - when typing unrestricted abstractions e_abs unr e, then it is no longer correct to simply capture the whole environment, as the environment may contain affine variables, which may be used multiple times, as the unrestricted abstraction is allowed to be called multiple times.
- Corresponding to two lemmas required for the soundness proof:
  - split_preserves_wf, which is used in the e_abs case, and states that if well-formed environments WfEnv ve te are split, then both halves are again well-formed; and
  - restr_preserves_et, which is used in the e_app case, and states that if an expression has a typing in an restricted type environment restrict te, then it has the same type in te.

# Subtyping

- Subtyping introduces a binary relation $\sqsubseteq$ between types, such that if $t \sqsubseteq t'$, then any expression of type $t$ can also be given type $t'$.
- We extend the STLC by the t_top type, such that t $\sqsubseteq$ t_top for all types t.
- Syntax, Semantics, and Type Soundness Theorem remain unchanged
- A subsumption rule is added to the type system

```
| et_sub :
    ∀ te e t1 t2,
    ExpTyp te e t1 →
    ExpSubTyp t1 t2 →
    ExpTyp te e t2.
```

# Subtyping

- As subtyping allows expressions to be evaluated to values, which have a subtype of the expression's type, we extend the value typing, such that closures now not only can have their arrow type t_arr t1 t2, but also any larger type t.
- Hence, the type soundness proof now requires
  - reflexivity and transitivity of $\sqsubseteq$; and
  - that a value typing ValTyp e t1 can be widened along subtyping ExpSubTyp t1 t2 to ValTyp e t2.
- Note, that the subtyping relation is formalized independenty of the choice of semantics, so the conventional proof methods can be used for the properties of subtyping:
  - reflexivity follows by straightforward induction over the type; and
  - transitivity follows by induction over the sum of the sizes of both subtyping derivations.

# Parametric Polymorphism (System F)

- Just as the simply typed lambda calculus allows to introduce variables ranging over values, the parametric polymorphism in System F allows to introduce variables ranging over types.
- For this purpose the type syntax is extended by type variables and universal quantification, and the expression syntax is extended by type abstractions and type applications.
- For example, we can write a polymorphic identity function as

  $$\Lambda\alpha.\lambda(x:\alpha).x \;:\; \forall\alpha.\alpha\to\alpha,$$

  and instantiate it to a given type $\tau$ as

  $$(\Lambda\alpha.\lambda(x:\alpha).x)[\tau] \;\equiv\; \lambda(x:\tau).x \;:\; \tau\to\tau.$$

# Parametric Polymorphism (System F)

- We specify the semantics of a type application $e[\tau]$ not by substituting $\tau$ for the type variable in $e$, but instead by pushing $\tau$ into the value environment, leaving the variable in $e$ intact.
- As a consequence, we need to introduce a type equivalence, that relates types with respect to their value environments.
- For example, a type $\tau$ with respect to the empty environment is equivalent to a type variable $\alpha$ with respect to the environment that maps $\alpha$ to $\tau$.
- Thus, the core lemmas of the soundness theorem are about the interaction of type equivalence with substitution used in the type system.

# Parametric Polymorphism (System F)

- Similar to subtyping, we need a lemma vt_widen, that allows to transfer a value typing ValTyp ve v t along a type equivalence TEq ve t ve' t', yielding ValTyp ve' v t'. The lemma is used in the cases of lambda and type applications to relate the value typing from our goal to the value typing of the closure values produced by the induction hypothesis for the closure body.
- Similar to subtyping, we need a lemma teq_refl, that states the reflexivity of the type equivalence TEq. The lemma is used in the cases of lambda and type abstractions to build the value typing in the current environment.
- The teq_subst lemma is used in the type application case. It states the type equivalence between the direct type substitution performed by the type system and the delayed type substitution performed by the semantics through extending the value environment with a type closure. Proving this lemma requires a fair amount of extra machinery as witnessed by the proof graph.

# Bounded Quantification (System $F_{<:}$)

- Combines Parametric Polymorphism and Subtyping.
- Variables introduced by type abstractions are bounded by subtyping
  - e.g. $\Lambda(\alpha <: \tau).\lambda(x : \alpha).x \ : \ \forall(\alpha <: \tau).\alpha \to \alpha$
- Interaction of polymorphism and subtyping is non-trivial
- While the subtyping relation used in the type system, is the same as for small-step semantics, the value typing now requires a special subtyping relation that incorporates the type equivalence mentioned for System F.
- This value subtyping requires proofs of transitivity, analogouly to STLC + subtyping.

# Bounded Quantification (System $F_{<:}$)

- Value Subtyping can be formulated logically or algorithmically:
  - The logical subtyping relation has an explicit rule for transitivity, which makes it easy to use transitivity, but hard to perform induction over the subtyping, as the case of transitivity has to be covered.
  - The algorithmic subtyping relation has transitivity rules only for type variables, which makes it easy to perform induction over the subtyping, but hard to use transitivity, as it has to proved first as a complicated lemma.
- To get the best of both worlds, we formalized type soundness using the algorithmic subtyping, and prove the algorithmic subtyping equivalent to the logical subtyping, which yields the transitivity of the algorithmic subtyping as a corollary.

# Bounded Quantification (System $F_{<:}$)

- For this purpose, we extend the proof from Pierce's TAPL for the equivalence of the logical and algorithmic subtyping relation, such that it still works with the incorporated type equivalence modulo value environments.
- The original proof is by induction over the size of the type in the middle. The proof breakes in the type variable case, where the type corresponding to the variable is in the value environment, and hence not smaller than the type in the middle.
- The solution we found is to perform induction over the size of the type in the middle + the recursive size of the value environments.
- The recursive size is used, as the value environment may contain type closures that have captured other value environments.

# Part IV
## Conclusion

# Conclusion

- Type soundness proofs using definitional interpreters seem to provide a viable alternative to proofs using small-step semantics.
- Both proof techniques, allow for handling non-termination by using only basic inductive proof techniques, instead of resorting to heavier machinery like coinduction.
- In contrast to small-step semantics, formalizations of systems with variables do not require a *substitution-preserves-typing* lemma, which doesn't hold in some languages, e.g. Dot Calculus.

# The End

Thanks for your attention!
Questions?

Coq Mechanizations
`https://github.com/m0rphism/definitional`.

Wright, Felleisen.
*A syntactic approach to type soundness*.
Information and computation

Rompf, Amin.
*From F to DOT: Type soundness proofs with definitional interpreters*.
arXiv preprint arXiv:1510.05216