A Syntactic Approach to Type Soundness

Andrew K. Wright, Matthias Felleisen

Summary by: Joshua Peignier

Plan

Introduction

- Type systems: useful to detect type errors before the execution of a program and prevent them.

- Type systems: useful to detect type errors before the execution of a program and prevent them.
  $\rightarrow$ For instance: preventing a program from trying to evaluate `1 + true`, or $(1\ f)$.

Introduction

- Type systems: useful to detect type errors before the execution of a program and prevent them.
  $\rightarrow$ For instance: preventing a program from trying to evaluate `1 + true`, or $(1\ f)$.
- Such errors are usually detected during the compilation, and stop the compilation if they occur.

Introduction

- Type systems: useful to detect type errors before the execution of a program and prevent them.
  $\rightarrow$ For instance: preventing a program from trying to evaluate `1 + true`, or $(1\ f)$.
- Such errors are usually detected during the compilation, and stop the compilation if they occur.
- Soundness properties: ensure that if a program is well-typed, then no such error can happen.

Type Soundness

- Notation: $\triangleright e : \tau$ in a type system if e has the type τ in that type system.

Type Soundness

- Notation: $\triangleright e : \tau$ in a type system if e has the type τ in that type system.
- Consider the partial function $eval : Programs \rightarrow V \cup \{\text{wrong}\}$.

Type Soundness

- Notation: $\triangleright e : \tau$ in a type system if e has the type τ in that type system.
- Consider the partial function $eval : Programs \to V \cup \{\text{wrong}\}$.
- Consider a partition $V = \biguplus_{\tau} V^{\tau}$.

Type Soundness

- Notation: $\triangleright e : \tau$ in a type system if e has the type τ in that type system.
- Consider the partial function $eval : Programs \to V \cup \{\text{wrong}\}$.
- Consider a partition $V = \biguplus_{\tau} V^{\tau}$.

Weak Soundness

If $\triangleright e : \tau$, then $eval(e) \neq \text{wrong}$.

Type Soundness

- Notation: $\triangleright e : \tau$ in a type system if e has the type τ in that type system.
- Consider the partial function $eval : Programs \to V \cup \{\text{wrong}\}$.
- Consider a partition $V = \biguplus_{\tau} V^{\tau}$.

Weak Soundness

If $\triangleright e : \tau$, then $eval(e) \neq \text{wrong}$.

Strong Soundness

If $\triangleright e : \tau$ and $eval(e) = v$, then $v \in V^{\tau}$.

Proving Type Soundness

- Proving type soundness of a given type system over a language: non-trivial for certain type systems.

Proving Type Soundness

- Proving type soundness of a given type system over a language: non-trivial for certain type systems.
  $\rightarrow$ For instance, Hindley/Milner-style type systems. (Including polymorphism and type inference.)

Proving Type Soundness

- Proving type soundness of a given type system over a language: non-trivial for certain type systems.
  $\rightarrow$ For instance, Hindley/Milner-style type systems. (Including polymorphism and type inference.)
- Existing proofs: often *ad hoc* proofs:

Proving Type Soundness

- Proving type soundness of a given type system over a language: non-trivial for certain type systems.
  $\rightarrow$ For instance, Hindley/Milner-style type systems. (Including polymorphism and type inference.)
- Existing proofs: often *ad hoc* proofs:
  - Difficult to combine existing proofs for two languages for another language including features of both.

Proving Type Soundness

- Proving type soundness of a given type system over a language: non-trivial for certain type systems.
  $\rightarrow$ For instance, Hindley/Milner-style type systems. (Including polymorphism and type inference.)
- Existing proofs: often *ad hoc* proofs:
  - Difficult to combine existing proofs for two languages for another language including features of both.
  - Different techniques are required depending on whether the semantics is specified as operational or denotational.

Proving Type Soundness

- Proving type soundness of a given type system over a language: non-trivial for certain type systems.
  $\rightarrow$ For instance, Hindley/Milner-style type systems. (Including polymorphism and type inference.)
- Existing proofs: often *ad hoc* proofs:
  - Difficult to combine existing proofs for two languages for another language including features of both.
  - Different techniques are required depending on whether the semantics is specified as operational or denotational.
- Contribution: Provide a new approach for proofs of type soundness for Hindley/Milner-style type systems.

Proving Type Soundness

- Proving type soundness of a given type system over a language: non-trivial for certain type systems.
  $\rightarrow$ For instance, Hindley/Milner-style type systems. (Including polymorphism and type inference.)
- Existing proofs: often *ad hoc* proofs:
  - Difficult to combine existing proofs for two languages for another language including features of both.
  - Different techniques are required depending on whether the semantics is specified as operational or denotational.
- Contribution: Provide a new approach for proofs of type soundness for Hindley/Milner-style type systems.
  $\rightarrow$ Based on *subject reduction* result and *rewriting* system as semantics.

Plan

Functional-ML

Expressions and Values in Functional-ML:

Expressions and Values in Functional-ML:

Functional-ML Syntax

$$e ::= v \mid e_1 \; e_2 \mid \text{let } x \text{ be } e_1 \text{ in } e_2$$

$$v ::= c \mid x \mid \mathrm{Y} \mid \lambda x.e$$

Functional-ML

Expressions and Values in Functional-ML:

Functional-ML Syntax

$$e ::= v \mid e_1\ e_2 \mid \mathrm{let}\ x\ \mathrm{be}\ e_1\ \mathrm{in}\ e_2$$

$$v ::= c \mid x \mid \mathrm{Y} \mid \lambda x.e$$

Consider a partial function $\delta : \mathit{Const} \times \mathit{ClosedVal} \rightarrow \mathit{ClosedVal}$.

Functional-ML

Expressions and Values in Functional-ML:

Functional-ML Syntax

$$e ::= v \mid e_1 \; e_2 \mid \text{let } x \text{ be } e_1 \text{ in } e_2$$

$$v ::= c \mid x \mid \mathrm{Y} \mid \lambda x.e$$

Consider a partial function $\delta : \mathit{Const} \times \mathit{ClosedVal} \rightarrow \mathit{ClosedVal}$.

Reduction relation

$$\begin{array}{rcl} c \; v & \rightarrow & \delta(c, v) \text{ when defined} \\ (\lambda x.e) \; v & \rightarrow & e[x \mapsto v] \\ \text{let } x \text{ be } v \text{ in } e & \rightarrow & e[x \mapsto v] \\ \mathrm{Y} \; v & \rightarrow & v \; (\lambda x.(\mathrm{Y} \; v) \; \; x) \end{array}$$

Functional-ML

Semantics based on the reduction $\rightarrow$ and on evaluation contexts:

Evaluation contexts

$$E ::= [\,] \mid E\ e \mid v\ E \mid \mathrm{let}\ x\ \mathrm{be}\ E\ \mathrm{in}\ e$$

Functional-ML

Semantics based on the reduction $\rightarrow$ and on evaluation contexts:

Evaluation contexts

$$E ::= [\,] \mid E\ e \mid v\ E \mid \mathrm{let}\ x\ \mathrm{be}\ E\ \mathrm{in}\ e$$

- $\longmapsto$ relation defined by : $E[e] \longmapsto E[e']$ iff $e \rightarrow e'$.

Functional-ML

Semantics based on the reduction $\rightarrow$ and on evaluation contexts:

Evaluation contexts

$$E ::= [\,] \mid E\ e \mid v\ E \mid \text{let } x \text{ be } E \text{ in } e$$

- $\longmapsto$ relation defined by : $E[e] \longmapsto E[e']$ iff $e \rightarrow e'$.
- $\longmapsto\!\!\!\!\rightarrow$: transitive and reflexive closure of $\longmapsto$.

Functional-ML

Semantics based on the reduction $\rightarrow$ and on evaluation contexts:

Evaluation contexts

$$E ::= [] \mid E\ e \mid v\ E \mid \mathrm{let}\ x\ \mathrm{be}\ E\ \mathrm{in}\ e$$

- $\longmapsto$ relation defined by : $E[e] \longmapsto E[e']$ iff $e \rightarrow e'$.
- $\longmapsto\!\!\!\!\rightarrow$: transitive and reflexive closure of $\longmapsto$.
- With this definition, there is at most one possible reduction at each step.

Functional-ML

Semantics based on the reduction $\rightarrow$ and on evaluation contexts:

Evaluation contexts

$$E ::= [\,] \mid E\ e \mid v\ E \mid \text{let } x \text{ be } E \text{ in } e$$

- $\longmapsto$ relation defined by : $E[e] \longmapsto E[e']$ iff $e \rightarrow e'$.
- $\longmapsto\!\!\!\!\rightarrow$: transitive and reflexive closure of $\longmapsto$.
- With this definition, there is at most one possible reduction at each step.
- Reduction have the form:

$$E[e_1] \longmapsto \cdots \longmapsto E[v] = E'[e'_1] \longmapsto \ldots$$

$$\longmapsto E'[v'] = E''[e''_1] \longmapsto \cdots \longmapsto v_0$$

Type system

Types in Functional-ML have the following form

Type system

Types in Functional-ML have the following form

Types

$$\tau ::= \iota_1 \mid \ldots \mid \iota_n \mid \alpha \mid \tau_1 \to \tau_2$$

Type system

Types in Functional-ML have the following form

Types

$$\tau ::= \iota_1 \mid \ldots \mid \iota_n \mid \alpha \mid \tau_1 \rightarrow \tau_2$$

- Type scheme: $\sigma = \forall \alpha_1 \ldots \alpha_n . \tau$

Type system

Types in Functional-ML have the following form

Types

$$\tau ::= \iota_1 \mid \ldots \mid \iota_n \mid \alpha \mid \tau_1 \rightarrow \tau_2$$

- Type scheme: $\sigma = \forall \alpha_1 \ldots \alpha_n . \tau$
  $\rightarrow$ Represents the set of types obtained by substitution of $\alpha_1, \ldots, \alpha_n$.

Type system

Types in Functional-ML have the following form

Types

$$\tau ::= \iota_1 \mid \ldots \mid \iota_n \mid \alpha \mid \tau_1 \rightarrow \tau_2$$

- Type scheme: $\sigma = \forall \alpha_1 \ldots \alpha_n . \tau$
  $\rightarrow$ Represents the set of types obtained by substitution of $\alpha_1, \ldots, \alpha_n$.
- Generalization relation $\succ$ over type schemes: $\sigma \succ \tau$ when σ' is obtained by substitution of bound type variables in σ.

Type system

Types in Functional-ML have the following form

Types

$$\tau ::= \iota_1 \mid \ldots \mid \iota_n \mid \alpha \mid \tau_1 \rightarrow \tau_2$$

- Type scheme: $\sigma = \forall \alpha_1 \ldots \alpha_n . \tau$
  $\rightarrow$ Represents the set of types obtained by substitution of $\alpha_1, \ldots, \alpha_n$.
- Generalization relation $\succ$ over type schemes: $\sigma \succ \tau$ when σ' is obtained by substitution of bound type variables in σ.
- Type environment Γ: map from free variables to type schemes.

Typability condition

δ-typability

If $\mathit{TypeOf}(c) \succ \tau' \to \tau$ and $\triangleright v : \tau'$, then $\delta(c, v)$ is defined and $\triangleright \delta(c, v) : \tau$.

Typability condition

δ-typability

If $\mathit{TypeOf}(c) \succ \tau' \to \tau$ and $\triangleright v : \tau'$, then $\delta(c, v)$ is defined and $\triangleright \delta(c, v) : \tau$.

This condition is required for soundness to hold.

Typing rules in Functional-ML

$$\Gamma \triangleright x : \tau \text{ if } \Gamma(x) \succ \tau$$

$$\Gamma \triangleright c : \tau \text{ if } \mathit{TypeOf}(c) \succ \tau$$

$$\Gamma \triangleright \mathrm{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2$$

$$\frac{\Gamma[x \mapsto \tau_1] \triangleright e : \tau_2}{\Gamma \triangleright \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \triangleright e_1 : \tau_1 \to \tau_2 \qquad \Gamma \triangleright e_2 : \tau_1}{\Gamma \triangleright e_1 \ e_2 : \tau_2}$$

$$\frac{\Gamma \triangleright e_1 : \tau_1 \qquad \Gamma[x \mapsto \mathit{Close}(\tau_1, \Gamma)] \triangleright e_2 : \tau_2}{\Gamma \triangleright \mathrm{let}\ x\ \mathrm{be}\ e_1\ \mathrm{in}\ e_2 : \tau_2}$$

Plan

Proving Type Soundness

Goal:

Well-typed program

If e is closed and if there exists τ such that $\triangleright e : \tau$, then e is a well-typed program.

Proving Type Soundness

Goal:

Well-typed program

If e is closed and if there exists τ such that $\triangleright e : \tau$, then e is a well-typed program.

Goal:

Syntactic Soundness

If $\triangleright e : \tau$, then either $e \Uparrow$, or $e \longmapsto\!\!\!\!\!\rightarrow v$ and $\triangleright v : \tau$.

Proving Type Soundness

Goal:

Well-typed program

If e is closed and if there exists τ such that $\triangleright e : \tau$, then e is a well-typed program.

Goal:

Syntactic Soundness

If $\triangleright e : \tau$, then either $e \Uparrow$, or $e \longmapsto\!\!\!\!\rightarrow v$ and $\triangleright v : \tau$.

Layout of the proof:

Proving Type Soundness

Goal:

Well-typed program

If e is closed and if there exists τ such that $\triangleright e : \tau$, then e is a well-typed program.

Goal:

Syntactic Soundness

If $\triangleright e : \tau$, then either $e \Uparrow$, or $e \longmapsto\!\!\!\!\!\rightarrow v$ and $\triangleright v : \tau$.

Layout of the proof:

- Showing type preservation through reduction (*subject reduction*).

Proving Type Soundness

Goal:

Well-typed program

If e is closed and if there exists τ such that $\triangleright e : \tau$, then e is a well-typed program.

Goal:

Syntactic Soundness

If $\triangleright e : \tau$, then either $e \Uparrow$, or $e \longmapsto\!\!\!\!\rightarrow v$ and $\triangleright v : \tau$.

Layout of the proof:

- Showing type preservation through reduction (*subject reduction*).
- Characterizing answers and faulty expressions (i.e. what do we expect to cause a type error ?).

Proving Type Soundness

Goal:

Well-typed program

If e is closed and if there exists τ such that $\triangleright e : \tau$, then e is a well-typed program.

Goal:

Syntactic Soundness

If $\triangleright e : \tau$, then either $e \Uparrow$, or $e \longmapsto\!\!\!\!\!\rightarrow v$ and $\triangleright v : \tau$.

Layout of the proof:

- Showing type preservation through reduction (*subject reduction*).
- Characterizing answers and faulty expressions (i.e. what do we expect to cause a type error ?).
- Showing that faulty expressions are untypable.

Subject reduction

Theorem

If $\Gamma \triangleright e_1 : \tau$ and $e_1 \rightarrow e_2$, then $\Gamma \triangleright e_2 : \tau$.

Subject reduction

Theorem

If $\Gamma \rhd e_1 : \tau$ and $e_1 \rightarrow e_2$, then $\Gamma \rhd e_2 : \tau$.

Corollary

If $\Gamma \rhd e_1 : \tau$ and $e_1 \longmapsto\!\!\!\rightarrow e_2$, then $\Gamma \rhd e_2 : \tau$.

Subject reduction

Theorem

If $\Gamma \rhd e_1 : \tau$ and $e_1 \rightarrow e_2$, then $\Gamma \rhd e_2 : \tau$.

Corollary

If $\Gamma \rhd e_1 : \tau$ and $e_1 \longmapsto\!\!\!\!\rightarrow e_2$, then $\Gamma \rhd e_2 : \tau$.

Proof by case analysis over $e_1 \rightarrow e_2$ (one step of reduction).

Subject reduction

Theorem

If $\Gamma \triangleright e_1 : \tau$ and $e_1 \rightarrow e_2$, then $\Gamma \triangleright e_2 : \tau$.

Corollary

If $\Gamma \triangleright e_1 : \tau$ and $e_1 \longmapsto\!\!\!\!\rightarrow e_2$, then $\Gamma \triangleright e_2 : \tau$.

Proof by case analysis over $e_1 \rightarrow e_2$ (one step of reduction).
Abstraction and let cases relie on the following lemma:

Lemma

If $\Gamma[x \mapsto \forall \alpha_1 \ldots \alpha_n.\tau] \triangleright e : \tau'$ and $x \notin Dom(\Gamma)$ and $\Gamma \triangleright v : \tau$ and $\alpha_1, \ldots, \alpha_n$ are not free in Γ, then $\Gamma \triangleright e[x \mapsto v] : \tau'$.

Faulty expressions

Faulty expressions: expected to cause a type error, for instance an invalid application, even if the evaluation of the whole expression is not necessarily stuck.

Faulty expressions

Faulty expressions: expected to cause a type error, for instance an invalid application, even if the evaluation of the whole expression is not necessarily stuck.

Definition

The faulty expressions are expressions containing a subexpression $(c\ v)$ where $\delta(c, v)$ is undefined.

Faulty expressions

Faulty expressions: expected to cause a type error, for instance an invalid application, even if the evaluation of the whole expression is not necessarily stuck.

Definition

The faulty expressions are expressions containing a subexpression $(c\ \ v)$ where $\delta(c, v)$ is undefined.

Example: $((\lambda y.2)\ (\lambda x.(+\ 1\ \texttt{true}))$.

Faulty expressions

Faulty expressions: expected to cause a type error, for instance an invalid application, even if the evaluation of the whole expression is not necessarily stuck.

Definition

The faulty expressions are expressions containing a subexpression $(c\ \ v)$ where $\delta(c, v)$ is undefined.

Example: $((\lambda y.2)\ (\lambda x.(+\ 1\ \texttt{true})))$.
This expression reduces to 2, but we can expect it to cause a type error.

Faulty expressions

Lemma

If e is faulty, there are no Γ, τ such that $\Gamma \rightarrow e : \tau$.

Faulty expressions

Lemma

If e is faulty, there are no Γ, τ such that $\Gamma \to e : \tau$.

Proven by case analysis on a faulty subexpression in e.

Faulty expressions

Lemma

If e is faulty, there are no Γ, τ such that $\Gamma \rightarrow e : \tau$.

Proven by case analysis on a faulty subexpression in e.Here, the only possible case is $(c\ v)$ with $\delta(c, v)$ undefined. Easily proven with contradiction.

Faulty expressions

Lemma

If e is faulty, there are no Γ, τ such that $\Gamma \rightarrow e : \tau$.

Proven by case analysis on a faulty subexpression in e.Here, the only possible case is $(c\ \ v)$ with $\delta(c, v)$ undefined. Easily proven with contradiction.

Lemma (Uniform Evaluation)

For closed e, if e cannot be reduced to a faulty expression, then either $e \Uparrow$ or $e \longmapsto\!\!\!\!\!\rightarrow v$.

Faulty expressions

Lemma

If e is faulty, there are no Γ, τ such that $\Gamma \rightarrow e : \tau$.

Proven by case analysis on a faulty subexpression in e.Here, the only possible case is $(c\ v)$ with $\delta(c, v)$ undefined. Easily proven with contradiction.

Lemma (Uniform Evaluation)

For closed e, if e cannot be reduced to a faulty expression, then either $e \Uparrow$ or $e \longmapsto\!\!\!\!\!\rightarrow v$.

Corollary: Syntactic Soundness

If $\triangleright e : \tau$, then either $e \Uparrow$, or $e \longmapsto\!\!\!\!\!\rightarrow v$ and $\triangleright v : \tau$.

Soundness results

Definition

Let *eval* denote the following function:

$$eval(e) = \begin{cases} \text{wrong} & \text{if } e \longmapsto\!\!\!\!\!\rightarrow e' \text{ and } e' \text{ is faulty} \\ v & \text{if } e \longmapsto\!\!\!\!\!\rightarrow v \end{cases}$$

Soundness results

Definition

Let *eval* denote the following function:

$$eval(e) = \begin{cases} \text{wrong} & \text{if } e \longmapsto\!\!\!\!\rightarrow e' \text{ and } e' \text{ is faulty} \\ v & \text{if } e \longmapsto\!\!\!\!\rightarrow v \end{cases}$$

Weak Soundness

If $\triangleright e : \tau$, then $eval(e) \neq \text{wrong}$.

Soundness results

Definition

Let *eval* denote the following function:

$$eval(e) = \begin{cases} \text{wrong} & \text{if } e \longmapsto\!\!\!\!\rightarrow e' \text{ and } e' \text{ is faulty} \\ v & \text{if } e \longmapsto\!\!\!\!\rightarrow v \end{cases}$$

Weak Soundness

If $\triangleright e : \tau$, then $eval(e) \neq \text{wrong}$.

Strong Soundness

If $\triangleright e : \tau$ and $eval(e) = v$, then $v \in V^{\tau}$.

Plan

Conclusion

- Proving type soundness for Hindley/Milner-style type systems: non-trivial

Conclusion

- Proving type soundness for Hindley/Milner-style type systems: non-trivial
- Previously: *ad hoc* proofs, strongly depending on the chosen semantics, hardly reusable for other proofs for close languages.

Conclusion

- Proving type soundness for Hindley/Milner-style type systems: non-trivial
- Previously: *ad hoc* proofs, strongly depending on the chosen semantics, hardly reusable for other proofs for close languages.
- In this article, the authors designed a new approach, based on subject reduction, defining faulty expressions and showing that they are not typable.

Conclusion

- Proving type soundness for Hindley/Milner-style type systems: non-trivial
- Previously: *ad hoc* proofs, strongly depending on the chosen semantics, hardly reusable for other proofs for close languages.
- In this article, the authors designed a new approach, based on subject reduction, defining faulty expressions and showing that they are not typable.
- This approach was used to prove type soundness for Functional ML.

Conclusion

- Proving type soundness for Hindley/Milner-style type systems: non-trivial
- Previously: *ad hoc* proofs, strongly depending on the chosen semantics, hardly reusable for other proofs for close languages.
- In this article, the authors designed a new approach, based on subject reduction, defining faulty expressions and showing that they are not typable.
- This approach was used to prove type soundness for Functional ML.
- Type soundness was also proven for extensions of Functional ML including references, exceptions, and first-class continuation.