Well-Typed Languages are Sound

Matteo Cimini (Indiana University)

SNAPL 17, Asilomar
9 May 2017
Program Analysis

pen&paper proofs
interactive theorem proving
automated proving
type systems
model checking
control flow analysis
abstract interpretation

Language Analysis

pen&paper proofs
interactive theorem proving
automated proving

... ??

Rich portfolio of techniques

Not so rich...
Our research program:
To port the type systems analysis technique to Language Analysis
One example: { Type soundness
Small-step semantics
A certain class of languages

Type Systems for Language Definitions

joint work with Dale Miller (INRIA) and Jeremy Siek (Indiana University)
Type Systems for Language Definitions

Vision

One example:

Type soundness
Small-step semantics
Across languages
A certain class of languages
Type Systems for Language Definitions

Vision

One example: { Type soundness
Across semantics approaches
Small-step semantics
Across languages
A certain class of languages }
Type Systems for Language Definitions

Vision

One example:

Many language properties

Type soundness

Across semantics approaches

Small step semantics

Across languages

A certain class of languages
Type Soundness, in half slide

What is type soundness? You can trust your types!

```kotlin
fun average(n1:Int, n2:Int) {
    return (n1 + n2) / 2
}
```
Example Language Definition

\[ \mathcal{L} = (\text{Types, Expressions, Values, Errors, Contexts, Type System, Dynamic Semantics}) \]

**Types**

\[ T ::= \text{Int} \mid T \to T \mid \text{List } T \]

**Expressions**

\[ e ::= x \mid \lambda x.e \mid e \cdot e \mid \text{nil} \mid \text{cons } e e \mid \text{head } e \mid \text{tail } e \mid \text{error} \]

**Values**

\[ v ::= \lambda x.e \mid \text{nil} \mid \text{cons } v v \]

**Errors**

\[ er ::= \text{error} \]

**Contexts**

\[ E ::= E e \mid v E \mid \text{cons } E e \mid \text{cons } v E \mid \text{head } E \mid \text{tail } E \]

**Type System**

\[
\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : \text{List } T}{\Gamma \vdash \text{cons } e_1 e_2 : \text{List } T}
\]

\[
\frac{\Gamma \vdash e : \text{List } T}{\Gamma \vdash \text{head } e : T}
\]

**Dynamic Semantics**

\[
(\lambda x.e) v \rightarrow e[v/x]
\]

\[
\text{head } \text{nil} \rightarrow \text{error}
\]

\[
\text{head } (\text{cons } v_1 v_2) \rightarrow v_1
\]

\[
\text{tail } \text{nil} \rightarrow \text{error}
\]

\[
\text{tail } (\text{cons } v_1 v_2) \rightarrow v_2
\]

\[
\ldots
\]
Example Language Definition

\[ \mathcal{L} = (\text{Types}, \text{Expressions}, \text{Values}, \text{Errors}, \text{Contexts}, \text{Type System}, \text{Dynamic Semantics}) \]

\[ \lambda x. e \mid \text{nil} \mid \text{cons} \]

\[
\begin{align*}
\Gamma \vdash e_1 : T & \quad \Gamma \vdash e_2 : \text{List } T \\
\Gamma \vdash \text{cons } e_1 e_2 : \text{List } T
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \text{List } T \\
\Gamma \vdash \text{head } e : T
\end{align*}
\]

...
Type Systems for Type Soundness

Our message: “Well-typed languages are sound”
Type Systems for Type Soundness

Our meta type system performs type checking on all the components of the language, that is:

Our meta type system automatically holds. We have proved the following theorem [3].

In subsequent work, we will extend our meta type system to allow for layered grammars such as

Given a typed language

"Well-typed languages are sound"

Our message: “Well-typed languages are sound”
Type checking the Type System

PL Design:

“organize the language operators in introduction and elimination forms”
“introduction forms build values of some specific data type”

Values $\vdash$ Type System

\[
\begin{align*}
\text{op} \in \text{Values} \\
\Gamma^* \vdash e_1 : ty_1 & \quad \ldots \quad \Gamma^* \vdash e_n : ty_n \\
\Gamma \vdash (\text{op} e_1 \cdots e_n) : (c \overrightarrow{T})
\end{align*}
\]
Type checking the Type System
(introduction forms)

Examples

\[
\text{cons } v \ v \ | \ \cdots \vdash \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : (\text{List } T)}{\Gamma \vdash (\text{cons } e_1 \ e_2) : (\text{List } T)} : \text{cons} \leftrightarrow \text{intro List}
\]

\[
\text{succ } v \ | \ \cdots \vdash \frac{\Gamma \vdash e : \text{Int}}{\Gamma \vdash (\text{succ } e) : \text{Int}} : \text{succ} \leftrightarrow \text{intro Int}
\]

\[
\lambda x.e \ | \ \cdots \vdash \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \rightarrow T_2} : \lambda \leftrightarrow \text{intro } \rightarrow
\]

\[
\text{inl } v \ | \ \cdots \vdash \frac{\Gamma \vdash e : T_1}{\Gamma \vdash \text{inl } e : T_1 + T_2} : \text{inl} \leftrightarrow \text{intro } +
\]

\[
\Lambda X.e \ | \ \cdots \vdash \frac{\Gamma, X \vdash e : T}{\Gamma \vdash \Lambda X.e : \forall X.T} : \Lambda \leftrightarrow \text{intro } \forall
\]
Type checking the Type System

PL Design:

“organize the language operators in introduction and elimination forms”
“elimination forms manipulate values of some specific data type”

Values $\vdash$ Type System

\[
\begin{align*}
\Gamma & \vdash e_1 : (c \, T) \\
\vdots & \\
\Gamma^* & \vdash e_n : ty_n \\
\Gamma & \vdash (op \, e_1 \ldots e_n) : ty
\end{align*}
\]

\[
\begin{align*}
op \notin \text{Values} \\
\text{Values} \vdash \Gamma \vdash e_1 : (c \, T) \\
\vdots \\
\Gamma^* \vdash e_n : ty_n \\
\Gamma \vdash (op \, e_1 \ldots e_n) : ty
\end{align*}
\]
Type checking the Type System
(elimination forms)

Examples

\[
\frac{\Gamma \vdash e : \text{List } T}{\Gamma \vdash \text{head } e : T} : \text{head} \mapsto \text{elim List}
\]

\[
\frac{\Gamma \vdash e : \text{Int}}{\Gamma \vdash \text{pred } e : \text{Int}} : \text{pred} \mapsto \text{elim Int}
\]

\[
\frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash \text{app } e_1 \ e_2 : T_2} : \text{app} \mapsto \text{elim } \rightarrow
\]

\[
\frac{\Gamma \vdash e_1 : T_1 + T_2 \quad \Gamma, x : T_1 \vdash e_2 : T \quad \Gamma, x : T_2 \vdash e_3 : T}{\Gamma \vdash \text{case } e_1 \text{ of } \text{inl } x \Rightarrow e_2 \mid \text{inr } x \Rightarrow e_3 : T} : \text{case} \mapsto \text{elim } +
\]

\[
\frac{\Gamma \vdash e : \forall X.T_2}{\Gamma \vdash (\text{appT } e \ [T_1]) : T_2[T_1/X]} : \text{appT} \mapsto \text{elim } \forall
\]
Type checking the Dynamic Semantics

"elimination forms manipulate values of some specific data type"

\[
\begin{align*}
\text{Values} & \quad v ::= \lambda x.e \mid \text{nil} \mid \text{cons } v \, v \\
\text{Contexts} & \quad E ::= \text{app } E \, e \mid \text{cons } E \, e \mid \text{cons } v \, E \mid \text{app } v \, E \mid \text{head } E
\end{align*}
\]

\[
\begin{align*}
\text{app } (\lambda x.e) \, v & \rightarrow e[v/x] \\
\text{head } \text{nil} & \rightarrow \text{error} \\
\text{head } (\text{cons } v_1 \, v_2) & \rightarrow v_1
\end{align*}
\]

Classification \(\vdash op_1 : \text{elim } c\) Classification \(\vdash op_2 : \text{intro } c\)

\[
\begin{array}{c}
\text{Values} \mid (op_2 \, \vec{e} \, v) \in \text{Values} \quad \text{Contexts} \mid (op_1 \, E \, \cdots) \\
\text{Contexts} \mid (op_1 \, \cdots \, E \, \cdots) \quad (\text{for all } ev_i = v_k)
\end{array}
\]

Values \mid Contexts \mid \vdash (op_1 \, (op_2 \, \vec{e} \, v) \, \vec{e'} \, v') \rightarrow \text{exp} : op_1 \mapsto \text{eliminates } op_2

\[
ev ::= e \mid v
\]
Type checking the Dynamic Semantics

**Examples**

| Classification ⊨ app : elim → | Classification ⊨ λ : intro → |
|-----------------------------|-----------------------------|
| λx.e ∈ Values | app E e ∈ Contexts | app v E ∈ Contexts |

Values | Contexts | Classification ⊨ app (λx.e) v → e[v/x] : app ⇔ eliminates λ

| Classification ⊨ head : elim List | Classification ⊨ cons : intro List |
|----------------------------------|-----------------------------------|
| cons v v ∈ Values | head E ∈ Contexts |

Values | Contexts | Classification ⊨ head (cons v_1 v_2) → v_1 : head ⇔ eliminates cons

| Classification ⊨ case : elim + | Classification ⊨ inl : intro + |
|-------------------------------|-------------------------------|
| inl v ∈ Values | case(x) E e e ∈ Contexts |

Values | Contexts | Classification ⊨ case(x)(inl v)e_2 e_3 → e_2[v/x_1] : case ⇔ eliminates inl
Applicability

Integers, booleans + if-then-else, pairs, lists, sums, tuples, universal types, recursive types, derived operators (fix, let, letrec), error handlers/exception mechanisms (try/catch).

CBV, CBN, parallel beta-reduction, Left-to-right, right-to-left, lazy lists/tuples.

Theorem (Well-typed languages are sound)

\[ \text{if } \vdash \mathcal{L} \text{ then } \mathcal{L} \text{ is type sound.} \]

(Cimini, Miller, Siek. ArXiv paper)
TypeSoundnessCertifier

A tool for type checking language definitions and certifying their soundness

(annotated Abella) abella-prover.org/
Spotting Design Mistakes

```
typeOf nil (list T).
typeOf (cons E1 E2) (list T) :- typeOf E1 T, typeOf E2 (list T).
step (head nil) error.
step (head (cons E1 E2)) E1 :- value E1, value E2.
value nil.
value (cons E1 E2) :- value E1, value E2.
% context cons E e.
% context cons v E.
% context head E.
```

“**head** is an elimination form but its principal argument is not an evaluation context, hence type soundness does not hold”

**“thanks to the high-level type system, error messages use the same jargon of language designers**
Automatic Certification of Soundness

Theorem canonical_form_list : 
forall E T1, {typeOf E (list T1)} -> {value E} ->
  E = nil / (exists Arg1 Arg2, E = (cons Arg1 Arg2) / value Arg1 / value Arg2).
intros Main Value. case Main.
case Value. search. % for nil
case Value. search. % for cons
case Value. ... case Value. % cases that are not values have trivial proofs

Theorem progress_head :
forall E1 T, {typeOf (head E1) T} -> progresses E1 -> progresses (head E1).
intros Main PrgsE1. case Main. ProgressCase : case PrgsE1.
Canonical : apply canonical_form_list to E1 ProgressCase. % type-driven emitted code
case Canonical. search. search. (V) % type-driven emitted code
search.(S) search.(E)

and so on   ...
Vision: A World of Language Type Checkers and Certifiers!

languages with stores
- effects
- dependent types
- linear types ...

big-step semantics
definitional interpreters
denotational semantics? ...

determinism/confluence
strong normalization
relational parametricity
data-race freedom
security-based properties ...

Language Definition

Type Checker

Proof

Proof Compiler

Domain-specific Error

Coq proof

Isabelle proof

Abella proof

...
Vision: A World of Language Type Checkers and Certifiers!

- languages with stores
- effects
- dependent types
- linear types
- determinism/confluence
- strong normalization
- relational parametricity
- data-race freedom
- security-based properties
- big-step semantics
- definitional interpreters
- denotational semantics
- proof
- Coq proof
- Isabelle proof
- Abella proof

Thank you!
Happy Certification!