Formalizing *Types and Programming Languages* in Isabelle/HOL

Martin Desharnais

École de technologie supérieure

Types and Programming Languages

Outline

Outline

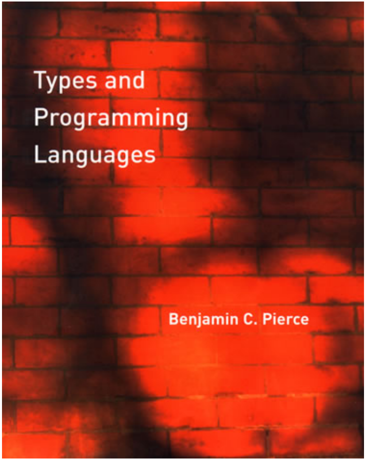
Motivation

Definition of the λ-Calculus

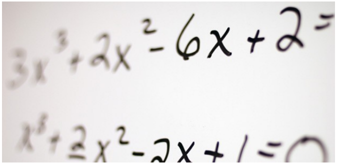
Augmenting the λ-Calculus with a Type System

Properties of the Typed λ-Calculus

Why this book?



Why a formalization?



Formalization = Definitions + Properties + Proofs

Why in Isabelle/HOL?

HOL = Functional Programming + Logic

Outline

Motivation

Definition of the λ-Calculus

Augmenting the λ-Calculus with a Type System

Properties of the Typed λ-Calculus

What is the λ-calculus?

$t ::=$			$\lambda x.\ x$
	x	variable	$\lambda x.\lambda y.\ x$
	$\lambda x.\ t$	abstraction	$\lambda f.\lambda x.\ f\ x\ x$
	$t_1\ t_2$	application	$\lambda f.\lambda g.\lambda x.\ f\ (g\ x)$

Bound variable names are irrelevent:

$$\lambda x.\ x \ = \ \lambda y.\ y$$

Function application:

$$(\lambda x.\ x\ x)\ y \ = \ y\ y$$

Formalization of terms

$t ::=$	
x	variable
$\lambda x.\ t$	abstraction
$t_1\ t_2$	application

```
datatype ulterm =
  ULVar nat |
  ULAbs ulterm |
  ULApp ulterm ulterm
```

De Bruijn indices

$$\lambda x.\ (\lambda y.\ y\ (\lambda z.\ z))(\lambda w.\ x\ w)$$

$$\lambda\ (\lambda\ 1\ (\lambda\ 1))\ (\lambda\ 2\ 1)$$

Formalization of single-step evaluation

$$\frac{t_1 \implies t_1'}{t_1\ t_2 \implies t_1'\ t_2}$$

$$\frac{t_2 \implies t_2'}{v_1\ t_2 \implies v_1\ t_2'}$$

$$(\lambda x.\ t_{12})\ v_2 \implies [x \mapsto v_2]\ t_{12}$$

Formalization of single-step evaluation

$$\frac{t_1 \implies t_1'}{t_1\ t_2 \implies t_1'\ t_2}$$

$$\frac{t_2 \implies t_2'}{v_1\ t_2 \implies v_1\ t_2'}$$

$$(\lambda x.\ t_{12})\ v_2 \implies [x \mapsto v_2]\ t_{12}$$

```
inductive eval1_UL :: "ulterm ⇒ ulterm ⇒ bool" where
  "eval1_UL t1 t1' ⟹ eval1_UL (ULApp t1 t2) (ULApp t1' t2)" |
  "is_value_UL v1 ⟹ eval1_UL t2 t2' ⟹ eval1_UL (ULApp v1 t2) (ULApp v1 t2')" |
  "is_value_UL v2 ⟹ eval1_UL (ULApp (ULAbs t12) v2)
    (shift_UL (-1) 0 (subst_UL 0 (shift_UL 1 0 v2) t12))"
```

Outline

Motivation

Definition of the λ-Calculus

Augmenting the λ-Calculus with a Type System

Properties of the Typed λ-Calculus

What are type systems?

Classification according to kind of values

Detection of errors Abstractions Maintenance

Inflexible Restrictive

Formalization of typing relation

$$\Gamma \vdash t : T$$

Formalization of typing relation

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

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

Formalization of typing relation

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

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

```
inductive has_type_L :: "lcontext ⇒ lterm ⇒ ltype ⇒ bool" ("((_)/ ⊢ (_)/ |:| (_))
  "(x, T) |∈| Γ ⟹ Γ ⊢ (LVar x) |:| T" |
  "(Γ |,| T1) ⊢ t2 |:| T2 ⟹ Γ ⊢ (LAbs T1 t2) |:| (T1 → T2)" |
  "Γ ⊢ t1 |:| (T11 → T12) ⟹ Γ ⊢ t2 |:| T11 ⟹ Γ ⊢ (LApp t1 t2) |:| T12"
```

Outline

Type safety

Safety = Progress + Preservation

The progress theorem

9.3.5 THEOREM [PROGRESS]: Suppose $\mathtt{t}$ is a closed, well-typed term (that is, $\vdash \mathtt{t} : \mathsf{T}$ for some T). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$. □

Proof: Straightforward induction on typing derivations. The cases for boolean constants and conditions are exactly the same as in the proof of progress for typed arithmetic expressions (8.3.2). The variable case cannot occur (because $\mathtt{t}$ is closed). The abstraction case is immediate, since abstractions are values.

The only interesting case is the one for application, where $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$ with $\vdash \mathtt{t}_1 : \mathsf{T}_{11} \to \mathsf{T}_{12}$ and $\vdash \mathtt{t}_2 : \mathsf{T}_{11}$. By the induction hypothesis, either $\mathtt{t}_1$ is a value or else it can make a step of evaluation, and likewise $\mathtt{t}_2$. If $\mathtt{t}_1$ can take a step, then rule E-APP1 applies to $\mathtt{t}$. If $\mathtt{t}_1$ is a value and $\mathtt{t}_2$ can take a step, then rule E-APP2 applies. Finally, if both $\mathtt{t}_1$ and $\mathtt{t}_2$ are values, then the canonical forms lemma tells us that $\mathtt{t}_1$ has the form $\lambda \mathtt{x}{:}\mathsf{T}_{11}.\mathtt{t}_{12}$, and so rule E-APPABS applies to $\mathtt{t}$. □

The progress theorem

9.3.5 **Theorem [Progress]:** Suppose t is a closed, well-typed term (that is, $\vdash \mathtt{t} : \mathtt{T}$ for some T). Then either t is a value or else there is some t′ with $\mathtt{t} \longrightarrow \mathtt{t}'$. □

Proof: Straightforward induction on typing derivations. The cases for boolean constants and conditions are exactly the same as in the proof of progress for typed arithmetic expressions (8.3.2). The variable case cannot occur (because t is closed). The abstraction case is immediate, since abstractions are values.

The only interesting case is the one for application, where $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$ with $\vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12}$ and $\vdash \mathtt{t}_2 : \mathtt{T}_{11}$. By the induction hypothesis, either $\mathtt{t}_1$ is a value or else it can make a step of evaluation, and likewise $\mathtt{t}_2$. If $\mathtt{t}_1$ can take a step, then rule E-App1 applies to t. If $\mathtt{t}_1$ is a value and $\mathtt{t}_2$ can take a step, then rule E-App2 applies. Finally, if both $\mathtt{t}_1$ and $\mathtt{t}_2$ are values, then the canonical forms lemma tells us that $\mathtt{t}_1$ has the form $\lambda \mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12}$, and so rule E-AppAbs applies to t. □

```
theorem progress:
  "∅ ⊢ t |:| T ⟹ is_closed t ⟹ is_value_L t ∨ (∃t'. eval1_L t t')"
proof (induction t T rule: has_type_L.induct)
  case (has_type_LIf Γ t1 t2 T t3)
  thus ?case by (cases "is_value_L t1")
    (auto intro: eval1_L.intros dest: canonical_forms simp: is_closed_def)
next
  case (has_type_LApp Γ t1 T11 T12 t2)
  thus ?case by (cases "is_value_L t1", cases "is_value_L t2")
    (auto intro: eval1_L.intros dest: canonical_forms simp: is_closed_def)
qed (simp_all add: is_value_L.intros is_closed_def)
```

The progress theorem

9.3.5 **Theorem [Progress]:** Suppose t is a closed, well-typed term (that is, $\vdash$ t : T for some T). Then either t is a value or else there is some t′ with t $\longrightarrow$ t′. □

Proof: Straightforward induction on typing derivations. The cases for boolean constants and conditions are exactly the same as in the proof of progress for typed arithmetic expressions (8.3.2). The variable case cannot occur (because t is closed). The abstraction case is immediate, since abstractions are values.

The only interesting case is the one for application, where $t = t_1\ t_2$ with $\vdash t_1 : T_{11} \to T_{12}$ and $\vdash t_2 : T_{11}$. By the induction hypothesis, either t_1 is a value or else it can make a step of evaluation, and likewise t_2. If t_1 can take a step, then rule E-App1 applies to t. If t_1 is a value and t_2 can take a step, then rule E-App2 applies. Finally, if both t_1 and t_2 are values, then the canonical forms lemma tells us that t_1 has the form $\lambda x{:}T_{11}.t_{12}$, and so rule E-AppAbs applies to t. □

```
theorem progress:
  "∅ ⊢ t |:| T ⟹ is_closed t ⟹ is_value_L t ∨ (∃t'. eval1_L t t')"
proof (induction t T rule: has_type_L.induct)
  case (has_type_LIf Γ t1 t2 T t3)
  thus ?case by (cases "is_value_L t1")
    (auto intro: eval1_L.intros dest: canonical_forms simp: is_closed_def)
next
  case (has_type_LApp Γ t1 T11 T12 t2)
  thus ?case by (cases "is_value_L t1", cases "is_value_L t2")
    (auto intro: eval1_L.intros dest: canonical_forms simp: is_closed_def)
qed (simp_all add: is_value_L.intros is_closed_def)
```

Other theorems proved

Determinacy of evaluation

Uniqueness of normal form

(Non-)termination of evaluation

Preservation of typing

Erasability of types

etc.

What I learned

- λ-calculus and why it is relevent
- Type systems as security net
- Isabelle/HOL as development platform
- Formalization is as concrete as programming
- How my intership was relevent

Thank you for your attention