CS 421 Lecture 20: Proof systems

- Lecture outline
  - Defining proof systems
    - Judgments
    - Axioms
    - Rules of inference
    - Proofs
  - T_{simp}: Simple proof system for types in Ocaml
  - OS_{simp}: Simple proof system for operational semantics of OCaml

Motivation

- Understanding and reasoning about programs and programming languages
- How does OCaml type inference work?
- Can we *prove* that a program is correct?
  - *I.e.*, that the result of the computation is always what we would expect.

Proof systems

- Proof system: formalized representation of mathematical proofs based on *axioms* and *rules of inference*.

- Can be used to formalize deductions for many purposes
  - **Type-checking** axioms and rules of inference allow proofs of assertions (a.k.a. "judgments") of the form "expression *e* has type τ".
  - **Operational semantics** rules allow proofs of judgments of the form "*e* evaluates to *v*".
  - **Axiomatic semantics** rules allow proofs of judgments of the form "If the variables in a program initially satisfy some conditions *C*, then after executing statement *S*, they will satisfy conditions *C'* ".

Proof systems

To define a proof system, we need to define three things:

- **Judgments**: A judgment J is an assertion whose truth is subject to proof.
- **Axioms**: Judgments that are assumed to be true without proof. There are usually an infinite number of axioms, so they can't all be listed, but they need to be described in some way. Written: $\overline{J}$
- **Rules of inference**: Rules that allow you to infer a judgment from one or more previously-inferred judgments. Written: $\dfrac{J_1 \ldots J_n}{J}$

Proofs

- Given a proof system, a **proof** is a tree labeled with judgments, such that:
  - Every judgment labeling a leaf node is an axiom
  - Every judgment labeling an internal node can be inferred from its children by a rule of inference.

- Notational notes:
  1. Axioms and rules of inference are usually given names, and these names are placed in the proof tree
  2. Proof trees are written with the root – the main judgment being proved at the bottom.

T_{simp} – simplified OCaml type system

- Types: int | $\tau \rightarrow \tau'$ (for any types τ and τ')
- Type environments Γ: mapping from variables to types
- Judgments: $\Gamma \vdash e : \tau$
- Expressions: constants, variables, abstractions, application

T_{simp} – simplified OCaml type system

Axioms

(Const) $\dfrac{}{\Gamma \vdash 0 : \text{int}} \qquad \dfrac{}{\Gamma \vdash 1 : \text{int}}$

$$\frac{}{\Gamma \vdash + : \text{int} \to \text{int} \to \text{int}}$$

(and many more)

(Var) $\dfrac{}{\Gamma \vdash x : \Gamma x}$

T_{simp} – simplified OCaml type system

Rules of inference:

(Application) $$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

(Abstraction) $$\frac{\Gamma[x : \tau] - e : \tau'}{\Gamma \vdash \text{fun } x \to e : \tau \to \tau'}$$

Example

fun x -> fun y -> (+ x) y : int → int → int

Example

fun g -> g(fun x-> x+1) : ((int → int) → int) → int

Notes on T_{simp}

- Given Γ, e, and τ:
  - The structure of the proof tree is completely determined by e – it is the same as the abstract syntax.
  - The content of the proof tree is almost completely determined by e and τ; however, in the application rule, even given Γ, e1, e2, and τ', τ is not uniquely determined.

- *Proving* $\emptyset \vdash e : \tau$ is called **type checking**.

- *Finding* τ such that $\emptyset \vdash e : \tau$ can be proved is called **type inference.**

OS_{simp} – simplified OCaml operational semantics

- The **operational semantics** of a language says, in an abstract way, how programs in a language are executed.
- For a functional language like Ocaml, the operational semantics should say how expressions are evaluated.
  - We will take the view that the evaluation of an expression involves transforming it to another, simpler expression.
- *E.g.*, "(fun x -> x*x) 4" evaluates to "16".

OS_{simp} – simplified OCaml operational semantics

- We give the operational semantics of a very simplified OCaml as a proof system. We need to define the judgments of the system, and then give the axioms and rules of inference.
  - **Expressions** (simplified Ocaml): constants, variables, fun x -> e, e1 e2, e1 $\oplus$ e2
  - **Values**: constants, closed abstractions (*i.e.*, fun x -> e, where e has no free variables other than x)
  - **Judgments**: $e \Downarrow v$ (where *e* is closed)

OS_{simp} – simplified OCaml operational semantics

- Axioms

(Const) $\dfrac{}{k \Downarrow k}$ for constants k

(Abstr) $\dfrac{}{\text{fun } x \to e \Downarrow \text{fun } x \to e}$ (fun x -> e closed)

OS_{simp} – simplified OCaml operational semantics

Rules of inference:

(Application)

$$\frac{e_1 \Downarrow \text{fun } x \to e \quad e_2 \Downarrow v' \quad e[v'/x] \Downarrow v}{e_1 e_2 \Downarrow v}$$

(δ rules)

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v = v_1 \oplus v_2}{e_1 \oplus e_2 \Downarrow v}$$

where $\oplus$ is any built-in function

Example

$+ (+ 3\ 4)\ 5 \Downarrow 12$

Example

(fun x -> + x x) (+ 3 4) ⇓ 14

Example

(fun f -> f(fun x -> x)) (fun y -> y) 4 ⇓ 4

Notes on OS_{simp}

- The structure of the proof tree for e ⇓ v is *similar* to the structure of e, but not the same.
  - It would be less similar if our simple language had recursion.

- However, the proof tree – structure and content – are completely, unambiguously determined by the expression e.
  - There is no intelligence or insight required; building the proof tree is completely mechanical.

Next two lectures

- Will present more complex and realistic proof systems for type-checking and operational semantics of OCaml.

- Type system
  - Polymorphism and the special role of "let".
  - Type-checking of references (*i.e.*, assignable variables)

- Operational semantics
  - Handling recursion