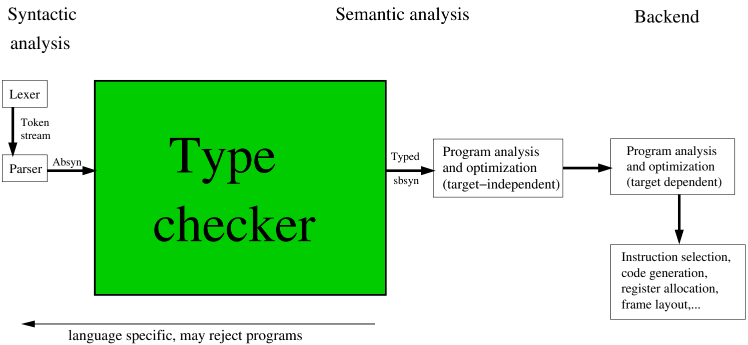
Type analysis

Lennart Beringer

COS320, Compiling Techniques, Spring 2011
See cos320/typelecture.pdf

February 2011

Where are we?



- Purpose & core challenges of type analysis
- Step-by-step development of type system for FUN-like (but slightly different) language

Motivation

Purpose of type systems (I)

For programmers:

Purpose of type systems (I)

For programmers:

- help to eliminate common programming mistakes, particularly those that may lead to runtime errors
- provide abstraction and modularization discipline: can substitute code with code of equal type without breaking surrounding code (interface/signature types)

For language designers:

Motivation

Purpose of type systems (I)

For programmers:

- help to eliminate common programming mistakes, particularly those that may lead to runtime errors
- provide abstraction and modularization discipline: can substitute code with code of equal type without breaking surrounding code (interface/signature types)

For language designers:

- structuring principle for programs
- basis for studying (interaction between) language features such as exceptions, references, IO-side effects,...
- formal basis for reasoning about program behaviour (verification, security analysis,...)

Motivation

Purpose of type systems (II)

For compiler writers:

Motivation

Purpose of type systems (II)

For compiler writers:

- provide information for later phases:
  - does value v fit into a single register? (size of data types)
  - how should stack frame for function f be organized? (number and types of parameters and return value)
  - support generation of efficient code: less code for error-handling (casting) needs to be inserted, sharing of representations (source of confusion eliminated by types)
  - post-Y2k-compilers: typed intermediate languages: model each intermediate code representations as separate language, use types to communicate structural code invariants and analysis results between compiler phases (example: different types for caller/callee-registers)
- "refined" type systems: provide alternative formalism for program analysis and optimization

Motivation

Language level errors

Can eliminate many programmer mistakes, and ensure “good” (safe!) runtime behaviour:

Memory safety: can’t dereference anything that’s not a pointer (can’t forge pointers), including nullPtr

Control flow safety: can’t jump to address that doesn’t contain code, can’t overwrite code (e.g. return address)

Type safety: typing predictions come true at run time (“this expression will produce a string”), so operator-operand mismatches eliminated

Contrast this with C, where lots of (implicit) casting happens, and lots of errors ensue (out of bounds, buffer overflows, seg faults, security violations,. . .).

Type systems: limitations

Static type systems are usually:

- unable to eliminate all runtime errors:
  - division by zero
  - exception behaviour often not modeled/enforced
- conservative, i.e. will reject some legal programs due to undecidability. Example:

  $$\textbf{if } f(x) \textbf{ then } 1 \textbf{ else } \; (5 + \textbf{tt})$$

  where f is some function that takes long to compute but always returns **tt**.

Nevertheless useful, even for more complex properties:

- termination, security
- resource consumption, adherence to usage protocols

Dynamic type systems not considered in this lecture.

Fundamental & algorithmic tasks

Practical tasks (compiler writer): develop algorithms for

type inference: given an expression e, calculate whether there is some type τ such that $e : \tau$ holds. If so, return the best such type, or (a representation of) all fitting types. May need program annotations.

type checking: given a fully type-decorated program, check that the decoration indeed respects the typing rules

Theoretical tasks (language designer):

uniqueness of typings, existence of best types

decidability & complexity of above tasks/algorithms

type soundness: give precise definition of “good behaviour” (runtime model, error model), and prove that well-typed programs don’t do wrong.

Common formalism: derivation system (cf. formal logic), i.e. set of judgments and typing rules, tree-shaped derivations

Type system for simple expressions (I)

Starting point: abstract syntax

$$
\begin{array}{rcl}
e & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid \textbf{tt} \mid \textbf{ff} \\
 & & \mid e \oplus e \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \\
\oplus & ::= & + \mid - \mid \times \mid \wedge \mid \vee \mid < \mid =
\end{array}
$$

Type system for simple expressions (I)

Starting point: abstract syntax

$$e ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid \mathbf{tt} \mid \mathbf{ff}$$
$$\mid e \oplus e \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e$$
$$\oplus ::= + \mid - \mid \times \mid \wedge \mid \vee \mid < \mid =$$

Step 1: define notion of types

Aim: separate integer expressions from boolean expressions, to prevent operations like $5 + \mathbf{tt}$.

Thus: $\tau ::= \mathbf{bool} \mid \mathbf{int}$

Type system for simple expressions (I)

Starting point: abstract syntax

$$e ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid \textbf{tt} \mid \textbf{ff}$$
$$\mid e \oplus e \mid \textbf{if } e \textbf{ then } e \textbf{ else } e$$
$$\oplus ::= + \mid - \mid \times \mid \wedge \mid \vee \mid < \mid =$$

Step 1: define notion of types
Aim: separate integer expressions from boolean expressions, to prevent operations like $5 + \textbf{tt}$.
Thus: $\tau ::= \textbf{bool} \mid \textbf{int}$

Next: define derivation system

Derivation systems

Judgments J: logical statement (claim) that may or may not be true. Truth can only be determined once an interpretation is defined (we use intuition...).

Inference rules: Axioms: $\textsc{Name}\ \dfrac{}{J}\ SC$

Rules : $\textsc{Name}\ \dfrac{J_{Hyp_0} \quad \ldots \quad J_{Hyp_n}}{J_{Concl}}\ SC$

Derivation system: inductive interpretation of rules, i.e. finite trees where nodes are rule instantiations (axioms in leaves), root is overall conclusion

Derivation systems

Judgments J: logical statement (claim) that may or may not be true. Truth can only be determined once an interpretation is defined (we use intuition...).

Inference rules: Axioms: $\textsc{Name}\ \dfrac{}{J}\ SC$

Rules : $\textsc{Name}\ \dfrac{J_{Hyp_0} \quad \ldots \quad J_{Hyp_n}}{J_{Concl}}\ SC$

Derivation system: inductive interpretation of rules, i.e. finite trees where nodes are rule instantiations (axioms in leaves), root is overall conclusion

Type inference: construct a proof tree for the root judgment
Type checking: check well-formedness of a purported proof tree
Type soundness: given an interpretation of judgments, prove that derivability implies validity. Proof typically by induction: axioms establish valid judgments, non-axioms preserve validity (assuming side conditions)

Type system for simple expressions (II)

Step 2: decide on forms of judgments

$$\vdash e : \tau$$

Intuitive interpretation: “evaluating expression e yields value of type τ.”

Type system for simple expressions (II)

Step 2: decide on forms of judgments

$$\vdash e : \tau$$

Intuitive interpretation: "evaluating expression e yields value of type τ."

Step 3: define inference rules, ideally syntax-directed: one rule/axiom for each syntax former

Type system for simple expressions (II)

Step 2: decide on forms of judgments

$$\vdash e : \tau$$

Intuitive interpretation: "evaluating expression *e* yields value of type τ."

Step 3: define inference rules, ideally syntax-directed: one rule/axiom for each syntax former

Axioms (for atomic expressions):

$$\text{TT}\ \frac{}{\vdash \mathbf{tt} : \mathbf{bool}} \qquad \text{FF}\ \frac{}{\vdash \mathbf{ff} : \mathbf{bool}}$$

$$\text{NUM}\ \frac{}{\vdash n : \mathbf{int}}\ n \in \{\ldots, -1, 0, 1, \ldots\}$$

Type system for simple expressions (III)

Rules for non-atomic expressions: one hypothesis for each subexpression.

Type system for simple expressions (III)

Rules for non-atomic expressions: one hypothesis for each subexpression.

Built-in operators: prevent application of built-in operators to wrong kinds of arguments.

Type system for simple expressions (III)

Rules for non-atomic expressions: one hypothesis for each subexpression.

Built-in operators: prevent application of built-in operators to wrong kinds of arguments.

$$\textsc{IOp}\ \frac{\vdash e_1 : \mathbf{int} \quad \vdash e_2 : \mathbf{int}}{\vdash e_1 \oplus e_2 : \mathbf{int}}\ \oplus \in \{+, -, \times\}$$

$$\textsc{BOp}\ \frac{\vdash e_1 : \mathbf{bool} \quad \vdash e_2 : \mathbf{bool}}{\vdash e_1 \oplus e_2 : \mathbf{bool}}\ \oplus \in \{\wedge, \vee\}$$

$$\textsc{COp}\ \frac{\vdash e_1 : \mathbf{int} \quad \vdash e_2 : \mathbf{int}}{\vdash e_1 \oplus e_2 : \mathbf{bool}}\ \oplus \in \{<, =\}$$

Type system for simple expressions (III)

Rules for non-atomic expressions: one hypothesis for each subexpression.

Built-in operators: prevent application of built-in operators to wrong kinds of arguments.

$$\textsc{IOp}\ \frac{\vdash e_1 : \textbf{int} \quad \vdash e_2 : \textbf{int}}{\vdash e_1 \oplus e_2 : \textbf{int}}\ \oplus \in \{+, -, \times\}$$

$$\textsc{BOp}\ \frac{\vdash e_1 : \textbf{bool} \quad \vdash e_2 : \textbf{bool}}{\vdash e_1 \oplus e_2 : \textbf{bool}}\ \oplus \in \{\wedge, \vee\}$$

$$\textsc{COp}\ \frac{\vdash e_1 : \textbf{int} \quad \vdash e_2 : \textbf{int}}{\vdash e_1 \oplus e_2 : \textbf{bool}}\ \oplus \in \{<, =\}$$

Conditionals: branch condition should be boolean, arms should agree on their type (τ), and overall type is τ, too

Type system for simple expressions (III)

Rules for non-atomic expressions: one hypothesis for each subexpression.

Built-in operators: prevent application of built-in operators to wrong kinds of arguments.

$$\text{IOP}\ \frac{\vdash e_1 : \mathbf{int} \quad \vdash e_2 : \mathbf{int}}{\vdash e_1 \oplus e_2 : \mathbf{int}}\ \oplus \in \{+, -, \times\}$$

$$\text{BOP}\ \frac{\vdash e_1 : \mathbf{bool} \quad \vdash e_2 : \mathbf{bool}}{\vdash e_1 \oplus e_2 : \mathbf{bool}}\ \oplus \in \{\wedge, \vee\}$$

$$\text{COP}\ \frac{\vdash e_1 : \mathbf{int} \quad \vdash e_2 : \mathbf{int}}{\vdash e_1 \oplus e_2 : \mathbf{bool}}\ \oplus \in \{<, =\}$$

Conditionals: branch condition should be boolean, arms should agree on their type (τ), and overall type is τ, too

$$\text{ITE}\ \frac{\vdash e_1 : \mathbf{bool} \quad \vdash e_2 : \tau \quad \vdash e_3 : \tau}{\vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_2 : \tau}$$

Type system for simple expressions (IV)

Inference can happen top-down or bottom-up.

Exercise

Perform syntax-directed inference for the expressions

- $3 + (\textbf{if}\ (3 < 5) \wedge ((2+2) = 5)\ \textbf{then}\ 7\ \textbf{else}\ (2 * 5))$
- $3 + (\textbf{if}\ (3 < 5) \wedge ((2+2) = 5)\ \textbf{then}\ 7\ \textbf{else}\ (5 + \textbf{tt}))$.

Are the derivations/final judgments unique?

Type system for simple expressions (IV)

Inference can happen top-down or bottom-up.

Exercise

Perform syntax-directed inference for the expressions

- $3 + (\textbf{if } (3 < 5) \wedge ((2+2) = 5) \textbf{ then } 7 \textbf{ else } (2 * 5))$
- $3 + (\textbf{if } (3 < 5) \wedge ((2+2) = 5) \textbf{ then } 7 \textbf{ else } (5 + \textbf{tt}))$.

Are the derivations/final judgments unique?

Exercise (homework)

Define a simple type system for above expressions *e* that counts the number of atomic subexpressions.

Type system for simple expressions (IV)

Inference can happen top-down or bottom-up.

Exercise

Perform syntax-directed inference for the expressions

- $3 + (\textbf{if } (3 < 5) \wedge ((2+2) = 5) \textbf{ then } 7 \textbf{ else } (2 * 5))$
- $3 + (\textbf{if } (3 < 5) \wedge ((2+2) = 5) \textbf{ then } 7 \textbf{ else } (5 + \textbf{tt}))$.

Are the derivations/final judgments unique?

Exercise (homework)

Define a simple type system for above expressions *e* that counts the number of atomic subexpressions.

Next: type system for languages with variables, functions, references, and products/records. These features require new types, judgment forms, and rules

Adding variables (I)

Starting point (absyn): extend syntax of expressions:

$$e ::= \ldots \mid x$$

where x ranges over identifiers

Step 1 (types): no changes – still only booleans and integers

Adding variables (I)

Starting point (absyn): extend syntax of expressions:

$$e ::= \ldots \mid x$$

where x ranges over identifiers

Step 1 (types): no changes – still only booleans and integers

Step 2 (judgments): expressions can contain variables, hence we can only associate types with expressions if we are given the types of the variables (assumptions).

Contexts

Adding variables (I)

Starting point (absyn): extend syntax of expressions:

$$e ::= \ldots \mid x$$

where x ranges over identifiers

Step 1 (types): no changes – still only booleans and integers

Step 2 (judgments): expressions can contain variables, hence we can only associate types with expressions if we are given the types of the variables (assumptions).

Contexts

A (typing) context Γ is a partial function mapping variables to types, usually written in the form $x_0 : \tau_0, \ldots x_n : \tau_n$, where all the x_i are distinct. Note: not all identifiers are required to occur.

Example: $\Gamma = \mathrm{x} : \textbf{int}, \mathrm{y} : \textbf{bool}, \mathrm{z} : \textbf{int}$

Variables (II)

Step 2 (ctd'): judgments with contexts: $\Gamma \vdash e : \tau$

Variables (II)

Step 2 (ctd'): judgments with contexts: $\Gamma \vdash e : \tau$

Step 3.1 (axioms): essentially no changes for constant expressions (just add Γ):

$$\text{TT}\ \frac{}{\Gamma \vdash \textbf{tt} : \textbf{bool}} \qquad \text{FF}\ \frac{}{\Gamma \vdash \textbf{ff} : \textbf{bool}}$$

$$\text{NUM}\ \frac{}{\Gamma \vdash n : \textbf{int}}\ n \in \{\ldots, -1, 0, 1, \ldots\}$$

Variables (II)

Step 2 (ctd'): judgments with contexts: $\Gamma \vdash e : \tau$

Step 3.1 (axioms): essentially no changes for constant expressions (just add Γ):

$$\text{TT}\ \frac{}{\Gamma \vdash \textbf{tt} : \textbf{bool}} \qquad \text{FF}\ \frac{}{\Gamma \vdash \textbf{ff} : \textbf{bool}}$$

$$\text{NUM}\ \frac{}{\Gamma \vdash n : \textbf{int}}\ n \in \{\ldots, -1, 0, 1, \ldots\}$$

Novel rule (context lookup): $\textsc{Var}\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$

Variables (II)

Step 2 (ctd'): judgments with contexts: $\Gamma \vdash e : \tau$

Step 3.1 (axioms): essentially no changes for constant expressions (just add Γ):

$$\text{TT}\ \frac{}{\Gamma \vdash \textbf{tt} : \textbf{bool}} \qquad \text{FF}\ \frac{}{\Gamma \vdash \textbf{ff} : \textbf{bool}}$$

$$\text{NUM}\ \frac{}{\Gamma \vdash n : \textbf{int}}\ n \in \{\ldots, -1, 0, 1, \ldots\}$$

Novel rule (context lookup): $\text{VAR}\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$

Step 3.2 (rules for composite expressions): essentially no changes (just add Γ everywhere)

Shortcoming?

Variables (II)

Step 2 (ctd'): judgments with contexts: $\Gamma \vdash e : \tau$

Step 3.1 (axioms): essentially no changes for constant expressions (just add Γ):

$$\text{TT}\ \frac{}{\Gamma \vdash \mathbf{tt} : \mathbf{bool}} \qquad \text{FF}\ \frac{}{\Gamma \vdash \mathbf{ff} : \mathbf{bool}}$$

$$\text{NUM}\ \frac{}{\Gamma \vdash n : \mathbf{int}}\ n \in \{\ldots, -1, 0, 1, \ldots\}$$

Novel rule (context lookup): $\text{VAR}\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$

Step 3.2 (rules for composite expressions): essentially no changes (just add Γ everywhere)

Shortcoming? cannot add a binding to variables.

Variables (III)

Extension by let-binding (ML-style)

Step 1: add new composite expression former:

$$e ::= \ldots \mid \textbf{let } x = e \textbf{ in } e \textbf{ end}$$

Variables (III)

Extension by let-binding (ML-style)

Step 1: add new composite expression former:

$$e ::= \ldots \mid \textbf{let } x = e \textbf{ in } e \textbf{ end}$$

Step 2: define update operation $\Gamma[x : \tau]$ on contexts: delete any binding for x in Γ (if existent), then add binding $x : \tau$. No changes in format of judgments

Step 3: new typing rule:

Variables (III)

Extension by let-binding (ML-style)

Step 1: add new composite expression former:

$$e ::= \ldots \mid \textbf{let } x = e \textbf{ in } e \textbf{ end}$$

Step 2: define update operation $\Gamma[x : \tau]$ on contexts: delete any binding for x in Γ (if existent), then add binding $x : \tau$. No changes in format of judgments

Step 3: new typing rule:

$$\textsc{Let}\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma[x : \sigma] \vdash e_2 : \tau}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau}$$

Variables (III)

Extension by let-binding (ML-style)

Step 1: add new composite expression former:

$$e ::= \ldots \mid \textbf{let } x = e \textbf{ in } e \textbf{ end}$$

Step 2: define update operation $\Gamma[x : \tau]$ on contexts: delete any binding for x in Γ (if existent), then add binding $x : \tau$. No changes in format of judgments

Step 3: new typing rule:

$$\textsc{Let}\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma[x : \sigma] \vdash e_2 : \tau}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau}$$

Exercise

Perform inference (i.e. find τ if existent) for

- $b : \textbf{bool} \vdash \textbf{if } b \textbf{ then let } x = 3 \textbf{ in } x \textbf{ end else } 4 : \tau$
- $x : \textbf{int}, y : \textbf{int} \vdash \textbf{let } x = x < y \textbf{ in if } x \textbf{ then } y \textbf{ else } 0 \textbf{ end} : \tau$
- $x : \textbf{int}, y : \textbf{int} \vdash \textbf{let } x = x < y \textbf{ in if } x \textbf{ then } y \textbf{ else } x \textbf{ end} : \tau$

Functions (I)

Starting point (absyn): two characteristic operations:

Functions (I)

Starting point (absyn): two characteristic operations:

Function formation

$$e ::= \ldots \mid \textbf{fun } f(x) = e_1 \textbf{ in } e_2 \textbf{ end}$$

declares function f with formal parameter x and body e_1. Name f may be referred to in e_1 (recursion) and e_2. Name x only in e_1.

Functions (I)

Starting point (absyn): two characteristic operations:

Function formation

$$e ::= \ldots \mid \textbf{fun } f(x) = e_1 \textbf{ in } e_2 \textbf{ end}$$

declares function f with formal parameter x and body e_1. Name f may be referred to in e_1 (recursion) and e_2. Name x only in e_1.

Function application

Denoted by juxtaposition : $e ::= \ldots \mid e\ e$

Functions (I)

Starting point (absyn): two characteristic operations:

Function formation

$$e ::= \dots \mid \textbf{fun } f(x) = e_1 \textbf{ in } e_2 \textbf{ end}$$

declares function f with formal parameter x and body e_1. Name f may be referred to in e_1 (recursion) and e_2. Name x only in e_1.

Function application

Denoted by juxtaposition : $e ::= \dots \mid e\ e$

Step 1 (types): Function/arrow type:

$$\tau ::= \dots \mid \tau_1 \to \tau_2$$

models functions with argument type τ_1 and return type τ_2

Functions (I)

Starting point (absyn): two characteristic operations:

Function formation

$$e ::= \ldots \mid \textbf{fun } f(x) = e_1 \textbf{ in } e_2 \textbf{ end}$$

declares function f with formal parameter x and body e_1. Name f may be referred to in e_1 (recursion) and e_2. Name x only in e_1.

Function application

Denoted by juxtaposition : $e ::= \ldots \mid e\ e$

Step 1 (types): Function/arrow type:

$$\tau ::= \ldots \mid \tau_1 \rightarrow \tau_2$$

models functions with argument type τ_1 and return type τ_2

Step 2 (judgment form): no change

Functions (II)

Aim: prevent application of functions to arguments of wrong type. And prevent applications $e\ e'$ where e is not a function.

Step 3: Rule for function formation:

Functions (II)

Aim: prevent application of functions to arguments of wrong type. And prevent applications $e\ e'$ where e is not a function.

Step 3: Rule for function formation:

$$\text{FUN}\frac{\Gamma[f : \tau_1 \to \tau_2][x : \tau_1] \vdash e_1 : \tau_2 \qquad \Gamma[f : \tau_1 \to \tau_2] \vdash e_2 : \tau}{\Gamma \vdash \textbf{fun}\ f(x) = e_1\ \textbf{in}\ e_2\ \textbf{end} : \tau}$$

First hypothesis verifies construction/declaration of f. Second hypothesis verifies its use. Note that types τ_1 and τ_2 have to be guessed.

Rule for function application:

Functions (II)

Aim: prevent application of functions to arguments of wrong type. And prevent applications $e\ e'$ where e is not a function.

Step 3: Rule for function formation:

$$\text{FUN}\frac{\Gamma[f : \tau_1 \to \tau_2][x : \tau_1] \vdash e_1 : \tau_2 \qquad \Gamma[f : \tau_1 \to \tau_2] \vdash e_2 : \tau}{\Gamma \vdash \textbf{fun } f(x) = e_1 \textbf{ in } e_2 \textbf{ end} : \tau}$$

First hypothesis verifies construction/declaration of f. Second hypothesis verifies its use. Note that types τ_1 and τ_2 have to be guessed.

Rule for function application:

$$\text{APP}\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

Functions (II)

Aim: prevent application of functions to arguments of wrong type. And prevent applications $e\ e'$ where e is not a function.

Step 3: Rule for function formation:

$$\textsc{Fun}\frac{\Gamma[f : \tau_1 \to \tau_2][x : \tau_1] \vdash e_1 : \tau_2 \qquad \Gamma[f : \tau_1 \to \tau_2] \vdash e_2 : \tau}{\Gamma \vdash \textbf{fun}\ f(x) = e_1\ \textbf{in}\ e_2\ \textbf{end} : \tau}$$

First hypothesis verifies construction/declaration of f. Second hypothesis verifies its use. Note that types τ_1 and τ_2 have to be guessed.

Rule for function application:

$$\textsc{App}\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

Exercise (homework)

Define an expression that declares and uses the factorial function, and write down its typing derivation.

References

Starting point (absyn): three characteristic operations:

References

Starting point (absyn): three characteristic operations:

Allocation, read, write (assign)

$$e ::= \ldots \mid \textbf{alloc}\ e \mid\ !e \mid e{:=}e$$

References

Starting point (absyn): three characteristic operations:

Allocation, read, write (assign)

$$e ::= \ldots \mid \textbf{alloc}\ e \mid !e \mid e{:=}e$$

Step 1 (types): $\tau ::= \ldots \mid \textbf{ref}\ \tau \mid \textbf{unit}$

Type **ref** τ models locations that can hold values of type τ.

Step 2 (judgment form): no change

Step 3 (rules): ALLOC

References

Starting point (absyn): three characteristic operations:

Allocation, read, write (assign)

$$e ::= \ldots \mid \textbf{alloc}\ e \mid !e \mid e{:=}e$$

Step 1 (types): $\tau ::= \ldots \mid \textbf{ref}\ \tau \mid \textbf{unit}$

Type **ref** τ models locations that can hold values of type τ.

Step 2 (judgment form): no change

Step 3 (rules): $\textsc{Alloc}\ \dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{alloc}\ e : \textbf{ref}\ \tau}$ $\textsc{Read}$

References

Starting point (absyn): three characteristic operations:

Allocation, read, write (assign)

$$e ::= \ldots \mid \textbf{alloc}\ e \mid !e \mid e{:=}e$$

Step 1 (types): $\tau ::= \ldots \mid \textbf{ref}\ \tau \mid \textbf{unit}$

Type **ref** τ models locations that can hold values of type τ.

Step 2 (judgment form): no change

Step 3 (rules): $\textsc{Alloc}\ \dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{alloc}\ e : \textbf{ref}\ \tau}$ $\quad \textsc{Read}\ \dfrac{\Gamma \vdash e : \textbf{ref}\ \tau}{\Gamma \vdash !e : \tau}$

$\textsc{Write}$

References

Starting point (absyn): three characteristic operations:

Allocation, read, write (assign)

$$e ::= \ldots \mid \textbf{alloc}\ e \mid !e \mid e{:=}e$$

Step 1 (types): $\tau ::= \ldots \mid \textbf{ref}\ \tau \mid \textbf{unit}$

Type **ref** τ models locations that can hold values of type τ.

Step 2 (judgment form): no change

Step 3 (rules):

$$\textsc{Alloc}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{alloc}\ e : \textbf{ref}\ \tau} \qquad \textsc{Read}\ \frac{\Gamma \vdash e : \textbf{ref}\ \tau}{\Gamma \vdash !e : \tau}$$

$$\textsc{Write}\ \frac{\Gamma \vdash e_1 : \textbf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1{:=}e_2 : \textbf{unit}}$$

Exercise (homework)

Redo factorial, but use a reference to hold the result.

Products

Starting point (absyn): two characteristic operations:

Products

Starting point (absyn): two characteristic operations:

Product formation, projections

$$e ::= \ldots \mid \langle e_1, \ldots, e_n \rangle \mid \#_n e$$

Products

Starting point (absyn): two characteristic operations:

Product formation, projections

$$e ::= \ldots \mid \langle e_1, \ldots, e_n \rangle \mid \#_n e$$

Step 1 (types): $\tau ::= \ldots \mid \langle \tau_1, \ldots, \tau_n \rangle$ ($n = 0$ amounts to **unit**)

Step 2 (judgment form): no change

Step 3 (rules): PROD

Products

Starting point (absyn): two characteristic operations:

Product formation, projections

$$e ::= \ldots \mid \langle e_1, \ldots, e_n \rangle \mid \#_n e$$

Step 1 (types): $\tau ::= \ldots \mid \langle \tau_1, \ldots, \tau_n \rangle$ ($n = 0$ amounts to **unit**)

Step 2 (judgment form): no change

Step 3 (rules): $\textsc{Prod}\ \dfrac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle}$

$\textsc{Proj}$

Products

Starting point (absyn): two characteristic operations:

Product formation, projections

$$e ::= \ldots \mid \langle e_1, \ldots, e_n \rangle \mid \#_n e$$

Step 1 (types): $\tau ::= \ldots \mid \langle \tau_1, \ldots, \tau_n \rangle$ ($n = 0$ amounts to **unit**)

Step 2 (judgment form): no change

Step 3 (rules):

$$\text{PROD}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle}$$

$$\text{PROJ}\ \frac{\Gamma \vdash e : \langle \tau_1, \ldots, \tau_n \rangle}{\Gamma \vdash \#_k e : \tau_k}\ 1 \leq k \leq n$$

Subtyping

Motivating observation

Expressions of type $\langle \tau_1, \ldots, \tau_n \rangle$ can be used as values of type $\langle \tau_1, \ldots, \tau_m \rangle$ for any $m \leq n$. Simply forget additional entries.

Indeed: any operation we may perform on an expression of the latter type (i.e. a projection $\#_k e$, which is only well-typed if $k \leq m$) is also legal on expressions of the former type.

Subtyping

Motivating observation

Expressions of type $\langle\tau_1, \ldots, \tau_n\rangle$ can be used as values of type $\langle\tau_1, \ldots, \tau_m\rangle$ for any $m \leq n$. Simply forget additional entries.

Indeed: any operation we may perform on an expression of the latter type (i.e. a projection $\#_k e$, which is only well-typed if $k \leq m$) is also legal on expressions of the former type.

General idea

Type τ is a subtype of σ if all values of type τ may also count as values of type σ. Operations that handle arguments of type σ must also handle arguments of type τ.

Axiomatize this idea in new judgment form subtyping: $\tau <: \sigma$. Again, we justify the axiomatization only informally.

Subtyping (II)

How to use subtyping: subsumption rule

How to use subtyping: subsumption rule

$$\textsc{Sub}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \sigma}\ \tau <: \sigma$$

Models the intuition that a τ-value may be provided whenever a σ-value is expected, i.e. interpretation as subset of values.

Subtyping (II)

How to use subtyping: subsumption rule

$$\textsc{Sub}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \sigma}\ \tau <: \sigma$$

Models the intuition that a τ-value may be provided whenever a σ-value is expected, i.e. interpretation as subset of values.

How to establish subtyping: Separate derivation system.

Subtyping (II)

How to use subtyping: subsumption rule

$$\textsc{Sub}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \sigma}\ \tau <: \sigma$$

Models the intuition that a τ-value may be provided whenever a σ-value is expected, i.e. interpretation as subset of values.

How to establish subtyping: Separate derivation system.

Pre-order rules

$$\textsc{SRefl}\ \frac{}{\tau <: \tau} \qquad \textsc{STrans}\ \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

These two rules deal with the base types **int**, **bool**, **unit**.
Next slides: rules that propagate subtpying through the various type formers.

Subtyping (III): propagation through products

Products (width): may truncate products

Subtyping (III): propagation through products

Products (width): may truncate products

$$\textsc{SProd}\ \frac{}{\langle \tau_1, \ldots, \tau_n \rangle <: \langle \tau_1, \ldots, \tau_m \rangle}\ m < n$$

Thought experiment: suppose $n < m$ instead. Take some e with, say, $\Gamma \vdash e : \langle \textbf{int}, \textbf{bool} \rangle$. By (hypothetical) rule SProd and Sub, have $\Gamma \vdash e : \langle \textbf{int}, \textbf{bool}, \textbf{int} \rangle$. So $\Gamma \vdash \#_3 e : \textbf{int}$ is well-typed. But this will crash!

Products: depth

Products (width): may truncate products

$$\text{SProd}\ \frac{}{\langle \tau_1, \ldots, \tau_n \rangle <: \langle \tau_1, \ldots, \tau_m \rangle}\ m < n$$

Thought experiment: suppose $n < m$ instead. Take some e with, say, $\Gamma \vdash e : \langle \textbf{int}, \textbf{bool} \rangle$. By (hypothetical) rule SProd and Sub, have $\Gamma \vdash e : \langle \textbf{int}, \textbf{bool}, \textbf{int} \rangle$. So $\Gamma \vdash \#_3 e : \textbf{int}$ is well-typed. But this will crash!

Products: depth

$$\text{PProd}\ \frac{\Gamma \vdash e : \langle \tau_1, \ldots, \tau_n \rangle}{\Gamma \vdash e : \langle \sigma_1, \ldots, \sigma_n \rangle}\ \forall i.\ \tau_i <: \sigma_i$$

Subtyping (IV): propagation through function type

Propagation of subtyping through functions

Subtyping (IV): propagation through function type

Propagation of subtyping through functions

$$\text{PFUN}\ \frac{\Gamma \vdash e : \tau_1 \rightarrow \tau_2}{\Gamma \vdash e : \sigma_1 \rightarrow \sigma_2}\ \sigma_1 <: \tau_1, \tau_2 <: \sigma_2$$

Return position covariant: weaker guarantee on result

Argument position contravariant: stronger constraint on arguments (e.g. longer products),

Subtyping (IV): propagation through function type

Propagation of subtyping through functions

$$\text{PFun}\ \frac{\Gamma \vdash e : \tau_1 \to \tau_2}{\Gamma \vdash e : \sigma_1 \to \sigma_2}\ \sigma_1 <: \tau_1, \tau_2 <: \sigma_2$$

Return position covariant: weaker guarantee on result

Argument position contravariant: stronger constraint on arguments (e.g. longer products),

Example: $f(x) = \textbf{let}\ z = \#_1 x\ \textbf{in}\ \langle \texttt{even}(z), z\rangle\ \textbf{end}.$

Have $\text{PFun}\ \frac{\Gamma \vdash f : \langle \textbf{int} \rangle \to \langle \textbf{bool}, \textbf{int} \rangle}{\Gamma \vdash f : \langle \textbf{int}, \textbf{int} \rangle \to \langle \textbf{bool} \rangle}$.

Rule thus correctly sanctions the application

$\textbf{let}\ arg = \langle 3, 4 \rangle\ \textbf{in let}\ res = f\ arg\ \textbf{in}\ \#_1 res\ \textbf{end end}.$

Subtyping (V): interaction with references

Guess

$$\textsc{PRef}\ \frac{\Gamma \vdash e : \textbf{ref}\ \tau}{\Gamma \vdash e : \textbf{ref}\ \sigma}\ ???$$

Subtyping (V): interaction with references

Guess

$$\text{PREF}\ \frac{\Gamma \vdash e : \textbf{ref}\ \tau}{\Gamma \vdash e : \textbf{ref}\ \sigma}\ ???\ \tau = \sigma\ \text{(invariance)}$$

Reason: read/write yield conflicting conditions

Read motivates

Subtyping (V): interaction with references

Guess

$$\text{PREF}\ \frac{\Gamma \vdash e : \textbf{ref}\ \tau}{\Gamma \vdash e : \textbf{ref}\ \sigma}\ ???\ \tau = \sigma\ \text{(invariance)}$$

Reason: read/write yield conflicting conditions

Read motivates $\frac{\tau <: \sigma}{\textbf{ref}\ \tau <: \textbf{ref}\ \sigma}$: if *e* evaluates to a reference holding τ values, and any (τ-)value we extract from that location (i.e. !*e*) can also be interpreted as a σ-value, we should be allowed to consider *e* as holding σ-values, so that $\vdash\ !e : \sigma$.

Write motivates

Subtyping (V): interaction with references

Guess

$$\text{PREF}\ \frac{\Gamma \vdash e : \textbf{ref}\ \tau}{\Gamma \vdash e : \textbf{ref}\ \sigma}\ ???\ \tau = \sigma\ \text{(invariance)}$$

Reason: read/write yield conflicting conditions

Read motivates $\dfrac{\tau <: \sigma}{\textbf{ref}\ \tau <: \textbf{ref}\ \sigma}$: if e evaluates to a reference holding τ values, and any (τ-)value we extract from that location (i.e. $!e$) can also be interpreted as a σ-value, we should be allowed to consider e as holding σ-values, so that $\vdash\ !e : \sigma$.

Write motivates $\dfrac{\sigma <: \tau}{\textbf{ref}\ \tau <: \textbf{ref}\ \sigma}$: if e evaluates to a reference to which we may write a τ value (i.e. $\Gamma \vdash e : \textbf{ref}\ \tau$), and if any σ-value (say $\Gamma \vdash e' : \sigma$) may be considered a τ-value, then we should be able to assign e' to e, i.e. allow $\Gamma \vdash e{:=}e' : \textbf{unit}$

HW 4: type inference/checking

Differences between FUN and above language:

- functions declared at top-level, annotated with argument and return types
- products start at 0

Challenge:

- subtyping destroys property that an expression has at most one type.
- rule SUB destroys syntax-directedness, and doesn't make the expression any smaller. Can apply SUB at any point.

Task:

- reformulate type system so that it is syntax-directed: modify the rules such that subtyping is integrated differently, **BUT EXACTLY THE SAME JUDGMENTS SHOULD BE DERIVABLE** using least common supertypes ("joins") and greatest common subtype ("meets"). Implement calculation of meets and joins.
- use these to implement syntax-directed inference