Topic 5: Types

COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

Types: potential benefits (I)

For programmers:

- help to eliminate common programming mistakes, particularly mistakes that might lead to runtime errors (or prevent program from being compiled)
- provide abstractions and modularization discipline: can substitute code with alternative implementation without breaking surrounding program context

Example (ML signatures):

```
sig
  type sorted_list
  val sort : int list -> sorted_list
  val lookup : sorted_list -> int -> bool
  val insert: sorted_list -> int -> sorted_list
end
```

Internal definition of sorted_list not revealed to clients, so can replace one implementation by another one!

Similarly for other invariants.

Types: potential benefits (II)

For language designers:

- yields structuring mechanism for programs – thus encodes abstraction principles that motivate development of new language
- basis for studying (interaction between) language features (references, exceptions, IO, other side effects, h-o-functions)
- formal basis for reasoning about program behavior (verification, security analysis, resource usage)

Types: potential benefits (III)

For compiler writers:

- filter out programs that backend should never see (and can't handle)
- provide information that's useful in later phases:
    - is that + a floating point add or an integer add?
    - does value v fit into a single register? (size of data types)
    - how should the stack frame for function f be organized (number and type/size of function arguments and return value)
    - support generation of efficient code:
        - less code needed for handling errors (and handling casting)
        - enables sharing of implementations (source of confusion eliminated by types)
    - postY2k: typed intermediate languages
        - model intermediate representations as full language, with types that communicate structural code properties and analysis results between compiler phases (example: different types for caller/callee registers)
    - "refined" type systems: provide alternative formalism for program analysis and optimization

Type-enforced safety guarantees

- Memory Safety – can’t dereference something not a pointer
- Control-Flow Safety – can’t jump to something not code
- Type Safety – typing predications (“this value will be a string”) come true at run time, so no operator-operand mismatches

All these errors are eliminated during development time, so applications much more robust!

Prevents programmer from writing code that “obviously” can’t be right.

Contrast with C (weakly typed): implicit casting, null pointers, array-out-of-bounds, buffer overruns, security violations

Type systems: limitations

- Can’t eliminate all runtime errors
  - division by zero (input dependence)
  - exception behavior often not modeled/enforced

- static type analyses are typically conservative: will reject some safe programs due to fundamental undecidability of perfectly predicting control flow

- Types typically involve some programmer annotations -
  - burden
  + documentation;
  + burden occurs at compile time, not runtime

- cryptic error messages

but trade-off against debugging/ tracing effort upon hitting segfault

Types: design & implementation tasks

Practical tasks (for compiler writer): develop algorithms for

- type inference: given an expression e, calculate whether there some type T such that e:T holds. If so, return (the best such) type T, or a representation of all such types.
  May need program annotations.
- type checking: given a fully type-annotated program, check that the typing rules are indeed applied correctly

Theoretical tasks (language designer): study

- uniqueness of typings, existence of best types
- decidability and complexity of above tasks / algorithms
- type soundness: give a precise definition of "good behavior (runtime model, error model) and prove that "well-typed programs can't go wrong" (Milner)

- Common formalism: derivation systems (cf formal logic)
  - formal judgments, derivation/typing rules, derivation trees

Defining a Formal Type System

- RE → Lexing
- CFG → Parsers
- Inductive Definitions → Type Systems / logical derivation systems

Components of a type system:

- a notion of *types*
- specification of *syntactic judgment forms* – A judgment is an assertion/claim, may or may not be true.
  - implicitly or explicitly underpinned by an interpretation (“validity”)
  - Typical judgement forms for type systems in PL: $\vdash e:T$, $\Gamma \vdash e:T$
- *inference rules* – tell us how to obtain new judgment instances from previously derived ones
  - should preserve validity so that only “true” judgments can be derived

Inference Rules

An inference rule has a set of **premises** $J_1, \dots, J_n$ and one **conclusion** J, separated by a horizontal line:

$$\frac{J_1 \quad \dots \quad J_n}{J}$$

Read:

- If I can establish the truth of the premises $J_1,\dots,J_n$, I can conclude: J is true.
- To check J, check $J_1,\dots,J_n$.

An inference rule with no premises is called an **Axiom** – J always true

But what IS a type?

Competing views:

1. Types are mostly syntactic entities, with little inherent meaning:
   - the types for this language are A, B, C; here are the typing rules
   - if you can't infer a type for e / check that e:T holds, reject e: untyped programs are not programs
   - intent / design goals of type system (partially) revealed by what you can do with well-typed programs (e.g. compile to efficient code)

But what IS a type?

Competing views:

1. Types are mostly syntactic entities, with little inherent meaning:
   - the types for this language are A, B, C; here are the typing rules
   - if you can't infer a type for e / check that e:T holds, reject e: untyped programs are not programs
   - intent / design goals of type system (partially) revealed by what you can do with well-typed programs (e.g. compile to efficient code)
2. Types have "semantic content", for example by capturing properties an execution may have
   - types as an algorithmic approximation to classify behaviors
   - if you can't derive a judgement e:T using the typing rules, but e still "has" the behavior captured by T, that's fine
   - => types describe properties of a priori untyped programs

But what IS a type?

traditional compiler view

Competing views:

1. Types are mostly syntactic entities, with little inherent meaning:
   - the types for this language are A, B, C; here are the typing rules
   - if you can't infer a type for e / check that e:T holds, reject e: untyped programs are not programs
   - intent / design goals of type system (partially) revealed by what you can do with well-typed programs (e.g. compile to efficient code)
2. Types have "semantic content", for example by capturing properties an execution may have
   - types as an algorithmic approximation to classify behaviors
   - if you can't derive a judgement e:T using the typing rules, but e still "has" the behavior captured by T, that's fine
   - => types describe properties of a priori untyped programs

Many variations possible, depending on goals!

often more modular

Type system for simple expressions

Starting point: abstract syntax

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

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

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

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

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

Type Checking Implementation

```
fun check (e: Expr, t: Type): bool :=
  case t of
    Bool => (case e of … )
  | Int => (case e of … );
```

type in the **host** language (the language the compiler is implemented in)

Type Checking Implementation

```
fun check (e: Expr, t: Type): bool :=
  case t of
    Bool => (case e of .. )



  | Int => (case e of … )
```

expressions/types of the **object** language, ie the language for which we're writing a compiler

Type Checking Implementation

```
fun check (e: Expr, t: Type): bool :=
  case t of
    Bool => (case e of
                         tt => true
                       | ff => true
                       | f e1 e2 => (case f of
                                          AND => check (e1,Bool) andalso check (e2, Bool)
                                        | (*similar case for OR *)
                                        | LESS => check (e1, Int) andalso check(e2, Int)
                                        | (*similar cases for EQ etc*)
                                        | _ => false)
                       | IF e1 THEN e2 ELSE e3 => check (e1, Bool) andalso
                                          check (e2, Bool) andalso (e3, Bool))
  | Int => (case e of … )
```

Alternative: swap nesting of case distinctions for expressions and types.

Type Inference Implementation

```
fun infer (e:Expr): Type option =
 case e of
    tt => Some Bool
  | ff => Some Bool
  | BINOP f e1 e2 => ???
  | …
```

Type Inference Implementation

```
fun infer (e:Expr): Type option =
 case e of
   tt => Some Bool
  | ff => Some Bool
  | BINOP f e1 e2 => (case f of
                    AND => if check(e1, Bool) andalso check(e2, Bool)
                           then Some Bool else None
                  | ...
```

Type Inference Implementation

```
fun infer (e:Expr): Type option =
 case e of
    tt => Some Bool
  | ff => Some Bool
  | BINOP f e1 e2 => (case f of
                        AND => if check(e1, Bool) andalso check(e2, Bool)
                               then Some Bool else None
                      | …
```

alternative that does not use **check**:

```
  case (infer e1, infer e2) of
  (Some bool, Some bool) => Some bool
  | (_, _) => None
```

Type Inference Implementation

```
fun infer (e:Expr): Type option =
 case e of
    tt => Some Bool
  | ff => Some Bool
  | BINOP f e1 e2 => (case f of
                         AND => if check(e1, Bool) andalso check(e2, Bool)
                                then Some Bool else None
                       | PLUS => if check (e1, Int) andalso check(e2, Int)
                                 then Some Int else None
                       | LESS => if check (e1, Int) andalso check (e2, Int)
                                 then Some Bool else None
                       | … )
  | IF e1 THEN e2 ELSE e3 => ???
```

alternative that does not use **check**:

```
  case (infer e1, infer e2) of
  (Some bool, Some bool) => Some bool
  | (_, _) => None
```

Type Inference Implementation

```
fun infer (e:Expr): Type option =
 case e of
    tt => Some Bool
  | ff => Some Bool
  | BINOP f e1 e2 => (case f of
                        AND => if check(e1, Bool) andalso check(e2,Bool)
                               then Some Bool else None
                      | (*similar cases for other binops*) )
  | IF e1 THEN e2 ELSE e3 => if check(e1, Bool)
                             then case (infer e2, infer e3) of
                                    (Some t1, Some t2) =>
                                         if t1=t2 then Some t1 else None
                                  | (_, _) => None
                             else None
  | (*other expressions*)
```

equality between types
(often defined by induction)

Improvement: replace return type "**Type** option" by type that allows informative error messages in case where inference fails.

Type system for simple expressions

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

Exercise

Perform syntax-directed inference for the expressions

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

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

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

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).

aka symbol table

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

Adding 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):

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

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

Adding 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): $\textsc{Var}\ \dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$

side condition

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

Shortcoming?

Adding 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): $\textsc{Var}\ \dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$

side condition

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

Shortcoming? cannot add a binding to variables.

Adding 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:

Adding 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:

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

Adding 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

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

Adding 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.

Adding 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 : $e ::= \ldots \mid e\ e$

Adding 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

Adding 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:

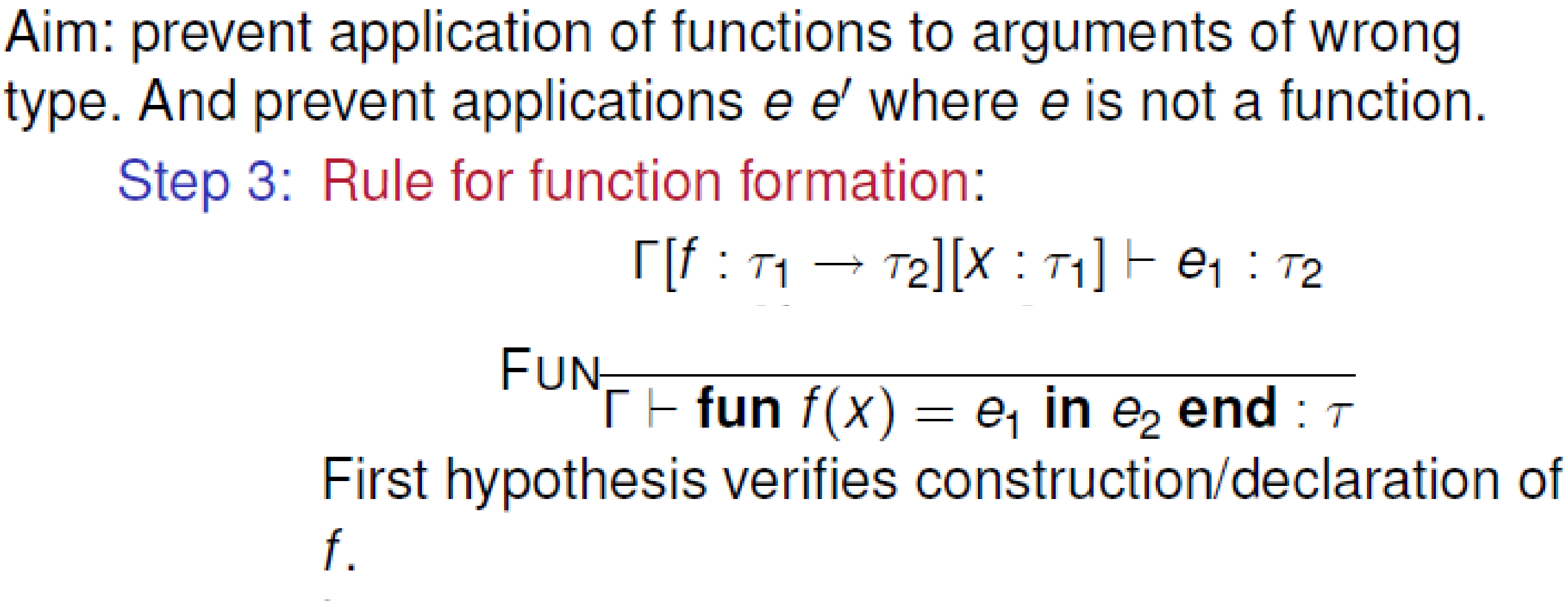
Adding functions (II)

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:

$$\textsc{Fun}\ \frac{???}{\Gamma \vdash \textbf{fun}\ f(x) = e_1\ \textbf{in}\ e_2\ \textbf{end} : \tau}$$

Adding 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:

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

First hypothesis verifies construction/declaration of f.

Adding 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:

Adding 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:

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

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:

$$\textsc{App}\frac{???}{\Gamma \vdash e_1 e_2 : \tau_2}$$

Adding 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:

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

Adding 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:

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

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:

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

Exercise (homework)

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

References (cf. ML-primer)

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/tuples

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):

Products/tuples

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

Products/tuples

Starting point (absyn): two characteristic operations:

Product formation, projections

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

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

Step 2 (judgment form): no change

Step 3 (rules):

$$\textsc{Prod}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle e_1, \dots, e_n \rangle : \langle \tau_1, \dots, \tau_n \rangle}$$

$$\textsc{Proj}\ \frac{\Gamma \vdash e : \langle \tau_1, \dots, \tau_n \rangle}{\Gamma \vdash \#_k e : \tau_k}\ 1 \leq k \leq n$$

Subtyping (I)

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 (I)

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.

Subtyping (I)

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.

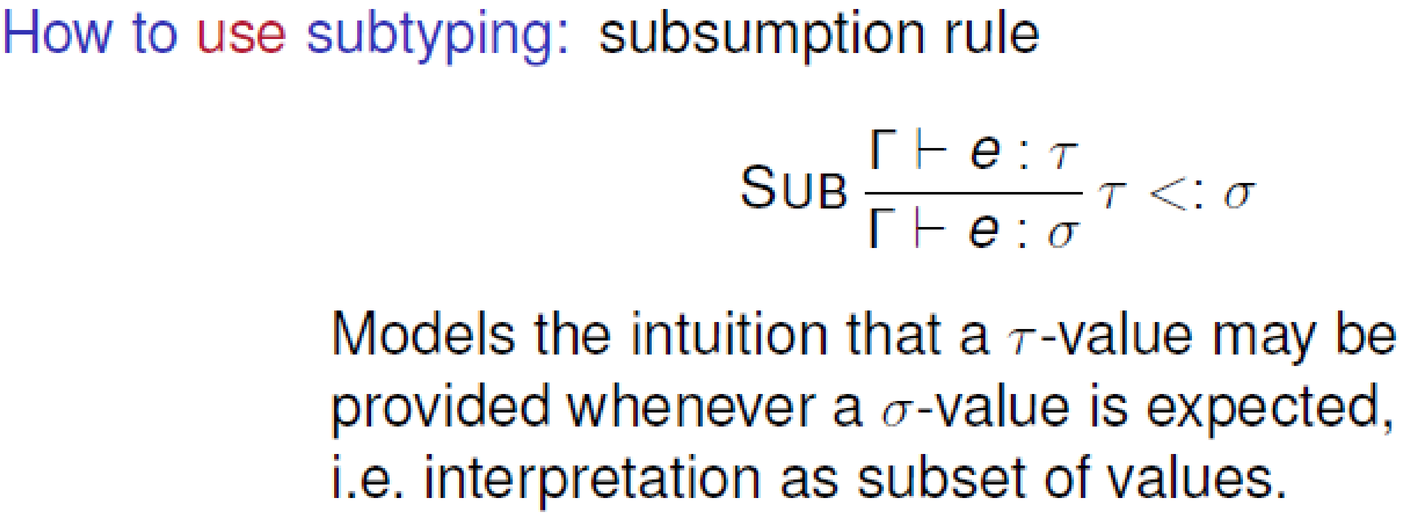
Indeed: any operation we may perform on an expression of the latter type (i.e. a projection $\#_k e$, which is only well-typed if $k \leq m$) is also legal on expressions of the former type.

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

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

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.

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): propagating 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!

Subtyping (III): propagating 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, $\Gamma \vdash e : \langle \mathbf{int}, \mathbf{bool} \rangle$. By (hypothetical) rule SProd and Sub, have $\Gamma \vdash e : \langle \mathbf{int}, \mathbf{bool}, \mathbf{int} \rangle$. So $\Gamma \vdash \#_3 e : \mathbf{int}$ is well-typed. But this will crash!

Products: depth

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

Subtyping (IV): propagating through functions

Propagation of subtyping through functions

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

Return position covariant: weaker guarantee on result

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

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

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

Rule thus correctly sanctions the application

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

Subtyping (V): propagating through references

Guess

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

Subtyping (V): propagating through references

Guess

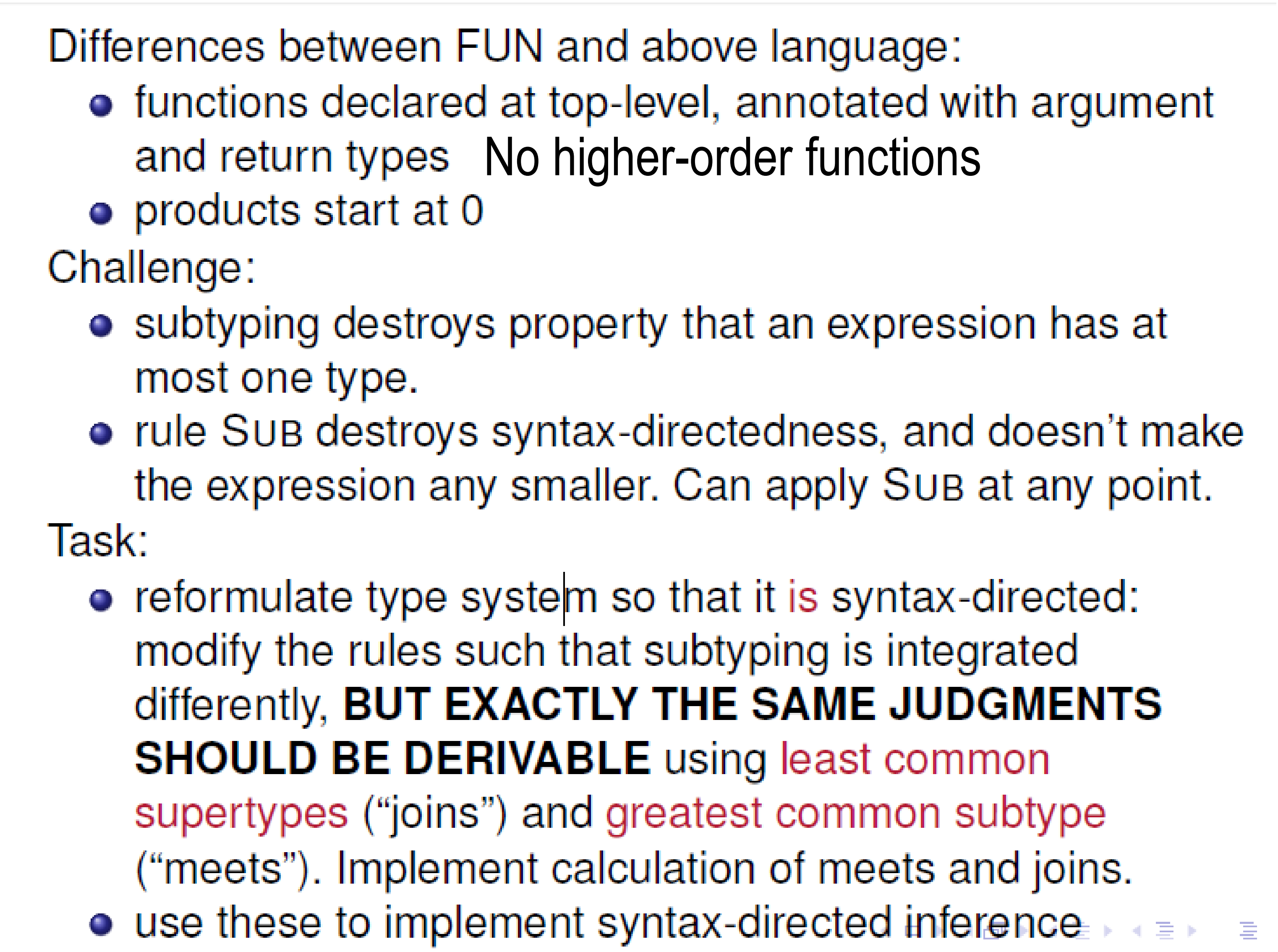
$$\text{PREF}\ \frac{\Gamma \vdash e : \mathbf{ref}\ \tau}{\Gamma \vdash e : \mathbf{ref}\ \sigma}\ ???\ \tau = \sigma\ \text{(invariance)}$$

Reason: read/write yield conflicting conditions

Read motivates $\frac{\tau <: \sigma}{\mathbf{ref}\ \tau <: \mathbf{ref}\ \sigma}$: 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 $\frac{\sigma <: \tau}{\mathbf{ref}\ \tau <: \mathbf{ref}\ \sigma}$: if e evaluates to a reference to which we may write a τ value (i.e. $\Gamma \vdash e : \mathbf{ref}\ \tau$), 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 $\Gamma \vdash e{:=}e' : \mathbf{unit}$

HW4: type analysis



Differences between FUN and above language:

- functions declared at top-level, annotated with argument and return types No higher-order functions
- 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.

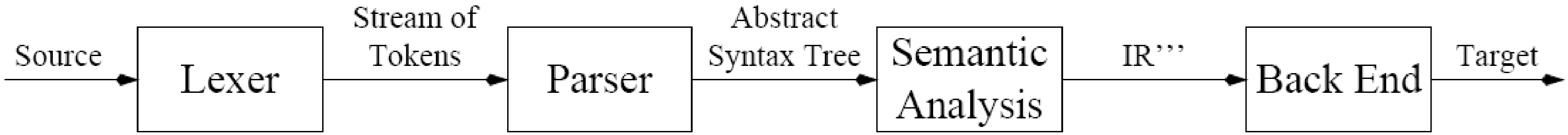
Task:

- 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

Additional aspects

- separate name spaces for type definitions vs variables/ functions/procedures ➔ separate environments (cf **Tiger**)
- when syntax-directedness fails, requiring user-supplied type annotations helps inference
  Example: functions declarations, in particularly (mutually) recursive ones
- not covered:
  - overloading (multiple typings for operators)
    example: arithmetic operations over int, float, double
  - additional syntactic category (eg statements): new judgements
  - Casting/coercion
    - explicit (ie visible in program syntax): similar to other operators
    - implicit: destroys syntax-directness similar to subtyping
    - often symbolizes/triggers change of representation (int -> double) that's significant for compiler backend
  - polymorphism (finite representations for infinitely many typings)
  - arrays, class/object systems incl inheritance, signatures/modules/interfaces

Next steps



- IR code generation
- But first: need to learn a bit how data will be laid out in memory: **activation records** / frame stack

Useful homework: read MCIL's sections on TIGER's semantic analysis (Chapter 5) and, if possible, TIGER's activation record layout (Chapters 6)!

Quiz 3: LR(0)

How many states does the LR(0) parse table for the following grammar have? (You may guess or draw the parse table ;-))

S → B $	P → ε	E → B
B → id P	P → (E)	E → B, E
B → id (E]		

(cf MCIL, page 85, exercise 3.11)