CS153: Compilers
Lecture 14: Type Checking

Stephen Chong

https://www.seas.harvard.edu/courses/cs153
Contains content from lecture notes by Steve Zdancewic and Greg Morrisett
Announcements

- HW4 Oat v1 out
  - Due Tuesday Oct 29 (12 days)
Today

• Type checking
• Judgments and inference rules
Basic Architecture

Source Code

Parsing

Elaboration

Lowering

Optimization

Code Generation

Target Code

Front end

Back end
Elaboration

Untyped Abstract Syntax Trees

Typed Abstract Syntax Trees
Undefined Programs

- After parsing, we have AST
- We can interpret AST, or compile it and execute
- But: not all programs are well defined
  - E.g., $3/0$, "hello" - $7$, $42(19)$, using a variable that isn’t in scope, ...

**Types** allow us to rule out many of these undefined behaviors
- Types can be thought of as an approximation of a computation
- E.g., if expression $e$ has type `int`, then it means that $e$ will evaluate to some integer value
- E.g., we can ensure we never treat an integer value as if it were a function
Type Soundness

• Key idea: a well-typed program when executed does not attempt any undefined operation

• Make a model of the source language
  • i.e., an interpreter, or other semantics
  • This tells us which operations are partial
  • Partiality is different for different languages
    • E.g., “Hi” + “ world” and “na”*16 may be meaningful in some languages

• Construct a function to check types: \( tc : \text{AST} \rightarrow \text{bool} \)
  • AST includes types (or type annotations)
  • If \( tc \ e \) returns true, then interpreting \( e \) will not result in an undefined operation

• Prove that \( tc \) is correct
Simple Language

type tipe =
  Int_t
| Arrow_t of tipe*tipe
| Pair_t of tipe*tipe

Note: function arguments have type annotation

type exp =
  Var of var | Int of int
| Plus_i of exp*exp
| Lambda of var * tipe * exp
| App of exp*exp
| Pair of exp * exp
| Fst of exp | Snd of exp
let rec interp (env:var->value)(e:exp) = 
  match e with
  | Var x -> env x
  | Int i -> Int_v i
  | Plus_i(e1,e2) ->
    (match interp env e1, interp env e2 of
     | Int_v i, Int_v j -> Int_v(i+j)
     | _,_ -> failwith "Bad operands!"
    )
  | Lambda(x,t,e) -> Closure_v{env=env,code=(x,e)}
  | App(e1,e2) ->
    (match (interp env e1, interp env e2) with
     | Closure_v{env=cenv,code=(x,e)},v ->
       interp (extend cenv x v) e
     | _,_ -> failwith "Bad operands!"
    )
let rec tc (env:var->tipe) (e:exp) =
  match e with
  | Var x -> env x
  | Int _ -> Int_t
  | Plus_i(e1,e2) ->
    (match tc env e1, tc env e with
    | Int_t, Int_t -> Int_t
    | _,_ -> failwith "..."
    )
  | Lambda(x,t,e) -> Arrow_t(t,tc (extend env x t) e)
  | App(e1,e2) ->
    (match (tc env e1, tc env e2) with
    | Arrow_t(t1,t2), t ->
      if (t1 != t) then failwith "..." else t2
    | _,_ -> failwith "..."
    )
Notes

• Type checker is almost like an approximation of the interpreter!
  • But interpreter evaluates function body only when function applied
  • Type checker always checks body of function

• We needed to assume the input of a function had some type $t_1$, and reflect this in type of function ($t_1 \rightarrow t_2$)

• At call site ($e_1 \ e_2$), we don’t know what closure $e_1$ will evaluate to, but can calculate type of $e_1$ and check that $e_2$ has type of argument
• Adding booleans...

```plaintext
type tipe = ... | Bool_t

type exp = ... | True | False | If of exp*exp*exp

let rec interp env e = ...
| True -> True_v
| False -> False_v
| If(e1,e2,e3) -> (match interp env e1 with
    True_v -> interp env e2
    | False_v -> interp env e3
    | _ -> failwith "...")
```
let rec tc (env:var->tipe) (e:exp) =
  match e with
...
  | True  -> Bool_t
  | False -> Bool_t
  | If(e1,e2,e3) ->
    (let (t1,t2,t3) = (tc env e1,tc env e2,tc env e3)
     in
      match t1 with
      | Bool_t  ->
        if (t2 != t3) then error() else t2
      | _       -> failwith "..."
Type Inference

- Type checking is great if we already have enough type annotations
  - For our simple functional language, sufficient to have type annotations for function arguments
- But what about if we tried to infer types?
  - Reduce programmer burden!
- Efficient algorithms to do this: Hindley-Milner
  - Essentially build constraints based on how expressions are used and try to solve constraints
  - Error messages for non-well-typed programs can be challenging!
Polymorphism and Type Inference

- **Polymorphism** is the ability of code to be used on values of different types.
  - E.g., polymorphic function can be invoked with arguments of different types
  - Polymorph means “many forms”

- OCaml has polymorphic types
  - E.g., `val swap : 'a ref -> 'a -> 'a = ...`

- But type inference for full polymorphic types is undecidable...

- OCaml has restricted form of polymorphism that allows type inference: **let-polymorphism** aka prenex polymorphism
  - Allow let expressions to be typed polymorphically, i.e., used at many types
  - Doesn’t require copying of let expressions
  - Requires clear distinction between polymorphic types and non-polymorphic types...
Type Safety

• “Well typed programs do not go wrong.”
  – Robin Milner, 1978

• Note: this is a very strong property.
  • Well-typed programs cannot “go wrong” by trying to execute undefined code (such as 3 + (fun x -> 2))
  • Simply-typed lambda calculus is guaranteed to terminate! (i.e. it isn't Turing complete)

• Depending on language, will not rule out all possible undefined behavior
  • E.g., 3/0, *NULL, ...
  • More sophisticated type systems can rule out more kinds of possible runtime errors
Judgements and Inference Rules

- We saw type checking algorithm in code
- Can express type-checking rules compactly and clearly using a type judgment and inference rules
Type Judgments

- In the judgment: \( E \vdash e : t \)
  - \( E \) is a typing environment or a type context
  - \( E \) maps variables to types. It is just a set of bindings of the form: \( x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n \)
- If \( E \vdash e : t \) then expression \( e \) has type \( t \) under typing environment \( E \)
  - \( E \vdash e : t \) can be thought of as a set or relation
- For example:
  \[ \begin{align*} &x : \text{int}, b : \text{bool} \vdash \text{if (b) 3 else x} : \text{int} \end{align*} \]
- What do we need to know to decide whether “if (b) 3 else x” has type int in the environment \( x : \text{int}, b : \text{bool} \)?
  - \( b \) must be a bool i.e. \( x : \text{int}, b : \text{bool} \vdash b : \text{bool} \)
  - \( 3 \) must be an int i.e. \( x : \text{int}, b : \text{bool} \vdash 3 : \text{int} \)
  - \( x \) must be an int i.e. \( x : \text{int}, b : \text{bool} \vdash x : \text{int} \)
Recall Inference Rules

• Inference rule
  • If the premises are true, then the conclusion is true
  • An **axiom** is a rule with no premises
  • Inference rules can be **instantiated** by replacing **metavariables** (e, e1, e2, x, i, ...) with expressions, program variables, integers, as appropriate.
Why Inference Rules?

• Compact, precise way of specifying language properties.
  • E.g. ~20 pages for full Java vs. 100’s of pages of prose Java Language Spec.

• Inference rules correspond closely to the recursive AST traversal that implements them

• Type checking (and type inference) is nothing more than attempting to prove a different judgment \( E \vdash e : t \) by searching backwards through the rules.

• Compiling in a context is nothing more than a collection of inference rules specifying yet a different judgment \( E \vdash \text{src} \Rightarrow \text{target} \)
  • Moreover, the compilation rules are very similar in structure to the typechecking rules

• Strong mathematical foundations
  • The “Curry-Howard correspondence”: Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  • See CS152 if you’re interested in type systems!
Simply-typed Lambda Calculus

- **INT**
  - \( E \vdash i : \text{int} \)

- **VAR**
  - \( x : T \in E \)
  - \( E \vdash x : T \)

- **ADD**
  - \( E \vdash e_1 : \text{int} \quad E \vdash e_2 : \text{int} \)
  - \( E \vdash e_1 + e_2 : \text{int} \)

- **FUN**
  - \( E, x : T \vdash e : S \)
  - \( E \vdash \text{fun } (x:T) \rightarrow e : T \rightarrow S \)

- **APP**
  - \( E \vdash e_1 : T \rightarrow S \quad E \vdash e_2 : T \)
  - \( E \vdash e_1 \ e_2 : S \)

• Note how these rules correspond to the code.
Type Checking Derivations

• A derivation or proof tree is a tree where nodes are instantiations of inference rules and edges connect a premise to a conclusion.
• Leaves of the tree are axioms (i.e. rules with no premises).
• Goal of the typechecker: verify that such a tree exists.
• Example: Find a tree for the following program using the inference rules on the previous slide:

\[ \vdash (\text{fun} \ (x:\text{int}) \ -> \ x + 3) \ 5 \ : \ \text{int} \]
Example Derivation Tree

\[
\begin{array}{c}
\text{VAR} \quad \text{ADD} \quad \text{FUN} \quad \text{APP} \\
\hline
\text{\(x: \text{int} \in x: \text{int}\)} & \text{\(\text{INT} \quad x: \text{int} \vdash 3: \text{int}\)} \\
\hline
\text{\(x: \text{int} \vdash x: \text{int}\)} & \text{\(x: \text{int} \vdash 3: \text{int}\)} \\
\hline
\text{\(x: \text{int} \vdash x + 3: \text{int}\)} & \text{\(\text{INT} \quad \vdash 5: \text{int}\)} \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{\(\vdash (\text{fun (x:int) -> x + 3}): \text{int} \rightarrow \text{int}\)} \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{c}
\vdash (\text{fun (x:int) -> x + 3}) 5: \text{int} \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{INT} \quad \text{VAR} \quad \text{ADD} \\
\hline
\text{\(\vdash i: \text{int}\)} & \text{\(\vdash x: T\)} \\
\hline
\text{\(\vdash x: T \in E\)} & \text{\(\vdash E, x: T \vdash e: S\)} \\
\hline
\text{\(\vdash E \vdash \text{fun (x:T) -> e}: T \rightarrow S\)} \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{ADD} \quad \text{APP} \\
\hline
\text{\(\vdash e_1: \text{int}\)} & \text{\(\vdash e_2: \text{int}\)} \\
\hline
\text{\(\vdash E \vdash e_1 + e_2: \text{int}\)} & \text{\(\vdash E \vdash e_1 e_2: S\)} \\
\hline
\text{\(\vdash E \vdash e_1: T \rightarrow S\)} & \text{\(\vdash E \vdash e_2: T\)} \\
\hline
\hline
\end{array}
\]
• Note: the OCaml function typecheck verifies the existence of this tree. The structure of the recursive calls when running `tc` is same shape as this tree!
• Note that $x : \text{int} \in E$ is implemented by the function `lookup`
Type Safety Revisited

**Theorem:** (simply typed lambda calculus with integers)

If \( \vdash e : t \) then there exists a value \( v \) such that \( e \downarrow v \).
Arrays

• Array constructs are not hard
• First: add a new type constructor: $T[]$

\[
\text{NEW} \quad \begin{array}{c}
E \vdash e_1 : \text{int} \\
E \vdash e_2 : T
\end{array}
\quad \frac{\begin{array}{c}
E \vdash \text{new } T[e_1](e_2) : T[]
\end{array}}{E \vdash e_1[e_2] : T}
\]

\[
\text{INDEX} \quad \begin{array}{c}
E \vdash e_1 : T[] \\
E \vdash e_2 : \text{int}
\end{array}
\quad \frac{\begin{array}{c}
E \vdash e_1[e_2] : T
\end{array}}{E \vdash e_1[e_2] = e_3 \text{ ok}}
\]

Note: These rules don’t ensure that the array index is in bounds – that should be checked \textit{dynamically}.

$e_1$ is the size of the newly allocated array. $e_2$ initializes the elements of the array.
Tuples

- ML-style tuples with statically known number of products
- First: add a new type constructor: $T_1 \times \ldots \times T_n$

**TUPLE**

$$E \vdash e_1 : T_1 \quad \ldots \quad E \vdash e_n : T_n$$

$$E \vdash (e_1, \ldots, e_n) : T_1 \times \ldots \times T_n$$

**PROJ**

$$E \vdash e : T_1 \times \ldots \times T_n \quad 1 \leq i \leq n$$

$$E \vdash \#i e : T_i$$
• **ML-style references** (note that ML uses only expressions)
• First, add a new type constructor: \( T \text{ ref} \)

```
\[ E \vdash e : T \]
\[ E \vdash \text{ref} \ e : T \text{ ref} \]
```

```
\[ E \vdash e : T \text{ ref} \]
\[ E \vdash !e : T \]
```

```
\[ E \vdash e_1 : T \text{ ref} \quad E \vdash e_2 : T \]
\[ E \vdash e_1 := e_2 : \text{unit} \]
```

Note the similarity with the rules for arrays…
Oat Type Checking

• For HW5 we will add typechecking to Oat
  • And some other features

• Some of Oat’s features
  • Imperative (update variables, like references)
  • Distinction between statements and expressions
  • More complicated control flow
    • Return
    • While, For, ...

• What does a type system look like for Oat?
Some Oat Judgments

• Split environment E into Globals and Locals

• Expression e has type t under context G;L
  • G; L ⊢ e : t

• Statement s is well typed under context G;L. If it returns, it returns a value of type rt. After s, the local context is L’.
  • G; L; rt ⊢ s ⇒ L’

• Where does G come from?

• Program is a list of global variable declarations and function declarations

• Use judgment to gather up global variable declarations
  • ⊢g prog ⇒ G
Example Derivation

```javascript
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```
Example Derivation

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

\[
D_1 = \frac{G_0; \vdash 0 : \text{int}}{\text{[INT]}}
\]
\[
\frac{G_0; \vdash 0 : \text{int}}{\text{[CONST]}}
\]
\[
\frac{G_0; \vdash \text{var } x_1 = 0 \Rightarrow \cdot, x_1 : \text{int}}{\text{[DECL]}}
\]
\[
\frac{G_0; \cdot; \text{int} \vdash \text{var } x_1 = 0 \Rightarrow \cdot, x_1 : \text{int}}{\text{[SDECL]}}
\]

\[
D_2 = \frac{\vdash \cdot : (\text{int}, \text{int}) \rightarrow \text{int}}{\text{[ADD]}}
\]
\[
\frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}}{\text{[VAR]}}
\]
\[
\frac{x_1 : \text{int} \in \cdot, x_1 : \text{int}}{\text{[BOP]}}
\]
\[
\frac{G_0; \cdot, x_1 : \text{int} \vdash x_1 : \text{int}}{\text{[VAR]}}
\]
\[
\frac{G_0; \cdot, x_1 : \text{int} \vdash x_1 : \text{int}}{\text{[BOP]}}
\]
\[
\frac{G_0; \cdot, x_1 : \text{int} \vdash x_1 + x_1 : \text{int}}{\text{[DECL]}}
\]
\[
\frac{G_0; \cdot, x_1 : \text{int}; \text{int} \vdash \text{var } x_2 = x_1 + x_1 \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}}{\text{[SDECL]}}
\]
Example Derivation

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

\[
D_3 = \quad \frac{\vdash -: (\text{int}, \text{int}) \rightarrow \text{int} \quad [\text{ADD}] \quad x_1: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int} \quad [\text{VAR}] \quad x_2: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int}}{G_0; \cdot, x_1: \text{int}, x_2: \text{int} \vdash x_1: \text{int} \quad [\text{VAR}] \quad G_0; \cdot, x_1: \text{int}, x_2: \text{int} \vdash x_2: \text{int} \quad [\text{BOP}]} \quad G_0; \cdot, x_1: \text{int}, x_2: \text{int} \vdash x_1 - x_2: \text{int} \quad [\text{ASSN}]
\]

\[
D_4 = \quad \frac{x_1: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int} \quad [\text{VAR}] \quad G_0; \cdot, x_1: \text{int}, x_2: \text{int} \vdash x_1: \text{int} \quad [\text{VAR}]}{G_0; \cdot, x_1: \text{int}, x_2: \text{int} \vdash \text{return} x_1; \Rightarrow \cdot, x_1: \text{int}, x_2: \text{int} \quad [\text{RET}]}
\]

Stephen Chong, Harvard University
Type Safety For General Languages

Theorem: (Type Safety)

If P is a well-typed program, then either:

(a) the program terminates in a well-defined way, or
(b) the program continues computing forever

• Well-defined termination could include:
  • halting with a return value
  • raising an exception

• Type safety rules out undefined behaviors:
  • abusing “unsafe” casts: converting pointers to integers, etc.
  • treating non-code values as code (and vice-versa)
  • breaking the type abstractions of the language

• What is “defined” depends on the language semantics…
Consider the source typing judgment for source expressions:

\[ C \vdash e : t \]

How do we interpret this information in the target language?

\[ \llbracket C \vdash e : t \rrbracket = ? \]

- \[ \llbracket C \rrbracket \] translates contexts
- \[ \llbracket t \rrbracket \] is a target type
- \[ \llbracket e \rrbracket \] translates to a (potentially empty) stream of instructions, that, when run, computes the result into some operand

**INVARIANT:** if \[ \llbracket C \vdash e : t \rrbracket = ty, \text{ operand, stream} \]
then the type (at the target level) of the operand is \( ty = \llbracket t \rrbracket \)
Example

• \( C \vdash 37 + 5 : \text{int} \)
• What is \( \llbracket C \vdash 37 + 5 : \text{int} \rrbracket \) ?

\[
\llbracket C \vdash 37 : \text{int} \rrbracket = (\text{i64}, \text{Const 37}, [])
\llbracket C \vdash 5 : \text{int} \rrbracket = (\text{i64}, \text{Const 5}, [])
\llbracket C \vdash 37 + 5 : \text{int} \rrbracket = (\text{i64}, \%\text{tmp}, [\%\text{tmp} = \text{add i64} (\text{Const 37}) (\text{Const 5})])
\]
What about the Context?

- What is $\llbracket C \rrbracket$?
- Source level C has bindings like: $x$:int, $y$:bool
  - We think of it as a finite map from identifiers to types
- What is the interpretation of $C$ at the target level?
- $\llbracket C \rrbracket$ maps source identifiers, “x”, to target types and $\llbracket x \rrbracket$
- What is the interpretation of a variable $\llbracket x \rrbracket$ at the target level?
  - How are the variables used in the type system?

\[
\begin{align*}
  x : t & \in L \\
  G; L \vdash x : t & \quad \text{TYP_VAR} \\
  \text{as expressions} & \quad \text{(which denote values)}
\end{align*}
\]

\[
\begin{align*}
  x : t & \in L \\
  G; L \vdash exp : t & \quad \text{TYP_ASSN} \\
  G; L; rt \vdash x = exp; \Rightarrow L & \quad \text{as addresses} \\
  & \quad \text{(which can be assigned)}
\end{align*}
\]
Interpretation of Contexts

• $[C] = \text{a map from source identifiers to types and target identifiers}$

• INVARIANT:
  \[ x: t \in C \] means that

  (1) \[ \text{lookup } [C] x = ([t]^*, \%id_x) \]

  (2) the (target) type of $\%id_x$ is $[t]^*$ (a pointer to $[t]$)
Interpretation of Variables

• Establish invariant for expressions:

\[
\frac{x : t \in L}{G; L \vdash x : t} \text{ TYP\_VAR}
\]

as expressions

(which denote values)

\[
\begin{align*}
\frac{x : t \in L}{G; L \vdash x : t} \text{ TYP\_VAR} \quad &\quad = (\%\text{tmp}, \ \%\text{tmp} = \text{load i64}\* \%\text{id}_x) \\
&\quad \text{where (i64, %id_x) = lookup [L] x}
\end{align*}
\]

• What about statements?

\[
\frac{x : t \in L \quad G; L \vdash exp : t}{G; L; rt \vdash x = exp; \Rightarrow L} \text{ TYP\_ASSN}
\]

as addresses

(which can be assigned)

\[
\begin{align*}
\frac{x : t \in L \quad G; L \vdash exp : t}{G; L; rt \vdash x = exp; \Rightarrow L} \text{ TYP\_ASSN} \quad &\quad = \text{stream @} [\text{store [t] opn, [t]* %id_x]} \\
&\quad \text{where ([t], %id_x) = lookup [L] x} \\
&\quad \text{and [G;L \vdash exp : t] = ([t], opn, stream)}
\end{align*}
\]