Lecture 16

EECS 483: COMPILER CONSTRUCTION

Announcements

- Midterm
  - Grades will be released after we review the results on Wednesday's class (3/20).
- HW4: OAT v.1.0
  - Parsing & translation to LLVM IR
  - Helps to start early!
  - **Due: Tuesday, March 26th**

Personal Announcement

- My wife and I are having a baby
  - Due date: March 30, but could be any day now
  - We will have guest lectures (by our GSI Eric or a guest professor) for at least 4 lectures after the baby comes.
  - I will have my regularly scheduled office hours remote only, will not be available for scheduled office hours.

Implementing First-Class Functions

- First attempt: Functions as Code
  - Represent a function value as its code
  - What about *local function definitions?*
    - `let f = fun x -> fun y -> x + y`
  - Every time we call `f 0, f 5, f 256`, we get a *different* function
  - In an substitution-based interpreter, we substitute a value in and get a different term each time we call the function.
  - Compilation
    - Infeasible to implement all of these possible functions statically in memory, impossible if the domain of the function is infinite!
    - Requires *runtime code generation*, which itself has high runtime overhead.
    - Usually not used for arbitrary first class functions, only in specialized situations

Implementing First-Class Functions

- Closures
  - Consider
    - `let f = fun x -> fun y -> x + y`
  - Each function `f 0, f 5, f 256,`… is implemented by substitution:
    - `(fun y -> x + y){ 0 / x }`
    - `(fun y -> x + y){ 5 / x }`
    - `(fun y -> 256 + y){ 256 / x }`
  - Idea: represent a first-class function as a ***pair*** of
    - A piece of code with **free** variables
    - An **environment** that provides all of the values of the free variables
  - In compilation
    - The code with free variables can be a code pointer to code that takes the environment as an argument
      - `fun (env, y) -> env.x + y`
    - The environment can be implemented in multiple ways
      - Array: fast access in the function
      - Linked list: more sharing between different closures

See fun.ml, cc.ml from lec15.zip

CLOSURES AND CLOSURE CONVERSION

Closure Conversion Summary

- A ***closure*** is a pair of an environment and a code pointer
  - the environment is a map data structure binding variables to values
  - environment could just be a list of the values (with known indices)
- Building a closure value:
  - code pointer is a function that takes an extra argument for the environment: $A \rightarrow B$ becomes $(Env * A \rightarrow B)$
  - body of the closure "projects out" then variables from the environment
  - creates the environment map by bundling the free variables
- Applying a closure:
  - project out the environment, invoke the function (pointer) with the environment and its "real" argument
- Hoisting:
  - Once closure converted, all functions can be lifted to the top level

Scope, Types, and Context

SEMANTIC ANALYSIS

Compilation in a Nutshell

Source Code
(Character stream)

```
if (b == 0) { a = 1; }
```

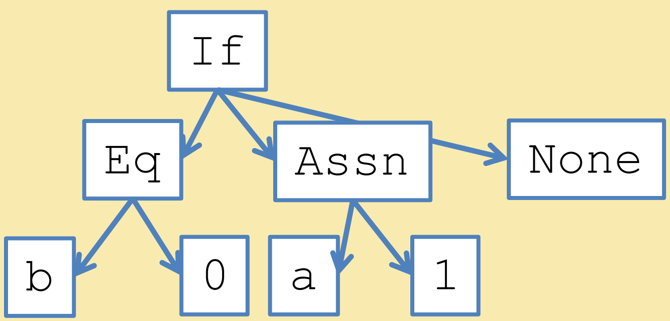
Lexical Analysis

Token stream:

if	(	b	==	0	)	{	a	=	0	;	}

Parsing

Abstract Syntax Tree:



Analysis & Transformation

Intermediate code:

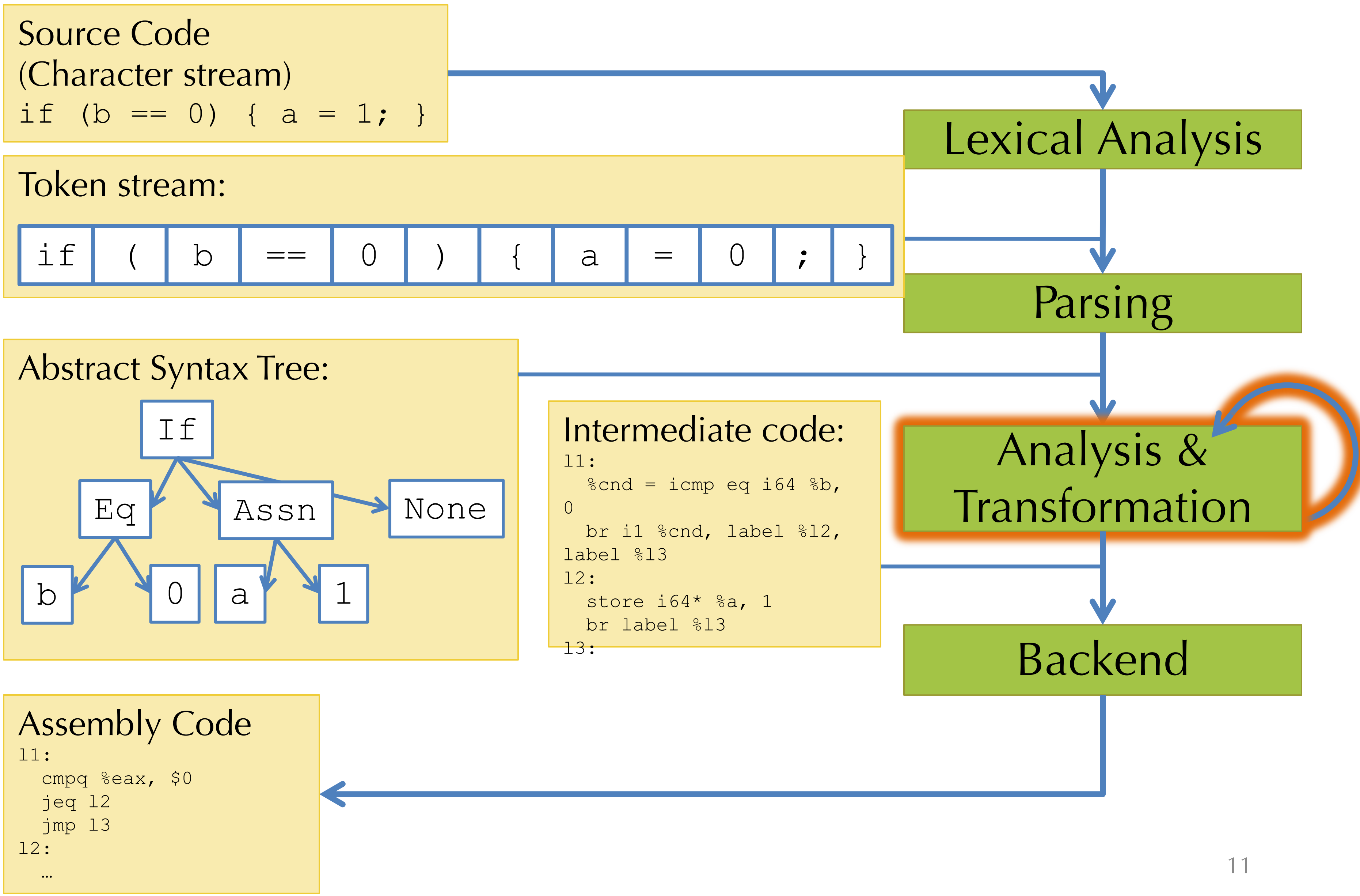
```
l1:
  %cnd = icmp eq i64 %b, 0
  br i1 %cnd, label %l2, label %l3
l2:
  store i64* %a, 1
  br label %l3
l3:
```

Backend

Assembly Code

```
l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2:
  …
```

Most of the Remainder of the Course



Variable Scoping

- Consider the problem of determining whether a programmer-declared variable is in scope.
- Issues:
  - Which variables are available at a given point in the program?
  - Shadowing – is it permissible to re-use the same identifier, or is it an error?

- Example: The following program is syntactically correct but not well-formed. (y and q are used without being defined anywhere)

```
int fact(int x) {
  var acc = 1;
  while (x > 0) {
    acc = acc * y;
    x = q - 1;
  }
  return acc;
}
```

Q: Can we solve this problem by changing the parser to rule out such programs?

Static Program Analysis

- Static program analysis is analysis of a program at compile-time
- Used for two main purposes in the compiler:
  - Last stage of the frontend: “Type checking” or “Semantic Analysis”
    - Not every program that passes parsing is valid
      - `int main() { return x; }`
      - `int main() { return “hello world”; }`
    - If the type checker fails, the program is rejected, like a parse error
    - After the program passes the frontend, we consider it well-formed and will compile it.
  - During optimization: “static analysis”
    - We can do more optimizations if we know more about the program
    - Are these equivalent programs?
      - `int main() { int y = f(); return 0; }`
      - `int main() { return 0; }`
    - We can optimize the first to the second if we establish that f is side-effect free.
    - Since they take place after the frontend, the analysis never rejects the program
- Next few weeks: type checking, after that optimization and analyis

Type Checking as Grammar

	Specification	Implementation
Lexing	Regular Expressions	DFA
Parsing	CFG LL(1) grammars LR(1) grammars	Pushdown automata Recursive descent Shift/reduce parser
Type checking	Inference rules	Manual recursive descent

Inference Rules

- We can read a judgment $G \vdash e$ as
  "the expression e is well scoped and has free variables in G"
- For any environment G, expression e, and statements s_1, s_2.

$$G \vdash \text{if } (e)\ s_1 \text{ else } s_2$$

  holds if $G \vdash e$ and $G \vdash s_1$ and $G \vdash s_2$ all hold.
- More succinctly: we summarize these constraints as an ***inference rule***:

Premises

$$\frac{G \vdash e \qquad G \vdash s_1 \qquad G \vdash s_2}{G \vdash \text{if } (e)\ s_1 \text{ else } s_2}$$

Conclusion

- Such a rule can be used for *any* substitution of the syntactic metavariables G, e, s_1 and s_2.

Judgments

- A ***judgment*** is a (meta-syntactic) notation that *names* a relation among one or more sets.
  - The sets are usually built from object-language syntax elements and other "math" sets (*e.g.*, integers, natural numbers, *etc.*)
  - We usually describe them using metavariables that range over the sets.
  - Often use domain-specific notation to ease reading.
  - The meaning of judgments, *i.e.*, which sets they represent, is defined by (collections of) inference rules

- Example: When we say "$G \vdash e$ is a judgment where G is a context of variables and e is a term, defined by these […] inference rules" that is shorthand for this "math speak":

  - Let Var be the set of all (syntactic) variables
  - Let Exp be the set $\{e \mid e$ is a term of the untyped lambda calculus$\}$
  - Let $\mathcal{P}$(Var) be the (finite) powerset of variables (set of all finite sets)
  - Define *well-scoped* $\subseteq$ ($\mathcal{P}$(Var), Exp) to be a relation satisfying the properties defined by the associated inference rules […]
  - Then "$G \vdash e$" is notation that means that (G, e) $\in$ *well-scoped*

Scope-Checking Lambda Calculus

- Consider how to identify "well-scoped" lambda calculus terms
  - Given: G, a *set* of variable identifiers, e, a term of the lambda calculus
  - *Judgment*: $G \vdash e$ "the free variables of e are included in G"

$$\frac{x \in G}{G \vdash x}$$

"the variable x is free, but in scope"

$$\frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1\ e_2}$$

"G contains the free variables of e_1 and e_2"

$$\frac{G \cup \{x\} \vdash e}{G \vdash \text{fun } x \rightarrow e}$$

"x is available in the function body e"

Scope-checking Code

- Compare the OCaml code to the inference rules:
  - structural recursion over syntax
  - the check either "succeeds" or "fails"

```
let rec scope_check (g:VarSet.t) (e:exp) : unit =
  begin match e with
  | Var x -> if VarSet.member x g then () else failwith (x ^ "not in scope")
  | App(e1, e2) -> ignore (scope_check g e1); scope_check g e2
  | Fun(x, e)   -> scope_check (VarSet.union g (VarSet.singleton x)) e
  end
```

$$\frac{x \in G}{G \vdash x}\ \boxed{\text{VAR}} \qquad \frac{G \vdash e_1 \quad G \vdash e_2}{G \vdash e_1\ e_2}\ \boxed{\text{APP}} \qquad \frac{G \cup \{x\} \vdash e}{G \vdash \text{fun } x \rightarrow e}\ \boxed{\text{FUN}}$$

- The inference rules are a *specification* of the intended behavior of this scope checking code.
  - they don't specify the order in which the premises are checked

Example Derivation Tree

$$
\text{APP}\;\dfrac{
  \text{FUN}\;\dfrac{
    \text{FUN}\;\dfrac{
      \text{APP}\;\dfrac{
        \text{VAR}\;\dfrac{x \in \{x,y\}}{\{x, y\} \vdash x}
        \qquad
        \text{VAR}\;\dfrac{y \in \{x,y\}}{\{x, y\} \vdash y}
      }{\{x, y\} \vdash x\ y}
    }{\{x\} \vdash \text{fun } y \text{ -> } x\ y}
  }{\{\} \vdash (\text{fun } x \text{ -> fun } y \text{ -> } x\ y)}
  \qquad
  \text{FUN}\;\dfrac{
    \text{VAR}\;\dfrac{z \in \{z\}}{\{z\} \vdash z}
  }{\vdash (\text{fun } z \text{ -> } z)}
}{\{\} \vdash (\text{fun } x \text{ -> fun } y \text{ -> } x\ y)\ (\text{fun } z \text{ -> } z)}
$$

- Note: the OCaml function **scope_check** verifies the existence of this tree. The structure of the recursive calls when running **scope_check** is the same shape as this tree!
- Note that $x \in E$ is implemented by the function **VarSet.mem**

Example Failed Derivation

VAR $\quad z \in \{x,y\}$ / $\{x, y\} \vdash z$ $\qquad$ VAR $\quad y \in \{x,y\}$ / $\{x, y\} \vdash y$

APP $\quad \{x, y\} \vdash z\ y$

FUN $\quad \{x\} \vdash \text{fun } y \text{ -> } z\ y$

FUN $\quad \{\} \vdash (\text{fun } x \text{ -> fun } y \text{ -> } z\ y)$ $\qquad$ VAR $\quad z \in \{z\}$ / $\{z\} \vdash z$ / FUN $\quad \vdash (\text{fun } z \text{ -> } z)$

APP $\quad \{\} \vdash (\text{fun } x \text{ -> fun } y \text{ -> } z\ y)\ (\text{fun } z \text{ -> } z)$

- This program is *not* well scoped
  - The variable z is not bound in the body of the left function.
  - The typing derivation fails because the VAR rule cannot succeed
  - (The other parts of the derivation are OK, though!)

Uses of the inference rules

- We can do proofs by induction on the structure of the derivation.
- For example:

Lemma: If $G \vdash e$ then $fv(e) \subseteq G$.

Proof.

By induction on the derivation that $G \vdash e$.

$$\frac{x \in G}{G \vdash x}$$

- case: VAR then we have $e = x$ (for some variable x) and $x \in G$. But $fv(e) = fv(x) = \{x\}$, but then $\{x\} \subseteq G$.

$$\frac{G \vdash e_1 \quad G \vdash e_2}{G \vdash e_1\ e_2}$$

- case: APP then we have $e = e_1\ e_2$ (for some $e_1\ e_2$) and, by induction, we have $fv(e_1) \subseteq G$ and $fv(e_2) \subseteq G$, so $fv(e_1\ e_2) = fv(e_1) \cup fv(e_2) \subseteq G$

$$\frac{G \cup \{x\} \vdash e_1}{G \vdash \text{fun } x \rightarrow e_1}$$

- case: FUN then we have $e = (\text{fun } x \text{ -> } e_1)$ for some x, e_1 and, by induction, we have $fv(e_1) \subseteq G \cup \{x\}$, but then we also have $fv(\text{fun } x \text{ -> } e_1) = fv(e_1) \setminus \{x\} \subseteq ((G \cup \{x\}) \setminus \{x\}) \subseteq G$

$fv(x)$	=	$\{x\}$	
$fv(\text{fun } x \rightarrow exp)$	=	$fv(exp) \setminus \{x\}$	*('x' is a bound in exp)*
$fv(exp_1\ exp_2)$	=	$fv(exp_1) \cup fv(exp_2)$	

See tc.ml

STATICALLY RULING OUT PARTIALITY: TYPE CHECKING

Adding Integers to Lambda Calculus

exp ::=	
| …	
| n	*constant integers*
| exp_1 + exp_2	*binary arithmetic operation*
val ::=	
| fun x -> exp	*functions are values*
| n	*integers are values*
n{v/x} = n	*constants have no free vars.*
$(e_1 + e_2)\{v/x\} = (e_1\{v/x\} + e_2\{v/x\})$	*substitute everywhere*

$$\frac{exp_1 \Downarrow n_1 \quad exp2 \Downarrow n_2}{exp_1 + exp_2 \ \Downarrow (n1\ [\![+]\!]\ n2)}$$

Object-level '+'   Meta-level '+'

NOTE: there are no rules for the case where exp1 or exp2 evaluate to functions! The semantics is *undefined* in those cases.

Type Checking / Static Analysis

- Recall the interpreter from the Eval3 module:

```
let rec eval env e =
  match e with
   | …
   | Add (e1, e2) ->
        (match (eval env e1, eval env e2) with
           | (IntV i1, IntV i2) -> IntV (i1 + i2)
           | _ -> failwith "tried to add non-integers")
   | …
```

- The interpreter might fail at runtime.
  - Not all operations are defined for all values (*e.g.*, 3/0, 3 + true, …)
- A compiler can't generate sensible code for this case.
  - A naïve implementation might "add" an integer and a function pointer

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: $x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n$

- For example: $x : \text{int}, b : \text{bool} \vdash \text{if (b) 3 else x} : \text{int}$

- What do we need to know to decide whether "if (b) 3 else x" has type int in the environment x : int, b : 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. $x : \text{int}, b : \text{bool} \vdash x : \text{int}$

Simply-typed Lambda Calculus

- For the language in “tc.ml” we have five inference rules:

INT

$$\frac{}{E \vdash i : int}$$

VAR

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

ADD

$$\frac{E \vdash e_1 : int \qquad E \vdash e_2 : int}{E \vdash e_1 + e_2 : int}$$

FUN

$$\frac{E, x : T \vdash e : S}{E \vdash fun\ (x:T) \text{->}\ e\ : T \text{->}\ S}$$

APP

$$\frac{E \vdash e_1 : T \text{->}\ S \qquad E \vdash e_2 : T}{E \vdash e_1\ e_2 : S}$$

- Note how these rules correspond to the code.

Type Checking Derivations

- A *derivation* or *proof tree* has (instances of) judgments as its nodes and edges that connect premises to a conclusion according to an inference rule.
- Leaves of the tree are *axioms* (i.e. rules with no premises)
  - Example: the INT rule is an axiom
- 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:

$$\vdash \texttt{(fun (x:int) -> x + 3) 5} \ : \texttt{int}$$

Example Derivation Tree

$$
\mathrm{APP}\;\dfrac{
  \mathrm{FUN}\;\dfrac{
    \mathrm{ADD}\;\dfrac{
      \mathrm{VAR}\;\dfrac{x : \mathrm{int} \;\in\; x : \mathrm{int}}{x : \mathrm{int} \vdash x \;: \mathrm{int}}
      \qquad
      \mathrm{INT}\;\dfrac{}{x : \mathrm{int} \vdash 3 \;: \mathrm{int}}
    }{x : \mathrm{int} \vdash x + 3 : \mathrm{int}}
  }{\vdash (\mathrm{fun}\ (x{:}\mathrm{int})\ \text{->}\ x + 3) : \mathrm{int}\ \text{->}\ \mathrm{int}}
  \qquad
  \mathrm{INT}\;\dfrac{}{\vdash 5 : \mathrm{int}}
}{\vdash (\mathrm{fun}\ (x{:}\mathrm{int})\ \text{->}\ x + 3)\ 5 \;: \mathrm{int}}
$$

- Note: the OCaml function **typecheck** verifies the existence of this tree. The structure of the recursive calls when running **typecheck** is the same shape as this tree!
- Note that $x : \mathrm{int} \in E$ is implemented by the function **lookup**

Notes about this Typechecker

- The interpreter evaluates the body of a function only when it's applied.
- The typechecker always checks the body of the function
  - even if it's never applied
  - We *assume* the input has some type (say t_1) and reflect this in the type of the function (t_1 -> t_2).

- Dually, at a call site (e_1 e_2), we don't know what *closure* we're going to get.
  - But we can calculate e_1's type, check that e_2 is an argument of the right type, and determine what type e_1 will return.

- Question: Why is this an approximation?
- Question: What if `well_typed` always returns `false`?