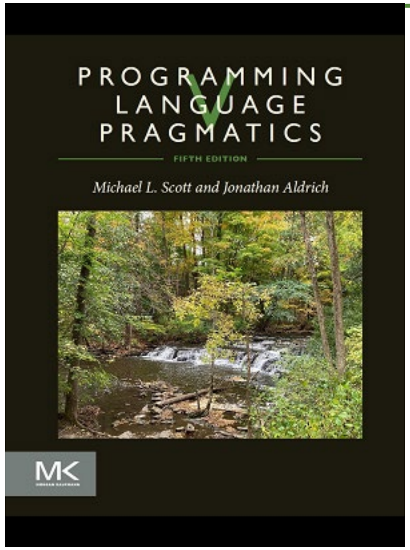
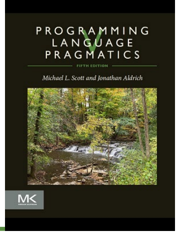
Chapter 4: Program Semantics



Programming Language Pragmatics, Fifth Edition

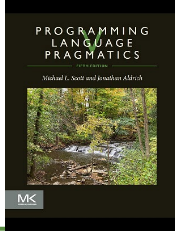
Michael L. Scott and Jonathan Aldrich

Program Semantics



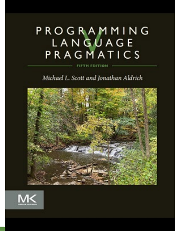
- While syntax describes the form of a program, semantics describes its *meaning*
- ***Dynamic semantics:*** How the program computes its output
  - We've studied big-step and small-step operational semantics as a way of defining dynamic semantics
- ***Static semantics:*** Language rules enforced at compile time
  - Example: checking for unbound identifiers (remember HW1?)
  - Example: checking that operations are passed the right types (today!)
    - `true + 5` is not a valid addition expression!
  - Checked in the *semantic analysis* phase of the compiler
  - These will be the focus of this lecture

Why type check?



- Finds program errors at compile time
  - `true + 5`
  - If we didn’t type check at compile time, then this program would result in an error at run time. It’s better to find it early!
  - Moreover, checking for these errors at run time slows the program down. If we check at compile time we can avoid these run time checks.
- Ensures that types are correct
  - Helpful because types are important documentation for programmers!
- Help generate code
  - `3+2` should generate different assembly code than `3.14+2.5`

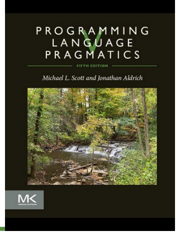
Type checking, at a high level



- A *type checker* traverses that abstract syntax tree (AST) of a program, computing the types of subtrees and flagging errors

- We can describe types with a grammar
  - For our calculator language with functions: $\tau \quad ::= \quad \mathrm{nat} \mid \tau \rightarrow \tau$
  - nat is the type of natural numbers
  - $\tau_1 \rightarrow \tau_2$ is the type of a function
    - takes arguments of type τ_1
    - returns a result of type τ_2

Type checking numbers and addition



- Consider our calculator language with functions:

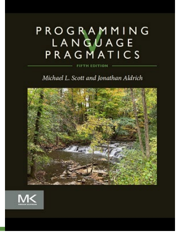
$$
\begin{array}{lcl}
e & ::= & x \mid n \mid e + e \mid \text{let } x = e \text{ in } e \mid \underline{x : \tau} \Rightarrow e \mid e(e) \\
\tau & ::= & \text{nat} \mid \tau \to \tau
\end{array}
$$

Note that function arguments now have a type annotation!

- We define a judgment for assigning types to expressions: $\boxed{e : \tau}$
- Now we can define rules for checking numbers and addition:

$$
\frac{}{n : \text{nat}}\ \textit{T-num} \qquad \frac{e_1 : \text{nat} \quad e_2 : \text{nat}}{e_1 + e_2 : \text{nat}}\ \textit{T-plus}
$$

Type checking variables

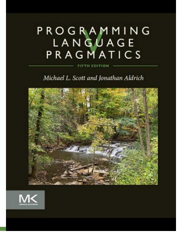


- How does a type checker know the type of a variable?
  - Type checkers rely on a type environment Γ
  - Γ maps each identifier to its type $\Gamma \quad ::= \quad \bullet \mid \Gamma, x : \tau$
- We revise our judgment form: $\boxed{\Gamma \vdash e : \tau}$
  - "In the context of type environment Γ, *e* has type τ"
- Now we can write typing rules for variables and let:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textit{T-var} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : \tau_2}\ \textit{T-let}$$

- The is a *hypothetical judgment*: *e* has type τ assuming (hypothetically) that the variables in *e* have the types given in Γ

Revising our previous rules

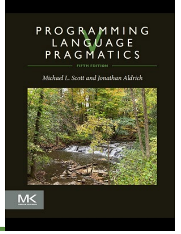


- Let’s revise the type checking rules for numbers and addition to match our new judgment form:

$$\frac{}{\Gamma \vdash n : \mathrm{nat}}\ \textit{T-num} \qquad \frac{\Gamma \vdash e_1 : \mathrm{nat} \quad \Gamma \vdash e_2 : \mathrm{nat}}{\Gamma \vdash e_1 + e_2 : \mathrm{nat}}\ \textit{T-plus}$$

- T-plus needs Γ because e_1 and e_2 might have variables in them
- T-num doesn’t use Γ, but we have to include it so the judgment form is consistent for all rules

Practice with typing rules



- Show a typing derivation for the following program: $\text{let } x = 1 \text{ in } x + 2$

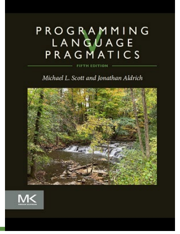
You can do this exercise on paper or with a drawing tool. Leave some space on the sheet for a second practice question; you'll turn in both later in the lecture for in-class participation credit.

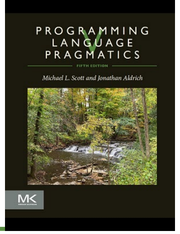
Here are our rules so far:

$$\frac{}{\Gamma \vdash n : \text{nat}}\ \textit{T-num} \qquad \frac{\Gamma \vdash e_1 : \text{nat} \quad \Gamma \vdash e_2 : \text{nat}}{\Gamma \vdash e_1 + e_2 : \text{nat}}\ \textit{T-plus}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textit{T-var} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}\ \textit{T-let}$$

Don’t peek until you did the problem! Answer next…



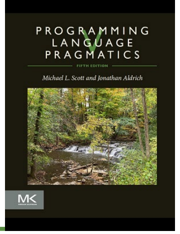


Practice with typing rules (SOLUTION)

- Show a typing derivation for the following program: $\text{let } x = 1 \text{ in } x + 2$

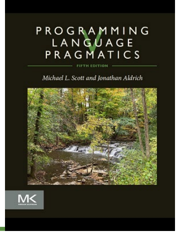
$$\frac{}{\bullet \vdash \text{let } x = 1 \text{ in } x + 2 : \text{nat}}\ \textit{T-let}$$

Practice with typing rules (SOLUTION)



- Show a typing derivation for the following program: $\text{let } x = 1 \text{ in } x + 2$

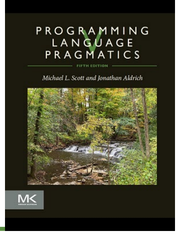
$$\dfrac{\dfrac{}{\bullet \vdash 1 : \text{nat}}\,\textit{T-num} \qquad \dfrac{\dfrac{}{\bullet, x : \text{nat} \vdash x : \text{nat}}\,\textit{T-var} \quad \dfrac{}{\bullet, x : \text{nat} \vdash 2 : \text{nat}}\,\textit{T-num}}{\bullet, x : \text{nat} \vdash x + 2 : \text{nat}}\,\textit{T-plus}}{\bullet \vdash \text{let } x = 1 \text{ in } x + 2 : \text{nat}}\,\textit{T-let}$$



Type checking functions

- The function typing rule checks the body of the function assuming that the argument has the annotated type τ_2:

$$\frac{\Gamma, x : \tau_2 \vdash e_1 : \tau_1}{\Gamma \vdash x : \tau_2 \Rightarrow e_1 : \tau_2 \to \tau_1} \; \textit{T-fn}$$



Practice writing typing rules

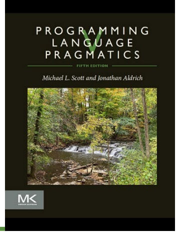
- Write a typing rule for application expressions of the form $e_1(e_2)$

- To get in-class participation credit, upload a picture or screenshot of your answers at https://forms.gle/HQ19Da9NfgRTP2jVA (or the QR code above) using your Andrew ID as the email.
- Here are some rules so far:

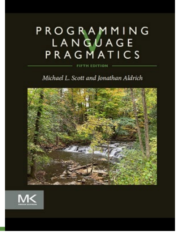
$$\frac{}{\Gamma \vdash n : \text{nat}}\ \textit{T-num} \qquad \frac{\Gamma \vdash e_1 : \text{nat} \quad \Gamma \vdash e_2 : \text{nat}}{\Gamma \vdash e_1 + e_2 : \text{nat}}\ \textit{T-plus}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textit{T-var} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}\ \textit{T-let}$$

Don’t peek until you did the problem! Answer next…



Practice writing typing rules

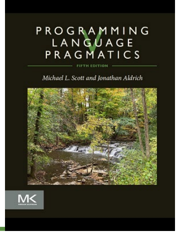


- Write a typing rule for application expressions of the form $e_1(e_2)$

- Solution: $$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1(e_2) : \tau_1}\ \textit{T-apply}$$

- The *T-apply* rule checks the following:
  - The expression in function position must have a function type
  - The expression in argument position must have a type matching the function's argument type
  - The overall expression's type is the function's return type

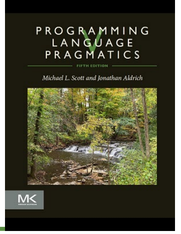
Implementing a type checker in Rust




```rust
fn typecheck(e: &Expr, ctx:&HashMap<String, Type>) -> Type {
    match e {
        Expr::Number(_) => Type::Int,
        Expr::Boolean(_) => Type::Bool,
        Expr::Plus(e1, e2) => {
            let ty1 = typecheck(e1, ctx);
            let ty2 = typecheck(e2, ctx);
            if ty1 != Type::Int || ty2 != Type::Int {
                panic!("Type mismatch: expected Int");
            }
            Type::Int
        },
        ...
    }
}
```

```rust
#[derive(PartialEq)]
#[derive(Clone)]
enum Type {
    Int,
    Bool,
}
```

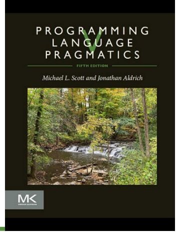
Implementing a type checker in Rust




```rust
fn typecheck(e: &Expr, ctx:&HashMap<String, Type>) -> Type {
    match e {
        ...
        Expr::Id(name) => {
            match ctx.get(name) {
                Some(ty) => ty.clone(),
                None => panic!(...),
            }
        },
        Expr::Let(name, rhs, body) => {
            let mut new_ctx = ctx.clone();
            if KEYWORD_LIST.contains(name) {
                panic!(“variable name is a keyword");
            }
            let ty1 = typecheck(rhs, &new_ctx);
            new_ctx = new_ctx.update(name.clone(), ty1);
            typecheck(body, &new_ctx)
        },
    }
}
```

```rust
#[derive(PartialEq)]
#[derive(Clone)]
enum Type {
    Int,
    Bool,
}
```

Fun Rust Tricks



- How do we make a global constant for `KEYWORD_LIST`?
  - You might think it’s easy, but Rust doesn’t permit global mutable state
  - The list is immutable, but have to use state to initialize it and `rustc` says no
- Some tricks to make this work:
  - A `LazyLock` allows lazy initialization
  - Actual stateful manipulation goes in a lamba (increment lambda is `|x|  x+1` in Rust)
  - The constants have type `&str`, so we convert the `Vec` to an iterator, `map` to `String` with `to_string()`, and `collect` into a new `Vec`

```
static KEYWORD_LIST : LazyLock<Vec<String>> =
    std::sync::LazyLock::new(
        || vec!["let", "if",...].into_iter().map(
                |s| s.to_string()
            ).collect()
    );
```