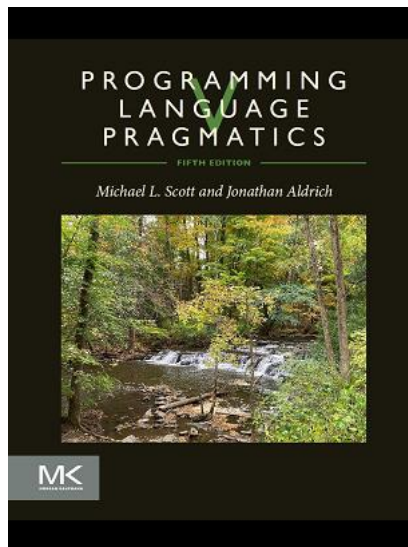
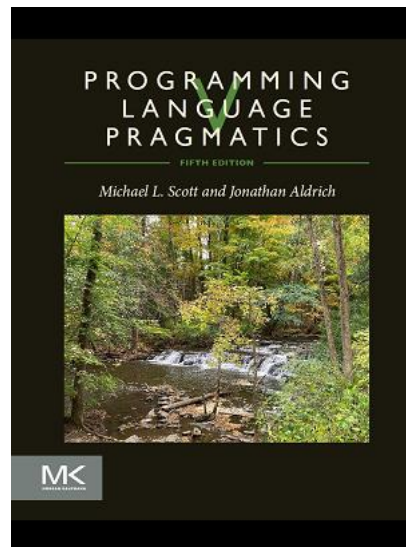


Chapter 4: Program Semantics



Programming Language Pragmatics, Fifth Edition
Michael L. Scott and Jonathan Aldrich

Section 4-4.1: Semantics & Abstract Syntax Trees



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*
 - In practice, syntax specifications are limited by the expressiveness of context-free grammars; semantics is everything that goes beyond this
- Semantics has two parts
 - Computing a program's output from its input
 - Enforcing rules, e.g. type consistency
- Some aspects of semantics may be left *undefined*
 - Then the implementation can choose what to do
 - Often the case for nonsensical operations
 - e.g. dereferencing a null pointer
 - Safe languages try to avoid undefined behavior (e.g. Java) or confine it to designated **unsafe** blocks (e.g. Rust)

Dynamic and static semantics

- **Dynamic semantics:** checks and computations done at run time
 - Could be the entire semantics, in the case of an interpreter
- **Static semantics:** language rules enforced at compile time
 - Example: checking for unbound identifiers
 - Example: checking that operations are passed the right types
 - `true + 5` is not a valid addition expression!
 - Checked in the *semantic analysis* phase of a compiler

Dynamic and static semantics

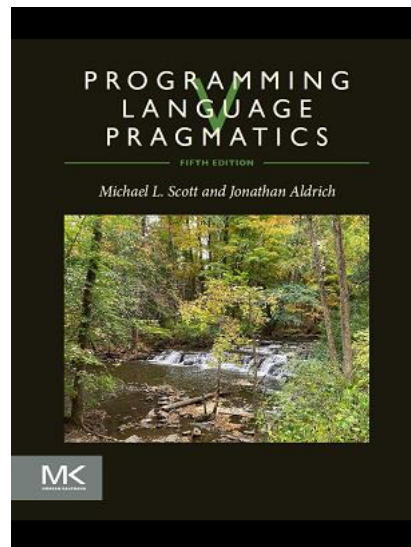
- **Dynamic semantics:** checks and computations done at run time
- **Static semantics:** language rules enforced at compile time
- Most languages check at least some rules at run time
 - e.g. checked type casts, null pointer and array out of bounds errors
- Enforcing some rules at compile time has advantages
 - Catches errors early
 - Can generate more efficient code and eliminates run time checks

Semantic analysis

- A *semantic analyzer* enforces the static semantics of the language
 - It also annotates the program with information for later compiler stages
- The analyzer is *sound* if the conclusions it reaches always agree with what will happen when the program runs
 - E.g. that a variable will always hold a certain type, or an array index will never be out of bounds
- Sometimes the analyzer is uncertain of a required property
 - It will then insert dynamic checks to verify the property at run time
- Tools outside the compiler can check additional properties
 - E.g. *linters* may check style rules, or alert the programmer to likely errors

Section 4.4: Static Semantics

Part 1: Typing rules



Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich

The semantic analyzer has two roles

- Enforce static semantic rules, for example:
 - Names may only be used in the scope of their declaration
 - Variables must be initialized before being used
 - All operations are passed data of the correct type
- Annotate the program with information needed by the code generator or interpreter
 - Clarifications
 - Does a variable x refer to a local declaration or a global one?
 - Is this addition an integer operation or a floating-point one?
 - Requirements for dynamic checks
 - This array access must be checked for an out-of-bounds error
 - This multiplication must be checked for integer overflow

One semantic analysis: type checking

Type checking has a number of benefits

- 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: $\tau ::= \text{nat} \mid \tau \rightarrow \tau$
 - `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:

$$e ::= x \mid n \mid e + e \mid \text{let } x = e \text{ in } e \mid x : \tau \Rightarrow e \mid e(e)$$

$$\tau ::= \text{nat} \mid \tau \rightarrow \tau$$

Note that function arguments now have a type annotation!

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

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

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 $\Gamma ::= \bullet \mid \Gamma, x : \tau$
- 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:

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

- This 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:

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

- 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: `let $x = 1$ in $x + 2$`

Here are our rules so far:

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

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

Practice with typing rules (SOLUTION)

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

• $\vdash \text{let } x = 1 \text{ in } x + 2 : \text{nat}$ *T-let*

Practice with typing rules (SOLUTION)

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

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

Type checking functions

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

$$\frac{\Gamma, x : \tau_2 \vdash e_1 : \tau_1}{\Gamma \vdash x : \tau_2 \Rightarrow e_1 : \tau_2 \rightarrow \tau_1} T\text{-fn}$$

Practice writing typing rules

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

- Here are some rules so far:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} T\text{-var} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} T\text{-let} \quad \frac{\Gamma \vdash e_1 : \text{nat} \quad \Gamma \vdash e_2 : \text{nat}}{\Gamma \vdash e_1 + e_2 : \text{nat}} T\text{-plus} \quad \frac{\Gamma, x : \tau_2 \vdash e_1 : \tau_1}{\Gamma \vdash x : \tau_2 \Rightarrow e_1 : \tau_2 \rightarrow \tau_1} T\text{-fn}$$

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

```
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
        },
        ...
    }
}
```

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

Implementing a type checker in 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) => {
            if KEYWORD_LIST.contains(name) {
                panic!("variable name is a keyword");
            }
            let ty1 = typecheck(rhs, &ctx);
            let mut new_ctx = ctx.update(name.clone(), ty1);
            typecheck(body, &new_ctx)
        },
    }
}
```

```
#[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 lambda (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()
    );
```