Type Systems

– lecture notes –

Contents

Introduction to Type Systems

Type systems are fundamental to understanding programming languages and ensuring program correctness. They provide a formal framework for classifying program values and preventing certain classes of errors before program execution. This is in contrast to testing-based approaches, which can only verify correctness for specific inputs.

The key advantage of type systems is their ability to provide *static guarantees* about program behavior:

- **Compile-time error detection:** Type errors are caught before the program runs
- **Documentation:** Types serve as machine-checked documentation
- **Optimization opportunities:** Type information enables better code generation
- **Program understanding:** Types help programmers reason about code behavior

Type systems are particularly well-suited for studying in the context of formal methods because:

- **Precision:** Type rules can be specified with mathematical rigor
- **Mechanization:** Proof assistants can verify type system properties automatically
- **Compositionality:** Type systems compose naturally with other language features
- **Practical relevance:** Understanding type systems is essential for modern programming
- **Foundational concepts:** Type theory underlies many programming language concepts

This chapter is designed for students learning about programming language theory and formal methods. We use Lean 4 as our proof assistant, which allows us to:

- Define type systems formally and precisely

- State and prove fundamental theorems about type safety
- Experiment with typing derivations interactively
- Verify that our type system implementation is correct

While defining and proving properties of type systems can be complex, proof assistants like Lean 4, Coq, and Isabelle greatly reduce the burden of this work. These tools provide:

- **Automation:** Tactics can solve many proof obligations automatically
- **Error detection:** Invalid proof steps are caught immediately
- **Confidence:** Completed proofs provide mathematical certainty
- **Exploration:** Interactive development aids understanding

The goal of this chapter is to develop a complete type system for a simple expression language, prove its fundamental properties (progress and preservation), and understand the implications of type soundness. By the end, students should understand:

- How to define typing rules formally
- What it means for a type system to be sound
- How to prove type safety properties
- The relationship between static typing and runtime behavior

1 Introduction

Type systems are essential in programming languages. The core concept of types is simple: values are categorised in certain *types* so that programmers can distinguish them and use them in particular contexts. For instance, Lean 4 is *strongly-typed*: it does have very strict typing rules that determine the compiler to throw errors at compile time. If a Lean function takes two integers, but it is called with a boolean argument, the compiler will complain:

```
-- Lean 4 type error example
#check true + false
-- application type mismatch
--   true + false
```

```
-- argument
--   true
-- has type
--   Bool : Type
-- but is expected to have a numeric type
```

Other programming languages are not that strict. The popular Javascript language is *weakly-typed*: the typing rules are loose and they typically allow values to be interpreted as having a particular type depending on the context. A disadvantage is that weak typing can lead to errors during the execution of a program or some unexpected behaviours in the eye of the unexperienced programmer. Here are some interesting examples using the Node.js interpreter in command line:

```
$ node
Welcome to Node.js v14.17.5.
> 2 + '4'
'24'
> 2 * '4'
8
> '2' * '4'
8
> 2 * 'a'
NaN
> 2 + false
2
> 2 + true
3
> false - true
-1
> 2 && true
true
> true && 2
2
> 0 && false
0
> false && 0
false
> [] == ![]
true
> [] == false
true
> ![] == false
true
```

Compared to Lean, the above examples seem to be a bit silly: why on earth would `2 + '4'` return `'24'`, while `2 * '4'` returns `8`? In the first case it looks like `2` is converted to `'2'` and then the strings are concatenated. In the second case, it looks like `'4'` is converted to `4` and the result is indeed `2 * 4`, i.e., `8`. At a closer look, this could make sense if `+` is used for both string concatenation and addition of numbers. Also, booleans are treated as numbers when convenient, and numbers can be treated as booleans. However, the last four examples are confusing for a newcomer.

Using numbers as booleans is common in C as well. For instance, in the program below, the value printed by the program depends on the value of the variable `x`:

```
#include <stdio.h>
int main() {
  int x = _;
  printf("x = %d, ", x);
  if (x) {
    printf("took the 'then' branch\n");
  } else {
    printf("took the 'else' branch\n");
  }
}
```

If the `_` is zero, the execution of the `if` statement will print `took the 'else' branch`. Otherwise, it will print `took the 'then' branch`. It is obvious that `x` is interpreted by the `if` statement as a boolean: zero is used to represent false and anything else is used to represent true.

Based on the above examples, one might be tempted to say that strongly-typed is good and weakly-typed is bad. But both approaches have their advantages and disadvantages. A strongly-typed approach provides some advantages in terms of correctness, documentation, safety, optimisation, abstraction. A weakly-typed approach provides the possibility to use the same value in different contexts and leaves the interpreter to choose the type. In the right hands, this can be turned into an advantage and can improve the productivity.

In this chapter we mix the arithmetic and boolean expressions of `IMP`, and we present a type system for them. We show the rules of the type system, we integrate them in Lean 4, and then we use them to prove properties of the proof system and typing properties for programs.

2 IMP Syntax of Expressions

The syntax of expressions that we are going to use in this chapter is shown here:

```
inductive Exp where
  | anum : Int -> Exp
  | avar : String -> Exp
  | aplus : Exp -> Exp -> Exp
  | amult : Exp -> Exp -> Exp
  | btrue : Exp
  | bfalse : Exp
  | bnot : Exp -> Exp
  | band : Exp -> Exp -> Exp
  | blessthan : Exp -> Exp -> Exp
  deriving Repr

open Exp

instance : OfNat Exp n where
    ofNat := anum n
instance : Coe String Exp where
  coe := avar

infixl:50 " +' " => aplus
infixl:40 " *' " => amult
infixl:60 " <' " => blessthan
infixl:75 " &&' " => band
```

It is now possible to mix booleans with numbers or other arithmetic expressions. The following checks do not fail:

```
#check (2 +' 2)
  -- 2 +' 2 : Exp

#check (2 +' btrue)
  -- 2 +' btrue : Exp

#check (band 2 2)
  -- band 2 2 : Exp
```

Exercise 2.1. *Rewrite the syntax of the IMP statements using the expressions above. Is it now possible to use numbers as booleans in conditional statements as we do in C? What about writing expressions as we did in our Javascript examples?*

3 Small-step SOS for Expressions

The small-step SOS semantics of the expressions is straightforward: we simply assemble all the small-step rules for arithmetic and boolean expressions:

```
abbrev Env := String -> Nat

def update (sigma : Env) (x : String) (v : Nat) : Env :=
  fun y => if x == y then v else sigma y

inductive eval : Exp -> Env -> Exp -> Prop where
  | const : forall i sigma,
      eval (anum i) sigma (anum i)
  | lookup : forall x sigma,
      eval (avar x) sigma (anum (sigma x))
  | add_l : forall a1 a2 sigma a1',
      eval a1 sigma a1' ->
      eval (a1 +' a2) sigma (a1' +' a2)
  | add_r : forall a1 a2 sigma a2',
      eval a2 sigma a2' ->
      eval (a1 +' a2) sigma (a1 +' a2')
  | add : forall i1 i2 sigma n,
      n = i1 + i2 ->
      eval (anum i1 +' anum i2) sigma (anum n)
  | mul_l : forall a1 a2 sigma a1',
      eval a1 sigma a1' ->
      eval (a1 *' a2) sigma (a1' *' a2)
  | mul_r : forall a1 a2 sigma a2',
      eval a2 sigma a2' ->
      eval (a1 *' a2) sigma (a1 *' a2')
  | mul : forall i1 i2 sigma n,
      n = i1 * i2 ->
      eval (anum i1 *' anum i2) sigma (anum n)
  | etrue : forall sigma,
      eval btrue sigma btrue
  | efalse : forall sigma,
      eval bfalse sigma bfalse
  | lessthan_l : forall a1 a2 sigma a1',
      eval a1 sigma a1' ->
      eval (a1 <' a2) sigma (a1' <' a2)
  | lessthan_r : forall a1 a2 sigma a2',
      eval a2 sigma a2' ->
      eval (a1 <' a2) sigma (a1 <' a2')
  | lessthan_ : forall i1 i2 sigma b,
      b = (if i1 < i2 then btrue else bfalse) ->
```

```
      eval (anum i1 <' anum i2) sigma b
  | not_1: forall b b' sigma,
      eval b sigma b' ->
      eval (bnot b) sigma (bnot b')
  | not_t : forall sigma,
      eval (bnot btrue) sigma bfalse
  | not_f : forall sigma,
      eval (bnot bfalse) sigma btrue
  | band_1 : forall b1 b2 b1' sigma,
      eval b1 sigma b1' ->
      eval (band b1 b2) sigma (band b1' b2)
  | band_true : forall b2 sigma,
      eval (band btrue b2) sigma b2
  | band_false : forall b2 sigma,
      eval (band bfalse b2) sigma b2

open eval
notation:98 A " -[ " S " ]-> " B => eval A S B
```

The reflexive-transitive closure of the evaluation relation is defined as follows:

```
inductive eval_closure : Exp -> Env -> Exp -> Prop where
  | refl : forall e sigma,
      eval_closure e sigma e
  | tran : forall e1 e2 e3 sigma,
      eval e1 sigma e2 ->
      eval_closure e2 sigma e3 ->
      eval_closure e1 sigma e3

open eval_closure
notation:99 A " -[ " S " ]>* " B => eval_closure A S B

def Env0 : Env := fun _ => 0
```

The semantics behaves as expected. Here are some examples:

```
example : (2 +' "n") -[ Env0 ]>* 2 := by
  apply tran (e2 := 2 +' 0)
  · apply add_r
    apply lookup
  · apply tran
    · apply add
      rfl
    · apply refl

```

```
example : ("n" +' "n") -[ Env0 ]>* 0 := by
  apply tran (e2 := 0 +' "n")
  · apply add_l
    apply lookup
  · apply tran (e2 := 0 +' 0)
    · apply add_r
      apply lookup
    · apply tran
      · apply add
        rfl
      · apply refl
```

However, this allows us to write some ill formed expressions. For instance:

```
example : (2 +' btrue) -[ Env0 ]>* ? := by
  sorry  -- Cannot be proven!
```

In a weakly-typed language, the compiler or the interpreter does some automatic conversions (e.g., remember the Javascript examples). For a strongly-typed language, an error is thrown. For instance, in Lean 4 you get an error right away, since Lean checks the types of the expressions *statically* (remember the Lean example at the beginning of this chapter). In Python, the error is thrown *dynamically*, that is, the types are checked during the execution:

```
$ python3
Python 3.9.6 (default)
>>> 3 + 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and '
    str'
```

Definitely, getting such errors during execution is not desirable. It would be easier for the programmer to have a tool that signals whether such terms could lead to runtime errors. Our semantics of the mixed expressions simply does not care about reducing such terms. So, `2 +' btrue` cannot be reduced to a value, and at the same time, it is not itself a value. We say that such terms (that are not values and cannot be reduced to a value) are *stuck*. How do we detect stuck terms? The answer is simple: define a type system for our language. The type system helps us to exclude nonsensical terms by simply checking their types.

4 A Type System for Expressions

Our expressions include both numbers and boolean values. In order to exclude terms that we do not want to have a meaning (e.g., `2 +' btrue`), we define a typing relation that relates terms to the types of their final (evaluated) results.

We are going to use some conventional notations. The set of basic types is denoted by $\mathcal{T}$. For our expressions, it is sufficient to choose the set $\mathcal{T} = \{Nat, Bool\}$. For more complex languages, this set can be larger. By $e : Nat$ we denote the fact that the expression e has type Nat. The typing rules are given using inference rules. Checking whether a given expression has a certain type reduces to finding a derivation (i.e., a proof) using the typing rules.

The typing rules for our expressions are shown below. The conclusions of all rules cover the entire syntax of our expressions. The rules are straightforward. The constants have the expected types. Unlike TTrue and TFalse, TNum and TVar have side conditions because the set of naturals and the set of variables are infinite. All the other rules have the expected premisses. For instance, TLt reads as follows: the expression e_1 `<'` e_2 has type $Bool$ if both e_1 and e_2 are of type Nat. Similarly, TPlus says that e_1 `+'` e_2 has type Nat when e_1 and e_2 are of type Nat. Therefore, it is impossible to find a derivation for `2 +' btrue` : Nat or `2 +' btrue` : $Bool$.

$$\text{TNum:}\quad \frac{\cdot}{n : \mathit{Nat}}\ n \in \mathbb{Z}$$

$$\text{TVar:}\quad \frac{\cdot}{x : \mathit{Nat}}\ x \in \mathit{Var}$$

$$\text{TPlus:}\quad \frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1\ \texttt{+'}\ e_2 : \mathit{Nat}}$$

$$\text{TMul:}\quad \frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1\ \texttt{*'}\ e_2 : \mathit{Nat}}$$

$$\text{TTrue:}\quad \texttt{btrue} : \mathit{Bool}$$

$$\text{TFalse:}\quad \texttt{bfalse} : \mathit{Bool}$$

$$\text{TNot:}\quad \frac{e : \mathit{Bool}}{\texttt{bnot}\ e : \mathit{Bool}}$$

$$\text{TAnd:}\quad \frac{e_1 : \mathit{Bool} \qquad e_2 : \mathit{Bool}}{\texttt{band}\ e_1 e_2 : \mathit{Bool}}$$

$$\text{TLt:}\quad \frac{e_1 : \mathit{Nat} \qquad e_2 : \mathit{Nat}}{e_1\ \texttt{<'}\ e_2 : \mathit{Bool}}$$

Here is an example of a derivation for $2\ \texttt{+'}\ x : \mathit{Nat}$, where $2 \in \mathbb{Z}$ and $x \in \mathit{Var}$ (i.e., x is a variable):

$$\dfrac{\dfrac{\dfrac{\cdot}{2 : \mathit{Nat}}\ \text{TNum} \qquad \dfrac{\cdot}{x : \mathit{Nat}}\ \text{TVar}}{2\ \texttt{+'}\ x : \mathit{Nat}}\ \text{TPlus}}$$

Here is another derivation for $2\ \texttt{+'}\ x\ \texttt{<'}\ x : \mathit{Bool}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot}{2 : \mathit{Nat}}\ \text{TNum} \qquad \dfrac{\cdot}{x : \mathit{Nat}}\ \text{TVar}}{2\ \texttt{+'}\ x : \mathit{Nat}}\ \text{TPlus} \qquad \dfrac{\cdot}{x : \mathit{Nat}}\ \text{TVar}}{2\ \texttt{+'}\ x\ \texttt{<'}\ x : \mathit{Bool}}\ \text{TLt}}$$

An important observation is that typing derivations are different from evaluating expressions. Obviously, if x is of type *Nat* then the expression $2\ \texttt{+'}\ x\ \texttt{<'}\ x$ cannot be true. But the expression $2\ \texttt{+'}\ x\ \texttt{<'}\ x$ is still typable and it does have type *Bool*. So, type checking is not the same thing as evaluating expressions.

Exercise 4.1. *What is the type of the expression:*

$$\texttt{band}\ (3\ \texttt{*'}\ x\ \texttt{+'}\ 7\ \texttt{<'}\ x)\ (\texttt{bnot}\ (y\ \texttt{>'}\ x))?$$

Write down a derivation that justifies your answer.

Exercise 4.2. *We define the syntax of an expression language called* PlayWithTypes. *The BNF syntax of expressions is shown below:*

$$E ::= \texttt{O} \mid \texttt{S}\ E \mid \texttt{isZero}\ E \mid \texttt{T} \mid \texttt{F} \mid \texttt{ite}\ EEE.$$

The constructors are written using this `font`. *We notice that the syntax of* PlayWithTypes *is in fact a mix of Peano naturals (*`O` *and* `S` *are the usual constructors) and booleans (*`T` *and* `F` *are well-known boolean constants). The* `ite` *construct is an* if-then-else*-like statement that returns an expression: it takes a boolean and two natural expressions and it should return a natural expression. The predicate* `isZero` *should tell us whether the given natural expression is zero or not.*

Based on the above assumptions, write down the typing rules for this language. Also, write examples that cover all the constructs of the language.

5 A Type System for Expressions in Lean 4

All the above definitions and rules can be easily encoded in Lean 4. We start by defining the set of basic types `Typ` (that is, the set $\mathcal{T}$):

```
inductive Typ where
  | Nat : Typ
  | Bool : Typ
  deriving Repr, DecidableEq

open Typ
```

The typing rules are shown below. In Lean 4 we use `type_of` instead of ':'. So, `type_of x Nat` stands for $x : Nat$.

```
inductive type_of : Exp -> Typ -> Prop where
  | t_num : forall n,
      type_of (anum n) Nat
  | t_var : forall x,
      type_of (avar x) Nat
  | t_plus : forall a1 a2,
      type_of a1 Nat ->
      type_of a2 Nat ->
```

```
      type_of (a1 +' a2) Nat
  | t_mult : forall a1 a2,
      type_of a1 Nat ->
      type_of a2 Nat ->
      type_of (a1 *' a2) Nat
  | t_true :
      type_of btrue Bool
  | t_false :
      type_of bfalse Bool
  | t_not : forall b,
      type_of b Bool ->
      type_of (bnot b) Bool
  | t_and : forall b1 b2,
      type_of b1 Bool ->
      type_of b2 Bool ->
      type_of (band b1 b2) Bool
  | t_lessthan : forall a1 a2,
      type_of a1 Nat ->
      type_of a2 Nat ->
      type_of (blessthan a1 a2) Bool
```

Note that the side conditions of TNum and TVar are enforced by the fact that the constructors `anum` and `avar` accept the right types for their arguments.

Our simple derivation example for `2 +' x` : *Nat* is trivial in Lean 4:

```
example : type_of (2 +' "n") Nat := by
  apply type_of.t_plus
  · apply type_of.t_num
  · apply type_of.t_var
```

Remember that we started this section by discussing the `2 +' btrue` example. Now that we have types and a Lean 4 implementation, our proof search fails to find a derivation for `2 +' btrue : Nat`:

```
example : type_of (2 +' btrue) Nat := by
  apply type_of.t_plus
  · apply type_of.t_num
  · -- Cannot prove that btrue has type Nat!
    sorry
```

In general, the fact that we cannot find a proof does not mean that there isn't one. Our point here is to show that it is impossible to derive a typing for `2 +' btrue`. The same happens for `2 +' btrue : Bool`. We address the properties of our type system later in a dedicated section.

Exercise 5.1. *Implement in Lean 4 the syntax and the type system that you developed for the* PlayWithTypes *language from Exercise 4.2. Also, use your implementation to automatically find the derivations for the examples that you are asked to write in Exercise 4.2.*

6 Properties of the Typing Relation

The syntax of our expressions allows us to write various expressions. Repetitive applications of the small step rules to an expression may produce either an infinite loop or a term that cannot be reduced anymore (i.e., none of the existing rules can be applied to it). Terms that cannot be reduced anymore are said to be in a *normal form*. Values (e.g., natural numbers or boolean constants) are in a normal form. Terms that cannot progress (e.g., `2 +' btrue`) are also in normal form. Such terms are stuck: they have no successors w.r.t. small-step rules.

Our goal is to use the typing relation to distinguish between expressions, so that expressions that do not make sense are excluded. But how do we know that our typing relation does this in a correct way? Can we prove some properties about our typing relation so that we can guarantee that it is correct? What are these properties?

In the rest of this section we are going to formulate and prove some common properties of typing relations: progress, type preservation and soundness.

6.1 Progress

Progress is a property that captures the fact that well-typed expressions make progress w.r.t. the small-step rules. Obviously, values cannot make progress because they are already reduced. First, we define what is a value in Lean 4:

```
1 inductive nat_value : Exp -> Prop where
2   | n_val : forall n, nat_value (anum n)
3
4 inductive bool_value : Exp -> Prop where
5   | b_true : bool_value btrue
6   | b_false : bool_value bfalse
7
8 def value (e : Exp) : Prop :=
9   nat_value e \/ bool_value e
```

The above definitions of the values agree with the typing relation. This is captured by the following lemmas:

```
theorem bool_canonical :
  forall e,
    type_of e Bool ->
    value e ->
    bool_value e := by
  intro e H H'
  cases H' with
  | inl Hn =>
    cases H with
    | t_true => constructor
    | t_false => constructor
    | t_not _ _ => cases Hn
    | t_and _ _ _ _ => cases Hn
    | t_lessthan _ _ _ _ => cases Hn
  | inr Hb => exact Hb

theorem nat_canonical :
  forall e,
    type_of e Nat ->
    value e ->
    nat_value e := by
  intro e H H'
  cases H' with
  | inl Hn => exact Hn
  | inr Hb =>
    cases H with
    | t_num => constructor
    | t_var => cases Hb
    | t_plus _ _ _ _ => cases Hb
    | t_mult _ _ _ _ => cases Hb
```

Now, we are ready to formulate and prove the progress property in Lean 4:

```
theorem progress :
  forall e T sigma,
    type_of e T ->
    (value e \/ exists e', e -[ sigma ]-> e') :=
by
  intro e T sigma H
  induction H with
  | t_num n => left; left; constructor
  | t_var x => right
```

```
              exists anum (sigma x)
              constructor
  | t_plus a1 a2 H1 H2 IH1 IH2 =>
    cases IH1 with
    | inl H1v =>
      cases IH2 with
      | inl H2v =>
        have Hn1 := nat_canonical a1 H1 H1v
        have Hn2 := nat_canonical a2 H2 H2v
        cases Hn1 with | n_val n1 =>
        cases Hn2 with | n_val n2 =>
        right
        exists (anum (n1 + n2))
        apply add
        rfl
      | inr H2v =>
        right
        have ⟨e2, He2⟩ := H2v
        exists (a1 +' e2)
        apply add_r
        exact He2
    | inr H1v =>
      right
      have ⟨e1, He1⟩ := H1v
      exists (e1 +' a2)
      apply add_l
      exact He1
  -- ... (similar cases for other constructors)
```

The proof is by induction on the typing relation.

6.2 Type Preservation

The type preservation property is critical: it says that if a well-typed expression is reduced in one step using the small-step rules, the obtained expression is also well-typed. The theorem is formulated below and its proof is by induction on the typing relation:

```
theorem preservation :
  forall e e' T sigma,
    type_of e T ->
    (e -[ sigma ]-> e') ->
    type_of e' T := by
  intro e e' T sigma H He
  revert e' He
```

```
  induction H with
  | t_num n =>
    intro e' He
    cases He with
    | const => constructor
  | t_var x =>
    intro e' He
    cases He with
    | lookup => constructor
  | t_plus a1 a2 H1 H2 IH1 IH2 =>
    intro e' He
    cases He with
    | add_l _ _ _ _ He1 =>
      apply type_of.t_plus
      · exact IH1 _ He1
      · exact H2
    | add_r _ _ _ _ He2 =>
      apply type_of.t_plus
      · exact H1
      · exact IH2 _ He2
    | add =>
      constructor
  -- ... (similar cases for other constructors)
  | t_lessthan a1 a2 H1 H2 IH1 IH2 =>
    intro e' He
    cases He with
    | lessthan_l _ _ _ _ He1 =>
      apply type_of.t_lessthan
      · exact IH1 _ He1
      · exact H2
    | lessthan_r _ _ _ _ He2 =>
      apply type_of.t_lessthan
      · exact H1
      · exact IH2 _ He2
    | lessthan_ i1 i2 _ _ He =>
      subst e'
      split <;> constructor
```

6.3 Type Soundness

Finally, we are ready to formulate and prove the soundness of our typing relation.

```
theorem soundness :
```

```
  forall e e' T sigma,
    type_of e T ->
    (e -[ sigma ]>* e') ->
    (value e' \/ exists e'', e' -[ sigma ]-> e'') := by
  intro e e' T sigma H Hclos
  induction Hclos with
  | refl e sigma =>
    exact progress e T sigma H
  | tran e1 e2 e3 sigma He12 He23 IH =>
    have H2 := preservation e1 e2 T sigma H He12
    exact IH H2
```

The property is essentially saying that a well-typed expression cannot reach a stuck state. In other words, if we reduce a well-typed expression using our small-step semantics we always reach a value (that is, a result) or an expression that can be further reduced.

The implications of this formal property are very strong: basically, if a program can be proved to be well-typed (and we can do that using the typing relation) then it is guaranteed not to produce erroneous results that can lead to unexpected behaviours in programs. Typing improves the quality of our programs and some common errors in programs can be avoided using type systems.

Exercise 6.1. *Formulate and prove the progress, type preservation and type soundness properties for the typing relation of the* PlayWithTypes *language (see Exercises 4.2 and 5.1).*

7 Conclusion

In this chapter, we have explored type systems and their fundamental role in programming languages. We have seen how type systems can prevent nonsensical expressions from being accepted, thereby improving program safety and reliability.

The key takeaways are:

- Type systems classify expressions into categories (types) and enforce rules about how these categories can be combined
- Strongly-typed languages catch type errors at compile time, while weakly-typed languages may allow implicit conversions
- A type system consists of typing rules that can be formalized as inference rules

- The three fundamental properties of a sound type system are:
  - Progress: well-typed expressions either are values or can take a step
  - Preservation: evaluation preserves types
  - Soundness: well-typed programs don't go wrong (they never get stuck)
- Proof assistants like Lean 4 allow us to mechanically verify that our type system has these properties

Understanding type systems is crucial for both language designers and programmers. Type systems provide guarantees about program behavior and help catch bugs before they manifest at runtime.