Why do languages include a type notation and a type checker?

1 The Pragmatics Question

While a 1970s programming language such as Smalltalk solves the problem of data representation and functional abstraction, it injects new problems into the work of software developers. As a Smalltalk program executes it may try to send a message to an object that does not come with a corresponding method. The language implementation's response is an error signal telling the user "method not found." Similarly, a program may try to retrieve the value of a non-existent field, try to modify such a field, and so on.

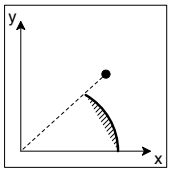
The models of the preceding two chapters reflect these novel problems via several novel failure cases in the CESK transition function:

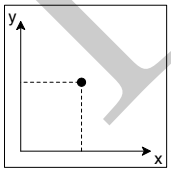
- object creation, due to a mismatch between the number of arguments and fields;
- method call expressions, due to a mismatch between the number of arguments and parameters;
- method call expressions, due to a lack of the named method;
- field access and modification, due to a lack of the named field; and
- the addition and division operations.

The last one is due to the extension of the set of values with objects. In addition to the (floating point) numbers of the *Core* model, the value set of *Classy* consists of the union of numbers and objects. Since the model does not constrain what a variable can stand for, it is therefore possible that expressions such as (x + y) or (x / y) have to deal with objects as arguments, which makes no sense.

```
(module Polar
  (class Polar
    (; the angle in degrees,
     ; counter-clockwise
     x
     ; the distance from (0,0)
     ; on the angled line
     y)
    ;; rotate `this` point by `δ`
    (method rotate (δ)
      (def myX   (this --> x))
      (def myY   (this --> y))
      (def angle (δ + angle))
      (new Polar (myX angle)))))
```

```
(module Cartesian
  (class Cartesian
    (; the distance (going right)
     ; on the x axis from (0,0)
     x
     ; the distance (going up)
     ; on the y axis from (0,0)
     y)
    ; translate `this` point by
    ; `other` point interpreted
    ; as a vector
    (method translate (other)
      (def myX (this --> x))
      (def myY (this --> y))
      (def otherX (other --> x))
      (def otherY (other --> y))
      (def X (myX + otherX))
      (def Y (myY + otherY))
      (new Cartesian (X Y)))))
```






```
(import Polar)
(import Cartesian)

(def one   1.4) ;; (sqrt 2)
(def half 45.0)
(def p (new Polar (half one)))
(def two  2.0)
(def c (new Cartesian (two two)))
(def d (c --> translate(p)))
(def dx (d --> x))
(def dy (d --> y))

(dx + dy)
```

Figure 45: Logical mistakes due to a lack of an isa check

Run-time errors pose problems to software developers, and they can cause havoc after deployment. Imagine software for assisting the airplane pilots. When such a system displays the text “method not found” or “addition expects numbers, given an object” and goes down during a critical phase, say take-off, the consequences could be catastrophic. Clearly, end users of software should experience as few run-time errors as possible.

The situation is worse than it appears: a system may terminate normally and output a number, without any indication that anything went wrong. For an example, consider figure 45. It displays a system in *Module* that consists of two independent modules (top) and a main program that relies on both (bottom). As the data interpretation comments in the module on the left explain, the module supplies a class for representing a point's coordinates in polar form; by contrast, the module on the right exports a class for representing point's coordinates in Cartesian form. Between them are two diagrams that illustrate the data interpretation comments.

Let's take a look at the main program. It imports both modules, and it constructs an instance each of the two classes:

- The properties of p tell us that its Cartesian coordinates would be near the coordinate (1,1), because the root of $1^2 + 1^2$ is close to 1.4 and 45 degrees splits a right angle in half.

- The second instance, c, sits at (2,2), just as its coordinates say.

Furthermore, the call to translate appears to have the goal of creating a point that is one step up and one step to the right—which is the vector interpretation of p. What we should expect is that the new instance of Cartesian has the coordinates (3,3) and therefore outputs 6. But, when this program is linked and run on the CESK machine for *Classy*, it yields a rather strange result, namely, 50.4.

Stop! Explain why running the program in figure 45 yields 50.4.

A close look reveals the problem. The translate method expects to receive an instance of Cartesian, but the method call supplies an instance of Polar. In this admittedly silly example, the method call doesn't fail because other does have x and y fields. Hence, the retrievals of the respective field values succeed, the rest of the method computes essentially meaningless numbers, and the final expression creates an instance of Cartesian.

The pragmatic problem is that the language creator has shifted the burden of enforcing this design discipline to the many users of this language. Every software developer using a language like *Classy* must use checks such as (other isa Cartesian) at the entry to methods such as translate. If this check is omitted, the call may produce numbers and objects without any meaningful interpretation.

Arguably this kind of problem is worse than the run-time errors discussed first. When a program signals a run-time error, the developer gets some information about the program's problems via the error message. If the language implementation is constructed to issue informative messages,

the error signal cuts down the search space for the bug. By contrast, when a program outputs strange numbers, the developer might not find out.

Deploying such code in real-world settings may cause serious problems. Consider the software for a pacemaker. If the software crashes due to a run-time error, the patient may die but a diagnosis may reveal that the software system issued an error message and trigger further investigations. If the software causes the instrument to put $500V$ instead of $5V$ on the "wire" neither a doctor nor a developer may suspect the software.

Language creators can accept the responsibility for the prevention of these run-time errors and logical mistakes. The key is to enrich the language with a form of validity checking that classifies how code behaves before it runs and that informs the developer of potential problems. In turn, the language creators may impose a different burden on the developer, namely, the task of massaging the code so that it satisfies these added validity checks.

1.1 Design Choices

By the 1990s, programming language creators had recognized that a type system not only helped compiler writers to generate fast code but that a type system also helped software developers at almost all stages of the engineering process. Roughly speaking, a type system adds a notation to the language that enables programmers to state claims about the remainder of the code. Thus, a piece of code may declare that a variable has a numeric type and therefore always represents a number in its lexical scope. To make sure that such claims are valid, a type system comes with a type checker. It is the task of the type checker to ensure that the types and the code are in sync throughout. For example, if a program states that some variable always stands for numbers but also contains an assignment statement that sets this variable to some object, the type checker must reject this program.

For the languages that *Classy* and *Module* model, creators developed three different approaches to type system design and implementation:

- nominal class types;

  C++, Java, and similar languages identify the name of a class as the type of its instances. Thus, if two classes have identical fields and identical methods, their objects are still of a distinct type.

- structural class types;

Academic language research tend to favor the alternative idea that if two objects have the same fields (by name and type) and the same methods (by name and signature), they have the same type. That is, the structure of the class definition gives rise to its type, not its name.

- module-global type inference.

  In the 1970s, the ML programming language introduced the idea that programmers could have a type system without having to write down (all) the types. The OCaml variant of ML includes a class-based language and modules not unlike the those in our *Module* model.

  ML's type system replaces missing types with type variables, uses the code in a module to set up equations in these type variables, and attempts to solve the equation system:

  – If there is *one* solution, the variables can be replaced with plain types, and the compiler can generate conventional code.
  – If there are *many* solutions, the type system assigns polymorphic types to some of the functions. On occasion, such polymorphic type assignments are desirable; in other cases, it causes the compiler to create slow-running code.
  – If there is *no* solution, type checking fails and the type inference systems tries to compose an type-error message. To this day, though, these systems tend to report such errors in terms that are often difficult to decipher for a software developer.

  See "Typeful Programming," a DEC SRC research report (1989, 1993).

  Note By the early 1990s, Luca Cardelli, a leader of the types research community, began to argue that when types are not written down, several software engineering steps suffer, starting with the systematic design of data representations and their functionality all the way to documentation. Many others began to embrace this point of view and, as a result, *module-level* type inference experienced a decline in popularity. Modern languages use *local* type inference instead.

The remainder of this chapter focuses mostly on the second idea, in preparation of the next chapter. It ignores module-level type and explains local type inference as an aside.

No matter which approach is chosen, the creator of a type system must confront the question whether to ensure that the type checker—which is essentially is just another validity checker—is supposed to be sound in the spirit of section 4.1 in chapter IV. While checking whether a variable in

an expression is declared is essentially trivial, checking types is a rather complex task. Indeed, in some cases it is so complex that some language creators opt to "do their best" but make no effort to state or prove a theorem about the relationship between the type checker and the language's semantics.

If the language creators aim for a mathematical soundness theorem, they tend to set up a formal *model* like those of this book, state the theorem for the validity checking and semantic functions of model, and prove it with logical means. Critically, they do not work with the language implementation, which tends to support many more linguistic constructs than a model. As a result, even if the goal is to deliver a typed language that satisfies a type soundness theorem, the implementation may come with bugs that violate the theorem.

As for their connection to pragmatics, type soundness plays a role but perhaps not the critical role that language theoreticians ascribe to this theorem. Hence, this chapter presents the relationship between pragmatics and languages with both sound as well as unsound type systems. Readers interested in type soundness may wish to check out traditional text books on programming languages.

1.2 Costs and Benefits

Although the dominance of C++ and Java in the marketplace seemed to have settled the argument in favor nominal types, the emergence of TypeScript and the addition of similar type systems to other languages resurrected the structural-class type idea. Both come with their own costs and benefits, and it is worth to compare those briefly.

From the perspective of a software developer and plain language user, the cost of using a type system comes in one form: a reduction in expressive power. A developer notices this loss in two ways: extensibility and safety/security. Any type system imposes restrictions on code. This statement is a tautology, because the very point of a validity checker is to eliminate code that *may* suffer from mistakes. Due to the undecidable nature of these problems, the emphasis is on "may" as in "a validity checker may reject code that, if type checking were disregarded, runs fine and computes the desired result."

This decrease in expressive power affects developers in rather different ways, depending on whether they use a language with a nominal or a structural type system. Most importantly, programs in the latter remain extensible in ways that the former cannot accomplish. When one developer

programs to a structural type, another developer can create a module with a structurally equivalent class that comes with a different name. The old code will work just fine with the new kinds of objects. By contrast, a software system in a nominally typed language forces the second developer to use the actual class to which the first one programmed; if this class isn't accessible, the software system is not extensible.

If the type system also supports sub-typing, structural types make functionality even more reusable than nominal typing. As long as an object has the required fields and methods that some method demands, everything works out, even if the object has additional fields and methods.

On the flip side of extensibility, software developers are concerned with safety and security. With a nominal type system, a developer can specify that only instances of one particular class are welcome inside some piece of functionality. Given some knowledge about the workings of this class's implementation, the developer knows that calling an instance's method cannot do any harm. Contrast this situation with a structurally typed language. Just because the type signatures of two methods agree does not mean their behavior agrees. While one may compute the desired result, another one may raise an exception, violate a security policy, or perform any other undesirable effect. A developer can know how a method behaves only its code resides in a particular known class.

This last concern, safety and security, shows that restrictions on the expressive power come with benefits in terms in reasoning about code. In this concrete context, while a type system restricts the space of well-formed and valid code, it empowers the developer to draw firm conclusion about the general behavior of code, not just about certain concrete runs. And this holds regardless of whether a language creator chooses a nominal or a structural type system; a nominal type system just happens to be more restrictive than a structural one, and therefore it is a bit easier to reason about by a language user.

From the perspective of a language creator, the design and implementation of a typed language require significantly more work than that of an untyped language:

- The design of a type system involves a modicum of syntactic work, namely coming with a notation for types.

- Next comes the development of type judgments and rules that establish such judgments. Theoreticians write down these rules with a notation borrowed from logic; language creators write them down in

English so that language users can understand them and, critically, understand error messages from the type system.

- If the language creators aim for a soundness theorem about the type system, they need to make mathematical models of both the type system and the semantics, develop a relationship, and formulate an argument as to why this work scales to the full language. Reasonable language creators tend to leave this work to theoretical language researchers in academia.

- Finally, the validation pass must be implemented. Checking its compliance with the informal rules requires extensive testing work. Compliance means that the informal rules and the actual algorithms are in sync. And extensive testing is currently the only method for arguing compliance for a reasonably modern and large language. Furthermore, the implementation will inevitably come with errors that violate the specification. Hence the test suite must be carefully curated and maintained for this complex piece of code.

Choosing a nominal approach simplifies this work somewhat. Starting with the typing judgments and their rules, designing a nominal type system is simpler than designing a structural one. Just consider what it means for types to match: on one side, we have a name comparison and, on the other, a comparison of complex structures. Unsurprisingly, the translation of the respective rules into type checking code is more troublesome for structural systems when compared with nominal ones.

The final consideration concerns the semantics—the generated run-time code to be precise—of typed languages when compared to languages without type system. When code comes with explicit type specification for program variables, a compiler can exploit this information for the generation of code, including the allocation of data structures, the move of data from one place to another, or the operations that manipulate the data. By contrast, a compiler for an untyped language must use approximation algorithms to infer comparable information about a given piece of code.

While this assistance for compiler writers motivated type systems in the early days of programming languages, the discovery of the soundness concept cast some doubt on the use of type information. If the validity checker for types is *unsound*, a compiler may generate run-time code that eventually moves bits from a string into an integer-typed variable or cause similar problems. As a result, the majority of language creators began to aim for

type soundness, though to this day, some opt for designing a "usable" type system without consideration for soundness.

2 Project Language: *Types*

Turning the *Module* model into a typed one mostly follows the design and implementation plan of the preceding section:

- The first step is to design a notation for types and to equip the existing grammar with type declarations.
- The second step concerns the meaning of "keeping code and types in sync." Academic research has come to use *judgment rules* for this purpose, which provide a concise notation for expressing a relationship.
- Finally, the last step is to create a system of rules that explains how to judge. Once type theoreticians noticed the relationship between the study of logics and the study of type systems, these rules have been stated as derivation rules also known as inference rules. Each rule states that some piece of code is in sync with its type and relative to its context—typically called a conclusion—if it is possible to make some other judgments, called antecedents.

The remaining subsections present these three steps, respectively.

```
Type       ::= REAL | Shape
Shape      ::= ((FieldType*) (MethodType*))
FieldType  ::= (FieldName Type)
MethodType ::= (MethodName (Type*) Type)
```

Figure 46: *Types*: the Notation

2.1 *Types*: the Notation

Types classify the results and effects of expressions, statements, and so on. In the context of *Module*, there are essentially two such properties:

- an expression may yield a number, or

- an expression may yield an instance of a class.

All other phrases merely play a role in creating one of these outcomes.

Accordingly a type notation must define a type as one of two possibilities: a literal symbol denoting "numeric outcome" (REAL) and a production describing an object, which we call a Shape: see figure 46. Since an object consists of fields and methods, the Shapes that describe them consist of two parts: one describing the types of fields and another describing the types of methods.

Exercise 50. Explain what kind of objects the following Shapes describe:

```
( ((x REAL) (y REAL))
  ((distanceToOrigin () REAL)) )
```

Define two distinct classes whose instances intuitively have this type.

Note These types, in particular Shapes, describe the *structure* of an object. It does not come with any information about the class to which it might belong. The latter—including a ClassName into the type as a determining element—is the alternative used in *nominal* type systems.

2.2 *Types*: the Grammar

The *Types* differs from *Module* in two ways. First, modules use the keyword tmodule, because we eventually wish to model a programming language that comes with typed and untyped pieces of code. Marking those with distinct keywords helps future readers. Second, modules in *Types* contain one more piece of code than those in *Module*: a Shape at the end, which specifies the type of the objects that the module's class can generate.

```
TypedSystem ::= (TypedModule*
                 Import*
                 Declaration*
                 Statement*
                 Expression)

TypedModule ::= (tmodule ModuleName Import* Class Shape)
```

Figure 47: *Types*: the Grammar

Here is the BNF of *Types*: see figure 47. Stop! Compare this BNF with the one of *Module* in the previous chapter.

You might wonder why this grammar does not come with typed variable declarations. In many languages, a developer would have to write def x : int = 4 to declare an integer variable, but here, variable declarations come without types.

Following contemporary developments in programming languages, we instead have the type checker *compute* the missing type from the right-hand side of a def construct. In *Types*, computing this type is feasible and easy because the expression sub-language is quite simplistic. Language researchers refer to this process as *local type inference*. A fair number of contemporary programming languages incorporate local type inference for a number of cases.

Note Local type inference radically differs from the module-global inference mentioned in section 1.1 in this chapter. In our model, computing the type is always possible. In real-world programming languages, local inference rarely fails; when it does, generating an appropriate error message is straightforward, though repairing the expression so that inference succeeds rarely is.

Exercise 51. Design an AST data representation for *Types*. Implement a parser for *Types* that maps an S-expression to an instance of AST. Re-use your solution from exercise 43 as much as possible.

2.3 *Types*: Validity

The *Module* comes with a number of validity rules. Adapting those rules to the new setting, *Types*, is a matter of ignoring the types. However, these rules also induce a couple of rules for the model of this chapter:

- Any two FieldNames in the Shape of a module must be distinct.
- Any two MethodNames in the Shape of a module must be distinct.

Clearly, these rules just correspond to the validity rules concerning Classes in the *Module* model. To simplify type checking, we impose a third rule:

> *The FieldNames in the class and those in the type of the class must show up in the same order.*

It is easier to check this constraint than to type-check classes or new expressions with a loose ordering on field names. By contrast, it makes no difference whether methods and method types are supplied in the same order.

Hence, to practice program design, consider examples that list methods and method types in distinct orders.

Exercise 52. Design and implement a validity checker that enforces the validity rules for *Types*. The checker consumes error-free ASTs from exercise 51; if it finds errors it annotates the AST appropriately.

3 Type Checking

A type checker is the most sophisticated validity checker. Language researchers have developed a distinct notation for specifying type checkers. This notation heavily leans on ideas and notations from the study of formal, also known as mathematical, logic.

Roughly speaking, a type checker *judges* each kind of phrase in a programming language. Since such phrases occur in syntactic contexts and since judging them usually needs information about this syntactic context, each judgment consists of at least two parts: information about the context and the piece of code. For some kind of phrases a judgment also includes a third part, namely an inferred kind of information.

Let's make this a bit more concrete in the context of *Types*. Consider the case of an expression, say (y + z). Such an addition succeeds only if y and z are going to evaluate to a number; if either one of them evaluates to an object, the addition fails. Hence, the type checker needs a classification of variables, which originates from the context. Furthermore, if the expression successfully computes a result, it is going to be a number; the type checker can thus classify the expression as REAL. Here is the formal way of expressing this judgment:

$$\{\textit{type information about } y \textit{ and } z\} \vdash (y + z) \Rightarrow \text{REAL}$$

Mathematics must be pronounced to be comprehended, and this "mathematical sentence" is best spelled out as "given type information about y and z from the program context of (y + z), the type checker classifies the expression as REAL."

Now consider an assignment statement, (x = (y + z)). In order to judge such a statement, the type checker needs to know the types of x, y, and z. The first one is needed to ensure a type-correct modification of x's value; the last two are needed to check the addition expression. In contrast to expression checking, though, checking a statement does not produce any new type information. Hence the formal statement is simpler:

$\{$*type information about x, y, and z*$\} \vdash (x = (y + z))$

Some judgments are plain wrong. Take $\varnothing \vdash (y + z) =>$REAL. This example states that without any information about the syntactic program context of the expression, the type checker should judge the expression as valid and classify it as REAL. Clearly, this specification would lead to an *unsound* validity check for types. So the question is how to come up with a system of *sound* judgments.

Following logicians, programming language researchers use so-called *judgment derivation rules* also known as *type inference rules*. We prefer the former term to avoid any confusion with local or module-global type inference, which is different concept.

In general, a derivation rule has the following shape:

some contextual information $\vdash$ one judgment
...
more contextual information $\vdash$ another judgment
$\_\_\_\_\_$ [generic rule]
contextual information $\vdash$ piece of code $\Longrightarrow$ Result

The judgments above the horizontal line are called *antecedents*. If the use of the derivation rules can validate all of these antecedents, then the judgment—called the *conclusion*—below the line is validated.

$\_\_\_\_\_$ [numerical literal]
SClasses, TVar $\vdash$ n $\Longrightarrow$ REAL

x has type REAL $\vdash$ x $\Longrightarrow$ REAL
y has type REAL $\vdash$ y $\Longrightarrow$ REAL
$\_\_\_\_\_$ [+]
x and y have type REAL $\vdash$ (x + y) $\Longrightarrow$ REAL

Figure 48: Two examples of judgment derivation rules

An *axiom* is a derivation rule without antecedents. Take a look at the first rule in figure 48, which is an axiom. By convention, n is a numeric literal, and the type checker should classify it as such. The second rule

in the same figure illustrates what a typical simple derivation rule looks like. Its antecedents extract from the contextual information that each of the variables has type REAL, and therefore the addition is going to produce a REAL, too.

3.1 Contextual Information and Conventions

Just like the specification of an abstract machine, the specification of a type checker relies on a fair number of notational conventions. The point of those is to present the specifications in a reasonably concise manner.

meta-variable	standing in for
Modules	a collection of modules
SClasses	Shapes of Classes, a mapping from ClassNames to Shapes
TVar	Types of Variables a mapping from Variabless to Types
T	any type
S	Shape type
Tf	the type of a field
Tm	the type of a method
sys	a system
mod	a module
imp	an import specification
cls	a class
meth	a method

Figure 49: Type Conventions

Figure 49 presents the conventions used here for both the statement of judgments and derivation rules. The first three name the most common pieces of contextual information: (1) sets of modules, (2) maps from ClassNames to Shapes, and (3) maps from variables to Types. The remaining ones are meta-variables for types and pieces of code. As you tackle this section's project, keep figure 49 in mind and consult it as needed.

3.2 Type Judgments

Figure 50 presents the complete collection of judgments that the type checker makes. These judgments are grouped into several collections: (1) for systems-

level pieces of code, (2) for class-level ones, (3) for the core of the language, which consists of declarations, statements, and expressions.

The entry point is the very first line, a judgment that merely states that a complete system is valid. It relies on the three following judgments, which, in turn, validate:

1. an individual module, given a set of modules as syntactic context;
2. a sequence of import specifications, which produces an SClasses from a set of modules; and
3. an individual import specification, which given both a set of modules and an SClasses, adds information to the latter.

All other judgments follow this schema. They either just validate a piece of code, given some information about the syntactic context in which they appear. Or, they validate and produce additional information, which is often added to the context.

For a last example, consider the judgment for validating a method. Its context consists of three pieces: (1) the mapping from ClassNames to Shapes; (2) the mapping from Variables to Types; and (3) the signature of the method. The last piece of information is needed so that the type checker can add the method's parameters and their respective types to the given TVar and compare the type of the return expression with the specified return type.

Exercise 53. Jointly with a programming partner, read the table in figure 50 and pronounce them out loud.

3.3 Rules for Deriving Type Judgments

Derivation rules make up the final part of the specification of a type system. As the preceding section explained the derivation rules for *Types* come in two distinct flavors and there is a fair number of them.

Figure 51 displays the derivation rules for system-level pieces of code, excluding the rules for the system's body. The first one tells a programmer that a system is valid according to the type checker if n+2 antecedents hold:

- The n modules in the system are valid relative to the modules that precede each module. Naturally the first module has to be valid in an empty set of modules.
- Next, the sequence of imports setting for the system's body must be valid and produce an SClasses mapping.

read judgment as	the formal judgment
systems and modules	
a system is valid	$\vdash$ sys
a module is valid	Modules $\vdash$ mod
a sequence of imports produces an SClasses	Modules $\vdash$ imp_1 ... imp_I $\Longrightarrow$ SClasses
an import adds information to the given SClasses	Modules, SClasses $\vdash$ imp $\Longrightarrow$ SClasses
classes and methods	
a class is valid	SClasses $\vdash$ cls
a method is valid	SClasses, TVar, ((T_a ...) T_r) $\vdash$ meth
bodies of blocks, systems, and methods	
a program produces a type T	SClasses $\vdash$ def* stmt* exp $\Longrightarrow$ T
a block is valid	SClasses $\vdash$ (block def* stmt*)
declarations	
a sequence of declaration produces a TVar	SClasses, TVar $\vdash$ def_1 ... def_d $\Longrightarrow$ TVar
a declaration adds information to the given TVar	SClasses, TVar $\vdash$ (def x exp) $\Longrightarrow$ TVar
statements	
a sequence of statements is valid	SClasses, TVar $\vdash$ $stmt_1$ ... $stmt_n$
one statement is valid	SClasses, TVar $\vdash$ stmt
expressions	
an expression produces a type T	SClasses, TVar $\vdash$ exp $\Longrightarrow$ T

Figure 50: Type judgments

- And finally, the system's body must type-check, and its result type must be numeric.

This last antecedent illustrates how the development of such rules is somewhat whimsical. We could easily have allowed the system's body to return any type, but focusing on number facilitates the comparison of implementations. Even real-world type systems incorporate such decisions, though for different reasons than ours.

$\varnothing \vdash \text{mod}_1$
for all i from 2 to K:
  $\{\text{mod}_1, ..., \text{mod}_{i\text{-}1}\} \vdash \text{mod}_i$
$\{\text{mod}_1, ..., \text{mod}_K\} \vdash \text{imp}_1 ... \text{imp}_I \implies \text{SClasses}$
$\text{SClasses}, \varnothing \vdash \text{def*}\ \text{stmt*}\ \text{exp} \implies \text{REAL}$

$\vdash (\text{mod}_1 ... \text{mod}_K\ \text{imp}_1 ... \text{imp}_I\ \text{def*}\ \text{stmt*}\ \text{exp})$ [system]

$\text{Modules} \vdash \text{imp}_1 ... \text{imp}_I \implies \text{SClasses}$
$\text{SClasses}\ [\text{C} : \text{S}] \vdash (\text{class C (f ...) meth ...})$

$\text{Modules} \vdash (\text{tmodule MN}\ \text{imp}_1 ... \text{imp}_I\ (\text{class C (f ...) (meth ...)})\ \text{S})$ [module]

$\text{SClasses}_0 = \varnothing$
for all i from 1 to I:
  $\text{Modules}, \text{SClasses}_{i\text{-}1} \vdash \text{imp}_i \implies \text{SClasses}_i$

$\text{Modules} \vdash \text{imp}_1 ... \text{imp}_I \implies \text{SClasses}_I$ [imports]

M is the name of mod in Modules
C is the name of the class defined in mod
S is the Shape of C in mod

$\text{Modules}, \text{SClasses} \vdash (\text{import M}) \implies \text{SClasses}\ [\text{C} : \text{S}]$ [an import]

Figure 51: Typing rules for systems and modules

The second rule specifies how an individual module is validated. Its sequence of import specifications produce an SClasses, which is then used to type-check the (one and only) class in the module. Note how the SClasses is first extended with an entry for C, the defined class, so that the type checker knows what the expected the Shape of the class is. It is a classic example of how syntactic context is incorporated into the contextual information to the left of the $\vdash$ symbol in a judgment.

Let's take the last two rules together. Rule 3 tells us that the type checker needs a function that turns a set of modules and a series of import specifications into a table from names of classes to their shapes. Starting with an empty table, the I antecedents gradually build up SClasses, one at a time. The result analyzing the last import specification, SClasses_I is the result for

the entire sequence of imps. Rule 4 specifies how the type checker must deal with a single import specification. Given the name of the module, it uses the given set of modules to retrieve the name of the class and its type: C and S, respectively. The pair is added to the given table, possibly replacing an existing entry for C.

$$\frac{\begin{array}{l}\text{SClasses[C] = S}\\ \text{S = (((}f_1{}^T\ Tf_1\text{) ... (}f_g{}^T\ Tf_g\text{)) ((}m_1{}^T\ Tm_1\text{) ... (}m_k{}^T\ Tm_k\text{)))}\\ f_1, ..., f_g = f_1{}^T, ... f_g{}^T\\ \{\text{NameOf[meth}_i\text{]}, ..., \text{NameOf[meth}_k\text{]}\} = \{m_1{}^T, ..., m_k{}^T\}\\ \text{for all } i \text{ from 1 to } k\text{:}\\ \quad \text{SClasses, } \varnothing\ \text{[this : S] , Tm}_j \vdash \text{meth}_i\ \text{, if NameOf[meth}_i\text{] is m}_j\end{array}}{\text{SClasses} \vdash \text{(class C (}f_1 \text{ ... } f_g\text{) meth}_1 \text{ ... meth}_k\text{)}}\ \text{[class]}$$

$$\frac{\text{SClasses, TVar [}y_1 : T^a{}_1\text{] , ... , [}y_n : T^a{}_n\text{]} \vdash \text{def* stmt* exp} \implies T_r}{\text{SClasses, TVar, ((}T^a{}_1 \text{ ... } T^a{}_n\text{) } T_r\text{)} \vdash \text{(method m (}y_1 \text{ ... } y_n\text{) def* stmt* exp)}}\ \text{[method]}$$

Figure 52: Typing rules for classes and methods

Figure 52 presents the most complex derivation rule for *Types*. To check whether a class named C and its type are in sync, the type checker is given a class-shape table, which contains the type for the to-be-checked class. In this context, it proceeds as follows according to the antecedents:

1. It retrieves the Shape S of C from the given table.
2. As for fields, C and S must contain the same *sequence* of FieldNames.
3. As for methods, C and S must contain the same *set* of MethodNames.
4. Once these two conditions are satisfied, the type checker is going to deal with each method. Checking a method uses three pieces of context information: (a) the given SClasses; (b) a type-variable table, populated with an entry for this/S; and (c) the appropriate method signature from S.

Stop! Compare this explanation with the mathematical specification in the first rule of figure 52. Explain 4(b) to a programming partner.

The second derivation rule in figure 52 shows how to type-check methods. The type-variable table is extended with one entry per parameter with its matching type from the given method signature—assuming the two sequences are of the same length. Given these two tables, the type checker uses the judgment for method and program bodies to confirm that the type of the final expression equals the return type from the method signature.

$$\frac{\begin{array}{l}\text{SClasses, TVar}_0 \vdash \text{def*} \implies \text{TVar}_1 \\ \text{SClasses, TVar}_1 \vdash \text{stmt*} \\ \text{SClasses, TVar}_1 \vdash \text{exp} \implies \text{T}\end{array}}{\text{SClasses, TVar}_0 \vdash \text{def* stmt* exp} \implies \text{T}}\ [\text{body}]$$

$$\frac{\begin{array}{l}\text{SClasses, TVar}_0 \vdash \text{def*} \implies \text{TVar}_1 \\ \text{SClasses, TVar}_1 \vdash \text{stmt*}\end{array}}{\text{SClasses, TVar}_0 \vdash \text{(block def* stmt*)}}\ [\text{block}]$$

Figure 53: Typing rules for bodies of blocks, systems, and methods

A system's body has the same shape as a method's body; in comparison a block lacks the final expression. The derivation rules for these grammatical production resemble each other. As figure 53 shows, both require type-checking the sequence of definitions first, and doing so enriches the given type-variable table. Next up, this extended table is used to check the sequence of statements. In the case of system and method bodies, the type checker also type checks the final expression, and whatever type this check produces, also becomes the type of the complete body.

By implication, type-checking a sequence of declarations has two distinct goals. First, the type checker needs to extend the given type-variable table with the types of the newly declared variables, one declaration at a times. The "declarations" rule in figure 54 specifies how this step works. Second, the type checker must ensure that the right-hand side expression has a type, which is then used as the type of the left-hand side variable. See the "one declaration" rule in the same figure.

Figure 55 displays the rules for sequences of statements and individual statements. Unsurprisingly, a sequence of statements is valid if each individual statement type checks. To get a sense of how individual statements

for *i* from 1 to d:
SClasses, $\text{TVar}_{i\text{-}1} \vdash (\text{def } x_i \text{ exp}_i) \implies \text{TVar}_i$

$\text{SClasses}, \text{TVar}_0 \vdash (\text{def } x_1 \text{ exp}_1) \ldots (\text{def } x_d \text{ exp}_d) \implies \text{TVar}_d$ [declarations]

$\text{SClasses}, \text{TVar}_0 \vdash \text{exp} \implies \text{T}$

$\text{SClasses}, \text{TVar}_0 \vdash (\text{def x exp}) \implies \text{TVar}_0[\text{x} : \text{T}]$ [one declaration]

Figure 54: Typing rules for declarations

are type checked, take a look at the "assignment" rule. Its antecedent tells the implementer of the type checker that the expression on the right-hand side must type check and that its type must be the same as the type associated with the left-hand variable in the type-variable table.

Consider the derivation named "loop" next. The CESK semantics of (while0 tst bdy) runs the loop if the value of the tst is 0; any other value causes the loop to terminate. In terms of types, the tst expression may have *any* type, and bdy must merely type-check as a statement.

Exercise 54. Languages such as Java demand that conditionals and looping constructs have Boolean-typed expressions in the analogous positions. Assuming *Types* came with a Boolean type plus true and false as literal constants of this type, how would you reformulate the "loop" and "conditional" rules in figure 55?

At this point, we have covered most derivation rules; the ones for expressions are missing. For the latter, it is best to proceed in two steps, starting with the expressions from *Core*, followed by those related to *Classy*.

Figure 56 covers the first set of rules: literal constants, variables, additions, divisions, and comparison. The first and the last rule are axioms, i.s. they validate that these expressions produce the type unconditionally. In both cases, the resulting type is REAL.

Exercise 55. Some typed programming languages do not implement unconditional comparison operators. Instead their type system ensures that both sides of == have the same type. Modify the "==" rule in figure 56 to model this kind of type checker.

for all i from 1 to n:
  SClasses, TVar $\vdash$ stmt_i

SClasses, TVar $\vdash$ stmt_1 ... stmt_n [statements]

TVar[x] = T; SClasses, TVar $\vdash$ exp $\Longrightarrow$ T

SClasses, TVar $\vdash$ x = exp [assignment]

SClasses, TVar $\vdash$ tst $\Longrightarrow$ T
SClasses, TVar $\vdash$ thn
SClasses, TVar $\vdash$ els

SClasses, TVar $\vdash$ (if0 tst thn els) [conditional]

SClasses, TVar $\vdash$ tst $\Longrightarrow$ T ; SClasses, TVar $\vdash$ bdy

---

SClasses, TVar $\vdash$ (while0 tst bdy) [loop]

SClasses, TVar $\vdash$ exp $\Longrightarrow$ T
TVar[o] = $(((\text{f}_1{}^T\ \text{Tf}_1)\ \ldots\ (\text{f}_g{}^T\ \text{Tf}_g))\ (\text{Tm}_1{}^T\ \ldots\ \text{Tm}_K{}^T))$
there exists an i such that
  $\text{f} = \text{f}_i{}^T$ and Tf_i = T

SClasses, TVar $\vdash$ (o –> f = exp) [field mutation]

Figure 55: Typing rules for statements

The remaining three rules have straightforward explanations. Since the validity checkers ensure that all variable occurrences refer to some variable declaration, a variable always type checks and its type is the one from the given type-variable table. When it comes to addition and division, our *Types* model's semantics demands that both operands are numbers; the two derivation rules are set up accordingly.

A language such as Type Script would allow the addition of numbers and strings, and its type checking rules need to accommodate this semantics.

Figure 57 covers the second set of derivation rules, those for expressions that deal with objects. A (new C(x)) expression is going to create an instance of C and therefore is going to have its shape—if it succeeds. The antecedents of the "new" rule tell us that SClasses comes with a Shape type S for C, which in turn, specifies the number and the types of the class's fields.

$$\frac{}{\text{SClasses, TVar} \vdash \text{n} \Longrightarrow \text{REAL}} \text{[numerical literal]}$$

$$\frac{\text{TVar[x] = T}}{\text{SClasses, TVar} \vdash \text{x} \Longrightarrow \text{T}} \text{[variable]}$$

$$\frac{\text{TVar[x] = REAL; TVar[y] = REAL}}{\text{SClasses, TVar} \vdash \text{(x + y)} \Longrightarrow \text{REAL}} \text{[+]}$$

$$\frac{\text{TVar[x] = REAL; TVar[y] = REAL}}{\text{SClasses, TVar} \vdash \text{(x / y)} \Longrightarrow \text{REAL}} \text{[/]}$$

$$\frac{}{\text{SClasses, TVar} \vdash \text{(x == y)} \Longrightarrow \text{REAL}} \text{[==]}$$

Figure 56: Typing rules for arithmetic expressions

In order for new to succeed, the number of arguments must be the same as the number of fields, and according to the third antecedent, the types of the arguments to new must equal the types of the fields specified in S.

When it comes to the derivation rule for isa expressions, a language creator has to make a choice. Given (o isa C), it is possible to determine the types of both o and C. Let's call them S_o and S_C, respectively. Based on this context, a type-system designer can choose from three alternatives:

1. to add $S_C = S_o$ to the antecedent;
2. to require that both are Shapes but not necessarily identical ones; or
3. to specify that S_C is a Shape without imposing any restrictions on o.

In the specific context of *Types*, alternative 1 make sense. Unless the two shapes are the same, the instance check is guaranteed to fail; it can succeed only if the two Shapes are the same *and* o is an instance of C—which cannot be decided by the type checker. All class-based languages come

$$\frac{\text{SClasses[C]} = \text{S} \quad \text{S} = (((\text{f}_1{}^T\ \text{Tf}_1)\ \ldots\ (\text{f}_g{}^T\ \text{Tf}_g))\ (\text{Tm}_1{}^T\ \ldots\ \text{Tm}_K{}^T)) \quad \text{TVar[a}_1\text{]} = \text{Tf}_1, \ldots, \text{TVar[a}_g\text{]} = \text{Tf}_g}{\text{SClasses, TVar} \vdash (\text{new C } (\text{a}_1\ \ldots\ \text{a}_g)) \Longrightarrow \text{S}} \quad [\text{new}]$$

$$\frac{\text{SClasses[C]} = \text{S}_C;\ \text{TVar[o]} = \text{S}_o}{\text{SClasses, TVar} \vdash (\text{o isa C}) \Longrightarrow \text{REAL}} \quad [\text{isa}]$$

$$\frac{\text{TVar[o]} = (((\text{f}_1{}^T\ \text{Tf}_1)\ \ldots\ (\text{f}_g{}^T\ \text{Tf}_g))\ (\text{Tm}_1{}^T\ \ldots\ \text{Tm}_K{}^T)) \quad \text{there exists an } i \text{ such that } \text{f} = \text{f}_i{}^T}{\text{SClasses, TVar} \vdash (\text{o} \text{ -> } \text{f}) \Longrightarrow \text{Tf}_i} \quad [\text{get}]$$

$$\frac{\text{TVar[o]} = ((\text{FT}_1{}^T\ \ldots\ \text{FT}_1{}^T)\ ((\text{m}_1{}^T\ \text{Tm}_1)\ \ldots\ (\text{m}_k{}^T\ \text{Tm}_k))) \quad \text{there exists an } i \text{ such that } \text{m} = \text{m}_i, \ \text{Tm}_i = ((\text{T}_1\ \ldots\ \text{T}_n)\ \text{T}_r), \text{ and TVar[a}_1\text{]} = \text{T}_1, \ldots, \text{TVar[a}_n\text{]} = \text{T}_n}{\text{SClasses, TVar} \vdash (\text{o} \text{ -> } \text{m } (\text{a}_1\ \ldots\ \text{a}_n)) \Longrightarrow \text{T}_r} \quad [\text{call}]$$

Figure 57: Typing rules for object-oriented expressions

with subtyping though, and a model of those setups would invalidate the rationale for the first choice.

Alternative 2 models a Java-like type check, even though the type system for *Types* is structural in contrast to Java's nominal one. It is the one stated in figure 57, because it is widely used. By contrast, alternative 3 resembles what the type system and semantics of Type Script implements. As long as the type of o is remotely sensible, the type checker validates the expression and leaves it to the underlying machine to decide whether o is an instance of the given class.

To bridge the gap between the type systems of Types and TypeScript, we would need to enrich the former with many more kinds of values and their types.

The derivation rule for get comes with two antecedents: one for retrieving the type of o, the target object; another one for retrieving the type of the referenced field. Here, the first antecedent provides access to the names and types of the fields and methods, respectively. Since the only way for o to come with a shape type is by [class] and [new], we know that its shape comes

with a set of field names. Hence, the second antecedent may demand that one element of this set is equal to the FieldName in the expression. The result type of o –> f is the type of the respective field.

Exercise 56. Explain the last derivation rule, [call], in the style of this section. **Hint** This rule combines aspects of [new] and [get].

3.4 Implementing a Type Checker

In the context of a model, an implementation of the derivation rules should aim for a line-by-line transliteration of the antecedents and consequence. With the use of auxiliary functions or methods, reaching this goal is straightforward. Often these auxiliary methods or functions come in handy in several contexts.

Consider the [system] derivation rule from figure 51. While the preceding section correctly suggests n+2 antecedents, a programmer may identify four groups:

1. checking the base case, namely just the first module;
2. checking the remaining K-1 modules in the context of the preceding ones;
3. extracting from the imports which classes and their types are available; and
4. checking the system's body and ensuring that its result type is REAL.

Recognizing this grouping of antecedents suggests how to organize the corresponding type-checking method for a complete system.

Figure 58 sketches a Java implementation of a type checking method for a complete system. It assumes that each stage comes with its own data representation. Since the type checker of our model runs after the validity checker has confirmed a number of basic properties, the AST classes with the type checker methods represent well-formed and valid ASTs, hence, the name WFandValidSystem. The key method needed for each such class is typeCheck. As the purpose statement in figure 58 says, the method type checks an instance of this system AST, throwing a runtime exception if it discovers a violation of the [system] rule itself or any of its antecedents.

Next take a look at the signature of the typeCheck method. It matches the format of the [system] judgment. For judgments such as this one, the method consumes no inputs, because the judgment has no contextual information

```
class WFandValidSystem {
  WFandValidModules modules = ...;
  WFandValidImports imports = ...;
  WFandValidBody    body = ...;

  // check the type of `this` complete system
  // EFFECT throw an exception if a violation is discovered
  void typeCheck() throws TypeError {
    this.modules.select(1).typeCheck(this.selectPreceding(0));

    for(i = 2; i <= modules.size(); i++) {
      this.modules.select(i).typeCheck(this.selectPreceding(i-1));
    }

    SClasses sc = this.imports.typeCheck(this.modules);

    Type actual = this.body.typeCheck(sc,new TVar());
    if (!actual.equals(TypesAST.REAL)) // the type for numbers
       throw new TypeError("wrong result type for system body");

    // ------------------------------------------------------------------
    return ;
  }

  // create a container from the first `i` modules in `this` system
  Container<WFandValidModule> selectPreceding(int i) {
    ...
  }

}
```

Figure 58: A Java-style implementation of the [system] derivation rule

to the left of $\vdash$; since it does not produce a result, the typeCheck method has a result type of void in Java. By contrast, for judgments such as [imports] that do require information about the context and that do produce a result, the signature of the corresponding method specifies inputs and outputs types. In the case of imports.typeCheck, the method must have access to the (well-formed and valid) ASTs of all modules so that it can create a mapping from ClassNames to their types, and its result is going to be the mapping. The boxed expression in figure 58 displays an illustrative call to this method.

With all these ideas in place, matching the method body of typeCheck with the judgment derivation rule is straightforward. Like the rule itself, the method body consists of four pieces. The horizontal rule in a comment indicates the separation of the antecedents from the part of the conclusion

that is the result of the judgment. Blank lines separate the four transliterations of the antecedents from each other. Each block corresponds to one of the antecedents. The last one uses three lines to explicate the steps involved in the use of this judgment: (1) compute the actual, derived type; (2) compare it with the expected type; and (3) signal an error if this comparison fails. This last part is implied in judgment derivation rules—when antecedent judgments do not hold, the conclusion cannot hold either. An implementation may turn this implicit idea into an exception. In this example, it is the only exception that the method raises directly; all other exceptions would come from calls to the auxiliary type-checking methods.

While the method clearly mirrors the specification—the [system] rule—this programming style comes with a cost. Notice how the method calls the (local and private) selectPrecedingModule method as many times as it typechecks an individual module. The purpose of the method is to create a container from the first *i* modules in this instance of the class. Although implementing this method poses no problem, calling it creates an inefficiency—acceptable for a programmer who creates an executable model, but unacceptable for a developer who creates an efficient type checker for *Types*.

Exercise 57. Can the first two blocks of typeCheck in figure 58 be merged? If so, how? If not, why not?

Exercise 58. How would you go about eliminating the inefficiency of repeatedly calling selectPrecedingModule? How would this distort the relationship between the specification via a rule and the implementation?

Figure 59 sketches the implementation of the [imports] rule. Recall its purpose: it specifies how a sequence of imports creates an instance of SClasses, a mapping from ClassNames to Shapes. As the purpose statement of the class says, the type-checking method for [imports] is located in the representation of such sequences. In Java, such a class may delegate to a collection class, say ArrayClass; this kind of detail does not matter here.

Like the judgment for [imports] itself, which assumes knowledge about all modules in the context of the imports sequence, the typeCheck method in WFandValidImports consumes one value: a container of modules. Its signature tells a future reader that it computes an instance of SClasses.

The method body of typeCheck consists of two blocks, each of which implements one of the antecedents in the corresponding derivation rule. Also notice how these two pieces are of the same size as the corresponding antecedents. With the first block, the method creates an empty instance of SClasses. The second block of code uses a Java-style for each loop to iterate through the sequence of import specifications, type checking one at a time.

```
// represents a sequence of import specifications
class WFandValidImports {
  WFandValidImports imports = ...;

  // check the type of `this` sequence of imports
  // EFFECT throw a runtime exception if a violation is discovered
  SClasses typeCheck(WFandValidModules modules) throws TypeError {
    SClasses sc = new SClasses();

    for(WFandValidImport oneImport : this.imports) {
      sc = oneImport.typeCheck(modules, sc);
    }

    return sc;
  }
}
```

Figure 59: A Java-style implementation of the [imports] derivation rule

From the corresponding derivation rule, we know that the method for type checking a single import consumes two pieces of contextual situation—the collection of modules plus the mapping of from ClassNames to Shape—and returns such a mapping. Once the method has dealt with all imports, the full mapping, sc, is returned as the result of the method. Note how this last part corresponds to the right of the ==> arrow in the conclusion of the derivation rule.

Exercise 59. While the method in figure 59 uses a for each loop, the one in figure 58 uses an index-based for loop. Could it use a for each loop? If so, re-factor the code; otherwise explain why not.

Figure 60 displays one last example of a typeCheck method that implements a judgment derivation rule. It illustrates how type checking works for the data representation of a well-formed and valid expression. Following the pattern for the previous two examples, the method consumes some pieces of contextual information and produces a data representation of Types, because this is the shape of a generic expression type judgment.

The two blocks inside of the method body correspond to the two antecedents; the return statement realizes the confirmation to the right of the ==> in the conclusion. Each of the two blocks retrieves the type of an operand of the addition expression and compares this type to Number. Unless this comparison succeeds, the antecedent fails to hold, causing the

```
// represents an addition expression
class WFandValidAddition {
  Variable leftOperand  = ...;
  Variable rightOperand = ...;

  // check the type of `this` expression
  // EFFECT throw a runtime exception if a violation is discovered
  Type typeCheck(SClasses sc, TEnv tenv) throws TypeError {
    Type leftType = tenv.lookup(this.leftOperand);
    if (!leftType.equals(TypeAST.REAL))
       throw new TypeError("wrong argument type for + [left]");

    Type rightType = tenv.lookup(this.rightOperand);
    if (!rightType.equals(TypeAST.REAL))
       throw new TypeError("wrong argument type for + [right]");

    return TypeAST.REAL;
  }
}
```

Figure 60: A Java-style implementation of the [+] derivation rule

method to signal an error.

Exercise 60. The two blocks of code resemble each other and call for an abstraction. Design the abstraction systematically so that each antecedent becomes just one simple line.

Exercise 61. Why is it guaranteed that the lookup method of tenv finds a type for the two Variables?

3.5 *Types*: Type Checking

At this point, we have everything in place to complete the model for *Types*: the parser for the *Types* language (exercise 51), the validity checker (exercise 52), and the ideas for translating the judgment derivation rules into code (for an object-oriented data representation of the syntax).

Exercise 62. Design a complete type-checking pass for *Types*. The implementation should adhere closely to the shape of the rule-based specification so that small changes to the specification directly translate to small changes in the code. Have your type checker signal an exception when it discovers a type violation. Try to formulate error messages that assist the *Types* programmer with the task of repairing type errors.

Exercise 63. An implementation of a type checker inside of an IDE cannot just signal an exception when it discovers a rule violation. Instead, it pretends to repair the AST so that it checks and adds a node to the AST—or mutates an AST node—so that the IDE can highlight several such violations at once.

Design a variant of your solution to exercise 62 that lives up to this standard. When checking an expression fails, use the type Number. Although this substitution is a simplistic repair, it gives you a first idea of how such an IDE-integrated type checker may work.

Exercise 64. Design and implement a pass that turns the AST of a well-formed and well-typed *Types* system into a *Module* AST so that the linker from exercise 48 can create a *Classy* program. The resulting program can then be run on the CESK machine.

Note Depending on your chosen implementation language and your choice of AST data representation, your linker might already work for *Typed* ASTs or you might find it easy to adapt the linker so that it works this way.

Exercise 65. Adapt the main function from exercise 49 so that it can run the entire process of parsing, validating, type checking, linking, and determining its meaning.

The function should issue error strings in the following order:

- "syntax error" if the parser discovers any mistakes;
- "duplicate module name" if the *Types*-specific validity checker finds two modules with the same name;
- "import of non-existing module" if the *Types*-specific validity checker discovers an import specification that does not refer to an already-existing module;
- "duplicate name error" if the *Types*-specific validity checker encounters a class with two identical field names, or a class with two identical method names;
- "undeclared name error" if the validity checker encounters an undeclared ClassName or variable;
- "type error" if the type checker encounters a type violation; or
- "runtime error" if the CESK transition function stops due to a final state with an error in its C register.

If running the machine delivers a value, it will be a number. Why? The main function prints this number.

4 Static Types Eliminate More Dynamic Checks

Checking the types of a *Types* program simplifies the CESK machine. Concretely, it renders a fair number of run-time checks superfluous and thus eliminates the corresponding transitions. The purpose of these checks is to catch inconsistencies between computational operations and arguments. For example, consider the transition for addition in figure 31. The figure specifies *two* transitions for an addition expression: one if the value of both variables are numbers and one if either one of them is an object. For several other expressions, the CESK machine for *Classy* comes with pairs of transitions for their evaluation: a "success" and a "failure" case.

The addition of types to the language and a type checker to the model *before* the program is handed to the abstract machine ensures that these consistency conditions on the transitions never matter. Compare the transition for + in figure 31 with the implementation of typeCheck in figure 60. Notice how typeCheck compares the type of each variable with Number and how the side condition on the machine transition confirms that the same fact. Since typeCheck is run *before* the program is (linked and) loaded onto the machine, the case-distinction is superfluous.

	Control		Environment	Store	Kontinuation
evaluate an addition					
before:	(y + z)		e	s	k
after:	(+ yn zn)		e	s	k
where	yl	=	e[y]		
and	zl	=	e[z]		
and	yn	=	s[yl]		
and	zn	=	s[zl]		

Figure 61: The CESK transition function for *Types*: simplified addition

Exercise 66. Inspect the CESK transitions in figures 31 through 36 and create a list of conditions that distinguish success and failure cases.

With this list of conditions in hand, inspect the derivation rules in this chapter to determine which of these conditions type checking renders superfluous. Mark the corresponding specifications in figures 31 through 36.

4.1 The Final Bit of Theory: Type Soundness

The elimination of run-time checks from the CESK machine for *Classy* demands trust in the type checking rules and its implementation. From the perspective of theory, the relationship raises the following question concerning the relationship between type-checking on one hand and the CESK machine on the other:

> *does the type checker eliminate all cases in which a Types program may apply a computational operation to the wrong kind of value?*

Here "computational operation" refers to those cases in the CESK transition function that come with success conditions:

- `+` and `/` are obvious candidates, because both can process numbers only;
- `new` must match the number of arguments to the number of fields;
- `o –> f` and `o –> f = e` succeed only if `o` is an object *and* this object comes with a field named `f`; and
- `o –> m(a ...)` needs three conditions to hold for a successful transition: (1) `o` is an object, (2) it has a method named `m`, and (3) the number of arguments agrees with the number of `m`'s parameters.

As far as the type checker is concerned, we need to adapt the basic idea from section 4.1 in chapter IV: A bad type checker can accept ill-typed systems. Take a look at the following typeCheck method for a *Types* system:

```
class WFandValidSystem {
 ...
 void typeCheck() throws TypeError {
  return ;
 }
 ...
}
```

It does not inspect any of its pieces; instead it accepts the existing (well-formed and valid) system AST as type correct. And this kind of type checker is not going to help a language implementer with the elimination of run-time checks.

Following the pattern of the preceding sections on theory, we can state a theorem that expresses how the type checker must relate to the CESK machine to be sound. To this end, we think of the CESK semantics as a

mathematical function, and we also consider the type checker sketched in this chapter as a function called *typeCheck*. Using these conventions, the so-called *Typed Soundness Theorem* has the following shape:

Some people use the phrase "type safety" instead.

> For all well-formed and valid systems S in the *Types* model, if *typeCheckSystem*(S) holds and if $\mathit{runMachine}_{CESK}(P) = R$, then R is either a number or a division-by-zero *Exception*.
>
> Here P is the program that results from erasing types from S and linking the resulting untyped system.

In other words, *runMachine* does not signal an exception due to any of the conditions listed concerning success-failure pairs of transitions. A corollary is that the corresponding failure cases in the transition function are superfluous.

If you followed this far, you may now wonder what such a theorem really means. As is, the statement concerns the mathematical nature of the judgment derivation rules and the CESK machine. Critically, it does not say anything about the *implementation* of the type checker or the CESK machine.

Seen this way, you may wonder whether a language implementer can truly rely on this theorem. Theoreticians tend to think so, because they assume that implementers faithfully follow the specification. Practitioners know that the chosen implementation language or algorithmic concerns often necessitate a deviation from the specification. Unsurprisingly, they accept that implementations come with bugs, that is, differences to the specified behavior of either the type checker or the run-time machinery.

These cases will be needed again in the next chapter.

Exercise 67. Figure 61 shows how an implementer who trusts the type checker may collapse the pair of cases in the specification of the CESK transition function into one. Use the result of exercise 66 to comment out the cases of the CESK transition function that are superfluous due to the type system.

5 The Pragmatics of Types

The addition of a type system to a programming language raises one of the most complex questions of pragmatics. Gaining a solid understanding of the pragmatics of a type system is only possible after a complete model has been developed. And the development of the model in this section shows that a type system comes with costs and benefits for both consumers and producers of a language.

5.1 The Costs and Benefits for the Language Creator

As the preceding sections indicate, the design and implementation of a type system imposes a significant effort on the language creators. In addition, types shift property checking from semantics to syntax, which simplifies the machine. Let's look at these various issues, one at a time.

Designing a type system starts from a goal and must then work through the collection of linguistic features with this goal in mind. Consider the type system in this chapter, whose goal could be formulated as

> *eliminate the compatibility checks from the CESK machine, listed in the preceding section.*

At first glance, this goal just means checking that the variables in an addition expression always stand for numbers. But, an addition expression may appear inside of a method and involve the method's parameters. Hence it is necessary to consider methods, that is, how values flow from the system body *into* methods. Conversely, a variable declaration may contain a method call instead of a numeric constant, meaning the type system also has to consider how values flow *out* of methods. More generally, every item in the list of section 4.1 forces a type-system designer to study which role each linguistic feature plays in moving such a constraint from the machine into the type system.

Once the type-system designer has written up the type judgments and the judgment derivation rules, the language implementer's task is to turn these rules into code. Section 3.4 illustrates with a few cases how this works, and it indicates how much work this translation of rules into code is. What it fails to show is the work involved in the design of auxiliary methods so that the typeCheck methods mirror the derivation rules as clearly as possible. Aiming for such an implementation is important because type systems evolve, and realizing such changes is simplest when the implementation is structured according to the specification.

All this work pays off when it comes to "selling" the language. Every compatibility check performed at run time slows down program execution. Conversely, the elimination of every check speeds up the execution. Furthermore, types assist language implementers in other dimensions too, including space-efficient representations of surface data as machine-level data and energy-efficient arrangement of instructions. In short, the creators of a typed programming language can "brag" more easily about its performance than those who create untyped languages.

Both topics are out scope for this book.

Besides performance, language creators tend to advertise that type systems accelerate bug discovery. That is, the language implementation can inform a software developer about incompatible operations and values during the creation of code instead of when the program runs and the given inputs force the code to exercise these operations. Given that such failed compatibility checks stop the program execution, accelerated bug discovery appears to provide another sales point for a language. After all, no developer wants the control software for an airplane to fail because it tried to add a number to an object.

Like all advertising, this last one needs a close look. Unless the language creators establish type soundness—both on theory side and the implementation side—this claim is patently false. The problem is that

> *if a language does not satisfy the type-soundness property and if its semantics does not perform run-time checks to ensure compatibility between operations and values, the code may perform entirely nonsensical computations.*

Take the example of an addition operation again and consider that both a number and an object are represented as a some fixed number of bits in an actual hardware machine. While the bit-level representation may distinguish numbers and objects, the machine's add instruction can, and will, add any two bunch of bits—except that this addition produces a meaningless result. In the best case, such meaningless calculations eventually result in so-called segmentation faults—but this may happen long after the add instruction used the bits of an object representation. In the worst case, the program execution terminates normally without much of a hint that something went wrong.

Establishing type soundness is a lot of work. Language creators typically perform this work in two phases. First, they develop mathematical models of the type checker and the semantics, like those presented in this book. Then they validate with a careful analysis of all cases in the semantics that the type system and the semantics are related in the desired manner at every step of the way. Even if the language creators perform all this work of establishing theoretical type soundness and carefully translate the derivation rules into type-checking methods, the implementation may still suffer from bugs due to a number of reasons. Most importantly, theory work covers a model, and a key attribute of a model is its simplicity compared to the actual language.

Second, to close this gap between mathematical models and the full-fledged language, language creators also derive extensive test suites from

the language grammar and the full-fledged type-system specification. These test suites must cover language constructs excluded from the model plus the corresponding judgments and derivation rules. Critically, these tests do not just ensure that the type checker works properly and the semantics works properly, separately; they must ensure that the combination works.

This specification is often formulated in structured English based on the type judgments and derivation rules.

Clearly, this kind of work is extensive and complicated. Unsurprisingly, software developers discover problems with type-soundness claims long after a language implementation has hit the market. Language creators with the ambition of maintaining type soundness will react with new releases when such bugs are discovered. It should be equally unsurprising that many language creators do *not* bother with type soundness, leaving their consumers in the dark as to how safe the language is.

5.2 The Benefits for the Software Developer

A software developer faces many more work situations than those covered in the preceding chapters of this book. It starts with the design of software and goes all the way to its maintenance over time. Types affect almost all of these situations, so it is time to take an expansive look at the combination of those and types.

Figure 62 presents a table whose rows describe work situations and whose three columns list the three kinds of languages we can imagine: one without a type system, an idealized one with a sound type system, and one with an unsound one. Each cell corresponds to a work situation combined with an approach to types in a programming language.

By filling in these cells, we get a comprehensive picture of the pragmatics of types. We start with just a few examples of row-specific and even cell-specific benefits. Consider these examples illustrative, and keep them in mind for the exercises at the end of this section.

Designing Code When developers design software, types assists with almost all steps, but definitely with the first and most important one: how to choose a data representation for the information that the code is going to process. A language of types describes the forms of available data, and it is only those forms that a developer can use to represent information. Once the data representation is chosen, the formulation of a data definition heavily relies on the type notation too (plus comments that interpret what each piece denotes).

In the context of a typed programming language—sound or unsound—a developer does not have much of a choice. The type notation and the type

work situation	language		
	without	with sound type system	with unsound
design code			
keyboard code			
— create			
— refactor			
— evolve			
— migrate			
testing			
debugging			
documentation			
deployed code			
— bug diagnosis			
maintenance			
— port to different PL			

Figure 62: Work situations and the role of types systems

rules are baked into the language, and they have to be used as is. A developer who has to work with an untyped language should still choose to follow a type discipline; but this developer can choose a type discipline that fits the problem. As the next chapter shows, these usually implicit choices still dictate a certain programming discipline and result in a reasonably small number of idioms.

The final question concerning this first row is whether a type checker is helpful. Only the right-most two columns in figure 62 combine a type system with this tool; the point of an untyped language is that programmers are allowed to violate the imagined rules of their type system—as long as they know what they are doing. Based on posts in a wide variety of forums, mailing lists, blog posts, and so on, it is clear that the vast majority of developers think that a type checker helps, though in all likelihood they conflate design with code creation inside of an IDE.

Creating Code The use of integrated development environments synthesizes code design—the creative act of thinking through the problem and coming up with a solution—with code creation—the act of keyboarding. Developers design as they keyboard, and they keyboard as they design. A typed, object-oriented setting supports keyboarding to the point where it

assists with some part of the design—if the data representation is chosen carefully. If so, it often boils down to the act of choosing the proper menu item after entering "o." into the editor.

Having clarified this much, we can turn to the question of what enables the IDE to offer a menu of possible completions for the "o." sequence of keystrokes. And the answer is *the type system*, of course. The very moment when a programmer enters the dot, the IDE has determined the type of the `o` via some background computation. As this chapter shows, the type contains the names of all methods and fields for an object.

As a matter of fact, if the developer selects a method `m` to continue from "o.", the IDE has type information about the shape of the method call. Using this information, the IDE can present a template for the method call that the programmer then fills in. In sum, having a type system is extremely helpful in this case of an object-oriented language, which partly explains the success of this kind of language.

The picture looks a bit different for typed functional or procedural languages. An IDE can still provide more support for keyboarding code than for an untyped one, but the support is less helpful than for an object-oriented language. Most importantly, the IDE catches type errors as the programmer adds characters to the program text, and this form of alert is useful to prevent design and code creation errors.

Testing and Debugging Let's skip some rows to analyze the pragmatics of testing and debugging. Here the three cells differ radically.

As noted, a typed language signals that an operation, say field access, is incompatible with the given value(s) during program creation, that is, as the programmer enters text into an IDE's editor. This observation clearly separates the first cell from the other two. Separating the second cell from the third requires a reminder of the role that type soundness plays.

When a sound type system validates code, the programmer knows that most semantic compatibility checks cannot fail. In the context of *Types*, a programmer may experience only one kind of semantic error: division by zero. By contrast, when an unsound type system validates code, it is still possible that programmers run into semantic mistakes during testing. Recall, however, that a language semantics does not have to perform these run-time checks at all. A language creator can remove them with or without arguing type soundness. In this case, a program may terminate and output results as if every operation had worked properly.

It is rare that the semantics of an untyped *language comes without compatibility checks.*

From this point of view, a developer may observe the following outcomes during test runs in the context of an unsound type system:

- A test run may terminate with the correct result. While improbable, it is possible that the execution misapplied operations but that the observable outputs do not give any hint that things went wrong. Clearly, this is the worst possible outcome, though luckily it is rare.

- A test run may terminate with an incorrect result. In turn, the developer must search for a mistake but does not know whether this is a semantic mistake or a logical mistake. The good news is that the developer is aware of a problem; the bad news is that the search space for the bug is large, due to lacking language support.

- A test run may terminate with a segmentation fault, because a misapplied operation causes a (bit-level) violation of an operating system or hardware constraint. At this point, the developer knows that the program execution used a primitive computational operation with the wrong values. Sadly, the developer cannot know when this misapplication took place during execution nor to which operation in the source code it relates.

This list significantly simplifies the set of possible scenarios, but it suffices to make the point about dangers with unsound type systems.

This list of testing scenarios and related debugging work clarifies how type soundness pays off. Yes, establishing type soundness imposes a significant amount of work on the small team of language implementers, but at the same time, the vast number of language users benefit a lot.

Exercise 68. Analyze how a type system may assist programmers who have to modify existing code so that it can deal with new variants of data. In your experience with procedural and object-oriented programming, is this task easier in one or the other?

Exercise 69. Analyze how a type system may assist software developers who must modify a program so that it can compute additional results for the existing forms of data. In your experience with procedural and object-oriented programming, is this task easier in one or the other?

Exercise 70. C# and Java come with so-called nominal type systems. By contrast, TypeScript has a structural type system, similar to the one of our *Types* model. In your experience, is a nomimal type system as accommodating as a structural one for the extensibility tasks described in the preceding two exercises?

Exercise 71. Analyze how a type system may assist with documenting the interface of a software component.

Exercise 72. Does the existence of a type system help with the task of porting software from one language to another?

5.3 How to Defeat a Programming Language

Even without filling in all of the cells of the table in figure 62, you can see that a language with a sound type system comes with tremendous benefits in many working situations. From a pragmatics perspective, such a language is the best choice. Experience demonstrates, however, that developers can miss out on all these benefits unless they work with the language and its tools, not against it.

To start with, a developer's choice of data representation matters most. Consider the "choose a data representation " exercises that show up in all of the chapter of this book. Every programming language offers an infinitude of choices. As demonstrated at the beginning of chapter VI, a developer could, in principle, encode any form of information as a number. Realistically, programmers often choose strings to represent information, even if the information suggests structure.

In the case of an abstract syntax tree, the represented information has two sources: the program text and the language grammar. While the first usually consists of either just a sequence of (Unicode) characters or a semi-structured collection of characters, the second is a highly structured form of information. A good data representation mirrors this information structure and accommodates pieces of the program text as needed.

Figure 63 displays a choice of data representation for *Module* ASTs. Here is what the creator of this data representation writes to represent an occurrence of the VariableName x in a *Module* system:

```
new AST("variable", new AST[0], "x")
```

Now imagine using this Java class to represent modules, import specifications, (*Module*) classes, methods, variable declarations, or an expression. So here is a representation of a *Module* module-free system:

```
AST xvar = new AST("variable", new AST[0], "x");
AST one  = new AST("number", new AST[0], 1.0);
AST x1[] = xvar, one;
AST decl = new AST("declaration", x1, "");
AST de[] = decl;
AST st[] = new AST[0];
AST expr = xvar;
AST bd[] = de, st, expr;
AST sys  = new AST("system", bd);
```

Stop! Before reading on, imagine designing one of these methods for this data representation.

```
// represents an AST node of Module
class AST {
  String kind;          // the BNF production `this` represents
  AST[] pieces;         // ... its pieces
  String name  = "";    // ... if it has a name
  double value = 666.0; // ... if it represents a literal number

  AST(String kind, AST[] pieces) {
    this.kind   = kind;
    this.pieces = pieces;
  }

  AST(String kind, AST[] pieces, String name) {
    ...
  }

  // no two modules have the same name
  AST noDuplicateModules();

  ...

  // all occurrences of a variable name refer to a declaration
  AST closed();
}
```

Figure 63: An ill-suited data representation of *Module* ASTs in Java

Obviously, a data representation for ASTs such as the one in figure 63 fails a developer in many ways. Most importantly from the perspective of this chapter, it fails to take advantage of Java's type system in many different ways. For example, when the developer enters "xvar." into the IDE, the drop-down menu contains the entry "noDuplicateModules" because it is one of the many methods that AST implements. But invoking the noDuplicateModules method on the data representation of a variable makes no sense. Similarly, in order to check whether the modules of a *Module* system come with distinct names, the noDuplicateModules has to iterate over all pieces, although only the first elements of this array should represent modules:

```
AST noDuplicateModules() {
  for(AST ast: pieces) {
    if (ast.kind == "module") {
      ...
    }
  }
}
```

Furthermore, as the sketch indicates the method must compare strings to find out whether any particular instance of AST is a module. Only then it make sense to check the name field of the instance and process it as a ModuleName.

Contrast this first data representation, with an approach that uses distinct Java classes to represent modules, import specifications, classes (those in *Module*), methods, variable declarations, and so on. In such a data representation, a system would come with one container that represents all of its modules. This container would be the only Java class that would need a noDuplicateModules method, meaning this method name would not pollute any drop down menus when the developer enters "someVariable." into the IDE. And, it is easy to see how this method can uniformly access all of the modules' names—without any further conditionals.

A good core college curriculum on programming focuses on the very principles that help you reap pragmatic benefits from programming languages. Such a curriculum conveys an understanding of software development as a systematic design activity tailored to the underlying philosophy of several different languages. This approach naturally introduces students to IDEs that properly assist with both syntactic validation and program execution. If your college course didn't not come with such a curriculum, the best course of action is to observe yourself (and others) interacting with the IDE, stop to reflect on these interactions, and experiment on how to get the best feedback from the IDE and the chosen language.

5.4 The Costs for the Software Developer

Benefits do not come for free. Most importantly, it is impossible to equip many well-behaved *Module* systems with types so that they can be run as *Types* systems. That is, *Types* comes with less expressive power than *Module*.

Figure 64 illustrates this point. Consider the *Module* code on the left side. This module defines a class, Coordinate, which comes with two fields, x and y, and one method, move. A mathematically inclined developer might have named this method translate, because it realizes a geometric translation of one point relative to another, interpreted as a vector. Critically, the method does not mutate this instance but creates a new one; it is a "functional" method.

Stop! Try to write down a Shape that describes Coordinate as a type.

Problem is, the types notation does not allow a programmer to refer to a Shape by name. In particular, a Shape for Coordinate would have to refer to itself, because move returns another instance of the class—meaning its

```
(module Cartesian
  (class Coordinate
    (; going right horizontally:
     x
     ; going up vertically:
     y)

    (method move (delta)
      (def myX    (this --> x))
      (def myY    (this --> y))
      (def deltaX (delta --> x))
      (def deltaY (delta --> y))
      (def nuX   (myX + deltaX))
      (def nuY   (myY + deltaY))
      (new Coordinate (nuX nuY)))))
```

```
class Coordinate {

  double x;
  double y;


  Coordinate(double x, double y) {
    this.x = x;
    this.y = y;
  }

  Coordinate move(Coordinate delta) {
    double myX    = this.x;
    double myY    = this.y;
    double deltaX = delta.x;
    double deltaY = delta.y;
    double nuX    = myX + deltaX;
    double nuY    = myY + deltaY;
    return new Coordinate (nuX, nuY);
  }
}
```

Figure 64: The loss of expressive power due to type constraints (1)

return type is the Shape we are in the process of writing down. Technically speaking, language researchers say that the type system of the *Types* model lacks recursive types.

The designers of type systems for real-world languages try to overcome this loss of expressive power with additional complexity. For example, Java's type system accommodates recursive class types. Take a look at the right side of figure 64, which displays a Java-style implementation of an equivalent class. As the boxed Coordinate in Java a method's signature shows, the name of a class is in scope of all its method types. Hence, it is possible to equip a functional method such as move with a signature.

Exercise 73. Add a system body to the code on the left side of figure 64 so that you can run it in your implementation of *Module*. The example should enable you to observe an output related to the processing of a Coordinate via move.

Exercise 74. If you have experience with Java, complete the code on the right side of figure 64 in the same manner as your solution of exercise 73. Run the program and compare the output to the one of your *Module* system.

Support for generics—occasionally called parametric polymorphism—in typed class-based, object-oriented languages makes this point as clearly

```
(module Null
  (class Null ()
    (method map ()
      this)))

(module Cons
  (import Null)

  (class Cons (one rem)

    (method map ()
      (def one  (this --> one))
      (def fone (one --> f()))
      (def rem  (this --> rem))
      (def frem (rem --> map()))
      (new Cons (fone frem)))))
```

```
interface WithF<T> {
  WithF<T> f();
}

interface IList<T extends WithFC> {
  IList<T> map();
}

class Null<T extends WithF>
     implements IList<T> {
  public IList<T> map() {
    return this;
  }
}

class Cons<T extends WithF>
     implements IList<T> {
  WithF<T> one;
  IList<T> rem;
  Cons(WithF<T> one, IList<T> rem) {
    this.one = one;
    this.rem = rem;
  }

  public IList<T> map() {
    WithF<T> fone = this.one.f();
    IList<T> frem = this.rem.map();
    return new Cons<T>(fone, frem);
  }
}
```

Figure 65: The loss of expressive power due to type constraints (2)

as possible. While Java and any typed object-oriented language with a type system supports recursive types, it initially lacked generics. Indeed, it took the Java team a decade to add this extremely useful feature to the type system. During this time, several researchers designed and explored a number of variants, meaning both theoretical and practical properties. In the end, the Java team selected an *unsound* alternative, mostly to stay backwards-compatible. By contrast, Microsoft's C# designers, who had also released the language without generics, opted for a sound alternative and modified the underlying virtual machine to accommodate this choice.

Figure 65 displays an example that demonstrates the need for generics and simultaneously documents another loss of expressive power. The left side of the figure presents an untyped *Module* system fragment. It consists of two modules, Null and Cons, which jointly create a linked-list data repre-

sentation. Using the exported classes, a *Module* programmer can form lists that contain distinct kinds of objects, that is, instances from distinct classes. As long as each of these classes supports a method named f, the programmer can also request a map over these lists. Here are two such classes:

```
(module F
  (class F (x)
    (method f()
      this)))
```

```
(module G
  (class G (x)
    (method f()
      (def x (this --> x))
      (def y 2.0)
      (x + y))))
```

Key is that the programmer knows that the result of this use of map is a list with the same kinds of objects, that is, instances of the respective classes. In the above example this would mean a list of Fs and a list of Gs, respectively. Since *Module* lacks a type system, however, the programmer cannot use the language's validity checkers to confirm this thought.

Neither could a Java programmer until 2004. Instead, the programmer would use Java's Object type to encode the fact that a list can accommodate all kinds of objects plus casts to confirm the result type. A cast tells the type system to accept a programmer's claim without checking it, and it informs the semantics to perform a run-time check. Hence, a pre-2004 Java programmer could confirm this thinking only with unit tests, not while editing the program in an IDE.

The right side of figure 65 displays a Java version of the *Module* fragment that uses generics. Critically, the programmer can express the thought "every class that has a method f" in the type system—see the boxed code—and the type system validates this thought across the entire program. In particular, it confirms that the map method inside of Cons may use f. Since the IList interface also specifies that the result of map is IList<T>, the type checker confirms that each application of map returns the kind of list on which it is invoked.

Technically, the type checker replaces the type variable T for each call to method with the type of the list elements.

Exercise 75. Using the two classes above, add a system body to the code on the left side of figure 65 so that you can run it in your implementation of *Module*. The example should enable you to observe an output related to the processing of these lists via map.

Exercise 76. If you have experience with Java, complete the code on the right side of figure 65 so that you can run the program and observe an output related to the processing of two distinct ILists via map.

While the addition of complex features, like generics, to a type system solves one problem, it creates another one. As the complexity of a type

system increases so does the slope of the learning curve. Yes, complex type systems enable developers to articulate their thinking about code as code and have the type checker confirm it. But, language designers must acknowledge that doing so slows down the learning process, potentially frustrates novices, and occasionally injects new pitfalls into the language.

Consider Java's generics. On one hand, they replace the Object and casts with useful pieces of code. On the other hand, Java's generics break the soundness theorem that applied to its initial releases. As mentioned in preceding sections, this lack of soundness reduces the trust into the program's results. Fortunately in Java's case the problem remains minor.

Similarly, the replacement of generics instead of Object and casts is not straightforward. It requires a thorough understanding of a non-trivial type notation and of the meaning of parametric type expressions. To get to the point where writing programs such as those in figure 65 comes easy, a developer is likely to get frustrated with efforts to get get code to conform to the type system.

Exercise 77. A structural type system, such as the one of the *Types* model, greatly simplifies the extension of a code base with new forms of data and additional pieces of functionality. If some functionality exists, say map, that applies to any object with a method f, a programmer can re-use this functionality with new classes.

How can this reuse endanger the code's integrity at run time?

How does a nominal type system, such as C#'s or Java's, prevent such novel uses?

The answer to these questions demonstrates that the design of a type system adds to the tension between the security of a piece of code and its extensibility.

6 The Costs and Benefits of Types: A Warning

Software developers began to understand the expressive power of type systems in the 1990s. They recognized that type systems enabled them to express intricate thoughts and have them checked; and they recognized that by doing so, sound type systems could thus guarantee the absence of certain run-time failures.

In response, academics decided to explore additional ways to add expressive power to type systems. They advertised types as theorems about code and type checkers as proof assistants that established the validity of these theorems. Enabling developers to state more theorems and prove

them correct would eliminate more run-time problems. Deployed software would suffer from fewer problems. These researchers seemed to overlook, however, that powerful type systems create a steep learning curve and that in such settings, developers have to work so much harder to sync their code with their types.

As this chapter's cost-benefit sections point out, the design of a type system is not a mathematical exercise but an engineering task. Engineers know that they maximize a benefit subject to resource constraints; or they minimize cost subject to safety constraints. Put differently, engineering is an optimization process. Engineers do not go for ideals, such as total safety.

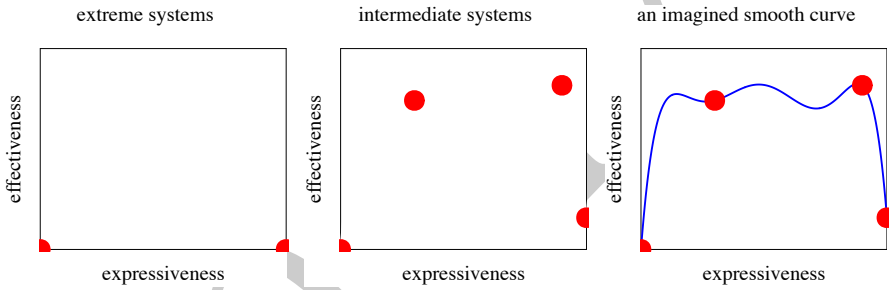


Figure 66: The Laffer curve of type systems

One way to illustrate this point is the use of the Laffer curve from economics. A Laffer curve records the benefits of some project over its resources. For simplicity, let's say the benefits of a type system is the prevention of bad behavior at run time, and let's similarly and equally naively say that the resource is the power of the type system to invalidate code.

While this setup is somewhat vague, it allows us to draw the first two points of the Laffer curve of types: see the left-most curve in figure 66. The point near the origin represents a type system that accepts every piece of code. Clearly such a type system comes without any benefits; all problems are (at best) discovered by the semantics. By contrast, the "extremist" type system on the right side of the graph rejects all programs. Although this guarantees that no program behaves badly at run time, it really means that no program can ever be run. Once again, it is appropriate to assign this type system a score of 0 effectiveness. After all, effectiveness implies developers get programs to run easily and behave well.

This chapter, and empirical evidence, shows that there are effective type systems with the kind of benefits that developers appreciate. Let's record this observation as red dots between the two extreme ones; see the graph in the middle of figure 66. For example, many programmers consider Java's type system expressive and reasonably safe. Academics may instead point to OCaml's type system or Haskell's; both of these bestow more benefits on developers than Java.

In short, we may wish to imagine a continuous graph of the effectiveness function of types as presented in the right-most part of figure 66. Given this understanding of the type-system design task, the goal of types researchers and type systems designers ought to be the discovery of optimal points, or as stated above, points on this curve that facilitate program design and yet make the resulting code reasonably safe.