result w of this evaluation is an integer, and if so use it as the result of the function application.

When evaluating $e_1\ e_2$, we evaluate e_1 to a functional value f and e_2 to an integer v, and then apply f to v as described above.

Extend the interpreter from Fig. 4.3 to handle these new constructions as described above.

References

1. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs. MIT Press (1996). Also downloadable from https://mitpress.mit.edu/sicp/full-text/sicp/book/
2. Steele, G.L., Sussman, G.J.: The Art of the Interpreter or, The Modularity Complex. Tech. Rep. AIM-453, Massachusetts Institute of Technology, Cambridge, MA, USA (1978)

Chapter 5
Type Checking

The most touching epitaph I ever encountered was on the tombstone of the printer of Edinburgh. It said simply:
He kept down the cost and set the type right.

Gregory Nunn (1955–)

Lexing and parsing will reject many texts as not being correct programs. However, many languages have well-formedness requirements that can not be handled exclusively by the techniques seen so far. These requirements can, for example, be static type correctness or a requirement that pattern-matching or case-statements are exhaustive.

These properties are most often not context-free, i.e., they can not be checked by membership of a context-free language. Consequently, they are checked by a phase that conceptually comes after syntax analysis (though it may be interleaved with it). These checks may happen in a phase that does nothing else, or they may be combined with the actual execution or translation to another language. Often, translation to another language (such as machine language) may exploit or depend on type information, which makes it natural to combine calculation of types with the actual translation or to pass type information from a type-check phase to the translation phase. In Chap. 4, we covered type-checking during execution, which is normally called *dynamic typing*. We will in this chapter assume that type checking and related checks are done in a phase previous to execution or translation (i.e., *static typing*), and similarly assume that any information gathered by this phase is available in subsequent phases.

© Springer International Publishing AG 2024

T. Æ. Mogensen, *Introduction to Compiler Design*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-46460-7_5

5.1 The Design Space of Type Systems

We have already discussed the difference between static and dynamic typing, i.e., if type checks are made *before* or *during* execution of a program. Additionally, we can distinguish *weakly* and *strongly* typed languages.

Strong typing means that the language implementation ensures that, whenever an operation is performed, the arguments to the operation are of types on which the operation is defined, so you, for example, do not try to concatenate a string and a floating-point number. This is independent of whether this is ensured statically (prior to execution) or dynamically (during execution).

In contrast, a weakly typed language gives no guarantee that operations are performed on arguments that make sense for the operation. The archetypical weakly typed language is machine code: Operations are performed with no prior checks, and if there is any concept of type at the machine level, it is fairly limited: Registers may be divided into integer, floating point and (possibly) address registers, and memory is (if at all) divided into only code and data areas. Weakly typed languages are mostly used for system programming, where you need to manipulate, move, copy, encrypt or compress data without regard to what that data represents.

Many languages combine both strong and weak typing or both static and dynamic typing: Some types are checked before execution and other types are checked during execution, and some types are not checked at all. For example, C is a statically typed language (since no checks are performed during execution), but not all types are checked, so it is somewhat weakly typed. For example, you can store an integer in a union-typed variable and read it back as a pointer or floating-point number. While Java is generally considered a statically typed language, even this has run-time type checks: When you do a downcast, this is in essence a dynamic type check. A more extreme example is JavaScript: If you try to multiply two strings, the interpreter will see if the strings contain sequences of digits and, if so, convert the strings to numbers and multiply these. This is a form of weak typing, as the multiplication operation is applied to arguments (strings) where multiplication, mathematically speaking, does not make sense. But instead of, like machine code, blindly trying to multiply the machine representations of the strings as if they were numbers, JavaScript performs a dynamic check and a conversion to make the values conform to the operation. I will still call this behaviour weak typing, as there is nothing that indicates that converting strings to numbers before multiplication makes any more sense than just multiplying the machine representations of the strings. The main point is that the language, instead of reporting a possible problem, silently does something that might make no sense.

Figure 5.1 shows a diagram of the design space of static versus dynamic and weak versus strong types, placing some well-known programming languages in this design space. Note that the design space is shown as a triangle: If you never check types, you do so neither statically nor dynamically, so at the weak end of the weak/strong axis, the distinction between static and dynamic is meaningless. We have put Standard ML (SML) in the bottom-left corner as it does all type checking at compile time and guarantees that no type-incorrect operation ever happens. Scheme is in the bottom-

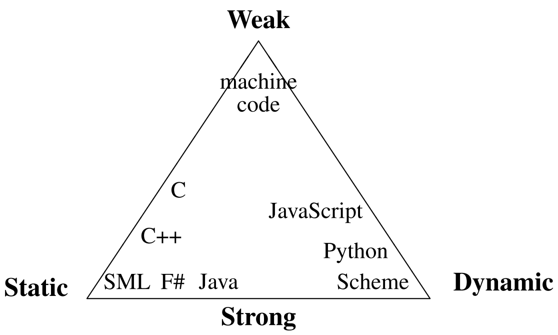

Fig. 5.1 The design space of type systems

right corner as all type checks are done at runtime, but errors are reported if the program attempts to do type-incorrect operations. The languages along the left edge are all statically typed, while those along the right edge are all dynamically typed.

5.2 Attributes

The checking phase operates on the abstract syntax tree of the program and may make several passes over this. Typically, each pass is a recursive walk over the syntax tree, gathering information, or using information gathered in earlier passes. Such information is often called *attributes* of the syntax tree. Typically, we distinguish between two types of attributes: *Synthesised attributes* are passed upwards in the syntax tree, from the leaves up to the root. *Inherited attributes* are, conversely, passed downwards in the syntax tree. Note, however, that information that is synthesised in one subtree may be inherited by another subtree or, in a later pass, by the same subtree. An example of this is a symbol table: This is synthesised by a declaration and inherited by the scope of the declaration. When declarations are recursive, the scope may be the a syntax tree that contains the declaration itself, in which case one pass over this tree will build the symbol table as a synthesised attribute, while a second pass will use it as an inherited attribute.

Typically, each *syntactic category* (represented by a type in the data structure for the abstract syntax tree or by a group of related nonterminals in the grammar) will have its own set of attributes. When we write a checker as a set of mutually recursive functions, there will be one or more such functions for each syntactic category. Each of these functions will take inherited attributes (including the syntax tree itself) as arguments, and return synthesised attributes as results.

We will, in this chapter, focus on type checking, and only briefly mention other properties that can be checked. The methods used for type checking can in most cases easily be modified to handle such other checks.

We will use the language from Sect. 4.2 as an example for static type checking.

5.3 Environments for Type Checking

In order to type check the program, we need symbol tables that bind variables and functions to their types. Since there are separate name spaces for variables and functions, we will use two symbol tables, one for variables and one for functions. A variable is bound to one of the two types `int` or `bool`. A function is bound to its type, which consists of the list of types of its arguments and the type of its result. Function types are written as a parenthesised list of the argument types, an arrow and the result type, e.g, `(int,bool)` $\rightarrow$ `int` for a function taking two parameters (of type `int` and `bool`, respectively) and returning an integer.

We will assume that symbol tables are persistent, so no explicit action is required to restore the symbol table for the outer scope when exiting an inner scope. We don't need to preserve symbol tables for inner scopes once these are exited (so a stack-like behaviour is fine).

5.4 Type Checking of Expressions

When we type check expressions, the symbol tables for variables and functions are inherited attributes. The type (`int` or `bool`) of the expression is returned as a synthesised attribute. To make the presentation independent of any specific data structure for abstract syntax, we will (like in Chap. 4) let the type-checker functions for pattern-matching purposes use a notation similar to the concrete syntax. But you should still think of it as abstract syntax, so all issues of ambiguity, etc., have been resolved.

For terminals (variable names and numeric constants) with attributes, we assume that there are predefined functions for extracting these. Hence, **id** has an associated function *getname*, that extracts the name of the identifier. Similarly, **num** has a function *getvalue*, that returns the value of the number. The latter is not required for static type checking, but we used it in Chap. 4, and we will use it again in Chap. **??**.

For each nonterminal, we define one or more functions that take an abstract syntax subtree and inherited attributes as arguments, and return the synthesised attributes.

In Fig. 5.2, we show the type-checking function for expressions. The function for type checking expressions is called $Check_{Exp}$. The symbol table for variables is given by the parameter *vtable*, and the symbol table for functions by the parameter *ftable*. The function **error** reports a type error. To allow the type checker to continue and report more than one error, we let the error-reporting function return.[1] After reporting a type error, the type checker can make a guess at what the type should have been and return this guess, allowing type checking to continue for the rest of the program. This guess might, however, be wrong, which can cause spurious type errors to be reported later on. Hence, all but the first type error message should be taken with a grain of salt.

[1] Unlike in Chap. 4, where the **error** function stops execution.

Fig. 5.2 Type checking of expressions

$Check_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
num	`int`
id	$t = lookup(vtable, getname(\mathbf{id}))$ $if\ \ t = unbound$ $then$ **error**(); `int` $else\ \ t$
$Exp_1 + Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $if\ \ t_1 =$ `int` $and\ t_2 =$ `int` $then$ `int` $else$ **error**(); `int`
$Exp_1 < Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $if\ \ t_1 = t_2$ $then$ `bool` $else$ **error**(); `bool`
`if` Exp_1 `then` Exp_2 `else` Exp_3	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $t_3 = Check_{Exp}(Exp_3, vtable, ftable)$ $if\ \ t_1 =$ `bool` $and\ t_2 = t_3$ $then\ \ t_2$ $else$ **error**(); t_2
id ($Exps$)	$t = lookup(ftable, getname(\mathbf{id}))$ $if\ \ t = unbound$ $then$ **error**(); `int` $else$ $\quad ((t_1, \ldots, t_n) \rightarrow t_0) = t$ $\quad [t'_1, \ldots, t'_m] = Check_{Exps}(Exps, vtable, ftable)$ $\quad if\ \ m = n\ and\ \ t_1 = t'_1, \ldots, t_n = t'_n$ $\quad then\ \ t_0$ $\quad else$ **error**(); t_0
`let` **id** $= Exp_1$ `in` Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, getname(\mathbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$

$Check_{Exps}(Exps, vtable, ftable) =$ `case` $Exps$ `of`	
Exp	$[Check_{Exp}(Exp, vtable, ftable)]$
Exp `,` $Exps$	$Check_{Exp}(Exp, vtable, ftable)$ $\quad :: Check_{Exps}(Exps, vtable, ftable)$

We will briefly explain each of the cases handled by $Check_{Exp}$.

- A number has type `int`.
- The type of a variable is found by looking its name up in the symbol table for variables. If the variable is not found in the symbol table, the lookup-function returns the special value *unbound*. When this happens, an error is reported and

the type checker arbitrarily guesses that the type is `int`. Otherwise, it returns the type returned by *lookup*.

- A plus-expression requires both arguments to be integers and has an integer result.
- Comparison requires that the arguments have the same type. In either case, the result is a boolean.
- In a conditional expression, the condition must be of type `bool` and the two branches must have identical types. The result of a condition is the value of one of the branches, so it has the same type as these. If the branches have different types, the type checker reports an error and arbitrarily chooses the type of the `then`-branch as its guess for the type of the whole expression. Note that the dynamic type checking done in Chap. 4 does not require that the branches have the same type: it only requires that the type of the chosen branch is consistent with how it is later used.
- At a function call, the function name is looked up in the function environment to find the number and types of the arguments as well as the return type. The number of arguments to the call must coincide with the expected number and their types must match the declared types. The resulting type is the return-type of the function. If the function name is not found in *ftable*, an error is reported and the type checker arbitrarily guesses the result type to be `int`.
- A `let`-expression declares a new variable, the type of which is that of the expression that defines the value of the variable. The symbol table for variables is extended using the function *bind*, and the extended table is used for checking the body-expression and finding its type, which in turn is the type of the whole expression. A `let`-expression can not in itself be the cause of a type error (though its subexpressions may), so no testing is done.

Since $Check_{Exp}$ mentions the nonterminal *Exps* and its related type-checking function $Check_{Exps}$, we have included $Check_{Exps}$ in Fig. 5.2.

$Check_{Exps}$ builds a list of the types of the expressions in the expression list. The notation is taken from SML: A list is written in square brackets with elements separated by commas. The operator :: adds an element to the front of a list.

Suggested exercises: 5.1.

5.5 Type Checking of Function Declarations

A function declaration explicitly declares the types of the arguments to the function. This information is used to build a symbol table for variables, which is used when type checking the body of the function. The type of the body must match the declared result type of the function. The type check function for functions, $Check_{Fun}$, has as inherited attribute the symbol table for functions, which is passed down to the type check function for expressions. $Check_{Fun}$ returns no information, it just checks for errors. $Check_{Fun}$ is shown in Fig. 5.3, along with the functions for *TypeId* and *TypeIds*, which it uses. The function Get_{TypeId} just returns a pair of the declared name and type,

Fig. 5.3 Type checking a function declaration

$Check_{Fun}(Fun, ftable)$ = `case` Fun `of`	
$TypeId\ (TypeIds)\ =\ Exp$	$(f, t_0) = Get_{TypeId}(TypeId)$ $vtable = Check_{TypeIds}(TypeIds)$ $t_1 = Check_{Exp}(Exp, vtable, ftable)$ *if* $t_0 \neq t_1$ *then* **error**()

$Get_{TypeId}(TypeId)$ = `case` $TypeId$ `of`	
`int` **id**	(*getname*(**id**), `int`)
`bool` **id**	(*getname*(**id**), `bool`)

$Check_{TypeIds}(TypeIds)$ = `case` $TypeIds$ `of`	
$TypeId$	$(x, t) = Get_{TypeId}(TypeId)$ $bind(emptytable, x, t)$
$TypeId\ ,\ TypeIds$	$(x, t) = Get_{TypeId}(TypeId)$ $vtable = Check_{TypeIds}(TypeIds)$ *if* $lookup(vtable, x) = unbound$ *then* $bind(vtable, x, t)$ *else* **error**(); $vtable$

and $Check_{TypeIds}$ builds a symbol table from such pairs. $Check_{TypeIds}$ also verifies that all parameters have different names by checking that a name is not already bound before adding it to the table. *emptytable* is an empty symbol table. Looking any name up in the empty symbol table returns *unbound*.

5.6 Type Checking a Program

A program is a list of functions, and is deemed type correct if all the functions are type correct, and there are no two function definitions defining the same function name. Additionally, there must be a function called `main` with one integer argument and an integer result.

Since all functions are mutually recursive, each of these must be type checked using a symbol table where all functions are bound to their type. This requires two passes over the list of functions: One to build the symbol table, and one to check the function definitions using this table. Hence, we need two functions operating over *Funs* and two functions operating over *Fun*. We have already seen one of the latter, $Check_{Fun}$. The other, Get_{Fun}, returns the pair of the function's declared name and type, which consists of the types of the arguments and the type of the result. It uses an auxiliary function Get_{Types} to find the types of the arguments. The two functions for the syntactic category *Funs* are Get_{Funs}, which builds the function symbol table and checks for duplicate definitions, and $Check_{Funs}$, which calls $Check_{Fun}$ for all functions. These functions and the main function $Check_{Program}$, which ties the loose ends, are shown in Fig. 5.4.

Fig. 5.4 Type checking a program

$Check_{Program}(Program) =$ `case` $Program$ `of`	
$Funs$	$ftable = Get_{Funs}(Funs)$ $Check_{Funs}(Funs, ftable)$ $if\ \ lookup(ftable,$ `main`$) \neq ($`int`$) \rightarrow$ `int` $then$ **error**()

$Get_{Funs}(Funs) =$ `case` $Funs$ `of`	
Fun	$(f,t) = Get_{Fun}(Fun)$ $bind(emptytable, f, t)$
$Fun\ Funs$	$(f,t) = Get_{Fun}(Fun)$ $ftable = Get_{Funs}(Funs)$ $if\ \ lookup(ftable, f) = unbound$ $then\ \ bind(ftable, f, t)$ $else$ **error**()$; ftable$

$Get_{Fun}(Fun) =$ `case` Fun `of`	
$TypeId\ (TypeIds)\ =\ Exp$	$(f, t_0) = Get_{TypeId}(TypeId)$ $[t_1, \ldots, t_n] = Get_{Types}(TypeIds)$ $(f, (t_1, \ldots, t_n) \rightarrow t_0)$

$Get_{Types}(TypeIds) =$ `case` $TypeIds$ `of`	
$TypeId$	$(x,t) = Get_{TypeId}(TypeId)$ $[t]$
$TypeId\ TypeIds$	$(x_1, t_1) = Get_{TypeId}(TypeId)$ $[t_2, \ldots, t_n] = Get_{Types}(TypeIds)$ $[t_1, t_2, \ldots, t_n]$

$Check_{Funs}(Funs, ftable) =$ `case` $Funs$ of	
Fun	$Check_{Fun}(Fun, ftable)$
$Fun\ Funs$	$Check_{Fun}(Fun, ftable)$ $Check_{Funs}(Funs, ftable)$

This completes type checking of our small example language.

Suggested exercises: 5.5.

5.7 Advanced Type Checking

Our example language is very simple and obviously does not cover all aspects of type checking. A few examples of other features and brief explanations of how they can be handled are listed below.

Assignments

When a variable is given a value by an assignment, it must be verified that the type of the value is the same as the declared type of the variable. Some compilers may additionally check if a variable can be used before it is given a value, or if a variable is not used after its assignment. While not exactly type errors, such behaviour is likely to be undesirable. Testing for such behaviour does, however, require somewhat more complicated analysis than the simple type checking presented in this chapter, as it relies on non-structural information. It can be done by *data-flow analysis*, see Chap. 10.

Data Structures

A data structure declaration may define a value with several components (e.g., an *array*, *struct*, *tuple*, or *record*), or a value that may be of different types at different times (e.g, a *union*, *variant*, or *sum*). To type check such structures, the type checker must be able to represent their types. Hence, the type checker may need a data structure that describes complex types. This may be similar to the data structure used for the abstract syntax trees of type declarations. Operations that build or take apart structured data need to be tested for correctness. If each operation on structured data has (implicitly or explicitly) declared types for its arguments and a declared type for its result, this can be done in a way similar to how function calls are tested. For example, the representation of an array type must include the number of dimensions of the array and the type of the elements. When the array is declared, the representation of the type is constructed, and when an array is used, e.g., at an array lookup, it is checked that the number of dimensions used in the lookup is correct, and the type of the result is found in the representation of the array type. The representation of a struct type must contain information about the names and types of the fields. These are found when the stuct type is declared. When a field of the record is accessed, the representation of the struct type is used to verify that the field name is one of the valid field names for the type, and the type of this field is returned as the type of the field access.

Overloading

Overloading means that the same name is used for several different operations over several different types. We saw a simple example of this in the example language, where $<$ was used both for comparing integers and for comparing booleans. In many languages, arithmetic operators like $+$ and $-$ are defined both over integers and floating point numbers, and possibly other types as well. If these operators are predefined, and there is only a finite number of cases they cover, all the possible cases may be tried in turn, just like in our example.

This, however, requires that the different instances of the operator have disjoint argument types. If, for example, there is a function *read* that reads a value from a text stream, and this is defined to read either integers or floating point numbers, the argument (the text stream) alone can not be used to select the right operator. Hence, the type checker must pass the *expected* type of each expression down as an inherited attribute, so this (possibly in combination with the types of the arguments) can be used to select the correct instance of the overloaded operator.

It may not always be possible to send down an expected type due to lack of information. In our example language, this is the case for the arguments to $=$ (as these may be either `int` or `bool`), and the first expression in a `let`-expression (since the variable bound in the `let`-expression is not declared to be a specific type). If the type checker for this or some other reason is unable to pick a unique operator, it may report "unresolved overloading" as a type error, or it may pick a default instance.

Type Conversion

A language may have operators for converting a value of one type to a value of another type, e.g. an integer to a floating point number. Sometimes these operators are explicit in the program and hence easy to check. However, many languages allow implicit conversion of integers to floats, such that, for example, $3 + 3.12$ is well-typed with the implicit assumption that the integer 3 is converted to a float before the addition. This can be handled as follows: If the type checker discovers that the arguments to an operator do not have the correct type, it can try to convert one or both arguments to see if this helps. If there is a small number of predefined legal conversions, this is no major problem. However, a combination of user-defined overloaded operators and user-defined types with conversions can make the type-checking process quite difficult, as the information needed to choose correctly may not be available at compile-time. This is typically the case in object-oriented languages, where method selection is often done at run-time. We will not go into details of how this can be done.

Polymorphism/Generic Types

Some languages allow a function to be *polymorphic* or *generic*, that is, to be defined over a large class of similar types, e.g. over all arrays no matter what the types of the elements are. A function can explicitly declare which parts of the type is generic/polymorphic, or this can be implicit (see below). The type checker can insert the actual types at every use of the generic/polymorphic function to create *instances* of the generic/polymorphic type. This mechanism is different from overloading, as the instances will be related by a common generic type, and because a polymorphic/generic function can be instantiated by (almost) *any* type, not just by a limited list of declared alternatives as is the case with overloading. We will, briefly, look at polymorphism in Sect. 12.1.

Implicit Types

Some languages (like Standard ML and Haskell) require programs to be well-typed, but do not require explicit type declarations for variables or functions. For such to work, a *type inference* algorithm is used. A type inference algorithm gathers information about *uses* of functions and variables and uses this information to infer the types of these. For example, if one branch of an if-then-else expression is a number constant, then both the other branch and the result of the if-then-else expression must be of number type. If there are inconsistent uses of a variable or function, or if the branches of an if-then-else expression have different inferred types, a type error is reported. A simple case is found in our example language, where the type of a variable bound in a let-expression is not declared, but is implicit the type of the expression to which it is bound.

Suggested exercises: 5.2.

5.8 Further Reading

Overloading of operators and functions is described in Sect. 6.5 of [1]. Section 6.7 of same describes how polymorphism can be handled.

Some theory and a more detailed algorithm for inferring types in a language with implicit types and polymorphism can be found in [2]. Types in general are covered in detail by [4]. See also [3].

5.9 Exercises

Exercise 5.1 We extend the language from Sect. 4.2 with Boolean operators as described in Exercise 4.1.

Extend the type-check function in Fig. 5.2 to handle these new constructions as described above.

Exercise 5.2 We extend the language from Sect. 4.2 with floating-point numbers as described in Exercise 4.2.

(a) Extend the type checking functions in Figs. 5.2, 5.3 and 5.4 to handle these extensions.
(b) We now add implicit conversion of integers to floats to the language, using the rules: Whenever an operator has one integer argument and one floating-point argument, the integer is converted to a float. Similarly, if a condition expression (`if-then-else`) has one integer branch and one floating-point branch, the integer branch is converted to floating-point. Extend the type checking functions from question a) above to handle this.

Exercise 5.3 The type check function in Fig. 5.2 tries to guess the correct type when there is a type error. In some cases, the guess is arbitrarily chosen to be `int`, which may lead to spurious type errors later on. A way around this is to have an extra type: `unknown`, which is only used during type checking. If there is a type error, and there is no basis for guessing a correct type, `unknown` is returned (the error is still reported, though). If an argument to an operator is of type `unknown`, the type checker should not report this as a type error but continue as if the type is correct. The use of an `unknown` argument to an operator may make the result `unknown` as well, so these can be propagated arbitrarily far.

Change Fig. 5.2 to use the `unknown` type as described above.

Exercise 5.4 We look at a simple language with an exception mechanism:

$$
\begin{aligned}
S &\rightarrow \texttt{throw}\ \mathbf{id} \\
S &\rightarrow \texttt{try}\ S\ \texttt{catch}\ \mathbf{id} \Rightarrow S \\
S &\rightarrow S\ \texttt{or}\ S \\
S &\rightarrow \texttt{other}
\end{aligned}
$$

A `throw` statement throws a named exception. This is caught by the nearest enclosing `try-catch` statement (i.e., where the `throw` statement is in the left sub-statement of the `try-catch` statement) that uses the same name, whereby the statement after the arrow in the `try-catch` statement is executed. An `or` statement is a non-deterministic choice between the two statements, so either one can be executed. `other` is a statement that does not throw any exceptions.

We want the type checker to ensure that all possible exceptions are caught and that no `try-catch` statement is superfluous, i.e., that the exception it catches can, in fact, be thrown by its left sub-statement.

Write type-check functions that implement these checks. Hint: Let the type of a statement be the set of possible exceptions it can throw.

Exercise 5.5 In Exercise 4.5, we extended the example language with functional values and implemented these in the interpreter.

Extend the type-checking functions in Figs. 5.2, 5.3 and 5.4 to statically type check the same extensions.

Hint: You should check a function definition when it is declared rather than when it is used.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers; Principles, Techniques and Tools. Addison-Wesley (2007)
2. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978)
3. Mogensen, T.Æ.: Programming Language Design and Implementation. Springer, Cham, Switzerland (2022). https://link.springer.com/book/10.1007/978-3-031-11806-7
4. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, MA, USA (2002)