*10*

## Type Systems

*Type systems are a subset of constraints – they implement type calculations and type checks. These can be relatively complex, so special support beyond general-purpose constraint checking is useful. In this chapter we discuss what type systems do in general, we discuss various strategies for computing types, and we provide the usual examples with Xtext, MPS and Spoofax.*

Let us start with a definition of type systems from Wikipedia:

A type system may be defined as a tractable syntactic framework for classifying phrases according to the kinds of values they compute. A type system associates types with each computed value. By examining the flow of these values, a type system attempts to prove that no type errors can occur. The type system in question determines what constitutes a type error, but a type system generally seeks to guarantee that operations expecting a certain kind of value are not used with values for which that operation makes no sense.

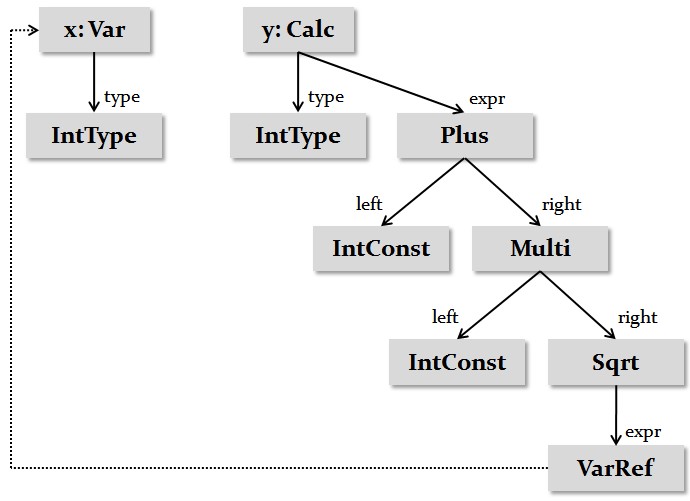
In summary, type systems associate types with program elements and then check whether these types conform to predefined typing rules. We distinguish between dynamic type systems, which perform the type checks as the program executes, and static type systems, where type checks are performed ahead of execution, mostly based on type specifications in the program. This chapter focuses exclusively on static type checks[[1]](#footnote-1).

### 10.1 Type Systems Basics

To introduce the basic concepts of type systems, let us go back to the example used at the beginning of the section on syntax. As a reminder here is the example code, and Fig. 10.1 shows the abstract syntax tree.

**var** x: **int**;

**calc** y: **int** = 1 + 2 \* sqrt(x)

F

Using this example, we can illustrate in more detail what type systems have to do:

*Declare Fixed Types* Some program elements have fixed types. They don’t have to be derived or calculated – they are always the same and known in advance. Examples include the integer constants **IntConst** (whose type is **IntType**), the square root concept **sqrt** (whose type is **double**), as well as the type declarations themselves (the type of **IntType** is **IntType**, the type of **DoubleType** is **Double- Type**).

*Derive Types* For some program elements, the type has to be derived from the types of other elements. For example, the type of a **VarRef** (the variable reference) is the type of the referenced variable. The type of a variable is the type of its declared type. In the example above, the type of **x** and the reference to **x** is **IntType**.

*Calculate Common Types* Most type systems have some kind of type hierarchy. In the example, **IntType** is a subtype of **DoubleType** (so **IntType** can be used wherever **DoubleType** is expected). A type system has to support the specification of such subtype relationships. Also, the type of certain program elements may be calculated from the arguments passed to them; in many cases the resulting type will be the "more general" one based on the subtyping relationship. Examples include the **Plus** and **Multi** concepts: if the left and right arguments are two **IntTypes**, the result is an **IntType**. In the case of two **DoubleType**s, the result is a **DoubleType**. If an **IntType** and a **DoubleType** are used, the result is a **DoubleType**, the more general of the two.

*Type Checks* Finally, a type system has to check for type errors and report them to the user. To this end, a language specifies type constraints or type checks that are checked at editing time by the type system based on the calculated types. In the example, a type error would occur if something with a **DoubleType** were assigned to an **IntType** variable.

The type of a program element is generally not the same as its language concept2. Different instances of the same concept can

have different types: a **+** calculates its type as the more general of the two arguments. So the type of each **+** instance depends on the types of the arguments of that particular instance.

Types are often represented with the same technology as the language concepts. As we will see, in the case of MPS types are just nodes, i.e. instances of concepts. In Xtext, we use **EObjects**, i.e. instances of **EClasses** as types. In Spoofax, any ATerm can be used as a type. In all cases, we can even define the concepts as part of the language. This is useful, because most of the concepts used as types also have to be used in the program text whenever types are explicitly declared (as in **var x: int**).

### 10.2 Type Calculation Strategies

Conceptually, the core of a type system can be considered to be a function **typeof** that calculates the type for a program element. This function can be implemented in any way suitable; after all, it is just program code. However, in practice, three approaches seem to be used most: recursion, unification and pattern matching. We will explore each of these conceptually, and then provide examples in the tool sections.

of the number

#### 10.2.1 Recursion

Recursion is widely used in computer science and we assume that every reader is familiar with it. In the context of type systems, the recursive approach for calculating a type defines a polymorphic function **typeof**, which takes a program element

and returns its type, while calling itself3 to calculate the types on which its own type depends. Consider. the following example grammar (using Xtext notation):

|  |
| --- |
| LocalVarDecl:  "var" name=**ID** ":" type=Type ("=" init=Expr)?; |

The following examples are structurally valid example sentences:

|  |  |  |
| --- | --- | --- |
| **var** i: | **int** | // 1 |
| **var** i: | **int** = 42 | // 2 |
| **var** i: | **int** = 33.33 | // 3 |
| **var** i = 42 | | // 4 |

|  |
| --- |
| typeof( LocalVarDecl lvd ) { **if** !isSpecified( lvd.type ) && !isSpecified( lvd.init ) **raise error**  **if** isSpecified( lvd.type ) && !isSpecified( lvd.init ) **return** typeof( lvd.type )  **if** !isSpecified( lvd.type ) && isSpecified( lvd.init ) **return** typeof( lvd.init ) |

Let’s develop the pseudo-code for **typeof** function the **LocalVarDecl**. A first attempt might look as follows:

|  |
| --- |
| typeof( LocalVarDecl lvd ) { **return** typeof( lvd.type )  }  typeof( IntType it ) { **return** it } typeof( DoubleType dt ) { **return** dt } |

Notice how **typeof** for **LocalVarDecl** recursively calls **typeof** for its **type** property. Recursion ends with the **typeof** functions for the types; they return themselves. This implementation successfully calculates the type of the **LocalVarDecl**, but it does not address the type check that makes sure that, if an **init** expression is specified, it has the same type (or a subtype) of the **type** property. This could be achieved as follows:

|  |
| --- |
| typeof( LocalVarDecl lvd ) { **if** isSpecified( lvd.init ) { assert typeof( lvd.init ) isSameOrSubtypeOf typeof( lvd.type )  } **return** typeof( lvd.type ) } |

Notice (in the grammar) that the specification of the variable type (in the **type** property) is also optional. So we have created a somewhat more elaborate version of the function:

// otherwise... **assert** typeof( lvd.init ) isSameOrSubtypeOf typeof( lvd.type ) **return** typeof( lvd.type ) }

#### 10.2.2 Unification

Unification is the second well-known approach to type calculation. Let’s start with a definition from Wikipedia:

Unification is an operation . . . which produces from . . . logic terms a substitution which . . . makes the terms equal modulo some equational theory.

While this sounds quite sophisticated, we have all used unification in high-school for solving sets of linear equations. The "equational theory" in this case is algebra. Here is an example:

1. 2 \* x == 10
2. x + x == 10
3. x + y == 2 \* x + 5

*Substitution* refers to assignment of values to **x** and **y**. A solution for this set of equations is **x := 5, y := 10**.

Using unification for type systems means that language developers specify a set of type equations which contain type variables (cf. the **x** and **y**) as well as type values (the numbers in the above example). Some kind of engine is then trying to make all equations **true** by assigning type values to the type variables in the type equations. The interesting property of this approach is that there is no distinction between typing rules and type checks. We simply specify a set of equations that must be **true** for the types to be valid4. If an equation

cannot be satisfied for any assignment of type values to type variables, a type error is detected. To illustrate this, we return to the **LocalVarDecl** example introduced above.

|  |  |  |
| --- | --- | --- |
| **var** i: | **int** | // 1 |
| **var** i: | **int** = 42 | // 2 |
| **var** i: | **int** = 33.33 | // 3 |
| **var** i = 42 | | // 4 |

The following two type equations constitute the complete type system specification. The **:==:** operator expresses type equation (left side must be the same type as right side), **:<=:** refers

to subtype-equation (left side must be same type or subtype of right side, the pointed side of **<** points to the "smaller", the more specialized type)[[2]](#footnote-2).

**typeof**( LocalVarDecl.type ) :>=: **typeof**( LocalVarDecl.init ) **typeof**( LocalVarDecl ) :==: **typeof**( LocalVarDecl.type )

Let us look at the four examples cases. We use capital letters for free type variables. In the first case, the **init** expression is not given, so the first equation is ignored. The second equation can be satisfied by assigning **T**, the type of the variable declaration, to be **int**. The second equations acts as a type derivation rule and defines the type of the overall **LocalVarDecl** to be **int**.

|  |  |
| --- | --- |
| // var i: **int**  **typeof**( **int** ) :>=: **typeof**( **int** ) | // ignore |
| **typeof**( T ) :==: **typeof**( **int** ) | // T := **int** |

In the second case the **type** and the **init** expression are given, and both have types that can be calculated independently of the equations specified for the **LocalVarDecl** (they are fixed). So the first equation has no free type variables, but it is **true** with the type values specified (two **int**s). Notice how in this case the equation acts as a type check: if the equation were not **true** for the two given values, a type error would be reported. The second equation works the same as above, deriving **T** to be **int**.

|  |  |
| --- | --- |
| // var i: **int** = 42  **typeof**( **int** ) :>=: **typeof**( **int** ) | // **true** |
| **typeof**( T ) :==: **typeof**( **int** ) | // T := **int** |

The third case is similar to the second case; but the first equation, in which all types are specified, is not **true**, so a type error is raised.

|  |  |
| --- | --- |
| // var i: **int** = 33.33  **typeof**( **int** ) :>=: **typeof**( double ) | // **error**! |
| **typeof**( T ) :==: **typeof**( **int** ) | // T := **int** |

Case four is interesting because no variable type is explicitly specified; the idea is to use *type inference* to derive the type from the **init** expression. In this case there are two free variables in the equations; substituting both with **int** solves both

equations6.

// var i = 42 inference! **typeof**( U ) :>=: **typeof**( **int** ) // U := **int typeof**( T ) :==: **typeof**( U ) // T := **int**

To further illustrate how unification works, consider the following example, in which we specify the typing rules for array types and array initializers:

**var** i: **int**[] **var** i: **int**[] = {1, 2, 3} **var** i = {1, 2, 3}

Compared to the **LocalVarDecl** example above, the additional complication in this case is that we need to make sure that *all* the initialization expressions (inside the curly braces) have the same or compatible types. Here are the typing equations:

|  |
| --- |
| **typevar** T **foreach** ( e: init.elements ) **typeof**(e) :<=: T  **typeof**( LocalVarDecl.type ) :>=: **new** ArrayType(T) **typeof**( LocalVarDecl ) :==: **typeof**( LocalVarDecl.type ) |

We introduce an additional type variable **T** and iterate over all the expression in the array initializer, establishing an equation between each of these elements and **T**. This results in a set of equations that *each* must be satisfied7. The only way to achieve

this is for all array initializer members to be of the same (sub)type. In the examples, this makes **T** to be **int**. The rest of the equations works as explained above. Notice that if we’d written **var i = {1, 33.33, 3}**, then **T := double**, but the equations would still work because we use the **:>=:** operator.

#### 10.2.3 Pattern Matching

In pattern matching we simply list the possible combinations of types in a big table. Cases that are not listed in the table will result in errors. For our **LocalVarDecl** example, such a table could look like this:

**typeof(type) typeof(init) typeof(LocalVarDecl)**

**int int int int** - **int** - **int int**

**double double double double** - **double** - **double double double int double**

To avoid repeating everything for all valid types, variables could be used. **T+** refers to **T** or subtypes of **T**.

**typeof(type) typeof(init) typeof(LocalVarDecl)**

**T T T T** - **T**

- **T T**

**T T+ T**

Pattern matching is used for binary operators in MPS and also for matching terms in Spoofax.

### 10.3 Xtext Example

|  |  |
| --- | --- |
| JVM’s type system is available9. It is not as versatile as it could |  |
| be, since it is limited to JVM-related types and cannot easily be used for languages that have no relationship with the JVM, such as C or C++.  As a consequence of this limitation and the fact that Xtext is widely used, two third-party libraries have been developed: |  |
| the Xtext Typesystem Framework (developed by the author10), |  |
| and XTypes (developed by Lorenzo Bettini11. In the remainder of this section we will look at the Xtext Typesystem Frame- |  |
| work12). |  |
| *Xtext Typesystem Framework* The Xtext Typesystem Framework is based on the recursive approach. It provides an interface **ITypesystem** with a method **typeof(EObject)** which returns the type for the program element passed in as an argument. In its simplest form, the interface can be implemented manually with arbitrary Java code. To make sure type errors are reported as part of the Xtext validation, the type system framework has to be integrated into the Xtext validation framework manually: |  |

|  |  |
| --- | --- |
| As we have discussed in Section 10.1, many type systems rely |  |
| on a limited set of typing strategies13. The **DefaultTypesystem** |  |

Up to version 1.0, Xtext provided no support for implementing type systems8. In version 2.0 a type system integrated with the

|  |
| --- |
| @Inject **private** ITypesystem ts;  @Check(CheckType.NORMAL)  **public void** validateTypes( EObject m ) { ts.checkTypesystemConstraints( m, **this** ); } |

class implements **ITypesystem** and provides support for declaratively specifying these strategies. In the code below, a simplified version of the type system specification for the cooling language, the **initialize** method defines one type (the type of the **IntType** is a clone of itself) and defines one typing constraint (the **expr** property of the **IfStatement** must be a Boolean). Also, for types which cannot be specified declaratively, an operation **type(..)** can be implemented to programmatically define types. The example below shows this for the **NumberLiteral**.

**public class** CLTypesystem **extends** DefaultTypesystem { **private** CoolingLanguagePackage cl = CoolingLanguagePackage.eINSTANCE;

type of an element from one of its properties, calculating the type as the common type of its two arguments.

|  |
| --- |
| @Override  **protected void** initialize() { useCloneAsType(cl.getIntType()); ensureFeatureType(cl.getIfStatement(), cl.getIfStatement\_Expr(), cl.getBoolType());  }  **public** EObject type( NumberLiteral s, TypeCalculationTrace trace ) { **if** ( s.getValue().contains(".")) { **return** create(cl.getDoubleType());  } **return** create(cl.getIntType()); }  } |

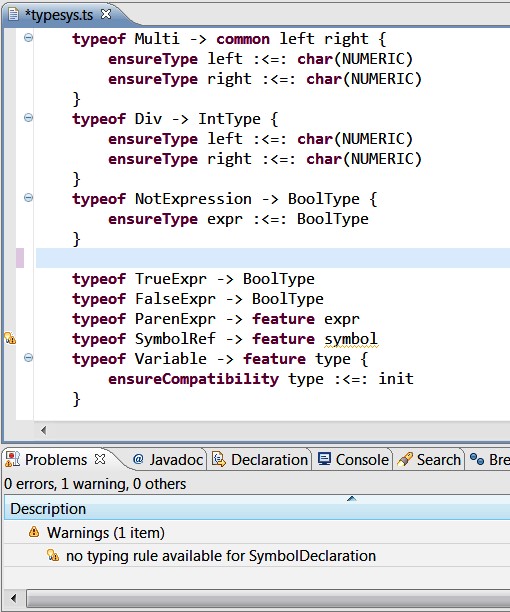
|  |  |
| --- | --- |
| to the specification in Java:   * The notation is much more concise compared to the API. * Referential integrity and code completion with the target language meta model is provided. * If the typing rules are incomplete, a static error is shown in the editor, as opposed to getting runtime errors during initialization of the framework (see the warning in Fig. 10.2). * **Ctrl-Click** on a property jumps to the typing rule that defines the type for that property.   *Type System for the Cooling Language* The complete type system for the cooling language is 200 lines of DSL code, and another 100 lines of Java code. We’ll take a look at some representative examples. Primitive types usually use a copy of |  |
| themselves as their type15: |  |

In addition to the API used in the code above, the Typesystem Framework also comes with a textual DSL to express typing rules (Fig. 10.2 shows a screenshot). From the textual type system specification, a generator generates the implementation of the Java class that implements the type system using the APIs14. The DSL provides the following advantages compared

**typeof** BoolType -> **clone typeof** IntType -> **clone typeof** DoubleType -> **clone typeof** StringType -> **clone**

Alternatively, since all primitive types extend an abstract meta class **PrimitiveType**, this could be shortened to the following, where the **+** operator specifies that the rule applied for the specified concept and all its subconcepts:

**typeof** PrimitiveType + -> **clone**



For concepts that have a fixed type that is different from the concept itself (or a clone), the type can be specified explicitly:

**typeof** StringLiteral -> StringType

|  |  |
| --- | --- |
| Type systems are most important, and most interesting, in the context of expressions. Since all expressions derive from the abstract **Expr** concept, we can declare that this class is abstract, |  |
| and hence no typing rule is given16: |  |

The notation provided by the DSL groups typing rules and type checks for a single concept. The following is the typing information for the **Plus** concept. It declares the type of **Plus** to be the common type of the **left** and **right** arguments (the "more general" one) and then adds two constraints that check that the **left** and **right** argument are either **int**s or **double**s[[3]](#footnote-3).

**typeof** Plus -> **common** left right { **ensureType** left :<=: IntType, DoubleType **ensureType** right :<=: IntType, DoubleType

}

The typing rules for **Equals** are also interesting. It specifies that the resulting type is **boolean**, that the **left** and **right** arguments must be **COMPARABLE**, and that the left and right arguments be compatible. **COMPARABLE** is a *type characteristic*: this can be considered as collection of types. In this case it is **IntType**, **DoubleType** and **BoolType**. The **:<=>:** operator describes unordered compatibility: the types of the two properties **left** and **right** must either be the same, or **left** must be a subtype or **right**, or vice versa.

|  |
| --- |
| **characteristic** COMPARABLE {  IntType, DoubleType, BoolType  } **typeof** Equals -> BoolType { **ensureType** left :<=: **char**(COMPARABLE) **ensureType** right :<=: **char**(COMPARABLE) **ensureCompatibility** left :<=>: right } |

There is also support for *ordered* compatibility, as can be seen from the typing rule for **AssignmentStatement** below. It has no type (it is a statement), but the **left** and **right** argument must exhibit ordered compatibility: they either have to be the same types, or **right** must be a subtype of **left**, *but not vice versa*:

**typeof** AssignmentStatement -> **none** { **ensureCompatibility** right :<=: left

}

The framework uses the generation gap pattern, i.e. from the DSL-based type specification, a generator creates a class **CLTypesystemGenerated** (for the cooling language) that contains all the code that can be derived from the type system specification. Additional specifications that cannot be expressed with the DSL (such as the typing rule for **NumberLiteral** shown earlier, or type coercions) can be implemented in Java[[4]](#footnote-4).

### 10.4 MPS Example

MPS includes a DSL for type system rule definition. It is based on unification, and pattern matching for binary operators. We

discuss each of them.

*Unification* The type of a **LocalVariableReference** is calculated with the following typing rule[[5]](#footnote-5). It establishes an equation between the type of the **LocalVariableReference** itself and the variable it references. **typeof** is a built-in operator that returns the type for its argument.

|  |
| --- |
| **rule** typeof\_LocalVariableReference { **applicable for concept** = LocalVariableReference **as** lvr **overrides false**  **do** {  **typeof**( lvr ) :==: **typeof**( lvr.variable ); }  } |

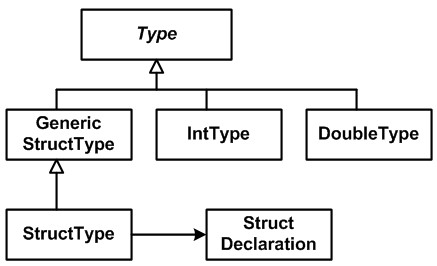
The rules for the Boolean **NotExpression** contains two equations. The first one makes sure that the negated expression is Boolean. The second one types the **NotExpression** itself to be Boolean20.

**typeof**( notExpr.expression ) :==: **new** node<BooleanType>(); they can be instantiated like any other

**typeof**( notExpr ) :==: <**boolean**>;

|  |  |
| --- | --- |
|  |  |
| A more interesting example is the typing of **struct**s. Consider the following C code: |  |

|  |
| --- |
| **struct** Person { **char**\* name; **int** age;  }  **int** addToAge( Person p, **int** delta ) { **return** p.age + delta; } |

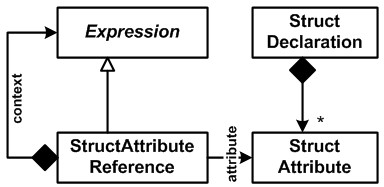
.

At least two program elements have to be typed: the parameter **p** as well as the **p.age** expression. The type of the **FunctionParameter** concept is the type of its **type** property. This is not specific to the fact that the parameter refers to a **struct**.

**typeof**( parameter ) :==: **typeof**( parameter.type );

|  |  |
| --- | --- |
|  |  |
| The language concept that represents the **Person** type in the parameter is a **StructType**. A **StructType** refers to the **Struct-** |  |
| **Declaration** whose type it represents, and extends **Type**, which acts as the super type for all types in mbeddr C21.  **p.age** is an instance of a **StructAttributeReference**. It is |  |

|  |
| --- |
| **concept** StructAttributeReference **extends** Expression **implements** ILValue  **children**:  Expression context 1  **references**:  StructAttribute attribute 1 |

defined as follows (see Fig. 10.4 as well as the code below). It is an **Expression**, owns another expression property (on the left of the dot), as well as a reference to a **StructAttribute** (**name** or **age** in the example).

The typing rule for the **StructAttributeReference** is shown in the code below. The **context**, the expression on which we use the dot operator, has to be a **GenericStructType**, or a subtype thereof (i.e. a **StructType** which points to an actual **StructDeclaration**). Second, the type of the whole expression is the type of the reference **attribute** (e.g., **int** in the case of **p.age**).

**typeof**( structAttrRef.context ) :<=: **new node**<GenericStructType>(); **typeof**( structAttrRef ) :==: **typeof**( structAttrRef.attribute );

This example also illustrates the interplay between the type system and other aspects of language definition, specifically scopes. The referenced **StructAttribute** (on the right side of the dot) may only reference a **StructAttribute** that is part of the the **StructDeclaration** that is referenced from the **StructType**. The following scope definition illustrates how we access the type of the expression from the scoping rule:

|  |
| --- |
| **link** {attribute} **search scope**:  (model, **scope**, referenceNode, linkTarget, enclosingNode)->join(  ISearchScope | sequence<node< >>) { node<> exprType = **typeof**( referenceNode.expression ); **if** (exprType.isInstanceOf(StructType)) { **return** (exprType **as** StructType).struct.attributes;  } **else** {  **return null**;  }  } |

*Pattern Matching* As we will discuss in the chapter on language extension and composition, MPS supports incremental extension of existing languages. Extensions may also introduce new types, and, specifically, may allow existing operators to be used with these new types. This is facilitated by MPS’ use for pattern matching in the type system, specifically for binary operators such as **+**, **>** or **==**. As an example, consider the introduction of complex numbers into C. It should be possible to write code like this:

**complex** c1 = (1, 2i); **complex** c2 = (3, 5i);

**complex** c3 = c1 + c2; // results in (4, 7i)

The **+** in **c1 + c2** should be the **+** defined by the original C language[[6]](#footnote-6). Reusing the original **+** requires that the typing rules

defined for **PlusExpression** in the original C language will now have to accept complex numbers; the original typing rules must be extended. To enable this, MPS supports *overloaded operations containers*. The following container, taking from the

the original plus and the new plus for complex numbers. This would not be very convenient from a usability perspective. By reusing the original plus we avoid this problem.

mbeddr C core language, defines the type of **+** and **-** if both arguments are **int** or **double**.

|  |
| --- |
| **overloaded operations rules** binaryOperation  **operation concepts**: PlusExpression | MinusExpression **left operand type**: <**int**> **right operand type**: <**int**>  **operation type**: (**operation**, leftOperandType, rightOperandType)->node<> {  <**int**>; }  **operation concepts**: PlusExpression | MinusExpression **left operand type**: <double> **right operand type**: <double> **operation type**: (**operation**, leftOperandType, rightOperandType)->node<> {  <double>;  } |

To integrate these definitions with the regular typing rules, the following typing rule must be written23. The typing rules tie in

with overloaded operation containers via the **operation type** construct:

|  |
| --- |
| **rule** typeof\_BinaryExpression { **applicable for concept** = BinaryExpression **as** binex  **do** {  node<> optype = **operation type**( binex , left , right ); **if** (optype != **null**) { **typeof**(binex) :==: optype;  } **else** { **error** "operator " + be.**concept**.name + " cannot be applied to " + left.**concept**.name + "/" + right.**concept**.name -> be;  }  }  } |

The important aspect of this approach is that overloaded operation containers are *additive*. Language extensions can simply contribute *additional* containers. For the complex number example, this might look like the following: we declare that as soon as one of the arguments is of type complex, the resulting type will be complex as well.

|  |
| --- |
| PlusExpression | MinusExpression **one operand type**: <complex> **operation type** :  (**operation**, leftOperandType, rightOperandType)->node<> {  <complex>;  } |

The type system DSL in MPS covers a large fraction of the type system rules encountered in practice. The type system for BaseLanguage, which is an extension of Java, is implemented in this way, as is the C type system in mbeddr. However, for exceptional cases, procedural BaseLanguage code can be used to implement typing rules as well.

### 10.5 Spoofax Example

Spoofax’ rewrite rules support both the recursive approach and pattern matching in specifying type systems. However, in most projects the recursive approach will be found. Therefore we will focus on it in the remainder of the section.

*Typing Rules in Spoofax* For typing rules in Spoofax, the basic idea is to use rewrite rules to rewrite language constructs to their types. For example, the following rule rewrites integer numbers to the numeric type. This is an example of assigning a fixed type to a language element.

type-of: Int(value) -> NumType()

Similarly, we can rewrite a **+** expression to the numeric type:

type-of: Add(exp1, exp2) -> NumType()

However, it is good practice to assign types only to well-typed language constructs. Thus, we should add type checks for the subexpressions:

|  |
| --- |
| type-of:  Add(exp1, exp2) -> NumType() **where**  <type-of> exp1 => NumType();  <type-of> exp2 => NumType() |

Spoofax allows for multiple typing rules for the same language construct. This is particular useful for typing overloaded operators, since each case can be handled by a separate typing rule. For example, when the operator **+** is overloaded to support string concatenation, we can add the following typing rule:

|  |
| --- |
| type-of:  Add(exp1, exp2) -> StringType() **where**  <type-of> exp1 => StringType();  <type-of> exp2 => StringType() |

*Persistence of Typing Information* Spoofax stores information about the definition sites of names in an in-memory data structure called the index. This typically includes information about types. For example, the type of property and variable references is initially not available at these references, but only at the declaration. But when Spoofax discovers a declaration, it stores its type in the index. Since declaration and references are annotated with the same URI, this information can also be accessed at references. Consider the following name binding rules which also involve type information:

Property(p, t): **defines** Property p **of type** t

Param(p, t): **defines** Variable p **of type** t

These rules match property and parameter declarations, binding their name to **p** and their type to **t**. Spoofax stores this type in the index as an information about the property or parameter name. In the typing rules for variable references and property accesses, we need to retrieve this type information from the index:

|  |
| --- |
| type-of:  Var(name) -> <index-lookup-type> name  type-of:  PropAccess(exp, name) -> <index-lookup-type> name |

Both rules rewrite references to the type of their definition sites. First, the definition of a reference is looked up in the index. Next, this definition is rewritten to its type. This uses the **index-lookup-type** rule, which implements the actual type lookup in the index.

In the previous example, the type was explicitly declared in property and parameter declarations. But the type of a definition site is not always explicitly declared. For example, variable declarations in Mobl come with an initial expression, but without an explicit type24.

expression. **var** x = 42;

The type of **x** is the type of its initial expression **42**, that is, **NumType()**. To make this type of **x** explicit, we need to calculate the type of the initial expression. The following name binding rule makes this connection between name binding and type system:

|  |
| --- |
| Declare(v, e):  **defines** Variable v **of type** t **in subsequent scope**  **where** e **has type** t |
| *Additional Types* In Spoofax, types are represented as terms. The constructors for these terms are specified in the syntax definition as labels to productions. Without the ability to define additional constructors, type systems are restricted to types which users can explicitly state in programs, for example in variable declarations. But many type systems require additional types which do not originate from the syntax of the language. Typical examples are top and bottom types in type | | |  |
| hierarchies25. For example, Java’s type system has a special | | |  |
| type for **null** values at the bottom of its type hierarchy, which | | |  |

cannot be used as a type in Java programs. Spoofax allows constructors for additional types in signatures to be defined:

|  |
| --- |
| **signature constructors**  FunType: List(Type) \* Type -> Type |

This defines an additional constructor **FunType** for **Type**. In general, a constructor definition is of the form **cons: Arg-1 \* ...\* Arg-n -> Sort**, where **cons** is the constructor name, **Sort** is the sort this constructor contributes to, and **Arg-1** to **Arg-n** are the sorts of its arguments. In the example, the first subterm should be a list of parameter types (**List(Type)**), while the second subterm should be the return type. We can employ the so-defined function type in the typing rules for function definitions and calls:

|  |
| --- |
| Function(f, p\*, t): **defines** Function f **of type** FunType(t\*, t)  **where** p\* **has type** t\* |
| type-of:  Call(name, arg\*) -> type **where**  <index-lookup-type> name => FunType(type\*, type) |

*Type Constraints* Like any other constraint, type constraints are specified in Spoofax by rewrite rules which rewrite language constructs to errors, warnings or notes. For example, we

can define a constraint on additions26:

|  |
| --- |
| constraint-error:  exp -> (exp, $[Operator + cannot be applied to arguments  [<pprint> type1], [<pprint> type2].])  **where**  !exp => Add(exp1, exp2); <not(type-of)> exp; type1 := <type-of> exp1; type2 := <type-of> exp2 |

|  |  |
| --- | --- |
|  |  |
| *Type Compatibility* Whether two types are compatible is again defined by rewrite rules. These rules rewrite a pair of types to the second element of the pair, if the first one is compatible with it. In the simplest case, both types are the same: |  |

is-compatible-to: (type, type) -> type

This rule only succeeds if it gets a tuple with two types that are identical (they both match the same variable **type**). A type might also be compatible with any type with which its supertype is compatible:

is-compatible-to:

(subtype, type) -> type **where**

supertype := <supertype> subtype; <is-compatible-to> (supertype, type)

Here, the subtype relation is defined by a rewrite rule, which rewrites a type to its supertype:

supertype: IntType() -> FloatType()

This approach only works for type systems in which each type has at most one supertype. When a type system allows for multiple supertypes, we have to use lists of supertypes and need to adapt the rule for **is-compatible-to** accordingly:

|  |
| --- |
| supertypes: IntType() -> [ FloatType() ]  is-compatible-to: (subtype, type) -> type **where** supertype\* := <supertypes> subtype;  <fetch-elem(is-compatible-to(|type))> supertype\* |

Here, **fetch-elem** tries to find an element in a list of supertypes, which is compatible to **type**. It uses a variant of the rule **is-compatible-to** in order to deal with a list of types. This variant does not rewrite a pair of types, but only the first type. The second type is passed as a parameter to the rewrite rule. It can be defined in terms of the variant for pairs:

is-compatible-to(|type2): type1 -> <is-compatible-to> (type1, type2)

The compatibility of types can easily be extended to compatibility of lists of types:

|  |
| --- |
| is-compatible-to:  (type1\*, type2\*) -> type\* **where** type\* := <zip(is-compatible-to)> (type1\*, type2\*) |

A list **type1\*** of types is compatible with another list **type2\*** of types, if each type in **type1\*** is compatible with the corresponding type in **type2\***. **zip** pairs up the types from both lists, rewrites each of these pairs by applying **is-compatible-to** to them, and collects the results in a new list **type\***.

With the extension for lists, we can define a constraint for function calls, which ensures that the types of the actual arguments are compatible with the types of the formal parameters:

|  |
| --- |
| constraint-error:  Call(name, arg\*) -> (arg\*,  $[Function [name] cannot be applied to arguments  [<pprint> arg-type\*].])  **where** fun-type := <index-lookup-type> name ; !fun-type => FunType(para-type\*, type) ; arg-type\* := <map(type-of)> arg\* ;  <not(is-compatible-to)> (arg-type\*, par-type\*) |

1. . [↑](#footnote-ref-1)
2. . [↑](#footnote-ref-2)
3. . [↑](#footnote-ref-3)
4. e [↑](#footnote-ref-4)
5. . [↑](#footnote-ref-5)
6. n [↑](#footnote-ref-6)