First Edition, Version 1.0

Please cite this book as:

Claudius Ptolemaeus, Editor,
System Design, Modeling, and Simulation using Ptolemy II, Ptolemy.org, 2014.
http://ptolemy.org/books/Systems.

14

The Type System

Edward A. Lee, Marten Lohstroh, and Yuhong Xiong

Contents

14.1 Type Inference, Conversion, and Conflict **508**
14.1.1 Automatic Type Conversion 511
Sidebar: What is a Lattice? 513
14.1.2 Type Constraints 514
14.1.3 Type Declarations 515
14.1.4 Backward Type Inference 517
14.2 Structured Types **520**
14.2.1 Arrays 520
14.2.2 Records 522
14.2.3 Unions 523
14.2.4 Functions 523
14.3 Type Constraints in Actor Definitions **524**
Sidebar: Object Types 525
Sidebar: Invoking Methods on Object Tokens 526
Sidebar: Petite and Unsigned Byte Data Types 526
14.4 Summary **531**
Sidebar: Monotonic Functions in Type Constraints 532

In a programming language, a **type system** associates a type with each variable. A **type** is logically a family of values that the variable can take on. For example, the type *double* is the set of all double-precision floating point numbers represented by 64 bits according to the IEEE 754 standard for floating-point arithmetic. A **strong type system** will prevent a program from using the 64-bit value of a *double* variable as, for example, a pointer into memory or a *long* integer (Liskov and Zilles, 1974). Java has a strong type system; C does not. A good introduction to type systems is given by Cardelli and Wegner (1985).

The Ptolemy II type system associates a type with each port and parameter in a model. Ptolemy II uses **type inference**, where the types of parameters and ports are inferred based on their usage. Types need not be declared by the model builder, usually.

A programming language where types are checked at compile time is said to be **statically typed**. In Ptolemy II, types are checked just prior to execution of a model, between the preinitialize and initialize phases of execution. Since this happens once, before execution of the model, we consider Ptolemy II to be statically typed.

Although the type system is a strong one (a port will not receive a token that is incompatible with its declared type, for example), there are loopholes. In particular, users can define their own actors in Java, and these actors may not behave well. For example, an actor may declare an output port to be of type *int* and then attempt to send a *string* through that port. To catch such errors, Ptolemy II also performs run-time type checks. Although it is rare (unless you write your own actors), it is possible to build models that will exhibit type errors only during execution. These errors will not be detected by the static type checker.

Fortunately, with the help of static type checking, run-time type checks can be performed automatically when a token is sent out from a port. The run-time type checker simply compares the type of a produced token against the (static) type of the output port. This way, a type error is detected at the earliest possible time (when the token is produced, rather than when it is used). However, this does not guarantee that a type of token that an actor accepts is indeed compatible with the operation it implements. A run-time type error may therefore also be thrown by the actor itself, particularly if the actor is written incorrectly.

The Ptolemy II type system supports **polymorphism**, where actors can operate on a variety of data types. To facilitate the construction of polymorphic actors, the type system offers a mechanism called **automatic type conversion**, which allows a component to receive multiple data types by automatically converting them to a single data type, assuming that the conversion can be done without loss of information. Polymorphism greatly increases the reusability of actors in the presence of static typing, especially in combination with **type inference**. In this chapter we describe how these mechanisms are integrated into the Ptolemy II static type checking framework, with emphasis on how they help build correct models.

14.1 Type Inference, Conversion, and Conflict

In Ptolemy II models, types of ports are inferred from the model, subject to constraints imposed by the actors and the infrastructure. We will explain exactly how these constraints come about and how the inference is performed, but first we can develop an intuitive understanding of what happens.

Example 14.1: Consider the example shown in Figure 14.1. This model calculates and displays $1 - \pi$. (This is a silly model, since the entire model could be replaced by the expression `1 - PI`, but it serves to illustrate the type system.) The model includes an attribute called *ShowTypes*, which can be found in the `Utilities→Analysis` library. This attribute causes Vergil to display the type of each port next to the port (in addition to the name of the port, if the name would otherwise be displayed). As you can see in the figure, initially the type of each port is *unknown*.

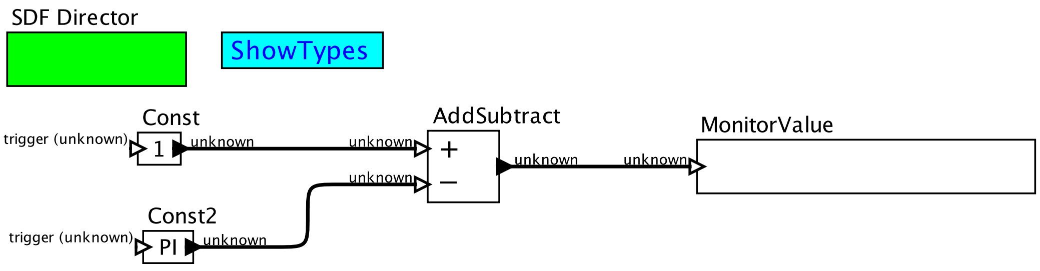


Figure 14.1: A simple example for illustrating type inference and conversion. [online]

During the first stage of execution, preinitialize, the types of the ports are inferred, after which the display is eventually updated as shown in Figure 14.2. The types of the output ports of the Const actors are determined by their *value* parameters, which are `1` and `PI`, respectively. If you change a *value* parameter to, for example, `1+i`, then the output type will become *complex*. If you change it to `{1, 2}`, then the output type will become *arrayType(int, 2)*, an array with *int* elements and length 2.

Notice in Figure 14.2 that the input ports of the AddSubtract actor have both resolved to *double*. The AddSubtract actor imposes a constraint that its two input ports must have the same type. When the AddSubtract actor receives an *int* token from the Const actor, the input port will automatically convert the token to type *double*. We will explain in detail what conversions are allowed, but intuitively, a conversion is allowed if no information is lost.

Notice in Figure 14.2 that the output port of the AddSubtract actor and the input port of MonitorValue have also resolved to *double*. The MonitorValue actor can accept any input type, since it simply displays a string representation of the token, and every token has a string representation.

The previous example illustrates that the type of a parameter in one part of a model can have far-reaching consequences in other parts of the model. The type system ensures consistency. It is common when building models to raise type errors, as illustrated by the following example.

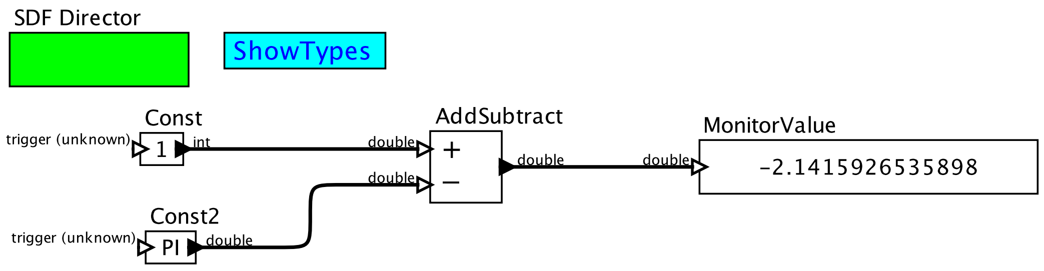


Figure 14.2: After execution.

Example 14.2: Consider the example shown in Figure 14.3. In that example, we have replaced MonitorValue with SequencePlotter, and we have changed the Const actor to produce a *complex* value. Type inference determines that the output of AddSubtract is *complex*. But SequencePlotter requires an input of type *double*. A *complex* token cannot be losslessly converted to a *double* token, so upon executing the model you will get the following exception:

```
ptolemy.actor.TypeConflictException:  Type conflicts occurred on
the following inequalities:
  (port .TypeConflict.AddSubtract.output:  complex) <=
  (port .TypeConflict.SequencePlotter.input:  double)
  in .TypeConflict
```

This error message reports that a type constraint in the model cannot be satisfied. That type constraint is that the type of the output port of AddSubtract must be *less than or equal to* ($\leq$) the type of the input port of SequencePlotter. Further, it reports that the output port has type *complex*, while the input port has type *double*. The offending ports and their containers are then highlighted in the model as shown in the figure.

In the above example, a type constraint is given as an inequality, an assertion that the type of one port must be "less than or equal to" the type of another. What does this mean?

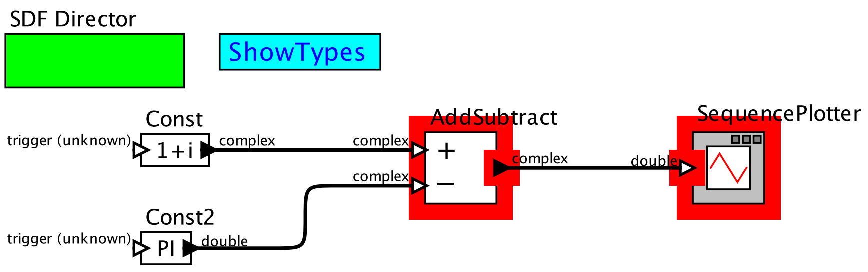


Figure 14.3: Type conflict.

Intuitively, one type is less than another if it can be lossless converted to that other type. We examine this inequality relation next.

14.1.1 Automatic Type Conversion

The allowed automatic type conversions are represented in Figure 14.4, which depicts the **type lattice** of Ptolemy II. In this diagram, a conversion from a first type to a second type is allowed if there is an upward path from the first type to the second type in the diagram. This relationship implies a partial order on types (see sidebar on page 513), so we might say that a conversion is allowed if the first type is less than or equal to the second type. This partial order has an elegant mathematical structure called a lattice (see sidebar on page 513) that enables efficient type inference and type checking.

Automatic conversions occur when an actor retrieves data from its input port.* The types of ports are determined prior to execution, and run-time type checking guarantees that tokens sent through an output port are compatible with the types of downstream input ports (i.e., a conversion to such type is allowed by the type lattice). This is due to the type constraints that are imposed by connections between ports. These type constraints are explained in Section 14.1.2. If a token is not compatible, the run-time type checker will throw an exception before the token is sent. Hence, run-time type errors are detected as early as possible.

A type conversion can also be forced in the expression language using the `cast` function, one of many built-in functions available in the expression language. An expression of the form `cast(newType, value)` will convert the specified value into the specified type. See Section 13.4.3 and Table 13.15 for information about the `cast` function.

The type lattice is constructed based on a principle of lossless conversion. A conversion is allowed automatically as long as important information about value of data tokens is not lost. Such conversions are referred to as **widening conversions** in Java. For instance, converting a 32-bit signed integer to a 64-bit IEEE double precision floating point number is allowed, since every integer can be represented exactly as a floating point number. On the other hand, data type conversions that lose information are not automatic.

* Some actors disable automatic type conversion on their input ports because they do not need the conversion. For example, AddSubtract and Display accept any token type, because they make use of methods that are inherited by all token types. These actors disable automatic conversion by invoking:
`portName.setAutomaticTypeConversion(false).`

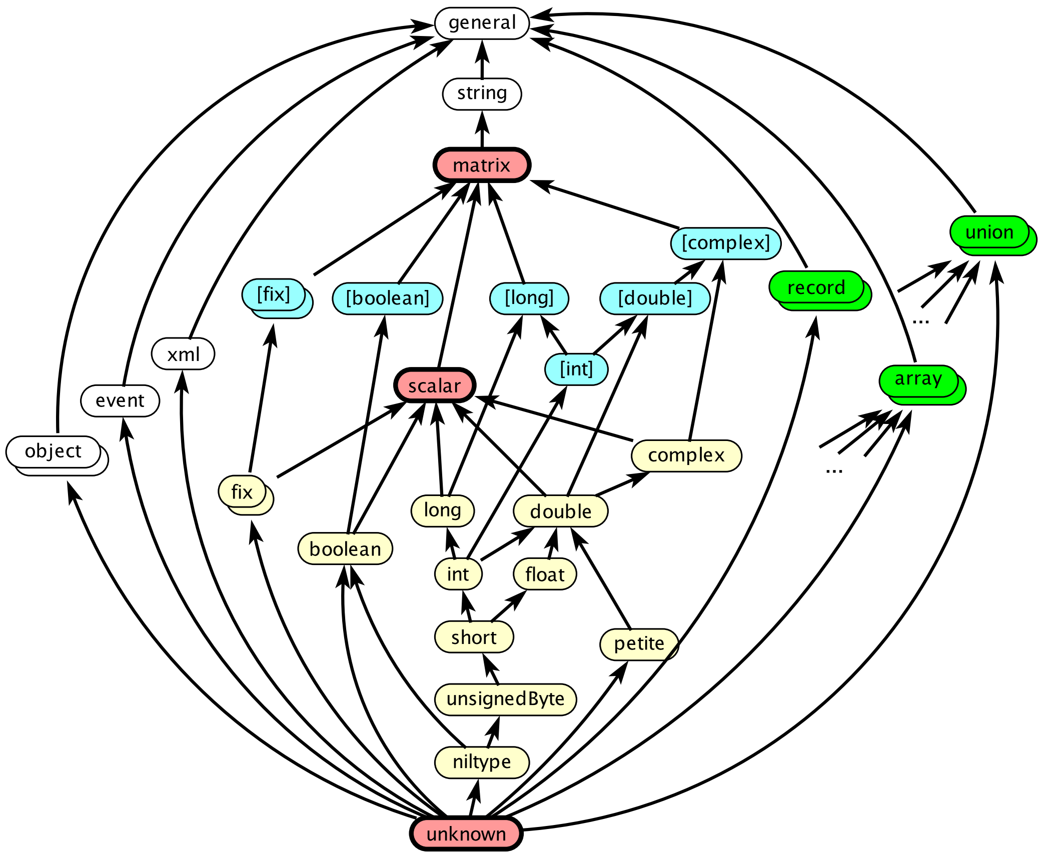


Figure 14.4: The Type Lattice. Types that cannot be instantiated are shown with bold outlines (and pink fill). Scalar types are lightly filled (in yellow), matrix types are slightly more darkly filled (in cyan), and composite types still more darkly filled (in green). The composite types and the *fix* types are infinite sublattices, as suggested by their double icons.

Sidebar: What is a Lattice?

A lattice is a mathematical structure that has properties that enable efficiently solving type constraints. A lattice is a set with particular kind of **order relation** related to CPOs (see sidebar on page 182). See Davey and Priestly (2002) for more details.

First, a total order is an ordering over a set S, denoted $\leq$, where any two elements of the set are ordered. Specifically, for any $x, y \in S$, either $x \leq y$ or $y \leq x$ (or both, in which case $x = y$). For example, if the set S is the set of integers, and $\leq$ denotes the ordinary arithmetic ordering, then $(S, \leq)$ is a total order.

A partial order relaxes the constraint that any two elements be ordered. An example of a partial order is $(S, \leq)$, where S is a set of sets and $\leq$ is the subset relation, usually denoted $\subseteq$. Specifically, if A, B are both sets in S, then it may be that neither $A \subseteq B$ nor $B \subseteq A$. E.g., let $A = \{1, 2\}$ and $B = \{2, 3\}$; then neither is a subset of the other.

Another partial order is the **prefix order** on strings. Let S be the set of sequences of alphabetic characters, for example. Then for two strings $x, y \in S$, we write $x \leq y$ is x is a prefix of y. E.g., if $x =$abc and $y =$abcd, then $x \leq y$. If $z =$bc, then neither $x \leq z$ nor $z \leq x$ holds. Formally, a partial order is a set S and a relation $\leq$, such that for all $x, y, z \in S$,

- $x \leq x$ (the order is reflexive),
- if $x \leq y$ and $y \leq z$, then $x \leq z$ (the order is transitive), and
- if $x \leq y$ and $y \leq x$, then $x = y$ (the order is antisymmetric).

The least upper bound (LUB), if it exists, of a subset $U \subseteq S$ of a partial order $(S, \leq)$ is the least element $x \in S$ such that for every $u \in U$, $u \leq x$. The **greatest lower bound** (**GLB**) of U, if it exists, is the greatest element $x \in S$ such that for every $u \in U$, $x \leq u$. E.g., in the prefix order, if $x =$abc, $y =$abcd, and $z =$bc, then the LUB of $\{x, y\}$ is y. The LUB of $\{x, z\}$ does not exist. The GLB of $\{x, y\}$ is x. The GLB of $\{x, z\}$ is the empty string, which is a prefix of all strings.

A **lattice** is a partial order $(S, \leq)$ for which every subset of S has a GLB and a LUB. The subset order is a lattice, because the LUB can be found with set union, and the GLB can be found with set intersection. The prefix order on strings, however, is not a lattice, because two strings may not have a LUB. The prefix order is a **lower semi-lattice**, however, because the GLB of a set of strings always exists.

14.1.2 Type Constraints

A model imposes a number of constraints that drive type inference. A constraint is expressed as an inequality between the types of two ports. It requires one port to have a type that is less than or equal to (losslessly convertible to) the type of the other port.

In a Ptolemy II topology, the **type compatibility rule** requires an output port to have a type that is less than or equal to all inputs to which it is connected, as follows:

$$\mathit{outType} \leq \mathit{inType} \tag{14.1}$$

This constraint guarantees that there is an allowed automatic conversion that can be performed during data transfer. Every connection between an output port and an input port establishes a type constraint that enforces this rule.

Example 14.3: In Figure 14.2, the Const actor produces type *int*, while the AddSubtract actor receives type *double*. In Figure 14.4, we see that *int* is less than *double*, so the type compatibility rule is satisfied.

In addition to the constraints imposed by the connections between actors, most actors also impose constraints. For example, the AddSubtract actor declares that its output type is greater than or equal to its input types, and that the types of its two input ports are equal. An equality constraint is equivalent to two inequality constraints, as in:

$$\begin{aligned} \mathit{plus} &\leq \mathit{minus} \\ \mathit{minus} &\leq \mathit{plus}, \end{aligned}$$

where *plus* and *minus* are the input ports of the AddSubtract actor.

Many actors impose a **default type constraint** that requires an unconstrained input to be less than or equal every unconstrained output. By default, actors implicitly include this type constraint for every set of input and output ports that have no explicit type constraint.

Example 14.4: Some actors operate on tokens without regard for the actual types of the tokens. For example, the DownSample does not care about the type of token

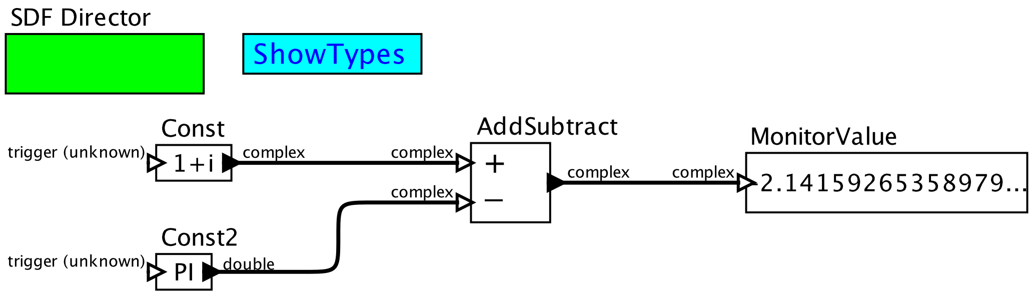


Figure 14.5: Type conflict of Figure 14.3 resolved by using an actor that can accept any input type, MonitorValue. [online]

going through it, so it does not explicitly declare any type constraints. The default type constraint enables type information to propagate through this actor from its input to its output.

By default, actors that leave their input ports undeclared will have the type of the input port determined by the upstream model (unless the model has enabled backward type inference, as explained below in Section 14.1.4).

Example 14.5: The MonitorValue actor, which displays the value of tokens it receives, can accept any type of input. By default, it leaves its input undeclared, which results in the type resolving to whatever is provided upstream. For example, Figure 14.5 resolves the type conflict of Figure 14.3 by replacing the SequencePlotter with a MonitorValue actor. The resolved type of the input, *complex*, is determined by the upstream actors.

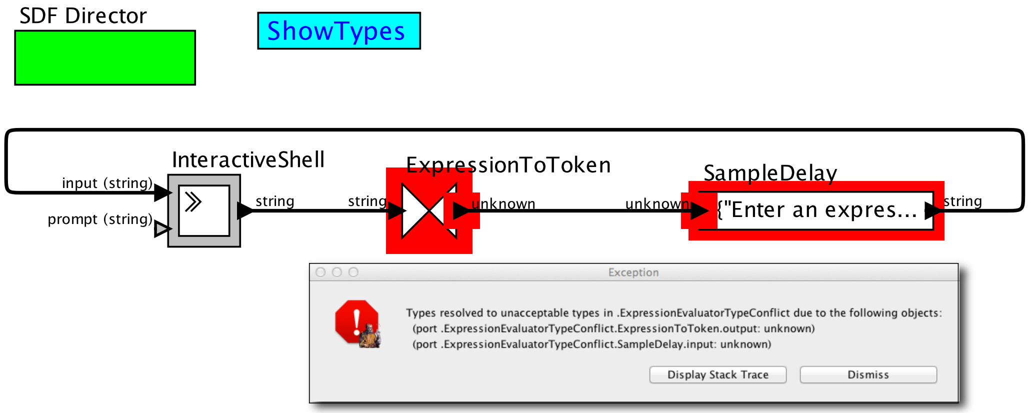
14.1.3 Type Declarations

Sometimes, there is not enough information in a model to infer types from the sources of data.

Example 14.6: Consider for example the model in Figure 14.6. This model is intended to evaluate expressions entered by a user in a shell, but type resolution fails, as shown. The ExpressionToToken actor takes an input string, which is expected to be an expression in the Ptolemy expression language (see Chapter 13), and evaluates the expression. The result of evaluation is produced on the output. There is no way to anticipate what the user might type in the shell, so there is not enough information to infer types. The type of the output of the ExpressionToToken remains *unkonwn*.

Such difficulties can be fixed by enabling backward type inference (discussed below), or by explicitly declaring the type of a port, as illustrated in the next example.

Example 14.7: We can force the type of output of the ExpressionToToken actor to be of type *general*, as shown in Figure 14.7. Downstream types resolve to *general* as well.



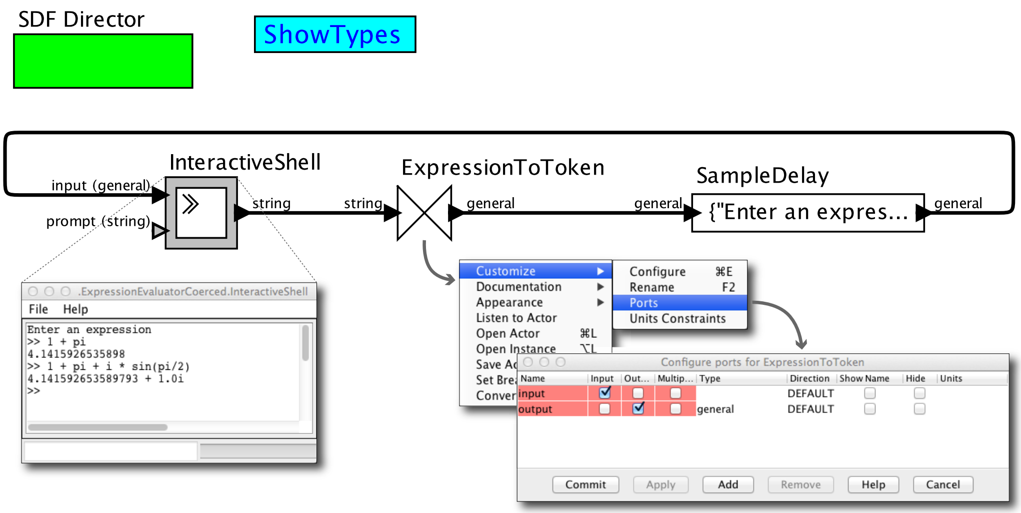
```
Types resolved to unacceptable types in .ExpressionEvaluatorTypeConflict due to the following objects:
  (port .ExpressionEvaluatorTypeConflict.ExpressionToToken.output: unknown)
  (port .ExpressionEvaluatorTypeConflict.SampleDelay.input: unknown)
```

Figure 14.6: A model intended to evaluate expressions entered by a user, but for which there is not enough information for types to be inferred from the sources of data.

In the type column of the port dialog, you can enter any expression in the expression language. Whatever type that expression resolves to will be the declared type of the corresponding port. For clarity, Ptolemy II provides some built in variables that designate a type. The variable named `general`, for instance, evaluates to a token of type *general*. Similarly, the variable named `double` evaluates to a token of type *double* (which happens to have value 0.0, but the value immaterial). Table 14.8 lists the predefined variable names and their corresponding types.

14.1.4 Backward Type Inference

In all the examples discussed so far, type inference propagates forward in models, with each constant or fixed output type causing downstream types to resolve. That is, type information travels in the same direction that the tokens are sent during execution. The type compatibility rule given by (14.1) imposes no useful constraints on output ports, because it is always satisfied if *outType* has type *unknown*, the bottom element of the type lattice. Nevertheless, **forward type inference** is usually sufficient because sources of data in most models provide specific type information about those data.



```
Enter an expression
>> 1 + pi
4.1415926535898
>> 1 + pi + i * sin(pi/2)
4.141592653589793 + 1.0i
>>
```

Name	Input	Out...	Multip...	Type	Direction	Show Name	Hide	Units
input	✓	☐	☐		DEFAULT	☐	☐	
output	☐	✓	☐	general	DEFAULT	☐	☐	

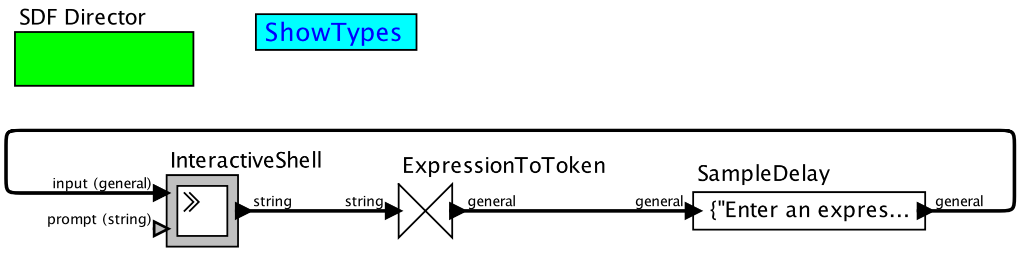
Figure 14.7: Types of ports declared by entering a type into the type column of the port configuration dialog. [online]

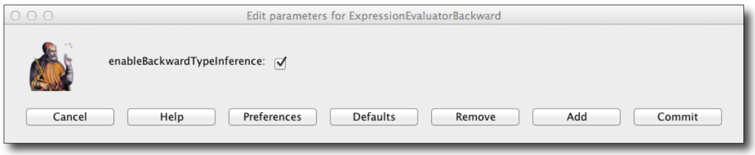
BaseType field	**Expression**	**Description**
UNKNOWN	`unknown`	bottom element of the data type lattice
ARRAY_BOTTOM		*array* of unknown type
BOOLEAN	`boolean`	boolean (true or false)
BOOLEAN_MATRIX	`[boolean]`	matrix of booleans
UNSIGNED_BYTE	`unsignedByte`	unsigned byte
COMPLEX	`complex`	complex number
COMPLEX_MATRIX	`[complex]`	complex matrix
FLOAT	`float`	32-bit IEEE floating-point number
DOUBLE	`double`	64-bit IEEE floating-point number
DOUBLE_MATRIX	`[double]`	matrix of doubles
FIX	`fixedpoint`	fixed-point data type
FIX_MATRIX	`[fixedpoint]`	matrix of fixed-point numbers
SHORT	`short`	16-bit integer
INT	`int`	32-bit integer
INT_MATRIX	`[int]`	matrix of 32-bit integers
LONG	`long`	64-bit integer
LONG_MATRIX	`[long]`	matrix of 64-bit integers
OBJECT	`object`	object type
ACTOR		actor type
XMLTOKEN		XML type
SCALAR	`scalar`	scalar number
MATRIX		matrix of unknown type
STRING	`string`	string
GENERAL	`general`	any type
EVENT		event (empty token)
PETITE		a double constrained to be between -1 and 1
NIL	`nil`	nil type
RECORD		record type

Figure 14.8: Type constants defined in the BaseType class with their corresponding name in the expression language, if there is one.

The models in Figures 14.6 and 14.7, however, do not have this property. First, since the model forms a loop, there is no clear "source" of data. Every actor is both upstream and downstream of every other actor. Moreover, the ExpressionToToken actor, by nature of what it does, cannot provide any specific information about the type of its output.

The Ptolemy II type system optionally provides **backward type inference** to solve this problem. To enable backward type inference, set the *enableBackwardTypeInference* parameter to true at the top level of the model, as shown in Figure 14.9. This has three effects. First, it causes certain actors that do not impose restrictions on the data received at input ports to declare those ports to have type *general*. This includes InteractiveShell and Display, for example. Second, it allows type constraints to propagate upstream. Specifically, it adds an additional constraint to the type compatibility rule of (14.1). The additional constraint is that the type of each output port is required to be greater than or equal to the greatest lower bound (GLB) of the types of all input ports to which it is connected. Third, for each actor that does not explicitly constrain the type relationships of its port, it imposes a default type constraint that the types of its input ports are greater than or equal to the GLB of the types of its output ports. These additional constraints are sufficient for the types to resolve to the same solution that we achieved with type coercion in Figure 14.7.




Edit parameters for ExpressionEvaluatorBackward

enableBackwardTypeInference: ✓

Figure 14.9: Enabling backward type inference allows type constraints to propagate upstream. [online]

The root of the problem is the ExpressionToToken actor, which cannot itself be specific about its output type. Any constraints on its output type would have to result from how its output tokens are used, rather than from the actor itself. In Figure 14.9, the InteractiveShell input accepts type *general*, which therefore propagates upstream to the output of the ExpressionToToken actor. When there are undeclared output port types, such as on the ExpressionToToken actor, backward type inference finds the the most general type that is compatible with downstream type constraints.

14.2 Structured Types

14.2.1 Arrays

Structured types include those tokens which aggregate other tokens of arbitrary type, such as array and record types. As described in Section 13.3, an array is an ordered list of tokens, all of which have the same type. Records contain a set of labeled tokens, like a struct in the C language. It is useful for grouping multiple pieces of related information together. In the type lattice in Figure 14.4, record types are incomparable with all the base types except *unknown*, *string*, and *general*. Array types are a bit more complex because any type is less than an array of that type in the type lattice. This is hinted at in the figure with the disconnected lines at the bottom of the array type. Note that the lattice nodes *array* and *record* actually represent an infinite number of types, so the type lattice is infinite.

For any type a, the following type relation holds,

$$a < \{a\}.$$

A value can be losslessly converted to an array of values. Moreover,

$$a < b \quad \Rightarrow \quad \{a\} < \{b\}.$$

Combining these, we see that the definition is recursive, so

$$a < \{a\} < \{\{a\}\} < \{\{\{a\}\}\} \cdots$$

Example 14.8: *int* $\leq$ *double*, so the following all hold:

$$\begin{array}{rcl} int & < & \{int\} \\ \{int\} & < & \{double\} \\ int & < & \{double\} \\ int & < & \{\{double\}\} \\ \cdots & & \end{array}$$

A consequence of these type relations is that there is an infinite path from any particular array type to the top of the type lattice. This can result in situations where type inference does not converge.

Example 14.9: In the model in Figure 14.10, the Expression actor constructs an array consisting of one element, its input token. When you attempt to run this model, an exception occurs that includes the message

```
Large type structure detected during type resolution
```

The reason for this is that there is no (finite) type that satisfies all the constraints. The SampleDelay actor requires its output to be at least an *int*, because it produces an initial *int*. But it also requires that its output be greater than equal to its input. The first input it will receive will have type $\{int\}$, and the second input will have type $\{\{int\}\}$, etc. The only possible type is an infinite nesting of arrays.

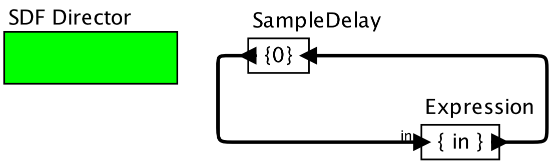


Figure 14.10: Example where type inference does not converge.

The Ptolemy II type system often (but not always) includes the *length* of an array in its type. If you explicitly query the type of a token, you can see this, as in the following command in the expression evaluator:

```
>> {1, 2}.getType()
object(arrayType(int,2))
```

Thus, `arrayType(int,2)` is the type of an array of length two, whereas `arrayType(int)` is the type of an array with indeterminate length. Including the length in the type allows the type system to detect more errors than otherwise.

Generally speaking, array types with specific length are incomparable with array types with different lengths, and can be converted to an array type with unknown length (and compatible element type). Scalars are convertible to array types with length 1.

One subtlety is that when you specify an array type with an expression `{ int }`, you are actually giving the type `arrayType(int, 1)`, which is more specific than you probably want. For this reason, unless you specifically want to constrain array types, it is better to specify an array type with `arrayType(int)`.

14.2.2 Records

The order relation between two record types follows the standard **depth subtyping** and **width subtyping** relations commonly used for such types (Cardelli, 1997). In depth subtyping, a record type c is a subtype of a record type d (i.e., $c \leq d$) if the fields of c are a subtype of the corresponding fields in d. For example,

```
    {x = string, y = int} <= {x = string, y = double}
```

In width subtyping, a record with more fields is a subtype of a record with fewer fields. For example, we have:

```
    {x = string, y = double, z = int} <= {x = string, y = double}
```

The width subtyping rule is a bit counterintuitive, as it implies a type conversion which loses information, discarding the extra fields of a record. However, it conforms with the "**is a**" interpretation of types, where $a \leq b$ if a is a b. In this case, the record with more fields "is an" instance of the record with fewer fields, whereas the reverse is not true.

14.2.3 Unions

Another structured type is the union type. It allows the user to create a token that can hold data of various types, but only one at a time. This is like the union construct in C. The union type is also called a **variant type** in the type system literature. The width subtyping relation for union type is the opposite to that of the record type. That is, a shorter union is a subtype of a longer one. Again, this corresponds with the "is a" interpretation of type relations.

A consequence of this width subtyping relation is that there are an infinite number of types from a particular union type to the top of the type lattice. This again means that type inference may not converge. The Ptolemy II type system truncates type inference after a finite number of steps, providing a heuristic that is a likely indicator of a type error.

14.2.4 Functions

One final structured type is the expression language function, described in Section 13.4.4. Functions can take several arguments and return a single value. The type system supports **function types**, where the arguments have declared types, and the return type is known. Function types are related in a way that is contravariant (oppositely related) between inputs and outputs. Namely, if `function(x:int, y:int) int` is a function with two integer arguments that returns an integer, then

```
function(x:int, y:int) int <= function(x:int, y:int) double
function(x:int, y:double) int <= function(x:int, y:int) int
```

The contravariant notion here is easiest to think about in terms of the automatic type conversion of one function into another. A function that returns *int* can be converted into a function that returns *double* by adding a conversion of the returned value from *int* to *double*. On the other hand, a function that takes an *int* cannot be converted into a function that takes a *double*, since that would mean that the function is suddenly able to accept *double* arguments when it could not before, and there is no automatic conversion from *double* to *int*. Functions that are lower in the type lattice assume less about their inputs and guarantee more about their outputs.

The names of arguments do not affect the relation between two function types, since argument binding is by the order of arguments only. Additionally, functions with different numbers of arguments (different **arity**) are considered incomparable.

The presence of function types that can be used as any other token results in what is commonly termed a higher-order type system. An example of the use of function tokens is given in Figure 2.44 and discussed in the sidebar on page 89.

14.3 Type Constraints in Actor Definitions

Section 12.4 introduces how to write actors in Java. In this section, we explain how to constrain types in the Java definition of an actor.

Prior to the execution of a model, during the setup phase, type constraints are gathered from all entities in the model (e.g., instances of TypedIOPort, TypedAtomicActor, or Parameter) that impose restrictions on their types. Actors can either set type constraints by storing them in the object instances of the concerning ports or parameters, or by setting them up using the `_customTypeConstraints` method of TypedAtomicActor.

A simple type constraint that is common to many actors is to ensure that the type of an output is greater than or equal to the type of a parameter. You can do so by putting the following statement in the constructor:

```
portName.setTypeAtLeast(parameterName);
```

or equivalently, by defining:

```
protected Set<Inequality> _customTypeConstraints() {
  Set<Inequality> result = new HashSet<Inequality>();
  result.add(new Inequality(parameterName.getTypeTerm(),
    portName.getTypeTerm()));
  return result;
}
```

This is called a **relative type constraint** because it constrains the type of one object relative to the type of another. Another form of relative type constraint forces two objects to have the same type, but without specifying what that type should be:

```
portName.setTypeSameAs(parameterName);
```

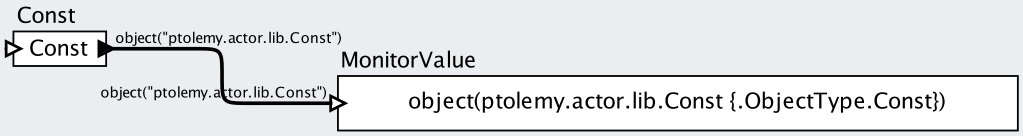
These constraints could be specified in reverse order,

```
parameterName.setTypeSameAs(portName);
```

which obviously means the same thing.

Sidebar: Object Types

The *object* type at the far left in Figure 14.4 is particularly powerful (and should be used with caution). A token of type *object* represents a Java object, such as a Ptolemy actor. Consider the following model:


Const
Const
object("ptolemy.actor.lib.Const")
object("ptolemy.actor.lib.Const")
MonitorValue
object(ptolemy.actor.lib.Const {.ObjectType.Const})

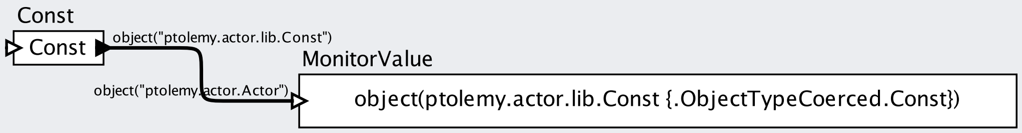
The *value* of the Const actor is set to `Const`, which evaluates to the Const actor itself. The Const actor's name is "`Const`," and the expression `Const` evaluates to the actor itself. Hence, the inferred type of the output port of Const is *object*, and its output will be the actor itself.

The *object* type is not one type, but an infinite number of types, as suggested by the double oval in Figure 14.4. There is a particular *object* type for every distinct Java class. The type of the output port above is `object("ptolemy.actor.lib.Const")`, because the Const actor is an instance of the Java class `ptolemy.actor.lib.Const`.

A type `object("A")` is less than `object("B")` if the Java class `A` is a subclass of Java class `B`. For example, `ptolemy.actor.lib.Const` implements the Java interface, `ptolemy.actor.Actor`, so

```
object("ptolemy.actor.lib.Const") <= object("ptolemy.actor.Actor")
```

In the following variant of the above model, the input port of the MonitorValue actor is coerced to `object("ptolemy.actor.Actor")`:


Const
Const
object("ptolemy.actor.lib.Const")
object("ptolemy.actor.Actor")
MonitorValue
object(ptolemy.actor.lib.Const {.ObjectTypeCoerced.Const})

A similar conversion can be accomplished with the `cast` function by doing

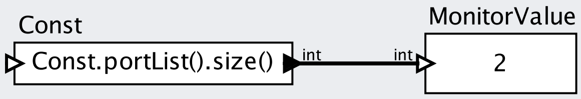
```
cast(object("ptolemy.actor.Actor"), Const)
```

The most general object type is *object* (without any argument). The pre-defined object token called `null` has this type.

Sidebar: Invoking Methods on Object Tokens

The expression language permits methods defined in the Java class of an object token (see sidebar on page 525). For example, in a model that contains an actor named C, the term `C` in an expression may refer to that actor.

Java methods may be invoked on the objects encapsulated in object tokens. For example, in the following model, the Const actor outputs the number of ports contained by the Const actor:



Sidebar: Petite and Unsigned Byte Data Types

The **petite** data type is used to represent real numbers in the range between −1.0 and 1.0 inclusive. It is used to emulate the behavior in certain specialized processors such as DSP processors (Digital Signal Processing), which sometimes use fixed-point arithmetic limited to this range of values. The *petite* type approximates this as a *double* limited to the range between −1.0 and 1.0 inclusive. In the expression language, a *petite* number is indicated by the suffix "p", and arithmetic operations saturate at −1.0 and 1 when results would lie outside this range. For example, using the expression evaluator, we get

```
>> 0.5p + 1.0p
1.0p
```

A data type that is sometimes useful for operating on raw data (e.g. packets arriving from a network) is the **unsigned byte**, designated as follows:

```
>> 1ub
1ub
>> -1ub
255ub
```

The same constraints can be expressed like this:

```
protected Set<Inequality> _customTypeConstraints() {
  result.add(new Inequality(parameterName.getTypeTerm(),
      portName.getTypeTerm());
  result.add(new Inequality(portName.getTypeTerm(),
      parameterName.getTypeTerm());
  return result;
}
```

The `_customTypeConstraints` method is particularly useful for actors that establish type constraints between ports that may be dynamically removed or added. For those actors, it is not safe to store the type constraints in the respective ports because constraints associated with no longer existing ports can persist and inadvertently cause type errors.

Another common type constraint is an **absolute type constraint**, which fixes the type of the port (i.e. making it **monomorphic** rather than polymorphic). This can specified as follows:

```
portName.setTypeEquals(BaseType.DOUBLE);
```

The above line declares that the port can only handle doubles. Figure 14.8 lists the type constants defined in the BaseType class. Another form of absolute type constraint imposes an upper bound on the type:

```
portName.setTypeAtMost(BaseType.COMPLEX);
```

which declares that any type that can be losslessly converted to ComplexToken is acceptable. By default, for any input port that has no declared type constraints, a default type constraint is automatically created that declares its type to be less than or equal to that of any output ports that have no declared type constraints. If there are input ports with no constraints, but no output ports lacking constraints, then those input ports will remain unconstrained. Conversely, if there are output ports with no constraints, but no input ports lacking constraints, then those output ports will remain be unconstrained. The latter is unacceptable, unless backward type inference is enabled. Default type constraints can be disabled by overriding the `_defaultTypeConstraints` method and having it return null.

A port can be declared to accept any token using following type constraint:

```
portName.setTypeAtMost(BaseType.GENERAL);
```

Example 14.10: An extension of Transformer of Figure 12.11 is shown in Figure 14.11. This SimplerScale is a simplified version of the Scale actor in the standard Ptolemy II library. This actor produces an output token on each firing with a value that is equal to a scaled version of the input. The actor is polymorphic in that it can support any token type that supports multiplication by the factor parameter. In the constructor, the output type is constrained to be at least as general as both the input and the factor parameter.

```java
public class SimplerScale extends Transformer {
     public SimplerScale(CompositeEntity container,
               String name)
               throws NameDuplicationException,
               IllegalActionException {
          super(container, name);
          factor = new Parameter(this, "factor");
          factor.setExpression("1");
          // set the type constraints.
          output.setTypeAtLeast(input);
          output.setTypeAtLeast(factor);
     }
     public Parameter factor;
     public Object clone(Workspace workspace)
               throws CloneNotSupportedException {
          SimplerScale newObject = (SimplerScale)super.
                    clone(workspace);
          newObject.output.setTypeAtLeast(newObject.input);
          newObject.output.setTypeAtLeast(newObject.factor);
          return newObject;
     }
     public void fire() throws IllegalActionException {
          if (input.hasToken(0)) {
               Token in = input.get(0);
               Token factorToken = factor.getToken();
               Token result = factorToken.multiply(in);
               output.send(0, result);
          }
     }
}
```

Figure 14.11: Actor with non-trivial type constraints.

Notice in Figure 14.11 how the `fire` method uses `hasToken` to ensure that no output is produced if there is no input. Furthermore, only one token is consumed from each input channel, even if there is more than one token available. This is generally the behavior of domain polymorphic actors. Notice also how it uses the `multiply` method of the `Token` class. This method is polymorphic. Thus, this scale actor can operate on any token type that supports multiplication, including all the numeric types and matrices.

An awkward complication when customizing type constraints is illustrated by the `clone` method in lines 12-18. In order for the actor to work properly in actor-oriented classes, relative type constraints that are set up in the constructor have to be repeated in the `clone` method.

The `setTypeAtLeast`, `setTypeAtMost`, `setTypeEquals`, and `setTypeSameAs` methods are part of the Typeable interface, which is implemented by ports and parameters. The `setTypeAtMost` method is usually invoked on input ports to declare a requirement that *input* tokens must satisfy, while the `setTypeAtLeast` method is usually invoked on *output* ports to declare a guarantee of the type of the output. The methods `_customTypeConstraints` and `_defaultTypeConstraints` are part of the base class TypedAtomicActor, and its subclasses can override those methods to customize the type constraints they impose.

Example 14.11: The constraint that the type of an input port can be no greater than double might be declared as:

```
inputPort.setTypeAtMost(BaseType.DOUBLE);
```

Note that the argument to `setTypeAtMost` and `setTypeEquals` is a type, whereas the argument to `setTypeAtLeast` is a Typeable object. This reflects the common usage, where `setTypeAtLeast` is declaring a dependency on externally provided types, whereas both `setTypeAtMost` and `setTypeEquals` declare constraints on externally defined types. The forms of the type inequalities that are specifiable by these methods also ensures that type inference is efficient and that the result of type inference is deterministic.

More complex type constraints arise from structured types, such as arrays and records. To declare that a parameter is an array of doubles, use:

```
parameter.setTypeEquals(new ArrayType(BaseType.DOUBLE));
```

This declares that a parameter or a port has a particular array type. A more flexible parameter might be able to contain an array of any type. This is expresses as follows:

```
parameter.setTypeAtLeast(ArrayType.ARRAY_BOTTOM);
```

In a more elaborate example, we might constrain the type of an output port to be no less than the element type of the array contained by a parameter (or an input port):

```
outputPort.setTypeAtLeast(ArrayType.arrayOf(parameter));
```

To declare that an output has a type greater than or equal to that of the elements of an input (or parameter) array, use:

```
outputPort.setTypeAtLeast(ArrayType.elementType(inputPort));
```

The above code implicitly constrains the input port to have an array type, but does not constrain the element types of that array. The above kinds of constraints appear in source actors such as DiscreteClock and Pulse, ArrayToSequence and SequenceToArray. Examining the source code for those actors can be instructive.

Another common constraint is that an input port of an actor receives a record with unconstrained fields. This constraint can be declared using the following code:

```
inputPort.setTypeAtMost(RecordType.EMPTY_RECORD);
```

Suppose you have an output port that may produce records with arbitrary fields. The above construct will not be sufficient since it does not declare any lower bound on the type, so at run time, the type will not be resolved to something useful. Instead, do:

```
outputPort.setTypeEquals(BaseType.RECORD);
```

This forces the type to resolve to the empty record. Any record with fields is a subtype of the empty record type, so this effectively declares the output to produce any record. Alternatively, enabling backward type inference allows the element type of the record to be inferred from the type constraints imposed by downstream actors.

To declare that a parameter can have a value that is any record, you can do:

```
param.setTypeAtMost(BaseType.RECORD);
```

but you will also need to specify a value for the parameter so that the type resolves to something concrete. To give a default value that is an empty record you can do:

```
    param.setToken(RecordToken.EMPTY_RECORD);
```

Two of the types, *matrix* and *scalar*, are union types. This means that an instance of this type can be any of the types immediately below them in the lattice. An actor may, for example, declare that an input port must be of type no greater than scalar:

```
    inputPort.setTypeAtMost(BaseType.SCALAR);
```

In this case, inputs of any type immediately below scalar in the type lattice will not be converted, except that the type of the input tokens will be reported as scalar. This is useful, for example, in actors that need to compare tokens, such as the Limiter actor. The `fire` method of that actor contains the code

```
if (input.hasToken(0)) {
    ScalarToken in = (ScalarToken) input.get(0);
    if ((in.isLessThan((ScalarToken) bottom.getToken()))
            .booleanValue()) {
        output.send(0, bottom.getToken());
    } else if ((in.isGreaterThan((ScalarToken) top.getToken()))
            .booleanValue()) {
        output.send(0, top.getToken());
    } else {
        output.send(0, in);
    }
}
```

This code relies on input port *in* and parameter *bottom* being declared to be at most scalar type, and ScalarToken being a base class for every token with type immediately below scalar. It then uses comparison methods defined in the ScalarToken class.

Type constraints in actors can get much more sophisticated than what we describe here. As always, the source code (and its extensive documentation) is the ultimate reference.

14.4 Summary

Ptolemy II includes a sophisticated type system that performs inference and checks for errors. It uses an efficient algorithm given by Rehof and Mogensen (1999), who prove that their algorithm has complexity that is linear in the number of occurrences of symbols in the type constraints. As a consequence, the algorithm scales well to large models. Most of the time, the type system makes it unnecessary for the builder of models to think about types. At it makes it easy to define actors that operate on a multiplicity of types, as most of the actors in the standard library do.

Sidebar: Monotonic Functions in Type Constraints

More sophisticated type constraints can be expressed using a **monotonic function** on the left-hand side of an inequality. A monotonic function f preserves the order of its arguments; that is

$$x_1 \leq x_2 \Rightarrow f(x_1) \leq f(x_2). \tag{14.2}$$

Using a monotonic function, it is possible to define a type that is dependent on other types in complicated ways. For example, the actor RecordDisassembler sets up a type constraint that forces each field of its input record to be of the same type as the output port with the same name as the field.

A monotonic function is specified by subclassing the abstract class MonotonicFunction and implementing the methods `getVariables` and `getValue`. The `getVariables` method returns the variables that the function takes as arguments. The `getValue` method returns the result of applying the function to the current value of the variables it depends on.

For example, the ConstructAssociativeType class subclasses MonotonicFunction. The variables returned by `getVariables` are the variables holding the types of a list of ports, such as the output ports of a RecordDisassembler actor. The `getValue` method returns a record type with fields matching the names of those ports and types matching the types of those ports.

It should be noted that the base class MonotonicFunction does not guarantee that its subclasses indeed behave monotonically. If a function that is not actually monotonic is used in a type constraint, then type resolution is no longer guaranteed to yield a unique result. The result may depend on the order in which constraints are applied.