5

Type System

Authors: *Edward A. Lee*
Steve Neuendorffer
Yuhong Xiong

5.1 Introduction

The computation infrastructure provided by the basic actor classes is not statically typed, i.e., the IOPorts on actors do not specify the type of tokens that can pass through them. This can be changed by giving each IOPort a type. One of the reasons for static typing is to increase the level of safety, which means reducing the number of untrapped errors [26].

In a computation environment, two kinds of execution errors can occur, trapped errors and untrapped errors. Trapped errors cause the computation to stop immediately, but untrapped errors may go unnoticed (for a while) and later cause arbitrary behavior. Examples of untrapped errors in a general purpose language are jumping to the wrong address, or accessing data past the end of an array. In Ptolemy II, the underlying language Java is quite safe, so errors rarely, if ever, cause arbitrary behavior.[1] However, errors can certainly go unnoticed for an arbitrary amount of time. As an example, figure 5.1 shows an imaginary application where a signal from a source is downsampled, then fed to a fast Fourier transform (FFT) actor, and the transform result is displayed by an actor. Suppose the FFT actor can accept ComplexToken at its input, and the behavior of the DownSample actor is to just pass every

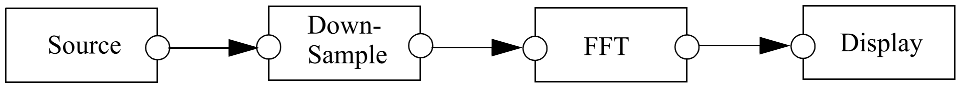

FIGURE 5.1. An imaginary Ptolemy II application

1. Synchronization errors in multi-thread applications are not considered here.

second token through regardless of its type. If the Source actor sends instances of ComplexToken, everything works fine. But if, due to an error, the Source actor sends out a StringToken, then the StringToken will pass through the sampler unnoticed. In a more complex system, the time lag between when a token of the wrong type is sent by an actor and the detection of the wrong type may be arbitrarily long.

In languages without static typing, such as Lisp and the scripting language Tcl, safety is achieved by writing defensive code. When safe execution is required, code must check manually at run-time whether the types of values are correct. In Ptolemy II, if we imitated this approach, we would have to require actors to check the type of the received tokens before using them. For example, the FFT actor would have to verify that the every received token is an instance of ComplexToken, or convert it to ComplexToken if possible. This approach places the burden of type checking on actor developers, distracting them from their development effort. It also relies on a policy that cannot be enforced by the system. Furthermore, since type checking is postponed to the last possible moment, the system does not have fail-fast behavior, so a system may generate an error message long after long after the error occurs, as illustrated in figure 5.1. To make matters worse, an actor may receive tokens from multiple sources. If a token with an incompatible type is received, it can be hard to identify the original source of the token. These potential problems can make debugging models unnecessarily difficult.

To address this and other issues discussed later, Ptolemy II includes static type checking. This approach is a significant extension of the simple type mechanism in Ptolemy Classic. In general-purpose statically-typed languages, such as C++ and Java, static type checking done by the compiler can find many potential program errors. However, execution of a model in Ptolemy II is more similar to an interpreted execution, and does not generally involve compilation. Nonetheless, static type checking of the model can still be used to detect modeling errors before actors fire. In figure 5.1, if the Source actor declares that its output port type is *String*, meaning that it will send out StringTokens upon firing, the static type checker will identify this type conflict in the topology.

In Ptolemy II, because actors may contain arbitrary Java code, static typing alone is not enough to ensure type safety at run-time. For example, even if the above Source actor declares its output type to be *Complex*, it may still attempt to send out a StringToken at run-time. For instance, the Source actor might contain a bug that incorrectly declares the type of a port. Hence run-time type checking is still necessary for the Ptolemy framework to guarantee that all actors receive tokens of an expected type. 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 type of the output port. This way, a type error is detected at the earliest possible time and less reliance on correct actor specifications is needed to ensure type safety. Additionally, actors can safely cast received tokens to the type of the input port without manually checking the type, making actor development easier.

We have found that type checking and type safety conversions can greatly increase our confidence in making use of reusable components. However, static typing does have some drawbacks. For instance, it often requires actor authors to explicitly declare what type(s) of data are allowed, making it more difficult to develop components. Ousterhout [116] also argues that static typing discourages the reuse of existing components.

> *"Typing encourages programmers to create a variety of incompatible interfaces, each interface requires objects of specific type and the compiler prevents any other types of objects from being used with the interface, even if that would be useful".*

In this chapter we will concentrate on two mechanisms for increasing the reusability of actors in the presence of static type checking. The first mechanism, called automatic type conversion, allows a component to receive multiple data types by automatically converting them to a single data type. A second mechanism, called type resolution or type inference, allows constructing data-polymorphic actors. Such actors operate in a similar way on different data types. This chapter will describe how these mechanisms are integrated into the Ptolemy II static type checking framework.

One mechanism that enables polymorphism in Ptolemy II is automatic type conversion. The allowed automatic data type conversions are represented in figure 5.2, called the *type lattice*. In this diagram, a conversion from one type to another is allowed if the first type appears below the second type in the diagram. This relationship implies a partial ordering of types, so we might say that a conversion is allowed if the first type is less than or equal to the second type.

Automatic conversions primarily occur during data transfer from one port to another. When a data token is received, it is automatically converted to the type of the input port receiving it. Along with the run-time type checking of sent data described earlier, this conversion implies that across every connection from an output port to an input, the type of the output must be the same as or lower than the type of the input. This requirement is called the type compatibility rule. For example, an output port with type *Int* can be connected to an input port with type *Double*, and tokens sent by the output port will be converted to type *Double* before being received. On the other hand, a *Double* to *Int* connection will generate a type error during static type checking, since no conversion is possible. These conversions

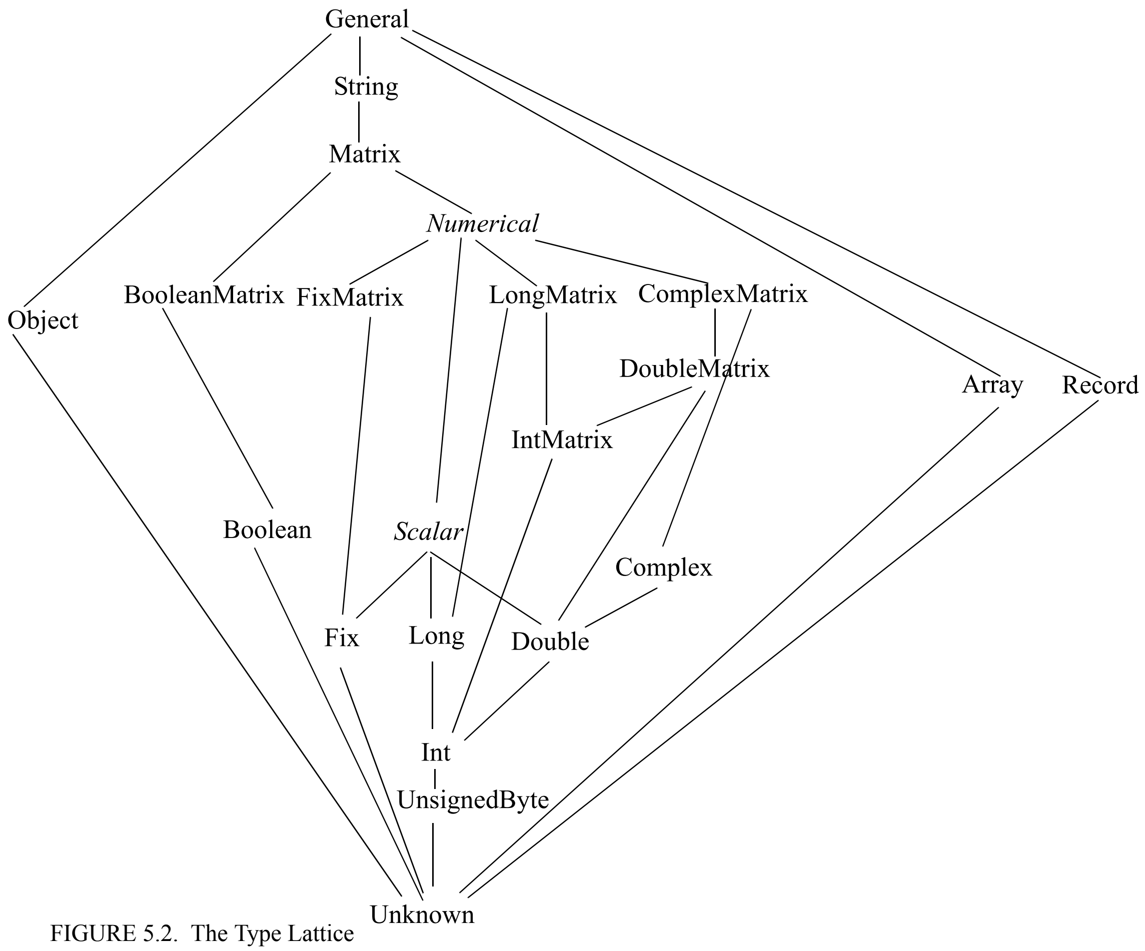

FIGURE 5.2. The Type Lattice

are performed transparently by the Ptolemy II system (actors are not aware it). Automatic conversions are also often performed in the data package when type-polymorphic operations are applied to values of different types.

The type lattice was constructed based on a principle of *lossless conversion*. A conversion is allowed automatically as long as important information about value of data tokens are 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 included in the type lattice of automatic conversions. In fact, the concentration on lossless conversions is somewhat arbitrary, but we find that it is relatively easy to use, since it minimizes unintentional loss of numerical precision.

While automatic type conversion allows an actor to receive data of different types, the operation performed by the actor is always performed on the same type of data, determined by the type of the ports. However, There are cases where an actor operates on tokens without regard for the actual types of the tokens. For example, the DownSample in figure 5.1 does not care about the type of token going through it; it works with any type of token. In general, the types on some or all of the ports of a polymorphic actor are not rigidly defined to specific types when the actor is written, so the actor can interact with other actors having different types, increasing reusability.

In Ptolemy Classic, the ports on type-polymorphic actors whose types are not specified are said to have ANYTYPE. ANYTYPE ports were allowed to be connected to ports of any other type. However, in the presence of such ports means that type safety cannot be ensured. Instead, Ptolemy II allows ports to have *undeclared type*, suggesting that the type of those ports has not been determined but cannot be assigned arbitrarily. Instead of being given as constants, the acceptable types on polymorphic actors are described by a set of type constraints. The type checker checks the applicability of a type-polymorphic actor in a model by finding specific types for ports that satisfy the type constraints. This process is called *type resolution* or *type inference,* and the specific types are called the resolved types. Assuming the type constraints of actors are consistent with the actor implementation, this technique can ensure the type safety of actor connections. Type constraints and the type resolution algorithm are described more completely in the next section.

In addition to ports, the parameters which are used to configure actors are also typed objects. By defining a uniform interface for setting up type constraints, Ptolemy II supports type constraints between parameters and ports, as well as between ports. This extends the range of type checking to allow parameters with arbitrary type, such as those that determine the values produced by source actors.

In Ptolemy II, typing does apply some restrictions on the interaction of actors. Particularly, actors cannot be interconnected arbitrarily if the type compatibility rule is violated. However, such models rarely make any sense, so the benefit of typing should far outweigh the inconvenience caused by this restriction. On the other hand, type declarations and type constraints help to clarify the interface of actors and makes them more manageable. Static typing also provide an opportunity for model compiler and circuit synthesis tools to generate type specialized code, when a Ptolemy system is synthesized to hardware, type information can be used for efficient synthesis. If the type checker asserts that a certain polymorphic actor will only receive IntTokens, then only hardware dealing with integers needs to be synthesized.

To summarize, Ptolemy II takes an approach of static typing coupled with run-time type checking. Lossless data type conversions during data transfer are automatically executed. Polymorphic actors are supported through type resolution.

5.2 Formulation

5.2.1 Type Constraints

In a Ptolemy II topology, the type compatibility rule imposes a type constraint across every connection from an output port to an input port. It requires that the type of the output port, *outType*, be the same as the type of the input port, *inType*, or less than *inType* under the type lattice. This can be written as an inequality:

$$outType \le inType \tag{5}$$

This constraint guarantees that there is an allowed automatic conversion that can be performed during data transfer. If both the *outType* and *inType* are declared, the static type checker simply checks whether this inequality is satisfied, and reports a type conflict if it is not.

In addition to the above constraint imposed by the topology, actors may also impose constraints. This happens when one or both of the *outType* and *inType* is undeclared, in which case the actor containing the undeclared port needs to describe the acceptable types through type constraints. All the type constraints in Ptolemy II are described in the form of inequalities like the one in (5). If a port or parameter has a declared type, its type appears as a constant in the inequalities. On the other hand, if a port or parameter has an undeclared type, its type is represented in the inequalities by a variable, called a type variable. The value of type variables are allowed to range over the elements of the type lattice. The type resolution algorithm resolves the values of type variables subject to the constraints of the model and the actors. If no solution exists, a type conflict error will be reported. As an example of the inequality constraints, consider figure 5.3.

The port of actor A1 has declared type *Int* and the ports of A3 and A4 have declared type *Double*. The types of the ports of A2, on the other hand, have been left undeclared. If the type variables of the undeclared types are α, β, and γ, then the type constraints from the topology are:

$Int \le \alpha$
$Double \le \beta$
$\gamma \le Double$

Now, assume A2 is a polymorphic adder, capable of doing addition for integer, double, and complex numbers, and the requirement is that it does not lose precision during the operation. Then the type constraints for the adder can be written as:

$\alpha \le \gamma$

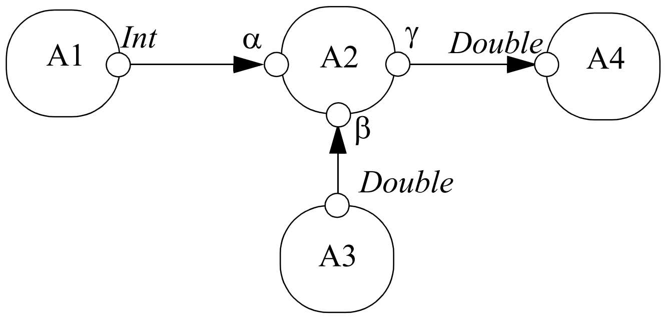

FIGURE 5.3. A topology with types.

$\beta \leq \gamma$

$\gamma \leq Complex$

The first two inequalities constrain the output precision to be no less than input, the last one requires that the data on the adder ports can be converted to *Complex* losslessly. These six inequalities form the complete set of constraints and are used by the type resolution algorithm to solve for α, β, and γ. Hence, the problem has been converted from type resolution into a problem of solving a set of inequalities. An efficient algorithm is available to solve constraints in finite lattices [121], which is described in the appendix through an example. This algorithm finds the set of most specific types for the undeclared types in the topology that satisfy the constraints, if they exist.

This inequality formulation is inspired by the type inference algorithm in ML [105]. There, equalities are used to represent type constraints. In Ptolemy II, the lossless type conversion hierarchy naturally implies inequality relation among the types. In ML, the type constraints are generated from program constructs. In a heterogeneous graphical programming environment like Ptolemy II, the system does not have enough information about the function of the actors, so actors must specify type information either by declaring port types, or by providing type constraints to describe the acceptable types of undeclared ports.

As mentioned earlier, the static type checker flags a type conflict error if the type compatibility rule is violated on a certain connection. There are other kind of type conflicts indicated by one of the following:

- The set of type constraints are not satisfiable.
- Some type variables are resolved to *Unknown*.
- Some type variables are resolved to an abstract type, such as *Numerical* in the type hierarchy.

The first case can happen, for example, if the port of actor A1 in figure 5.3 has declared type *Complex*. The second case can happen if an actor does not specify any type constraints on an undeclared output port. This is due to the nature of the type resolution algorithm where it assigns all the undeclared types to *Unknown* at the beginning. If the type constraints do not restrict a type variable to be greater than *Unknown*, it will stay at *Unknown* after resolution. The third case is considered a conflict since an abstract type does not correspond to an instantiable token class.

5.2.2 Run-time Type Checking and Lossless Type Conversion

The declared type is a contract between an actor and the Ptolemy II system. If an actor declares an output port to have a certain type, it asserts that it will only send out tokens whose types are less than or equal to that type. If an actor declares an input port to have a certain type, it requires the system to only send tokens that are instances of the class of that type to that input port. Run-time type checking enforces this contract, regardless of whether individual actors respect it. When a token is sent out from an output port, the run-time type checker queries its type and compares the type with the declared type of the output port. If the type of the token is not less than or equal to the declared type, a run-time type error will be generated.

As discussed before, type conversion is performed automatically when a token sent to an input port has a type less than the type of the input port. This conversion enables an actor to safely cast a received token to the type of the port. On the other hand, when an actor sends out tokens, the tokens being sent do not have to have the exact declared output port type. Any type that is less than the declared type is acceptable. For example, if an output port has declared type *Double*, the actor can send *IntToken* from that port. As can be seen, the automatic type conversion simplifies the input/output handling of the actors.

Note that even with the convenience provided by the type conversion, actors should still declare the input types to be the most general that they can handle and the output types to be the most specific type that includes all tokens they will send. This maximizes their applications. In the previous example, if the actor only sends out *IntToken*, it should declare the output type to be *Int* to allow the port to be connected with an input with type *Int*.

If an actor has ports with undeclared types, its type constraints can be viewed as both a requirement and an assertion from the actor. The actor requires the resolved types to satisfy the constraints. Once the resolved types are found, they serve exactly the same role as declared types at run time. The the type checking and type conversion system guarantees the type of tokens received by an actor, and the actor guarantees the types of tokens sent by the actor. These assumptions and guarantees are summarized for all possible types by the type constraints of the actor.

5.3 Structured Types

Structured types include those tokens which aggregate other tokens of arbitrary type, such as array and record types. As described in the Data Package chapter, an ArrayToken contains an array of tokens, and the element tokens can have arbitrary type. For example, an ArrayToken can contain an array of StringTokens, or an array of ArrayTokens. In the latter case, the ArrayToken can be regarded as a two dimensional array. RecordToken contains a set of labeled tokens, like the structure in the C language. It is useful for grouping multiple pieces of related information together. In the type lattice in figure 5.2, array and record types are incomparable with all the base types, except the top and the bottom elements of the lattice. Note that the lattice nodes Array and Record actually represent an infinite number of types, so the type lattice becomes infinite.

The order relation between two array types is that type *{B}* (the type of arrays containing elements of type *B*) is less than type *{A}* if *B* is less than *A*. This is a recursive definition if the element types *A* and *B* are themselves structured types. For example, *{Int}* ≤ *{Double}*, *{{Int}}* ≤ *{{Double}}*, where *{{Int}}* is an array of array. Note that *{Int}* and *{{Double}}* are incomparable.

The order relation between two record types follows the standard depth subtyping and width subtyping relations.[26] In depth subtyping, a record type *C* is a subtype of a record type *D* if the type of some fields of *C* is a subtype of the corresponding fields in *D*. In width subtyping, a record with more fields is a subtype of a record with less fields. For example, we have:

{x = *String*, y = *Int*} ≤ {x = *String*, y = *Double*}

{x = *String*, y = *Double*, z = *Int*} ≤ {x = *String*, y = *Double*}

Here, we use the {label = type, label = type, ...} syntax to denote record types. Notice that the width subtyping rule implies a type conversion which loses information, discarding the extra fields of a record.

One final structured type is the type of function closures. Each function closure is represented by an instance of the FunctionToken class. Function closures 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 that of 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. Note particularly that 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 are considered incomparable. Eventually, we intend to provide an actor token as well, which would have both contravariance of the types of input and output ports as well as allowing width subtyping, similarly to records. The presence of function types that can be used as any other token results in what is commonly termed a *higher-order* type system.

Type constraints can be specified between the element type of a structured type and the type of a Ptolemy object. For example, a type constraint can specify that the type of a port is no less than the type of the elements of an ArrayToken.

5.4 Implementation

5.4.1 Implementation Classes

All the classes for representing the types and the type lattice are under the data.type package, as shown in figure 5.4. The Type interface defines the basic operations on a type. BaseType contains a type-safe enumeration of primitive types. For example, *Unknown*, the bottom element of the type lattice which can be resolved to any type is represented by the field BaseType.UNKNOWN. ArrayType and RecordType are derived from an abstract class StructuredType. Each type has a convert() method to convert a token lower in the type lattice to one of its type. For base types, this method just calls the same method in the corresponding tokens. For structured types, the conversion is done within the concrete structured type classes.

The Typeable interface defines a set of methods to set type constraints between typed objects. It is implemented by the Variable class in the data.expr package and the TypedIOPort class in the actor package. The TypeConstant class encapsulates a constant type. It implements the InequalityTerm interface and can be used to set up type constraints between a typed object and a constant type.

In the actor package, the Actor interface, the AtomicActor, CompositeActor, IOPort and IORelation classes are extended with TypedActor, TypedAtomicActor, TypedCompositeActor, TypedIOPort and TypedIORelation, respectively, as shown in figure 5.5. The container for TypedIOPort must be a ComponentEntity implementing the TypedActor interface, namely, TypedAtomicActor or TypedCompositeActor. The TypedIORelation class is only able to connect instances of the TypedIOPort. TypedIOPort has a declared type and a resolved type. Declaring a type of BaseType.UNKNOWN allows the type system to infer the resolved type of a port. If a port has a declared type that is not BaseType.UNKNOWN, the resolved type will be the same as the declared type.

5.4.2 Type Checking and Type Resolution

Static type checking and type resolution are performed by the resolveTypes() method of the TypedCompositeActor class. This method finds all connections within the composite by first finding the output ports on deep contained entities, and then finding input ports deeply connected to those output ports. Transparent ports are ignored for type checking. For each connection, if the types on both ends are declared, static type checking is performed using the type compatibility rule. If the model contains other opaque TypedCompositeActors, this method recursively calls the _checkDeclaredTypes() method of the contained actors to perform type checking on the entire hierarchy. Hence, if resolveTypes() is called with the top level TypedCompositeActor, type checking is performed through out the hierarchy.

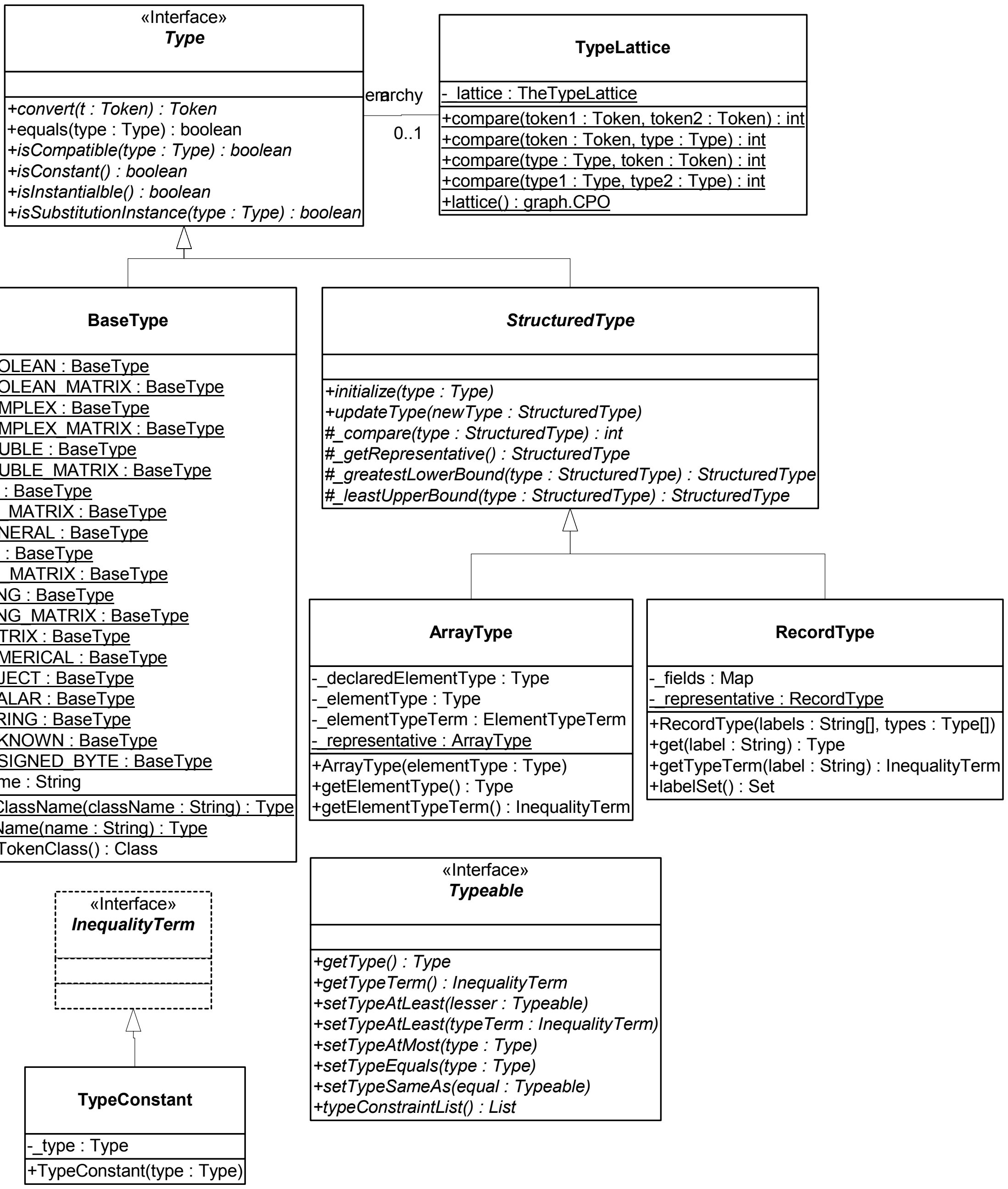
«Interface»
Type
+convert(t : Token) : Token
+equals(type : Type) : boolean
+isCompatible(type : Type) : boolean
+isConstant() : boolean
+isInstantialble() : boolean
+isSubstitutionInstance(type : Type) : boolean
enarchy
0..1
TypeLattice
-_lattice : TheTypeLattice
+compare(token1 : Token, token2 : Token) : int
+compare(token : Token, type : Type) : int
+compare(type : Type, token : Token) : int
+compare(type1 : Type, type2 : Type) : int
+lattice() : graph.CPO
BaseType
+BOOLEAN : BaseType
+BOOLEAN_MATRIX : BaseType
+COMPLEX : BaseType
+COMPLEX_MATRIX : BaseType
+DOUBLE : BaseType
+DOUBLE_MATRIX : BaseType
+FIX : BaseType
+FIX_MATRIX : BaseType
+GENERAL : BaseType
+INT : BaseType
+INT_MATRIX : BaseType
+LONG : BaseType
+LONG_MATRIX : BaseType
+MATRIX : BaseType
+NUMERICAL : BaseType
+OBJECT : BaseType
+SCALAR : BaseType
+STRING : BaseType
+UNKNOWN : BaseType
+UNSIGNED_BYTE : BaseType
-_name : String
+forClassName(className : String) : Type
+forName(name : String) : Type
+getTokenClass() : Class
StructuredType
+initialize(type : Type)
+updateType(newType : StructuredType)
#_compare(type : StructuredType) : int
#_getRepresentative() : StructuredType
#_greatestLowerBound(type : StructuredType) : StructuredType
#_leastUpperBound(type : StructuredType) : StructuredType
ArrayType
-_declaredElementType : Type
-_elementType : Type
-_elementTypeTerm : ElementTypeTerm
-_representative : ArrayType
+ArrayType(elementType : Type)
+getElementType() : Type
+getElementTypeTerm() : InequalityTerm
RecordType
-_fields : Map
-_representative : RecordType
+RecordType(labels : String[], types : Type[])
+get(label : String) : Type
+getTypeTerm(label : String) : InequalityTerm
+labelSet() : Set
«Interface»
InequalityTerm
TypeConstant
-_type : Type
+TypeConstant(type : Type)
«Interface»
Typeable
+getType() : Type
+getTypeTerm() : InequalityTerm
+setTypeAtLeast(lesser : Typeable)
+setTypeAtLeast(typeTerm : InequalityTerm)
+setTypeAtMost(type : Type)
+setTypeEquals(type : Type)
+setTypeSameAs(equal : Typeable)
+typeConstraintList() : List

FIGURE 5.4. Classes in the data.type package.

If a type conflict is detected, i.e., if the declared type at the source end of a connection is greater than or incomparable with the type at the destination end of the connection, then the ports at both ends of the connection are recorded and will be returned in a List at the end of type checking. Note that type checking does not stop after detecting the first type conflict, so the returned List contains all the ports that have type conflicts. This behavior is similar to a regular compiler, where compilation will generally continue after detecting errors in the source code.

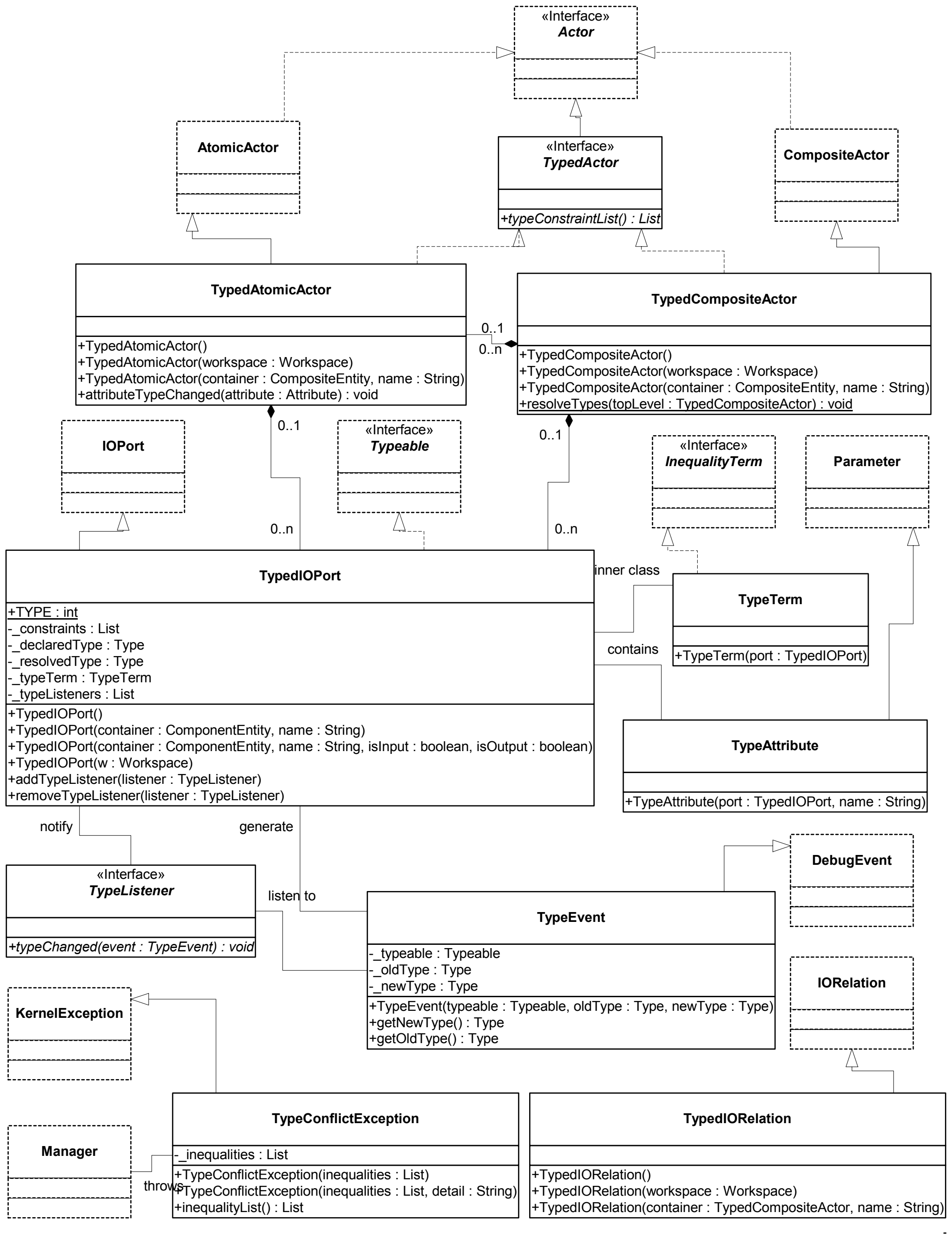
«Interface» Actor
AtomicActor
«Interface» TypedActor
+typeConstraintList() : List
CompositeActor
TypedAtomicActor
+TypedAtomicActor()
+TypedAtomicActor(workspace : Workspace)
+TypedAtomicActor(container : CompositeEntity, name : String)
+attributeTypeChanged(attribute : Attribute) : void
TypedCompositeActor
+TypedCompositeActor()
+TypedCompositeActor(workspace : Workspace)
+TypedCompositeActor(container : CompositeEntity, name : String)
+resolveTypes(topLevel : TypedCompositeActor) : void
0..1
0..n
IOPort
«Interface» Typeable
«Interface» InequalityTerm
Parameter
TypedIOPort
+TYPE : int
-_constraints : List
-_declaredType : Type
-_resolvedType : Type
-_typeTerm : TypeTerm
-_typeListeners : List
+TypedIOPort()
+TypedIOPort(container : ComponentEntity, name : String)
+TypedIOPort(container : ComponentEntity, name : String, isInput : boolean, isOutput : boolean)
+TypedIOPort(w : Workspace)
+addTypeListener(listener : TypeListener)
+removeTypeListener(listener : TypeListener)
inner class
contains
TypeTerm
+TypeTerm(port : TypedIOPort)
TypeAttribute
+TypeAttribute(port : TypedIOPort, name : String)
notify
generate
listen to
«Interface» TypeListener
+typeChanged(event : TypeEvent) : void
DebugEvent
TypeEvent
-_typeable : Typeable
-_oldType : Type
-_newType : Type
+TypeEvent(typeable : Typeable, oldType : Type, newType : Type)
+getNewType() : Type
+getOldType() : Type
IORelation
KernelException
Manager
throws
TypeConflictException
-_inequalities : List
+TypeConflictException(inequalities : List)
+TypeConflictException(inequalities : List, detail : String)
+inequalityList() : List
TypedIORelation
+TypedIORelation()
+TypedIORelation(workspace : Workspace)
+TypedIORelation(container : TypedCompositeActor, name : String)

FIGURE 5.5. Classes in the actor package that support type checking.

The TypedActor interface declares a typeConstraintList() method, which returns the type constraints of this actor. The TypedAtomicActor base class provides a default implementation of this method, which requires that the type of any input port with undeclared type must be less than or equal to the type of any undeclared output port. Ports with declared types are not included in the default constraints. If all of an actor's ports have declared types, no constraints are generated. This default is appropriate for many type-polymorphic actors such as the Commutator actor, the Multiplexer actor, and the DownSample actor in figure 5.1. In addition, the typeConstraintList() method also collects all the constraints from the contained Typeable objects, which are TypedIOPorts and Variables.

The typeConstraintList() method in TypedCompositeActor collects all the constraints for a model, including the constraints for actors and the constraints for connections between actors. It works in a similar fashion as the _checkDeclaredTypes() method, by recursively traversing the containment hierarchy. It also scans all the connections and forms additional type constraints on connections involving undeclared types. As with _checkDeclaredTypes(), if this method is called on the top level container, all the type constraints within the entire model are returned.

The Manager class has a resolveTypes() method that performs both type checking and resolution. It uses the InequalitySolver class in the graph package to solve the constraints. If type conflicts are detected during type checking or after type resolution, this method throws a TypeConflictException. This exception contains a list of inequalities where type conflicts occurred. The resolveTypes() method is invoked by the Manager of a model between the preinitialize() and initialize() phases, and after any mutations are processed.

Run-time type checking is performed in the send() method of TypedIOPort. The checking is simply a comparison of the type of the token being sent with the resolved type of the port. If the type of the token is less than or equal to the resolved type, type checking is passed, otherwise, an IllegalActionException is thrown.

Type conversion, if needed, is also done in the send() method. The type of the destination port is the resolved type of the port containing the receivers that the token is sent to. If the token does not have that type, the convert() method on that type is called to perform the conversion.

5.4.3 Setting Up Type Constraints

The class Inequality in the graph package is used to represent type constraints. This class references two objects implementing the InequalityTerm interface, one for each side of the inequality. The InequalityTerm interface is implemented by inner classes of TypedIOPort, Variable, ArrayType, and RecordType, to encapsulate the type of the port, the variable, and the element type of structured types. In most cases, type constraints can be set up easily through the methods in the Typeable interface. 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 setTypeEquals() method can also be used to force the type of typeable objects to a particular data type. As an example, the constraint that the type of an input port can be no greater than *Double* might be declared as:

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

and a constraint that the type of an output port can be no less than the type of a parameter:

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

The second type constraint is commonly seen when parameter values are used to compute values produced from an output port. Note that the methods above can take either a Type or a Typeable object.

More complex type constraints arise from structured types, such as arrays and records. These types contain internal type variables that determine the type of data contained in the array or record. As an example, the following code can be used to specify that a parameter can only contain an ArrayToken, and to constrain the type of a port to be no less than the element type of that array:

```
parameter.setTypeEquals(new ArrayType(BaseType.UNKNOWN));
ArrayType arrayType = (ArrayType)parameter.getType();
InequalityTerm elementTerm = arrayType.getElementTypeTerm();
port.setTypeAtLeast(elementTerm);
```

These kinds of constraints appear in source actors such as Clock and Pulse, where the actor outputs a sequence of values specified by an ArrayToken. Another common constraint is that an input port of an actor receives a RecordToken with unconstrained fields. This constraint can be declared using the following code:

```
String[] labels = new String[0];
Type[] types = new Type[0];
RecordType declaredType = new RecordType(labels, types);
inputPort.setTypeAtMost(declaredType);
```

In some actors, simple constraints between variables are not capable of representing the type constraints between ports and parameters. In such cases, monotonic functions can be used to specify more complex type constraints. That is, constraints in the form f(α) ≤ β are admitted, where f(α) is a monotonic function of α, and β can be a constant or a variable. An example of this appears in the AbsoluteValue actor in the actor library. Here, one of the type constraints is: If the input type is not *Complex*, the output type is the same as the input type, otherwise, the output type is *Double*. This constraint can be expressed as f(inputType) ≤ outputType, where

```
f(inputType)  = inputType,  if inputType ≠ Complex
f(inputType)  = Double,     if inputType = Complex.
```

This function is implemented by an inner class of AbsoluteValue that implements InequalityTerm. The evaluation is done in the getValue() method of InequalityTerm as:

```
public Object getValue() {
  // _port is the input port
  Type inputType = _port.getType();
  return inputType == BaseType.COMPLEX ? BaseType.DOUBLE : inputType;
}
```

Directly implementing the InequalityTerm interface is actually rather complex, and is implemented in the same pattern for all monotonic function constraints. The MonotonicFunction base class, which implements the uninteresting parts of the InequalityTerm interface, allows actors to easily implement new monotonic function constraints. Lastly, if the methods in Typeable are not sufficient for specifying complicated constraints, or the default implementation of the typeConstraints() method in the TypedAtomicActor is not appropriate, this method can be overridden, but this is rarely needed.

5.4.4 Some Implementation Details

The implementation of the structured types is more involved than the base types. This is because the base types are atomic, but structured types that contain type variables are mutable entities. For example, the declared type of a port can be *{Unknown}*, meaning that it is an array of undefined element type. After type resolution, that type may be updated to *{Double}*. Types that are mutable are variable types. The isConstant() method in Type determines if a type contains a type variable. Type variables are represented by a type initialized to BaseType.UNKNOWN.

When a typed object is cloned, if its type is a variable structured type, that type must be cloned because the original and the cloned Typeable objects may have different types in the future. Similarly, when constructing structured types with variable structured types as element types, the element types must be cloned. However, constant structured types do not need to be cloned. This means that an instance of a constant StructuredType can be shared by many objects, but an instance of a variable StructuredType can only have one user. To ensure this, structured types are always cloned when ports and parameters that contain them are cloned. This incurs some redundant cloning, but the overhead is small.

A variable type can be updated to another type, provided that the new type is compatible with the variable type. For example, a type variable α can be updated to any type, $\{\alpha\}$ can be updated to *{Int}*. However, $\{\alpha\}$ cannot be updated to *Int*. If a variable type can be updated to a new type, the new type is called a substitution instance of the variable type. This term is borrowed from type literature. Formally, a type is a substitution instance of a variable type if the former can be obtained by substituting the type variables of the latter to another type. The method isSubstitutionInstance() in the Type base class performs this check.

The updateType() method in StructuredType is used to change the variable element type of a structured type. For example, if the types of two ports are *{Int}* and $\{\alpha\}$ respectively, and a type constraint is that the second port is no less than the type of the first, that is, $\{Int\} \leq \{\alpha\}$, the type resolution algorithm will change the resolved type of the second port to *{Int}*. This step cannot be done by simply changing the type reference in the second port to an instance of *{Int}*, since type constraints may be set up between α and another typed objects. Instead, updateType() only changes the type reference for α to *Int*.

5.5 Examples

5.5.1 Polymorphic DownSample

In figure 5.1, if the DownSample is designed to do downsampling for any kind of token, its type constraint is just *samplerIn* ≤ *samplerOut*, where *samplerIn* and *samplerOut* are the types of the input and output ports, respectively. The default type constraints works in this case. Assuming the Display actor just calls the *toString()* method of the received tokens and displays the string value in a certain window, the declared type of its port would be *General*. Let the declared types on the ports of FFT be *Complex*, the The type constraints of this simple application are:

sourceOut ≤ *samplerIn*

samplerIn ≤ *samplerOut*

samplerOut ≤ *Complex*

Complex ≤ *General*

Where *sourceOut* represents the declared type of the Source output. The last constraint does not involve a type variable, so it is just checked by the static type checker and not included in type resolution. Depending on the value of *sourceOut*, the ports on the DownSample actor would be resolved to different types. Some possibilities are:

- If *sourceOut* = *Complex*, the resolved types would be *samplerIn* = *samplerOut* = *Complex*.
- If *sourceOut* = *Double*, the resolved types would be *samplerIn* = *samplerOut* = *Double*. At run-time, DoubleTokens sent out from the Source will be passed to the DownSample actor unchanged. Before they leave the Downsample actor and are sent to the FFT actor, they are converted to ComplexTokens by the system. The ComplexToken output from the FFT actor are instances of Token, which corresponds to the *General* type, so they are transferred to the input of the Display without change.
- If *sourceOut* = *String*, the set of type constraints do not have a solution, a typeConflictException will be thrown by the static type checker.

5.5.2 Fork Connection

Consider two simple topologies in figure 5.6. where a single output is connected to two inputs in 5.6(a) and two outputs are connected to a single input in 5.6(b). Denote the types of the ports by *a1, a2, a3, b1, b2, b3,* as indicated in the figure. Some possibilities of legal and illegal type assignments are:

- In 5.6(a), if *a1* = *Int*, *a2* = *Double*, *a3* = *Complex*. The topology is well typed. At run-time, the IntToken sent out from actor A1 will be converted to DoubleToken before transferred to A2, and converted to ComplexToken before transferred to A3. This shows that multiple ports with different types can be interconnected as long as the type compatibility rule is obeyed.
- In 5.6(b), if *b1* = *Int*, b2 = *Double*, and *b3* is undeclared. The the resolved type for *b3* will be *Double*. If *b1* = *Int* and *b2* = *Boolean*, the resolved type for *b3* will be *String* since it is the lowest element in the type hierarchy that is higher than both *Int* and *Boolean*. In this case, if the actor B3 has some type constraints that require *b3* to be less than *String*, then type resolution is not possible, a type conflict will be signaled.

5.6 Actors Constructing Tokens with Structured Types

The SDF domain contains two actors that perform conversion between a sequence of tokens and an ArrayToken. Type constraints in these actors ensure that the type of the array element is the same as

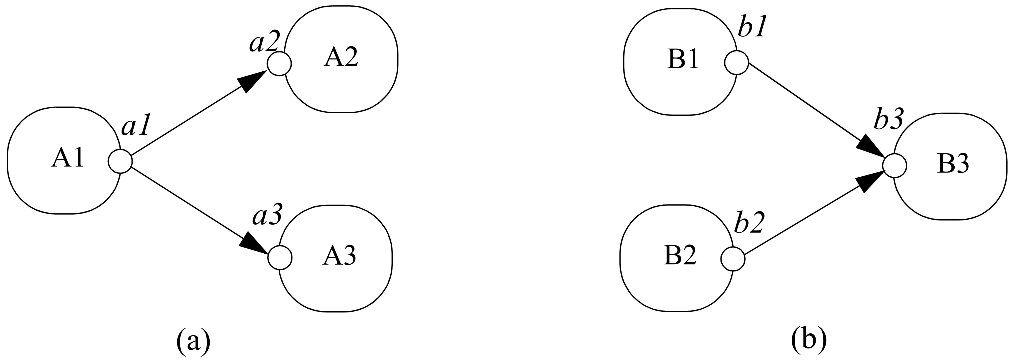

FIGURE 5.6. Two simple topologies with types.

the type of the sequence tokens. When two SequenceToArray actors are cascaded, the output of the second actor will be an array of array. Cascading ArrayToSequence with SequenceToArray restores the sequence. In these actors, the *arrayLength* parameter determines the size of the produced or consumed array, and also determines the number of tokens produced or consumed in each firing. If the ArrayToken received by ArrayToSequence does not have specified length and the *enforceArrayLength* parameter is true, an exception will be thrown.

The actor.lib package contains two actors that assemble and disassemble RecordTokens: RecordAssembler and RecordDisassembler. The former assembles tokens from multiple input ports into a RecordToken and sends it to the output port, the latter does the reverse. The labels in the RecordToken are the names of the input ports. Type constraints ensure that the type of the record fields is the same as the type of the corresponding ports.

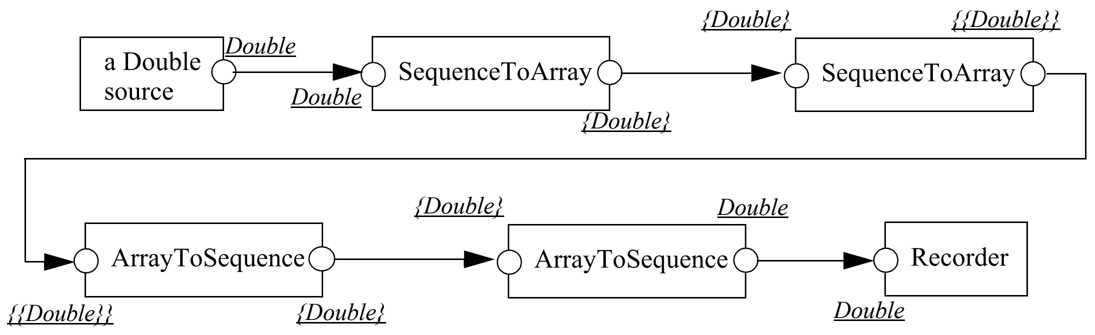

FIGURE 5.7. Conversion between sequence and array.

Appendix B: The Type Resolution Algorithm

The type resolution algorithm starts by assigning all the type variables the bottom element of the type hierarchy, *Unknown*, then repeatedly updating the variables to a greater element until all the constraints are satisfied, or when the algorithm finds that the set of constraints are not satisfiable. The kind of inequality constraints the algorithm can determine satisfiability are the ones with the greater term (the right side of the inequality) being a variable, or a constant. The algorithm allows the left side of the inequality to contain monotonic functions of the type variables, but not the right side. The first step of the algorithm is to divide the inequalities into two categories, *Cvar* and *Ccnst*. The inequalities in *Cvar* have a variable on the right side, and the inequalities in *Ccnst* have a constant on the right side. In the example of figure 5.3, *Cvar* consists of:

$Int \le \alpha$
$Double \le \beta$
$\alpha \le \gamma$
$\beta \le \gamma$

And *Ccnst* consists of:

$\gamma \le Double$
$\gamma \le Complex$

The repeated evaluations are only done on *Cvar*, *Ccnst* are used as checks after the iteration is finished, as we will see later. Before the iteration, all the variables are assigned the value *Unknown*, and *Cvar* looks like:

$Int \le \alpha(Unknown)$
$Double \le \beta(Unknown)$
$\alpha(Unknown) \le \gamma(Unknown)$
$\beta(Unknown) \le \gamma(Unknown)$

Where the current value of the variables are inside the parenthesis next to the variable.

At this point, *Cvar* is further divided into two sets: those inequalities that are not currently satisfied, and those that are satisfied:

Not-satisfied	Satisfied
$Int \le \alpha(Unknown)$	$\alpha(Unknown) \le \gamma(Unknown)$
$Double \le \beta(Unknown)$	$\beta(Unknown) \le \gamma(Unknown)$

Now comes the update step. The algorithm takes out an arbitrary inequality from the Not-satisfied set, and forces it to be satisfied by assigning the variable on the right side the least upper bound of the values of both sides of the inequality. Assuming the algorithm takes out $Int \le \alpha(Unknown)$, then

$$\alpha = Int \vee Unknown = Int \qquad (6)$$

After α is updated, all the inequalities in *Cvar* containing it are inspected and are switched to either the Satisfied or Not-satisfied set, if they are not already in the appropriate set. In this example, after this step, *Cvar* is:

Not-satisfied	Satisfied
$Double \le \beta(Unknown)$	$Int \le \alpha(Int)$
$\alpha(Int) \le \gamma(Unknown)$	$\beta(Unknown) \le \gamma(Unknown)$

The update step is repeated until all the inequalities in *Cvar* are satisfied. In this example, β and γ

will be updated and the solution is:

$$\alpha = Int,\ \ \beta = \gamma = Double$$

Note that there always exists a solution for *Cvar*. An obvious one is to assign all the variables to the top element, *General*, although this solution may not satisfy the constraints in *Ccnst*. The above iteration will find the least solution, or the set of most specific types.

After the iteration, the inequalities in *Ccnst* are checked based on the current value of the variables. If all of them are satisfied, a solution to the set of constraints is found.

This algorithm can be viewed as repeated evaluation of a monotonic function, and the solution is the fixed point of the function. Equation (6) can be viewed as a monotonic function applied to a type variable. The repeated update of all the type variables can be viewed as the evaluation of a monotonic function that is the composition of individual functions like (6). The evaluation reaches a fixed point when a set of type variable assignments satisfying the constraints in *Cvar* is found.

Rehof and Mogensen [121] proved that the above algorithm is linear time in the number of occurrences of symbols in the constraints, and gave an upper bound on the number of basic computations. In our formulation, the symbols are type constants and type variables, and each constraint contains two symbols. So the type resolution algorithm is linear in the number of constraints.