Maximally Permissive Composition of Actors in Ptolemy II

Marten Lohstroh

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-19
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-19.html

March 20, 2013

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #0931843 (CPS: Large: ActionWebs), and #1035672 (CPS: Medium: Ptides)), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota.

Maximally Permissive Composition of Actors in Ptolemy II

by
Marten Lohstroh

A thesis submitted in partial fulfilment
of the requirements for the degree of

Master of Science
in
Computer Science

at the
University of Amsterdam
Informatics Institute

in collaboration with
Department of
Electrical Engineering & Computer Science
University of California, Berkeley

Committee in charge:

Professor Edward A. Lee (University of California, Berkeley)
Dr. Andy D. Pimentel (University of Amsterdam)
Dr. Clemens Grelck (University of Amsterdam)
Dr. Inge Bethke (University of Amsterdam)

February 2013

Maximally Permissive Composition of Actors in Ptolemy II

by

Marten Lohstroh

Abstract

Maximally Permissive Composition of Actors in Ptolemy II

by

Marten Lohstroh

Master of Science in Computer Science

University of Amsterdam

The "Cyber" and "Physical" worlds are merging. *Cyber-Physical Systems* (CPS) are no longer isolated, but start to reach into the *Cloud*, thereby composing a network which realizes the concept that became known as *The Internet of Things*. The dynamic nature of the applications in this domain poses significant technical challenges concerning the assurance of important system properties like reliability, robustness, adaptability, and security. Modeling has proven itself to be a valuable tool in gaining better understanding of complex systems, but existing modeling platforms may lack the expressivity to model these new, much more dynamic, and opportunistically composed systems in which the data they handle typically does not conform to a rigid structure.

This thesis addresses the problem of handling *dynamic data*, in the *statically typed*, actor-oriented modeling environment called Ptolemy II. It explores the possibilities of using type inference to statically type dynamic data and leverage dynamic type checking to invoke error handling strategies that enhance robustness. The goal is to achieve *maximally permissive composition*, and the presented solution comes in the form of *backward type inference*. Backward inferred types are specific enough not to limit composability and general enough not to impose unnecessary constraints on the data. The type constraints imposed by downstream actors determine the type of the otherwise underdetermined output ports of actors that mediate access to untyped resources. This is achieved using additional type constraints, without changing Ptolemy II's original type resolution algorithm, and with no significant impact on the run-time of type resolution. The proposed solution was implemented successfully and has been adopted as an extension of the Ptolemy II type system.

As a byproduct, this thesis gives a thorough case study of the Ptolemy II type system. It uncovers two (independent) obscurities: interference between automatic type conversion and dynamic dispatch; and unsafe access to record elements. Both issues are discussed extensively and possible improvements are suggested.

Acknowledgements

I would like to express my gratitude to everyone who has contributed to this work, in the literal sense, by means of support, or as a source of inspiration. My adviser, Edward Lee, contributed immensely in every respect. I could not have wished for a more dedicated, perspicacious, and inspiring mentor. Edward, you have been a lot more than the person whom without this thesis would never have been written. It is a pleasure working with you and I am deeply grateful for your support.

Also, I would also like to thank the other members of the thesis committee: Andy Pimentel, Inge Bethke, and Clemens Grelck. I have known Inge and Andy since my first year in college, and have been both their student and their teaching assistant. Andy, if it were not for your recommendation, it would have been unlikely for me to end up in Berkeley. Inge, Clemens and Andy each have contributed valuable parts of my education in Computer Science.

My special thanks go out to Chris Shaver, David Broman, and Vasco Visser. I spent countless hours with Chris, discussing the ideas that have eventually materialized in this thesis. He helped me structure my thoughts, improve my writing, and sparked new ideas. Chris, you are a remarkable person and a great friend. David, thanks for the brief introduction to type systems that you were happy to give me over lunch. You gave me valuable pointers and helpful reviews. Vasco, your reviews were helpful as always, and I greatly appreciate our friendship.

Other members, visitors, or collaborators of the Ptolemy group that have in greater or lesser extend contributed to my research are: Christopher Brooks, Stavros Tripakis, Patricia Derler, Ben Lickly, Hallvard Traetteberg, Beth Latronico, Christos Stergiou, Alain Girault, and Eleftherios Matsikoudis.

Mira, even though we had to spent so much time apart, I have always kept you closest to my heart. I am thankful to my parents Jan & Yolanda, who have supported me throughout my studies and enabled me to pursue what turned out to be a life changing experience abroad. Thank you Tjitske, for being my awesome sister. Thank you Frank, for giving me a head start in CS by teaching me the basics of computer architecture at the age of 10. Finally, I would like to thank my friends, close by or far away, who have been there for me when I needed them the most, especially: Oscar de Boer, Kristina Kangas, Gideon Koekoek, and Sebastian Conrady.

Dedicated to Mira.

Contents

Index **61**

List of Figures

Chapter 1

Introduction

The frontiers of semiconductor and information technology are closing in. Whereas semiconductor components have kept becoming smaller, faster, and more power-efficient, the Internet has evolved into an infrastructure that weaves together the bits and pieces of our digital lives. The tentacles of the Internet reach out into virtually every corner of the world, connecting an ever increasing number of devices that serve as end-points of our physical reality. What results is a global nervous system that taps into our physical world by means of sensors and actuators. This global network of Cyber-Physical Systems (i.e., integrations of computation with physical processes [Lee, 2008]), is often referred to as the "Internet of Things" (IoT).

This term was coined by Kevin Ashton [Ashton, 2009] in 1999 to describe a system that captures data from the physical world through networked sensors, instead of relying on human beings to provide input. His observation was that people have limited time, attention and accuracy, hence they are not very good at structurally capturing information about physical entities and processes. He envisioned a network of things that autonomously would capture, generate, and process information without any immediate human interaction.

Another influential idea that finds its origin in the 1990s, is the concept of "Grid Computing", which encompasses a computing paradigm that allows consumers to obtain computing power on demand [Foster *et al.*, 2008]. Grid computing is a form of distributed computing that allows organizations to share and acquire resources for the execution of computational tasks. Whereas the Grid has been mostly geared towards applications in high-performance computing, the same ideas in slightly different shapes became mainstream under the title of "Cloud Computing". The Cloud is a service-oriented platform that is driven by economies of scale, centered around abstraction, virtualization, and scalability of systems that deliver storage and computing power.

Tremendous progress has been made in the development of Internet infrastructure [Dutta and Bilbao-Osorio, 2012] and industry predicts Internet-connected mobile devices such as smart phones, laptops, and tablets, will outnumber the people in the world before the end of 2013 [Cisco, 2013]. These mobile devices have become the primary peripherals of the Cloud, capable of accessing shear amounts of information and extremely powerful processing capabilities. A newly emerging outermost peripheral layer of the Cloud that is key to the full realization of the IoT, is identified as "The Swarm" [Rabaey, 2011] — a collection of networked sensors and actuators that interact with the physical world around us.

All of the former sketches a contour of what the IoT might eventually look like, but it remains an open question as to how all of these networked "things" can interoperate universally in a meaningful way. This is also the main question that is addressed by the recently launched TerraSwarm Research Center that is lead by the University of California at Berkeley [Lee *et al.*, 2012]. The term "TerraSwarm" sprung from the industry prediction that ten years from now there will be thousands of smart sensing devices per person on the planet, yielding a ubiquitous swarm with trillions of nodes.

One of the main themes identified by TerraSwarm Research Center is the ability to model system components and their interactions in a setting where components and subsystems can be dynamically recombined. Because components in Swarm- (or, IoT-) applications will be dynamically composed and recomposed, the distinction between "design-time" and "run-time" becomes blurred. This means that design-time testing and verification are no longer adequate and run-time validation strategies will need to be developed to assure key properties like reliability, robustness, adaptability, and security. In this thesis it is investigated how lightweight formal methods like *type systems* can play a role in achieving this goal.

1.1 Motivation

In the Ptolemy project [Lee, 1999], which is aimed at modeling Cyber-Physical Systems, it was recognized very early on that modern computing systems tend to be heterogeneous in the sense of being composed of subsystems with very different characteristics [Eker *et al.*, 2003]. The Ptolemy approach to tame heterogeneity is to combine abstract actor-oriented semantics with hierarchical component design. This approach extends very well to the IoT domain which essentially involves the networking of Cyber-Physical Systems.

Ptolemy II is an open-source software framework for experimentation with actor-oriented design that has been developed during the past decade in the Ptolemy project. In order to employ Ptolemy II for modeling IoT applications, a natural first step would be to develop components that mediate access to information that is dynamically accessible through the Cloud. The first hurdle to clear in order for this to succeed, is to bridge the gap between on the one hand, online data that is often unreliable, inconsistent, and subject to change, and on the other hand, much more static and precisely defined Ptolemy II models. We desire models to be resilient enough to cope with components of which their execution relies on interactions with external resources that are deployed in a "live" environment. What makes this particularly challenging, is that Ptolemy II is *statically* typed, whereas resources in the IoT domain are generally *untyped*. This yields an immediate obstacle; Ptolemy II models require all ports and variables to be assigned a type, prior to execution. This is a difficult task if the type of a resource is unknown until the moment it is accessed.

Clearly, in order to assign a type to dynamic data, an assumption needs to be made about that data. That assumption should be reasonable. If at runtime, the data conforms to the type that it was assigned statically, then safe execution of the model must be guaranteed. The type should not be too general in the sense that it limits composability because it imposes unnecessary constraints on the rest of the model. Neither should the type be too specific in the sense that it imposes unnecessary constraints on the data. Hence, the goal is *maximally permissive composition*.

The Ptolemy II type system makes use of *type inference*, which relieves the user from having to manually annotate every port or variable in a model with explicit types. If there is a structural way to deduce types for dynamic data (i.e., data whose type is not known statically), then presumably the existing type reconstruction mechanisms can be used to infer those types. This idea lead to the following hypothesis:

> *We can use type inference methods to statically type dynamic data and leverage dynamic type checking to invoke error handling strategies that enhance robustness.*

If the inferred types are indeed maximally permissive, then they provide the exact conditions where if those are not met, the component that mediates access to an external data source should consider alternate execution paths in order to keep the entire model from failing. This allows a component to define error handling strategies without the need to specify the exact error conditions, other than "there is a run-time type checking error".

1.2 Contributions

- A comprehensive formalization of Ptolemy II's type inference mechanism is given;
- An obscure and undesirable interference between Ptolemy II's automatic type conversion and Java's dynamic dispatch is identified, analyzed, and possible improvements are suggested;
- A recommendation is formulated to improve the type safety of records;
- The Ptolemy II type system is augmented with *backward type inference*, which allows actors to infer maximally permissive output types, regardless of whether it can be decided prior to execution what type of output they will actually produce.

1.3 Organization

The remainder of this thesis is structured as follows:

Chapter 2 provides a brief introduction to Cyber-Physical Systems, actor-oriented modeling, and the syntax and semantics of Ptolemy II. Secondly, it glances over the immensely rich field of research involved with type systems. This chapter intends to provide the necessary context for the matter that is discussed in subsequent chapters.

Chapter 3 presents a thorough case study of the Ptolemy II type system and in particular, its type inference mechanism. It formalizes the constraint solving problem that underlies the type inference mechanism; it delivers a comprehensive interpretation of type constraints in terms of the dependencies they establish between type variables; and it describes the problem of underdetermined type variables.

Chapter 4 explains how by adding additional type constraints, the original constraint solving algorithm can be used to achieve backward type inference, resulting in a maximally permissive type assignment outputs that would otherwise remain underdetermined.

Chapter 5 discusses the solutions proposed in the preceding chapter, sums up conclusions, and mentions future work.

Chapter 2

Background

This chapter introduces the ideas and motivation behind actor-oriented modeling and provides a high level introduction to Ptolemy II. An overview of type systems and their underlying concepts are given to provide the necessary background for the work that was done on the Ptolemy II type system. The section about type systems is inspired by [Broman *et al.*, 2006], which gives an introduction to type systems in the context of Modelica, an equation based language to model complex physical systems.

2.1 Actor-oriented modeling in Ptolemy II

Ptolemy II is an open-source platform that is designed to model Cyber-Physical Systems. The key idea behind it is to use a common abstract syntax that can be assigned a concrete semantics depending on the behavioral requirements imposed on the system that is to be modeled. Ptolemy II models can be composed hierarchically using an abstract semantics.

2.1.1 Cyber-Physical Systems

Cyber-Physical Systems (CPS) are *complex systems* that integrate computational and physical processes. These processes are coupled, usually by feedback loops, such that they influence one another. Therefore, in order to understand a CPS it does not suffice to consider its computational and physical components separately.

The abstractions present in conventional programming models are very suitable for algorithmic computation, but do not fit the characteristics of physical processes very well. Especially proper abstractions of the concepts of *concurrency* and *time*, which are indispensable in the physical world, are lacking in existing frameworks. In Computer Science, concurrency is often understood as interleavings of sequences of computational steps, but physical processes are continuous and influence each other without interruption. The notion of time is simply absent in most programming models. These discrepancies pose considerable challenges to the development of CPS, the relevance and potential of which is underlined by the recent TerraSwarm effort [Lee *et al.*, 2012]. Ptolemy II is a valuable tool in the search for better abstractions.

2.1.2 Models

A model can be a great help in gaining a deeper understanding of a complex system, because unlike a real system, a model can also be taken apart to analyze how it does what it does (or fails to do what is expected). Likewise, a model also allows experimentation by replacing components or adding new ones, which eventually may lead to new designs, or give rise to improved design principles. Needless to say, in order for conclusions that are drawn from a model to be meaningful, the model has to be a proper abstraction of the actual system i.e., it does not omit essential details.

An important aspect of complex systems is that they are characterized by *heterogeneity* as they are build up out of many different components that may operate in different domains, each having their own dynamics. This aspect must be reflected by the expressivity of the modeling environment, allowing the components of a system to be modeled differently. Ptolemy II meets this demand by taking a modeling approach based on concurrent communicating components called *actors*, where a diversity of orchestration strategies govern the execution and interaction of components [Lee, 2011].

Actors

Actors in Ptolemy II are related to the concept of *actors* that was introduced by Hewitt [Hewitt, 1977] in the 1970s which described a semantics of message passing between autonomously reasoning agents. This idea was later formalized by Agha [Agha and Mason, 1997] and became known as the "Actor model," or "Actor model of computation". The Actor model is *inherently concurrent* as the communication between actors is not specified by some interleaving of steps. Moreover, messages can be *timestamped*, which allows timing to be part of the interaction semantics.

However, there are several differences between Ptolemy II actors and the actors in Agha's formalization. Agha's actors have an independent thread of control and communicate via asynchronous message passing, where recipients of messages are identified by address. Ptolemy II actors operate concurrently but do not require an independent thread of control. Moreover, *tokens* (messages) are sent through *ports* that are connected using *relations*, and communication need not be asynchronous.

2.1.3 Syntax

The syntax of Ptolemy II models comprise the actors and the graphs that connect them. For simplicity, we focus on the *concrete syntax* offered by Ptolemy II, which is a graphical syntax where *actors* are depicted as boxes, with small triangles that represent their *ports*. Ports can be connected to one another by means of a *relation*. Each port has a type assigned to it, either manually or by means of inference. Actors can only send tokens through ports of which the type of is compatible with the type of the port. Other entities that can be part of a model are *attributes*. The most important attribute is the *director*, which governs the interactions between actors. Actors can either be *atomic* or *composite*. A composite actor can contain other entities whereas an atomic actor cannot. A composite actor that contains a director is called *opaque* and one that does not is said to be *transparent*. From the outside, an opaque actor looks just like an atomic actor. It is internally governed by its own director, but interacts with its external environment through the same interface as atomic actors do. On the other hand, the execution of a transparent composite is coordinated by nearest director up the hierarchy. The top-level component of a model must always be an opaque composite.

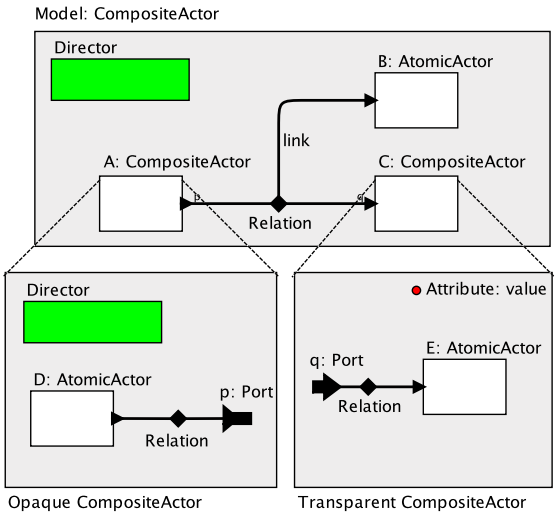


Figure 2.1: An example of a Ptolemy II model.

An example of a Ptolemy II model is depicted in Figure 2.1 (from [Lee, 2011], used with permission from the author). The model consists of a director, three actors (A, B, and C) and one relation (the black diamond, usually omitted for one-to-one relations). Each actor has one port and is connected through the relation. Upon execution actor A sends tokens to actors B and C. Actor A is an opaque composite as it contains a director. Inside A, actor D is connected to the output port of actor A. Actor B is an atomic actor, and actor C is a transparent actor that, along with actor E, carries an attribute that has a value.

2.1.4 Semantics

The *abstract semantics* of Ptolemy II models are defined by the Executable interface implemented by the actors, also referred to as the *actor semantics*. The Executable interface prescribes a number of methods, the most important ones are: preinitialize(), initialize(), fire(), and postfire(). Before execution, preinitialize() performs actions that may influence static analysis, and then initialize() resets local state, initializes parameters and prepares initial outputs, if any. During execution, at each iteration, fire() reads inputs and/or produces new outputs, after which postfire() updates the local state in response to any input.

The *concrete semantics* are captured in a *Model of Computation* (MoC) which governs the interaction of components in a model. MoCs share the same abstract semantics so that an MoC can be implemented by means of a director. The execution of a model is conducted by the director through invocation of the methods defined in the Executable interface. Because models can be constructed hierarchically and each composite can have its own director, actor models offer a disciplined approach to heterogeneity. A great variety of MoCs is implemented in Ptolemy II, among which are: process networks, dataflow, discrete events, finite state machines, continuous time, and rendezvous.

For more information about Ptolemy II, see [Ptolemy.org, 2012].

2.2 Type systems

Formal methods are mathematical tools that can help ensure that a system exhibits some specified behavior. Among these are Hoare logic, algebraic specification languages, modal logic's, and denotational semantics. These methods have great value for their ability to express very general correctness properties, but they are considered a heavy practice that requires levels of sophistication that go far beyond the capacities of most programmers.

Type systems fall in the category of so called *lightweight formal methods*, that are less powerful but at lot easier to handle. Although type systems cannot remove all potential errors in a program, they can eliminate a great fraction of common programming errors by preventing execution errors.

Pierce gives the following definition:

> "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." [Pierce, 2002]

2.2.1 Purpose

The fundamental purpose of a type system is to prevent the occurrence of errors during the execution of a program [Cardelli, 1996]. Two different kinds of execution errors can be identified; *trapped errors* and *untrapped errors*. A trapped error causes a computation to stop immediately whereas an untrapped error goes unnoticed and causes arbitrary behavior later in time. Examples of untrapped errors are jumping to the wrong address, or accessing data past the end of an array.

2.2.2 Type safety & Good behavior

In [Cardelli, 1996], the following definitions with regard to type safety are given. A program fragment is considered *safe* if its execution does not yield untrapped errors, or *unsafe* otherwise. A *safe language* only allows safe program fragments. Usually type systems are also concerned eliminating a great number of trapped errors, which together with the untrapped errors designate the set of *forbidden errors*. A program fragment is said to be well-behaved if it does not allow any forbidden errors to occur. A language where the set of forbidden errors includes all untrapped errors, and where all legal programs are well-behaved, is called *strongly checked* (or, *strongly typed*). Languages that do not include all untrapped errors in their set of forbidden errors are said to be *weakly checked* (or, *weakly typed*). Strong typing is also referred to as *type safety*, *safety*, or *security*. Examples of type safe languages are Java [Syme, 1999] and Standard ML [Milner, 1978], C is statically typed but unsafe; assembler languages untyped and unsafe.

2.2.3 Type inference

Types may be part of the syntax of a language, in which case the language is *explicitly typed*. The programmer must then annotate each variable with a type. Most statically typed languages are explicitly typed. Some languages are *implicitly typed*, meaning that all types are inferred. If some, but not all type annotations are already present, the process of type inference is also referred to as *type reconstruction*.

2.2.4 Formalization

Similarly to how the syntax of a programming language can be captured in a formal grammar, the type rules of a programming language can be given a formal mathematical representation.

> "Once a type system is formalized, we can attempt to prove a type soundness theorem stating that well-typed programs are well behaved. If such a soundness theorem holds, we say that the type system is sound. [Cardelli, 1996]"

A *sound* type system is said to be *safe*. For literature on soundness theorems, see [Milner, 1978] and [Wright and Felleisen, 1994].

Types, judgments & rules

A type is a description that characterizes the expected form of the result of a computation [Macqueen, 2012]. A *typing judgment*: $\Gamma \vdash e : \tau$ binds a type τ to an expression e, asserting that when e is evaluated (and its evaluation terminates), its value adheres to τ. Moreover, a typing judgment requires that the expression is *well-typed* i.e., derivable using a system of *type rules*. Free variables in a typing judgment are considered with respect to some context that assigns those variables a type, a *static type environment* Γ, which one can think of as a finite function that maps variables to types. Consider the following example of a type rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{e_1 e_2 : \tau_2} \quad \text{(T-APP)}$$

This type rule expresses that a unary function can only be applied to arguments that match the domain of the function. The conclusion $e_1 e_2 : \tau_2$, which appears below the horizontal line, concerns a function application where e_1 denotes the function and e_2 denotes the argument. The premises are located above the line. The first premise denotes a function that maps arguments of type τ_1 to values of type τ_2, and the second one denotes a value of type τ_1.

2.2.5 Type checking

Type checking involves finding a type for an expression e provided a context Γ, such that $\Gamma \vdash e : \tau$. If such a type does not exist, a type error is reported. If the type of an expression depends only on the types of its subexpressions i.e., the type rules are *compositional*, type checking is straightforward. The type of e can then be constructed by recursing over its structure, computing the types of its components, and composing the types. When non-compositional rules are involved, type checking gets more complicated.

Type checking can be done statically at compile time, or dynamically at run time. An *untyped* (or, dynamically typed) language can enforce type safety by performing the necessary run-time checks to rule out untrapped errors. Statically typed programming languages may support features like *downcasting* that require run-time checks as well. In that case, static and dynamic type checking can be combined in order to ensure type safety. This can be done using a technique called *soft typing* [Cartwright and Fagan, 1991] , where the static type checker does not reject programs, but instead transforms source programs and judiciously inserts run-time checks to ensure that errors are detected. In [Cardelli, 1991], and later in [Meijer and Drayton, 2004] it is argued that it is desirable to use static typing where possible and dynamic typing when needed.

The choice between static and dynamic checking, or combinations of both, involves trade-offs. Static typing has the advantage that it allows earlier error checking, better documented code because the use of type signatures, more opportunity for compiler optimizations and less overhead from consistency checks performed at run-time. Dynamic typing allows constructs like eval functions which execute arbitrary data as code, which in general would be illegal in a static typing.

The datatype Dynamic was proposed in [Abadi *et al.*, 1991] to safely deal with dynamic data in a statically typed language, but this solution relies on explicitly added introduction and elimination expressions. Alternatively, *gradual typing* [Siek and Taha, 2007] also uses the type Dynamic, but inserts implicit coercions which may cause types to mismatch at run-time. This approach is very flexible, but does only give static guarantees for the statically typed terms. The gradual typing approach is now also applied in the field of Cyber-Physical modeling. Modelyze [Broman and Siek, 2012] is a host language for embedding modeling languages as domain-specific languages (DSLs), whereas Ptolemy II implements different MoCs as directors. Modelyze is gradually typed and does not have subtyping or type inference. Many other solutions have been proposed for mixing dynamic and static typing, a comprehensive list of which is also given in [Broman and Siek, 2012].

2.2.6 Subtyping & Inheritance

A very popular language construct that is found in many programming languages, is subtyping. A *subtype* is a data type that relates to a *supertype* by some notion of *substitutability*. The *principle of safe substitution* entails that if S is a subtype of T, denoted $S <: T$, then S can be used safely in all contexts where T is expected [Pierce, 2002]. This relation is captured in the *rule of subsumption*:

$$\frac{\Gamma \vdash e : S \qquad S <: T}{\Gamma \vdash e : T} \quad \text{(T-SUB)}$$

This rule establishes that any expression of type S, is a also an expression of type T.

Another concept that is prevalent in modern programming languages is *inheritance*, which is a tool for code re-use. Inheritance is not subtyping [Cook *et al.*, 1989]. Through inheritance it is possible to create new objects from existing ones by creating a *base class* and extending it, which results in a *subclass* that inherits properties from the base class. This idea is fundamental to the notion of *Object Oriented Programming (OOP)*.

Subtyping relations (or type equivalence relations, for that matter) can be *opaque* or *transparent*. Opaque types declarations are used in the context of *nominal typing* where the names of types are considered as identifiers of types. On the other hand, transparent type declarations, allow for types to be expressed (recursively) in terms of other types. This is also referred to as *structural typing*.

2.2.7 Polymorphism

Typed languages where every function or procedure and hence their operands have a unique type, are said to be *monomorphic*. In contrast, in *polymorphic* languages, function arguments can have multiple types. In [Cardelli and Wegner, 1985] two main categories of polymorphism are identified; *universal polymorphism* and *ad-hoc polymorphism*. Whereas universally polymorphic functions work on well-defined, possibly infinite sets of types, ad-hoc polymorphic functions only work on finite sets of potentially unrelated types. In this categorization, universal polymorphism is further

subdivided in *inclusion polymorphism* and *parametric polymorphism*. Ad-hoc polymorphism comes in the varieties *overloading* and *coercion*.

Parametric polymorphism

Parametric polymorphism involves the use of implicit or explicit type parameters which determine the types of the arguments for each application of a polymorphic function. Functions that exhibit parametric polymorphism are also referred to as *generic functions*, or in short *generics*. Templates in C++ and generics in Java, are examples of *explicit parametric polymorphism*, where the type parameter must be declared explicitly. In C++ separate instances of the templated class or function are generate for every permutation of type parameters it is used with. Java on the other hand, uses type erasure, which means that at compile time generic types are replaced with ordinary types and casts are inserted where necessary. The exact opposite is done in Standard ML, which implements *implicit parametric polymorphism*, meaning that type parameters are computed during compilation by means of type inference.

Inclusion polymorphism

Inclusion polymorphism is based on the idea that an object can be viewed as belonging to many different classes, one of which can be included in the other. Subtyping is an instance of inclusion polymorphism. A type can be substituted by its subtype conform the (T-SUB) rule given in Section 2.2.6. By the same token, an expression can be safely ascribed a supertype of the type that would naturally be assigned by the type checker. This action is referred to as *upcasting* and can be viewed as a form of *abstraction* — representing a value such that certain parts are hidden. The inverse operation, *downcasting* which involves ascribing a type to an expression that is a subtype of its assigned type, is not safe. This is because it might cast to a type that is not included in its original type. However, in order for polymorphic functions to be useful, their arguments must be downcast inside of the function, which enables a kind of "poor-man's polymorphism", because the safety of downcasts must be enforced dynamically [Pierce, 2002].

Overloading

A generic function can be considered to be a single value that has multiple types. In overloading, the same symbol is used to denote different functions. Given a context, the appropriate function is inserted. If overloading is done statically, at compile-time, this can be observed as a syntactic shorthand rather than a semantic operation. Overloading can also take place at run-time, which is often referred to as *dynamic dispatch*, *dynamic look-up*, or *multi-method dispatch* [Mitchell and Krzysztof, 2002].

Coercion

Coercion, *implicit type conversion*, or *automatic type conversion* is a semantic operation that takes an argument of a function and converts it into the type that the function expects. Coercions can be inserted statically, at compile-time, or dynamically by the run-time type checker. The distinction between overloading and coercion is not always clear as it cannot be unambiguously derived from an expression which of the two mechanisms play what role. Consider e.g. `3 + 4.0`, which could be evaluated by coercing `3` into `3.0` before doing a double addition, or by having `+` be overloaded to add integers and doubles.

Chapter 3

The Ptolemy II type system

The type system deployed in Ptolemy II was developed by Yuhong Xiong and was extensively described in his PhD thesis [Xiong, 2002]. Over the years, only minor changes have been made to the source code that implements the type system. The first two sections of this chapter, which cover the description of some of the key features of the type system and their characteristics, are drawn from both Xiong's thesis and the actual implementation itself, such that it is sound with the current implementation. The third section focuses particularly on the subject of type resolution, or type inference. It provides a deeper theoretical insight in the mathematics and algorithms that drive type inference in Ptolemy II. It gives rise to a mathematical interpretation that is used in subsequent chapters to explain the contributions made in this thesis. Finally, the remainder of this chapter highlights some aspects of the combination of polymorphism and subtyping in Ptolemy II that are rather unexpected and therefore worth mentioning.

3.1 Characteristics

The type system combines static typing with run-time type checking. As advocated in [Cardelli, 1991], static typing is used as much as possible, and dynamic checking used when necessary. The strict observance of both secures the absence of unchecked run-time type errors, hence Ptolemy II is *strongly typed* (or rather, *strongly checked* [Cardelli, 1996]). The ports of the actors in a model can be annotated with types and type inference is used to resolve types for ports that are left undeclared. The types are established statically i.e., prior to execution. However, since a model essentially only governs the interactions between actors that are treated as black-box components that may exchange tokens between connected ports, there is really no guarantee that actors will live up to the restrictions that are imposed by the types assigned to their ports. Therefore, in order to warrant type safety during run-time, each token is checked for compatibility with the ports it is attempted to be sent through. This way, a type error is detected at the earliest possible time. The Java programming language in which Ptolemy II is implemented also relies on run-time type checking to enforce type safety [Syme, 1999].

In [Xiong, 2002] the design decisions behind the key features of the type system are motivated. We only briefly discuss the most essential parts here.

Automatic type conversion Because type conversion between primitive types happens frequently in programs, the required conversions are done automatically at run-time if they can be done losslessly. Implicit type conversion, or *coercion* is form of ad-hoc polymorphism that is very similar to overloading [Cardelli and Wegner, 1985]. Whereas coercion is a semantic operation that is needed to convert an argument to the type expected by a function, with overloading the same operator or function name is used to denote different functions.

Subtyping Subtyping is a very powerful mechanism that allows for the ordered extension of large software systems [Cardelli, 1991] and it defines the core of many popular object-oriented languages like Java or C++. In order to facilitate object-oriented design the type system has to recognize the subtyping relation among types.

Polymorphism The key benefit of component-based design is the ability to reuse components. In order to use the same actors in different settings with different types, polymorphism is required. This keeps the library of actors more compact and comprehensible.

Support for design optimization When different type assignments are possible, the type system should choose types that have the lowest cost of implementation. This is especially relevant if actor models are used to generate code for deployment on resource constrained platforms like embedded systems.

Structured types Structured types are very useful for organizing related data in a way that makes programs more readable. Records or arrays can be used to bundle multiple datums in a single token and transfer them in one round of communication, which makes execution more efficient and simplifies the topology of a model by reducing the number of ports.

Extensibility It is possible to add new types to the design environment as it is expected to continue developing. The type system accommodates for this.

3.2 Mathematical foundations

3.2.1 Partial orders & Lattices

A partially ordered set (poset) is a binary relation ($\leq$) defined over a set, indicating that one element precedes another, thereby arranging the elements of the set in some order. The relation $\leq$ is reflexive ($x \leq x$), transitive ($x \leq y \wedge y \leq z \Rightarrow x \leq z$) and antisymmetric ($x \leq y \wedge y \leq x \Rightarrow x = y$). The order is partial because not all elements need be related. Two unrelated elements are said to be incomparable. If the order is total, meaning all elements are related and thus comparable, the elements are ordered linearly.

Consider a poset P, an element $x \in P$, and a non-empty subset $S \subset P$. If x is greater than or equal to every element in S, then x is an *upper bound* of S. Conversely, if x is less than or equal to every element in S, then x is a *lower bound* of S. A *least upper bound* (LUB), also called *supremum* or *join*, is an upper bound that is less than or equal to all upper bounds. An LUB need not exist, as upper bounds may be incomparable, but if it exists, it is unique. A *greatest lower bound* (GLB), also called *infimum* or *meet*, is a lower bound that is greater than or equal to all lower bounds. Again, a GLB need not exist, but if it exists, it is unique. In mathematical notation we use the $\sqcup$

symbol for the LUB, and the $\sqcap$ symbol for the GLB. These symbols are used interchangeably as binary operators and set operators:

$$x \sqcap y = \sqcap\{x, y\} \tag{3.1}$$

$$x \sqcup y = \sqcup\{x, y\} \tag{3.2}$$

A poset P of which each of its directed subsets $S \subset P$ (i.e., S is non-empty and every pair of elements in S has an upper bound in S) has a join is called a *complete partial order* (CPO). A *lattice* is a CPO in which any pair of elements has a meet and a join. If any two elements in a poset have a meet but not necessarily a join, this structure is called a *meet-semilattice*. Similarly, a poset that has a join for any two elements but does not have meet for each pair, is called a *join-semilattice*. A lattice is *bounded* if it has a greatest element ($\top$) and a least element ($\bot$). A *complete lattice* requires that every subset $S \subset P$ has a join and a meet in P. Every complete lattice is a bounded lattice.

3.2.2 Inequality constraints

The inequality constraints used to express constraints on type variables, therefore also referred to as *type constraints*, are of the following form:

$$\left.\begin{array}{c} \tau \\ f(\tau_1, \tau_2, ..., \tau_n) \\ c \end{array}\right\} \leq \left\{\begin{array}{l} \tau \\ c \end{array}\right. \tag{3.3}$$

Type variables τ, constants c, and monotonic functions $f : \mathcal{L} \to \mathcal{L}$ are defined over a set of types that are ordered in a lattice $\mathcal{L}$. A constant is an immutable type variable that is strictly greater than $\bot$. Monotonic functions (see Section 3.2.3) are only allowed on the left-hand side of the inequality, which makes the inequality *definite*. This is a desirable property because the constraint solving algorithm in [Rehof and Mogensen, 1999] (discussed in Section 3.4.2) only admits inequalities of this particular kind.

3.2.3 Monotonic functions

A function $f : X \to Y$ where X and Y are ordered sets, is *monotonic* if it preserves the given order i.e. for any $x_1, x_2 \in X$ we have that:

$$x_1 \leq x_2 \quad \Rightarrow \quad f(x_1) \leq f(x_2) \tag{3.4}$$

Repeated evaluation of a monotonic function with increasing inputs will yield a monotonically increasing output sequence. Likewise, evaluation of a series of decreasing inputs will yield a monotonically decreasing output sequence. This is particularly useful when operating on finite partial orders because that allows one to express certain guarantees about the convergence of such series of function evaluations.

A monotonic function $f : P \to Q$ where P and Q are partially ordered sets, is Scott continuous if $\sqcup f(S) = f(\sqcup S)$ for every directed subset $S \subset P$.

3.2.4 Fixed points

A fixed point of a function is a value for which the function maps that value to itself i.e., $f(x) = x$. A function can have multiple fixed points, and if defined over a partial order, have a greatest and least fixed point. The *greatest fixed point* is greater than or equal to all other fixed points. The *least fixed point* is less than or equal to all other fixed points. If multiple fixed points exist, but all of them are incomparable, then no greatest or least fixed point exists.

Kleene fixed point theorem

The Kleene fixed point theorem states that a Scott continuous function $f : \mathcal{L} \to \mathcal{L}$ has a unique least fixed point, which is the least upper bound of the ascending Kleene chain of f [Davey and Priestley, 2002]:

$$lfp(f) = \sqcup\{f^n(\bot) \mid n \in N\} \tag{3.5}$$

The Kleene chain is obtained by iterating f by starting out with $\bot$ until $f(x) = x$, which gives the sequence:

$$\bot \leq f(\bot) \leq f(f(\bot)) \leq ... \leq f^n(\bot) \leq ... \tag{3.6}$$

3.3 Subtyping & Lossless convertibility

A subtype relationship is generally understood to guarantee that no type errors occur when objects are used in place of supertype objects [Leavens and Dhara, 2000]. The notion of behavioral subtyping is much stronger as it guarantees that substitution of an object by an instance of its subtype goes without changing the behavior of the program [Liskov and Wing, 1994]. Java does not enforce behavioral contracts, it only guarantees the availability of certain fields and certain methods. The subtyping in Ptolemy II is based on Java subtyping in combination with coercion, the latter of which can be viewed as ad-hoc subtyping [Mitchell, 1984].

3.3.1 Type lattice

The types in Ptolemy II are ordered in a lattice. A graphical representation of the type lattice is shown in Figure 3.1. By virtue of this ordering, subtype relations are reflexive, transitive, and antisymmetric. The ordering of types is based on the property of lossless convertibility. If there is a path upward from type τ_1 to τ_2 then τ_1 is losslessly convertible into τ_2. An Int can be converted into a Double, but an Int cannot be losslessly converted into a Boolean. The types Int and Boolean are said to be *incomparable* as no path exists from either to one another; Int is not a subtype of Boolean, nor is Boolean a subtype of Int.

Types correspond to tokens, which are the objects exchanged by actors. Both types and tokens are implemented in Java classes. Tokens are immutable containers that carry some state which can be accessed through a set of methods that make up for the interface of the token. Types are used to annotate ports and they provide the conversion methods that are used for automatic type conversion.

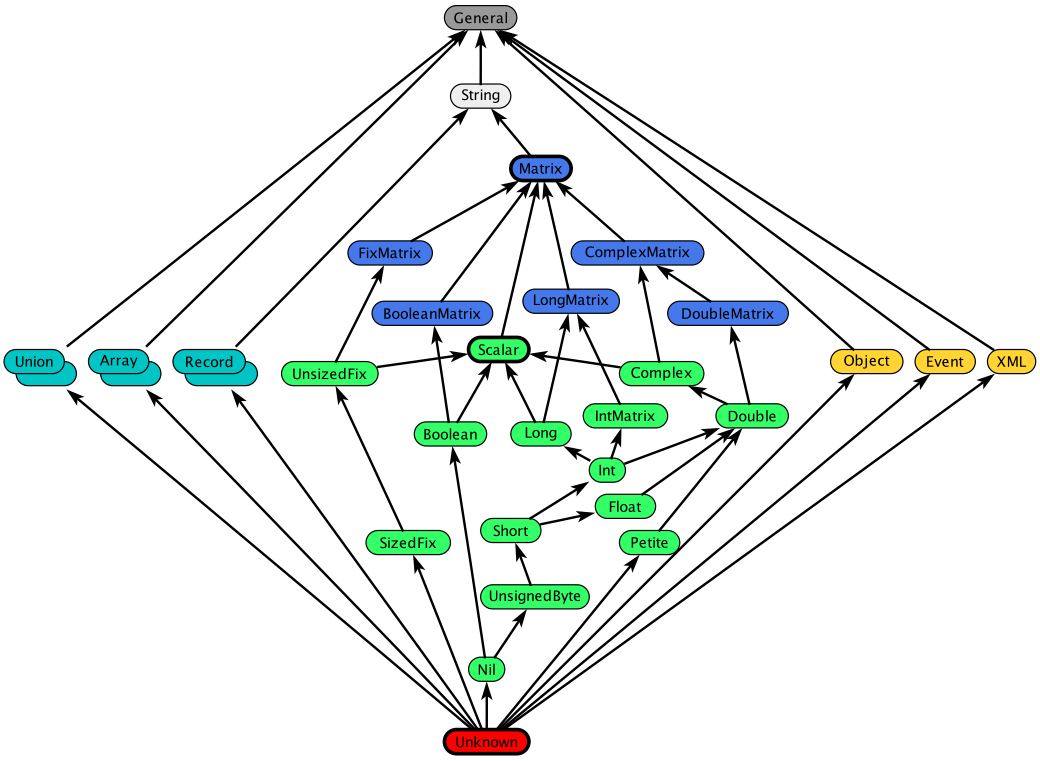


Figure 3.1: A Hasse diagram of the type lattice.

Aside the subtyping relations defined in the type lattice, there exists a Java subtyping relation between token classes. The $\top$ element of the lattice, type General, corresponds to Token, the base class of all tokens. The type Unknown, the $\bot$ element of the lattice, does not correspond to any token class. Although every token class extends Token, not all subtyping relations in the lattice are reflected in the type hierarchy of the Java implementation of the tokens. E.g., the type Scalar is a subtype of Matrix in the type lattice, but ScalarToken is not a a subtype of MatrixToken.

Not all types correspond to instantiable tokens. Both ScalarToken and MatrixToken are abstract classes. Conversion to such non-instantiable type does not actually constitute a conversion, it merely asserts compatibility through a common interface.

Throughout this document, $\top$ is always assumed to correspond to General and $\bot$ is used as a shorthand of Unknown.

3.3.2 Basic Types

A basic type **B** is subtype of another type **A** if and only if **A** is reachable from **B** through the directed edges in the type lattice. This defines a nominal subtyping relation [Pierce, 2002]; the subtype relation is purely declarative as it cannot be derived from the structure of the type. The types are ordered such that a subtype relation implies lossless convertibility between a type and its supertype.

3.3.3 Structured types

In contrast to the nominal subtyping relation between basic types, the subtyping relation between structured types is determined by the inner structure of types. Structures can be arbitrarily nested, and the leaves of these tree-like structures are again basic types.

Arrays

The subtyping relation for arrays is covariant, meaning lossless conversion is directed from a specialized type to a more general type, therefore:

$$\tau_1 \leq \tau_2 \Rightarrow \{\tau_1\} \leq \{\tau_2\} \tag{3.7}$$

Optionally, arrays types have a length parameter. Array types with different lengths are incomparable, but a bounded array can always be losslessly converted in an unbounded array:

$$\{\tau, l\} \leq \{\tau\} \tag{3.8}$$

Records

Records feature two distinct subtyping relations, depth subtyping and width subtyping. Depth subtyping regards the types of individual record elements, all the way down to their basic type. If for each label in a record **B** the type is less than or equal to the type corresponding to the same label in a record **A**, then **B** is a subtype of **A**, for example:

$$\{name = string, value = int\} \leq \{name = string, value = double\} \tag{3.9}$$

Width subtyping is concerned with the presence of labels. A record type with more labels is more specific i.e., less general than a record type with fewer labels, for example:

$$\{name = string, value = int\} \leq \{value = int\} \tag{3.10}$$

At run-time, it is always safe to convert a token into a token of a more general type because it does not incur any loss of information. However, at first sight it does not seem lossless at all to erase fields from a record in order to convert it to a record of greater type. But from a typing perspective this makes perfect sense.

A type expresses some guarantee, or at least an expectation, about the value of the thing it describes. For records, this claim is expressed in terms of a set of labels; a token is compatible, only if all expected labels are present. Additional labels can be safely ignored at run-time, because the types are static. If the subtyping relation would be inverted, then a conversion would entail padding records with extra fields i.e., create something out of nothing. One could argue that this is very similar to what happens when e.g. a Float is converted into a Double, but this padding occurs only on the level of the data representation. Semantically, the two different representations are the same. A record with added fields that hold some arbitrary value, is semantically different from a record without those fields. Hence, one must bare in mind that a record type that is smaller in size, is indeed greater in type.

Unions

Subtyping among unions is also defined in terms of depth subtyping and width subtyping. The depth subtyping for unions is the same as for records:

$$\{|\ name = string, value = int\ |\} \leq \{|\ name = string, value = double\ |\} \tag{3.11}$$

The width subtyping for unions is opposite to that of records:

$$\{|\ name = string\ |\} \leq \{|\ name = string, value = double\ |\} \tag{3.12}$$

A record must contain an element in its structure for each label in its type, but a union instance always has only one element present at the time. Therefore, a union type with more labels is indeed more general as it allows for more instances to fit the type description.

3.4 Type inference

In Ptolemy II, some actors declare types for the data they produce or consume, but some do not. Other actors constrain their input or output to be at least or at most of a certain type because their operations are not universally applicable to just any type. In order to have a well-typed model, all of these constraints must be satisfied using some assignment of types. The automated process that finds this assignment is referred to as *type inference* or *type reconstruction*.

The type inference mechanism that is deployed in Ptolemy II is inspired by the Hindley-Milner algorithm [Hindley, 1969] used in the programming language ML, but is different in many respects. These differences mainly arise from the fact that the Ptolemy II type system has a notion of subtyping, whereas Hindley-Milner does not. As a result, type rules cannot be captured in a set of equations defined over a set of type variables. Instead, type constraints are used that take the form of definite inequalities, as discussed in Section 3.2.2.

The Hindley-Milner algorithm finds the *principal type* (i.e., the most general type) for each type variable. This has the advantage that modules can be compiled separately and recomposed without the need to redo type inference, because the found solution is already most general. Ptolemy II does the exact opposite in so much as that it finds the most specific set of type assignments. In [Xiong, 2002] it is argued that the most specific type has the advantage of having the least implementation cost. The loss of composability of precompiled modules is not a concern because type resolution is always performed on a complete model prior its execution. Moreover, each occurrence of the same actor has a separate instance with its own type assignments. This means there is no need to compute the principal type in order to achieve component re-use.

The remainder of this section formulates the problem of type resolution in mathematical terms and provides an intuition for what causes it to fail or succeed.

3.4.1 Problem formulation

From Equation 3.3, we can tell apart two types of type constraints; one with a type variable on the right-hand side and one with a constant on the right-hand side. It is relevant to discuss them separately because they express different things. By aggregating these inequalities in two separate groups we can obtain two comprehensive type constraints that jointly describe the complete constraint system.

Type constraints that have a *type variable* on the right-hand side of the inequality can be generalized in the form shown in Equation 3.13, by substituting type variables by identity functions and constants by constant functions. By definition, these functions are monotonic.

$$f(\tau_1, ..., \tau_n) \leq \tau_i, \quad \tau_i \in \mathcal{T} \tag{3.13}$$

If we group these inequalities by the type variable on the right-hand side, we can form a single constraint for each type variable by taking the least upper bound (LUB, $\sqcup$) of the left-hand sides of the inequalities in each group. Assuming $\alpha \leq \tau$ and $\beta \leq \tau$, τ is an upper bound for α and β, therefore $\alpha \sqcup \beta \leq \tau$. Conversely, if $\alpha \sqcup \beta \leq \tau$ is assumed, by definition we have $\alpha \leq \alpha \sqcup \beta$ and $\beta \leq \alpha \sqcup \beta$, then by transitivity $\alpha \leq \tau$ and $\beta \leq \tau$. The resulting function as follows:

$$F = f_1(\tau_1, \tau_2..., \tau_n) \sqcup f_2(\tau_1, \tau_2..., \tau_n) \sqcup \cdots \sqcup f_3(\tau_1, \tau_2, ..., \tau_n) \leq \tau_i = \begin{cases} f_1(\tau_1, \tau_2..., \tau_n) \leq \tau_i \\ f_2(\tau_1, \tau_2..., \tau_n) \leq \tau_i \\ \vdots \\ f_n(\tau_1, \tau_2..., \tau_n) \leq \tau_i \end{cases} \tag{3.14}$$

After obtaining a single inequality for each right-hand side occurring type variable, we can express them jointly in the following form:

$$P(\vec{\mathcal{T}}) = \begin{bmatrix} F_1(\vec{\mathcal{T}}) \\ F_2(\vec{\mathcal{T}}) \\ \vdots \\ F_N(\vec{\mathcal{T}}) \end{bmatrix} \leq \vec{\mathcal{T}} \tag{3.15}$$

The resulting inequality $P(\vec{\mathcal{T}}) \leq \vec{\mathcal{T}}$ is the part of the constraint system that describes the ordering relations between possible assignments of type variables. Note that $\vec{\mathcal{T}}$ is a tuple that contains all type variables in the system. Because types are ordered in a lattice, the values of tuple $\vec{\mathcal{T}}$ are ordered by a *tuple lattice*. Let $\vec{\top}$ be the top of this lattice where each element in the tuple is assigned $\top$. Then $\vec{\bot}$ represents the tuple in which each element is assigned $\bot$. The ordering relation in the tuple lattice as follows:

$$\forall\, i \in N,\ \vec{\mathcal{T}}_i \leq \vec{\mathcal{S}}_i \iff \vec{\mathcal{T}} \leq \vec{\mathcal{S}} \tag{3.16}$$

A trivial solution that would satisfy $P(\vec{\mathcal{T}}) \leq \vec{\mathcal{T}}$ would thus be $\vec{\top}$. From a type resolution perspective, this is typically not a sensible assignment at all. It means that every token could be of any type because every port that is typed $\top$ would accept it. This renders static type checking practically meaningless.

However, $P(\vec{\mathcal{T}}) \leq \vec{\mathcal{T}}$ only embodies the first part of the constraint system. The other part is formed by the constraints that have a *constant* on the right-hand side of the inequality. Similarly as shown in Equation 3.13, we can rewrite those inequalities as:

$$g(\tau_i, ..., \tau_n) \leq c, \quad \tau_i \in \mathcal{T}, \quad c \in \mathcal{C} \tag{3.17}$$

Grouping by the constant on the right-hand side, gives:

$$G = g_1(\tau_1, \tau_2..., \tau_n) \sqcup g_2(\tau_1, \tau_2..., \tau_n) \sqcup \cdots \sqcup g_3(\tau_1, \tau_2, ..., \tau_n) \leq c = \begin{cases} g_1(\tau_1, \tau_2..., \tau_n) \leq c \\ g_2(\tau_1, \tau_2..., \tau_n) \leq c \\ \vdots \\ g_n(\tau_1, \tau_2..., \tau_n) \leq c \end{cases} \tag{3.18}$$

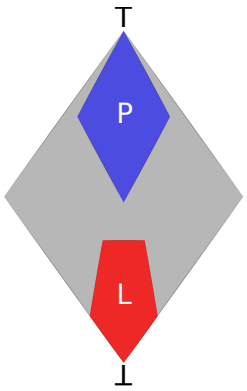
$$L(\vec{\mathcal{T}}) = \begin{bmatrix} G_1(\vec{\mathcal{T}}) \\ G_2(\vec{\mathcal{T}}) \\ \vdots \\ G_N(\vec{\mathcal{T}}) \end{bmatrix} \leq \vec{\mathcal{C}} \tag{3.19}$$

This inequality makes up for the second part of the constraint system. The constraint $L(\vec{\mathcal{T}}) \leq \vec{\mathcal{C}}$ imposes an upper bound on the solutions that satisfy $P(\vec{\mathcal{T}}) \leq \vec{\mathcal{T}}$. As will be established in the remainder of this section, the constraint $P(\vec{\mathcal{T}}) \leq \vec{\mathcal{T}}$ imposes a lower bound on the solutions that satisfy $L(\vec{\mathcal{T}}) \leq \vec{\mathcal{C}}$.

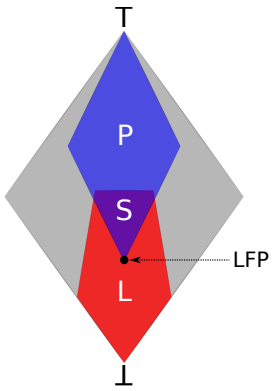
The constraints that jointly constitute $P(\vec{\mathcal{T}}) \leq \vec{\mathcal{T}}$ provide the mechanism for type variables to influence one another. In search of a solution of the constraint system, assigning a type to one variable in order to satisfy some constraint might require another variable to be a assigned a higher or lower type in order to satisfy some other constraint. By transitivity, since $\alpha \leq \beta \wedge \beta \leq \gamma \Rightarrow \alpha \leq \gamma$, assigning a type to any variable α during the process of type resolution (i.e., constraint solving), might force another variable γ to be assigned a higher type. Hence, these type constraints provide the infrastructure to propagate type information between dependent type variables, therefore we call them *propagation constraints*. It should be noted that the dependencies between type variables established by the propagation constraints are also the ones that warrant lossless convertibility between tokens sent from one port to another.

The constraints that construct $L(\vec{\mathcal{T}}) \leq \vec{\mathcal{C}}$ determine the ceiling of the solution space. These constraints, all of the form $\alpha \leq c$, we call *limitation constraints*. They each impose an upper bound on whatever is found on the left-hand side of the inequality, usually a type variable or a monotonic function. Obviously, a type constraint between two constants is trivially resolved; it either satisfies or not, regardless of type assignments made to any type variable. Since monotonic functions again depend on type variables, the limitation constraints ultimately express relations between type variables and constants.

All possible type assignments are graphically represented in the type lattice in Figure 3.2. The blue region **P** includes all possible type assignments that satisfy the propagation constraints and the red region **L** captures all type assignments that satisfy the limitation constraints.



(a) The intersection of region **P** which represents the solutions for $P(\vec{\mathcal{T}}) \leq \vec{\mathcal{T}}$ and region **L** which represents the solutions for $L(\vec{\mathcal{T}}) \leq \vec{\mathcal{C}}$, is empty. Type conflicts remain and the constraint system is not satisfiable.



(b) The intersection of regions **P** and **L**, region **S**, represents all type assignments that satisfy both $P(\vec{\mathcal{T}}) \leq \vec{\mathcal{T}}$ and $L(\vec{\mathcal{T}}) \leq \vec{\mathcal{C}}$. All solutions that satisfy the constraint system are thus found in **S**.

Figure 3.2: Graphical representation of the possible type assignments of $\vec{\mathcal{T}}$.

Region **P** always includes the top of the tuple lattice, because assigning $\top$ to every type variable trivially solves all propagation constraints; $\alpha \leq \top$ is always true since α ranges over the types in the type lattice. The lowest point of **P** in the tuple lattice corresponds to the smallest, or most specific set of type assignment that satisfies $P(\vec{\mathcal{T}}) \leq \vec{\mathcal{T}}$.

Assuming that the tuple lattice $\mathcal{L}$ is of finite height, then $P : \mathcal{L} \to \mathcal{L}$ being monotonic, is also Scott continuous [Edwards, 1997]. This means it preserves all directed suprema, i.e., $\sqcup P(\vec{\mathcal{T}}) = P(\sqcup\vec{\mathcal{T}})$. The Kleene fixed point theorem (Section 3.2.4) states that a Scott continuous function P has a unique least fixed point. In [Edwards, 1997] it is shown that the least fixed point is also the least *prefix* point $P(\vec{\mathcal{T}}) \leq \vec{\mathcal{T}}$, meaning there is indeed no smaller solution to this inequality. Therefore, the least fixed point is the lowest point in **P**. The presence of infinite chains in the type lattice can break continuity of P and thereby cause there not to exist a least fixed point. However, if it exists, then it is the least upper bound of the chain $\{\vec{\bot}, P(\vec{\bot}), P^2(\vec{\bot}), ...\}$.

A solution for the complete constraint system is found in the intersection of **P** and **L**. Figure 3.2a illustrates the situation where no solution exists as the two regions do not overlap. More specifically, if the least fixed point of $P(\vec{\mathcal{T}})$ does not satisfy $L(\vec{\mathcal{T}}) \leq \vec{\mathcal{C}}$, then there is no solution. Figure 3.2b shows at least one solution; the least fixed point. Any prefix point within **S**, is a solution too.

In addition to the propagation and limitation constraints captured in the inequalities $P(\vec{\mathcal{T}}) \leq \vec{\mathcal{T}}$ and $L(\vec{\mathcal{T}}) \leq \vec{\mathcal{C}}$ respectively, there is a third criterion that must be satisfied:

$$\forall\, \tau \in \mathcal{T}. \bot < \tau \tag{3.20}$$

This criterion however, which states that every type variable must be assigned a type strictly greater than $\bot$, can generally not be expressed using an inequality in the tuple lattice. Since greater than or equal to bottom includes bottom itself, strictly greater than bottom would mean: greater than or equal to the types directly above bottom. Unless the type lattice consists of a single chain,

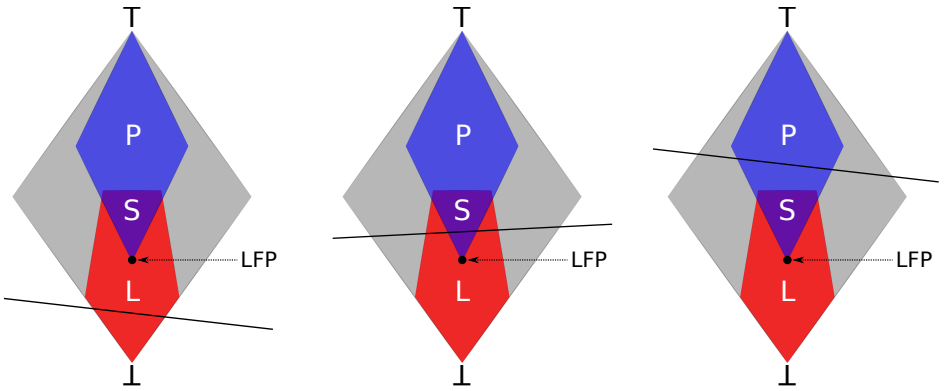
which is typically not the case, then there are multiple types directly above bottom. Setting up an inequality for every type directly above bottom would not work either. If c_x and c_y are right above $\bot$ and $c_x \leq \tau$ and $c_y \leq \tau$, then $c_x \sqcup c_y \leq \tau$. By definition, the least upper bound of c_x and c_y is higher than either c_x or c_y, or both. Hence, these constraints resolve to a solution that is at minimum not one, but two levels higher than $\bot$, which is too constraining. Ptolemy II attempts to find a solution that satisfies the constraint system as is, but simply rejects this solution if it contains any type assignments that are equal to $\bot$.

The $\bot$ element does not represent an instantiable type, but together with the $\top$ element, it gives the partial order of types the structure of a lattice. This elegant mathematical structure allows for the deployment of an efficient type resolution algorithm described in Section 3.4.2. More specifically, this algorithm requires there to exists a greatest lower bound for any subset of types. Introducing a $\bot$ element provides all types with a meet, but there is no obvious way to associate $\bot$ with a concrete type. As opposed to $\top$, which is the type that anything could be losslessly converted into, $\bot$ is the type that must be losslessly convertible into anything. In other words, having a token instance that is typed $\top$, means this token could be anything. But having a token of type $\bot$, means it must be an instance of a token that represents *everything*. It is hard to imagine a piece of data that could consistently be captured as such, meaning that all representations would indeed truthfully reflect the data such that they could be used interchangeably.

Assuming it would be possible, a finite amount of data represented by an instance of type $\bot$ would need to be convertible into the unboundedly many incomparable types that are present in the type lattice. There are only two ways to achieve this. The first one would be for infinitely many different representations to be encoded in the data, which contradicts the data being finite. The second one entails any missing data to be generated in some arbitrary way, as a token of type $\bot$ is converted into something else. The latter breaks the property of lossless convertibility and the following example shows how. Assume we have a token that is typed $\bot$, and binds the label x to some floating point value. In order to convert this datum into a record of type $\{x = float, y = int\}$, some value needs to be generated to associate with label y. If this would be considered a lossless conversion, then by transitivity — because $\{x = float, y = int\}$ can be losslessly converted into $\{y = int\}$ — $\{x = float\}$ would be losslessly convertible into $\{y = int\}$. This contradicts the width subtyping relation of records, out of which it follows that $\{x = int\}$ and $\{y = float\}$ are incomparable.

The threshold that separates type assignments that have no $\bot$ elements from ones that do, is graphically represented in the tuple lattice in Figure 3.3. Tuples of possible type assignments that are greater than or equal to the threshold have no $\bot$ elements, and tuples that are less than the threshold contain at least one $\bot$ element. If the threshold lies below or intersects with the least fixed point as is depicted in 3.3a, then all solutions in region **S** are acceptable since none of the type variables are assigned $\bot$.

As shown in Figure 3.3c if the threshold lies above **S**, then there is no solution that does not have one or more $\bot$ assignments. In this case, even if the constraint system can be satisfied, it cannot be satisfied without including $\bot$ assignments. Therefore all solutions are rejected.



(a) The least fixed point (LFP) is above the threshold. Hence, all solutions in **S** are accepted.

(b) The threshold is above the LFP but within **S**. Only the solutions in **S** above the threshold are accepted.

(c) Every solution in **S** is below the threshold. Hence, no solutions are accepted.

Figure 3.3: Graphical representation of the threshold $\forall\, \tau \in \mathcal{T}.\bot < \tau$.

The most interesting case is illustrated in Figure 3.3b, where the threshold lies strictly above the least fixed point, which splits up **S** into two separate regions. Like in the other cases, any solution below the threshold is rejected. However, there may very well be a solution i.e., another fixed point greater than the least fixed point, that lies above the threshold. As will be explained in the next chapter, the core of this thesis involves finding precisely such solutions.

3.4.2 Type resolution algorithm

As was concluded in the previous section, the least fixed point is unique, and yields the set of most specific type assignments, given some set of type constraints. To reach this fixed point, Ptolemy II uses an efficient algorithm given by Rehof and Mogensen. This is a linear time algorithm for deciding satisfiability of sets of inequality constraints involving monotonic functions, defined over a finite meet-semilattice. As explained in [Rehof and Mogensen, 1999], the algorithm leverages the notion of *definite inequalities* to formulate a class of constraint solving problems that is tractable. Moreover, Rehof and Mogensen prove that *any* strict extension of definite inequalities will lead to **NP**-hard problems.

A meet-semilattice is a partially ordered set which has a greatest lower bound (meet) for any non-empty finite subset [Davey and Priestley, 2002]. The types in Ptolemy II are ordered in a lattice, of which the basic types form a complete sublattice (because it is non-empty and finite), but the structured types form infinite sublattices. Because of this, type resolution does not always converge. In [Xiong, 2002] a particular corner case is highlighted where in the case of arrays, the chain $[\bot], [[\bot]], [[[\bot]]], \ldots$ may cause divergence. However, it is observed that this kind of infinite iteration is detectable and can therefore be obviated by setting a bound on the depth of structured types that contain $\bot$. Moreover, it is argued that for any structured type that does not contain $\bot$ elements, all chains to the top of the lattice have finite length. This also holds for records types since the width subtyping relation is defined such that the supertype of a record always contains fewer elements. Therefore, any upward chain from a record that does not contain $\bot$ elements, is finite and ends at $\top$.

Definite inequalities either have a constant or variable expression on the right-hand side of the inequality. These two kinds of type constraints identify as limitation and propagation constraints, respectively. Monotonic functions are only allowed on the left-hand side of the inequality. The type constraints in Ptolemy II, described in Equation 3.3, conform to these rules and thus are definite. Something that is not mathematically encoded in this system of definite inequalities, is a mechanism to extract a type variable that is embedded in a structured type. Assume we have a type variable $ArrayType(\tau_{array})$, then we might want to express the inequality $\tau_{array} \leq \tau$. To this end, $ArrayType$ was augmented with a method to obtain a reference to τ_{array}. This extension is more elaborately discussed in [Xiong *et al.*, 2005].

If a least fixed point exists for the entire constraint system, i.e., the least fixed point of the propagation constraints lies within the region of solutions enclosed by the limitation constraints, then this fixed point can be found using the Rehof-Mogensen algorithm. After constructing a list of all propagation constraints, it starts out with assigning $\perp$ to all type variables. Then it begins iterating through the list. For each constraint it takes the least upper bound of both the left- and right-hand side of the inequality and assigns it to the right-hand variable. This motion perpetuates until all constraints are satisfied and a fixed point is reached. Then finally, the limitation constraints are evaluated using the type assignments obtained from the fixed point computation. If all limitation constraints are satisfied, then the solution is accepted, or otherwise it is rejected. In the Ptolemy II implementation we have the additional acceptance criterion that requires every type assignment to be strictly greater than $\perp$.

The Rehof-Mogensen algorithm is linear in the number of symbols in the constraints. More precisely, the worst-case complexity is $\mathcal{O}(3h(L) \times |C|)$, where $h(L)$ is the height of the type lattice and $|C|$ represents the number of symbols used in the entire set of constraints. As monotonic functions can take an arbitrary number of arguments, the number of symbols can vastly exceed the number of constraints. Moreover, the maximum number of constraints scales quadratically with the number of ports in a model. However, since the topology of a model is typically only sparsely connected, the number of symbols drawn from the type constraints of a model is a lot more modest.

3.4.3 Alternative interpretations

Formulating the type inference problem as a constraint solving problem is attractive for different reasons. Constraints are very suitable for handling recursion and feedback loops, because once the constraints are set up the program structure itself is no longer considered. Another advantage of abstracting away from the program structure, or model topology rather, is that is that actor designers only need to understand how to formulate fitting type constraints for their actors without requiring deep insight in how the constraints are resolved. Also, since type constraints are aware of types, but not visa versa, additional types can be added over time, which makes the type system extensible.

In order to develop an intuition about the process of type resolution, the abstraction of type constraints away from structure, is not very helpful. Alternatively, the complete constraint system can be translated into a directed graph $G = (V, E)$ with vertices that are type variables or constants, and directed edges that signify dependency relations between them. More specifically, a dependency relation entails that during the process of type resolution, if α increases, then β *might* increase (or will stay the same, since due to monotonicity β shall never decrease).

For inequalities of the form $\alpha \leq \beta$, where α and β are constants or type variables, the mapping from type constraints to dependency relations is as follows:

$$\alpha \leq \beta \qquad \mapsto \qquad \alpha \to \beta \tag{3.21}$$

Note that the dependency relation $\alpha \to \beta$ means β depends on α i.e., α may influence β. Therefore, $\alpha \to \beta$ is a weaker relationship than $\alpha \leq \beta$ itself. This is important because α can be a monotonic function. For a monotonic function it holds that if $\alpha \leq \beta \Rightarrow f(\alpha) \leq f(\beta)$, but it is not required to be *strictly* monotonic i.e., $\alpha < \beta \Rightarrow f(\alpha) < f(\beta)$. Hence, if the argument of the function increases, the function result need not necessarily increase; although f depends on α, and β depends on f, an increase in α is not bound to increase β. An example of a function that is monotonic but not strictly monotonic, is the greatest lower bound; $(\bot, \bot) < (\bot, int)$ but $GLB(\bot, \bot) = \bot$ and $GLB(\bot, int) = \bot$ and $\bot \not< \bot$. For functions like these, if a function argument is pushed up the lattice, it is not bound to affect variables that depend on the function. However, because we have defined the dependency relationship as a *possible* influence during type inference, dependency relations can still be drawn from inequalities with (non-strict) monotonic functions on the left-hand side. As the function depends on its arguments, by transitivity, the type variable on the right-hand side of the inequality depends on those arguments too. Hence, the mapping from constraints of the form $f(\mathcal{A}) \leq \beta$ to dependency relations is as follows:

$$f(\mathcal{A}) \leq \beta \qquad \mapsto \qquad \{\alpha \to \beta \mid \alpha \in \mathcal{A}\} \tag{3.22}$$

In contrast to monotonic functions that map non-bottom arguments to $\bot$, there also exist monotonic functions that map $\bot$ arguments to types higher than bottom. Such a function thus directly affects variables that depend on it, without the need for any of its arguments to rise above $\bot$ (the initial value of any type variable). A trivial example of a function that produces a value independently, would be a function that does not depend on any arguments; a constant function. More generally, it is easy to construct a monotonic function with an arbitrary number of arguments that maps to a type higher than $\bot$ if all of its arguments are $\bot$. However, this very type can then be embedded as a constant in the definition of the function. So in order to capture all dependency relations that arise from the inequality $f(\mathcal{A}) \leq \beta$, it does not suffice to only set up dependency relations between β and f's arguments $\alpha \in \mathcal{A}$. In addition, for every monotonic function found on the left-hand side of an inequality it must be checked whether it evaluates to a type strictly higher than $\bot$ if all of its arguments are valued $\bot$. If this is the case, then a dependency must be drawn between this type as a constant and the term on the right-hand side of that inequality. This is reflected in the following expression:

$$f(\mathcal{A}) \quad \leq \quad \beta \quad \wedge \quad f(\vec{\bot}) = c > \bot \qquad \mapsto \qquad c \to \beta \tag{3.23}$$

Since the type resolution algorithm approaches the least fixed point after starting out with every type variable being assigned $\bot$, if the output of a monotonic function increases *after* the first iteration, this can only be due to one or more of its arguments having increased. This however is already captured by the dependency relations established between the arguments of the function and the dependent type variables.

Using the three simple rules listed in Table 3.1, an entire constraint system can be translated into a dependency graph. The dependency relations drawn from the equality constraints give rise to dependency chains between type variables that form connected clusters of variables with dependencies between them. Clusters are exclusively connected to the rest of the graph by constants, or not

α	$\leq$	β			$\alpha \to \beta$
$f(\mathcal{A})$	$\leq$	β			$\{\alpha \to \beta \mid \alpha \in \mathcal{A}\}$
$f(\mathcal{A})$	$\leq$	β	$\wedge$	$f(\vec{\perp}) = c > \perp$	$c \to \beta$

Table 3.1: Translation from type constraints to dependency relations. Terms α and β represent type constants or variables. Monotonic functions only take as arguments the variables they depend on.

connected at all. Constants are immutable and therefore independent, so a chain of dependent type variables always ends because there are simply no more relations that extend the chain, or because a constant ends it. An example of such a dependency graph is given in Figure 3.4.

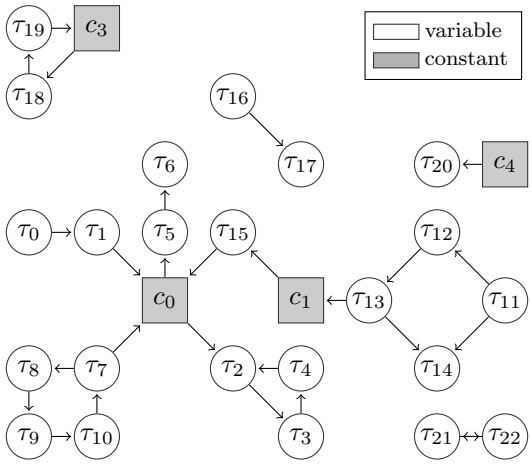
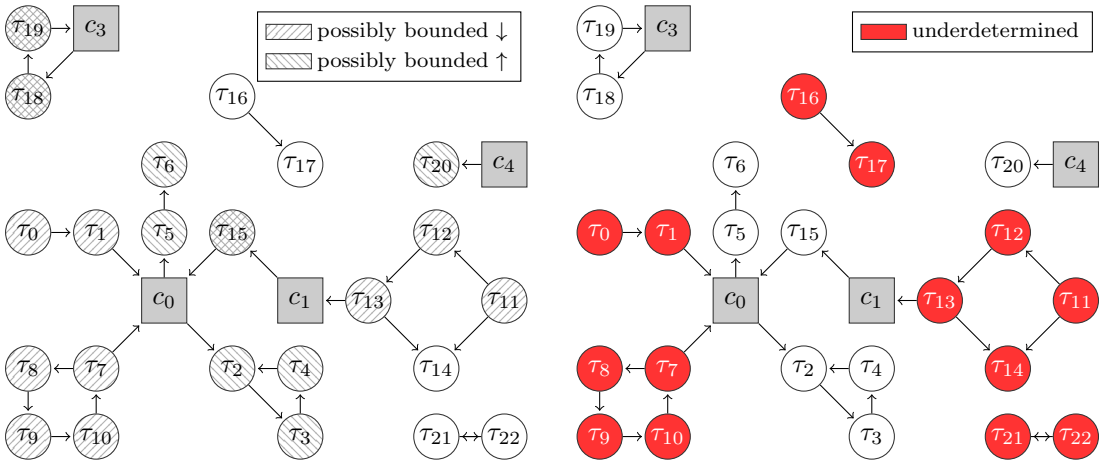


Figure 3.4: An example dependency graph that contains a number of specific features; disconnected clusters (e.g. $c_4 \to \tau_{20}$), disconnected clusters that consist of connected components (e.g. $\tau_{18} \to \tau_{19} \to c_3 \to \tau_{18}$), connected clusters (e.g. $\tau_0 \to \tau_1$), and connected clusters that consist of strongly connected components (e.g. $\tau_7 \to \tau_8 \to \tau_9 \to \tau_{10} \to \tau_7$).

What happens during the process of type resolution, can be thought of as type information being propagated along the edges of dependency relations. This happens by means of iteratively pushing up type variables until a fixed point is reached. At this point, a type assignment is found that satisfies all constraints. As discussed in Section 3.4.1, the propagation constraints, generally of the form $\alpha \leq \tau$, are responsible for pushing up type variables. The sources of the propagated type information are the type constants that, in contrast to the type variables, have an initial type greater than $\perp$. Therefore, the only way a type variable can be bounded from below ($\uparrow$) is when that type variable is reachable by a constant via a path of dependencies between type variables in the directed graph ($c \to ... \to \tau$). If there exists such a path, because of possible influence of non-strict monotonic functions, there is no guarantee that the variable is actually bounded from below. However, the inverse is true; if there is no such path, then the variable cannot possibly be bound from below.

The limitation constraints, generally of the form $\alpha \leq c$, impose limits on the type assignments of type variables. Although dependency relations $\alpha \to \beta$ drawn from such inequalities are defined as: if α increases, then β might increase; then β will of course never increase if β is a constant. In



(a) The markings of the type variables visualize their boundedness. A constant potentially bounds a variable from above (↓) if there is a path from the variable to that constant. A variable is potentially bounded from below (↑) if there is a path from a constant to that variable.

(b) The type variables that are impossibly bounded from below (↯) are underdetermined. This means they are only possibly bounded from above, or not bounded at all. These variables are are colored in red. There exists no path from a constant to any of the variables in a red cluster.

Figure 3.5: Alternative coloring of the same graph as in Figure 3.4.

fact, if a constant would be required to increase in order to satisfy a constraint, this would yield a type conflict. The path from a type variable to a constant ($\tau \to ... \to c$) signifies what one could describe as an anti-dependency. The constant is not propagated to other type variables, but if type information can travel along dependency relations from a type variable to a constant, the constant could impose an upper bound on the type assigned to the variable. If a variable is indeed bounded from above (↓), there must be a constant that defines the ceiling. Moreover, there has to exist a path from the type variable to that constant. Again, the presence of such a path does not guarantee that a variable is bounded from above, but if no such path exists, there cannot exist an upper bound for that variable.

A type variable is *underdetermined* if it is not bounded from below. This means there is no infrastructure in the dependency graph for type information to propagate from any constant to that variable. Consequently, the variable will remain stuck at $\bot$. A cluster is underdetermined if all of its connected variables are underdetermined. It should be noted that the added edges from constants to type variables for functions that map non-bottom elements to $\bot$ (as denoted in Equation 3.23), are critical for the purpose of making these claims.

The graphs in Figure 3.5 reveal cases where purely based on the type constraints, so without regarding the internals of any monotonic function, it can be concluded that type variables are underdetermined. For a constraint system that yields a dependency graph that has underdetermined variables, it is guaranteed that no solution can be found that has no $\bot$ type assignments in it. However, if the dependency graph does not have any underdetermined type variables, solving the underlying constraint system *still might fail*. The purpose of these graphs is to illustrate the information flow between type variables and emphasize that the absence of certain dependencies can

cause variables to remain underdetermined.

From the structure of these graphs we can draw two different approaches to get past the problem of underdeterminacy. One way is to add dependencies that provide pathways from constants to underdetermined type variables. This is achieved by adding more type constraints. The other way is to replace type variables with constants, which is the equivalent of doing manual type annotations. In terms of the representation of type assignments in the tuple lattice (figure 3.3), both of these measures have the potential of pushing up the least fixed point to a point where it exceeds the threshold of having no $\perp$ assignments. Replacing variables with constants has the result of increasing region **L**, whereas adding dependencies may reduce region **P**.

We can illustrate these strategies by considering examples of underdetermined clusters in Figure 3.5b. The cluster $(\tau_1 \rightarrow \tau_2)$ can be lifted up in its entirety either by substituting τ_1 with a constant, or by adding an incoming dependency that connects τ_1 with a constant or a subgraph that is not underdetermined. Possible solutions are: $(c_0 \rightarrow \tau_5 \rightarrow \tau_6) \rightarrow \tau_0$, $(c_1 \rightarrow \tau_{15}) \rightarrow \tau_0$, $(c_3 \rightarrow \tau_{18} \rightarrow \tau_{19}) \rightarrow \tau_0$, or $(c_4 \rightarrow \tau_{20}) \rightarrow \tau_0$. It is only possible to lift up an entire cluster if there is a path from a constant to *every* variable, otherwise the underdetermined cluster would just split up in a determined part and one or more underdetermined parts. For example, if τ_{14} was substituted with a constant or directly connected with another cluster to become reachable from another constant, the remainder of the cluster $(\tau_{11} \rightarrow \tau_{12} \rightarrow \tau_{13})$ would remain to be underdetermined.

Type *conflicts* are a lot less evidently exposed by the dependency graph. Any path from a constant to another constant might bring about a type conflict as the monotonic functions operating on the type variables that propagate values along such a path might lead to a type assignment that causes $\tau \not\leq c$. The strategies to eliminate conflicts are to remove dependencies to break down these paths, or to increase the constants that impose upper bounds that are too strict. In the type lattice representation, this corresponds to bringing regions **P** and **L** closer together such that they eventually might overlap. Removing dependencies may increase **P** by lowering its least point, whereas increasing constraints raises the ceiling of **L**.

Although the dependency graphs provide a useful insight in the mechanisms that drive type resolution, in order to decide which strategies are actually practically fitting to overcome underdeterminacy, we would again have to consider the semantics of the actual actor models. In the end, the aim of the type system is to prevent unsafe paths of execution, so type inference can only be meaningfully enhanced in ways that preserve this property.

3.5 Polymorphism, Coercion & Inheritance

An important aspect of actor-oriented modeling in Ptolemy II is the exploitation of component reuse by means of polymorphism. This allows for actors to be arranged and composed in a way that is in a high degree independent from the type of data that is exchanged between actors. Models with polymorphic actors are more flexible and the libraries that retain such actors are more orderly and compact. Subtyping usually refers to compatibility of interfaces, but in Ptolemy II the subtyping order implies lossless convertibility. Instead of relying on the property that if **S** is a subtype of **T**, then an instance of **S** can be safely substituted for an instance of **T**, an instance of **S** can be losslessly convert into an instance of **T**.

In an object-oriented framework, subtyping often goes hand in hand with the concept of inheritance, which is a language feature that allows new objects to be defined from existing ones [Mitchell and Krzysztof, 2002]. In typed object-oriented languages the subtyping relation is typically based on the inheritance hierarchy [Cook *et al.*, 1989]. The Ptolemy II type system has a notion of subtyping, but its expression language features no inheritance. Nevertheless, Ptolemy II is implemented in Java and leverages many of its features, including its inheritance-based subtyping relation. This is reflected in all of its components, including the actors themselves, the token types, and token instances that are exchanged between actors. The interaction between the Java type system and the Ptolemy II type system can lead to some unexpected behavior.

As mentioned in Section 3.3 the ordering of types in the type lattice is not identical to the Java subtyping hierarchy of the corresponding token classes. Polymorphism is achieved using two distinct mechanisms; coercion and inclusion polymorphism. Type coercion involves the instantiation of a new token based on the value contained by a token instance of another class. The premise for such conversion is lossless convertibility. There is no need for a Java subtyping relation to exist for coercion to be possible. An example of this is the conversion between an Int and an IntMatrix; the Ptolemy II subtype relation permits this conversion, but the Java class IntToken does not extend MatrixToken. This means that although coercion might be considered lossless with regard to the value that a token pertains, it can alter the interface and inheritance properties of a token in a way that is not lossless at all.

Some actors implement parametric polymorphism, meaning they operate uniformly on all types. An example of this is the Distributor actor, which distributes tokens from an input stream over a set of output streams, regardless of their type. However, most actors implement inclusion polymorphism, which relies on interface inheritance provided by Java subtyping. Although typing in Java is done statically, it is not possible to determine the concrete class types of symbols at compile time. It is deferred until run-time to do name binding i.e., to associate concrete objects with identifiers. Dynamic binding (also known as dynamic dispatch) lets a method invocation be dispatched to the most concrete implementation that is provided by the object instance at hand. This mechanism comes into play when an actor declares or infers its input type to be of a non-instantiable type like General or Scalar. Automatic type conversion into such types constitutes nothing more than the assertion that a token is a subtype of the corresponding Java class. Note that this assertion always holds for type General as every token class extends Token. Relying on the interface exposed by the abstract Java type, an actor can invoke the methods provided by a token polymorphically by means of dynamic dispatch.

The interaction between automatic type conversion and Java's inclusion polymorphism in Ptolemy II is best described as *coercive default inheritance*. When a token arrives at the input port of an actor, automatic type conversion either asserts that the token inherits some expected interface, or the token is converted into an instance that exposes that interface. In other words, coercion offsets the token to inherit properties from a different base class. From that point on, Java's dynamic binding mechanism takes over. There are subtle ways in which these these mechanisms can interfere with one another.

If an actor expects a more general interface than the type that was inferred for its input port, an automatic type conversion might be unnecessary, or even improper. Consider the example in Figure 3.7 (model originally created by Patricia Derler) where Display does not declare its input type, CurrentTime declares its output to be of type Double and CurrentMicrostep declares its output to be

of type Int. The default constraint forces the input of Display up to Double during type inference, which at run-time causes the output of CurrentMicrostep to be converted from Int to Double before it is received by the Display. The Display just uses the toString() method to print a String representation of incoming tokens. The toString() method is part of the Token interface and therefore defined for both IntToken and DoubleToken. The automatic conversion of from IntToken to DoubleToken however causes the tokens sent by CurrentMicrostep to be displayed as `1.0` instead of `1`.

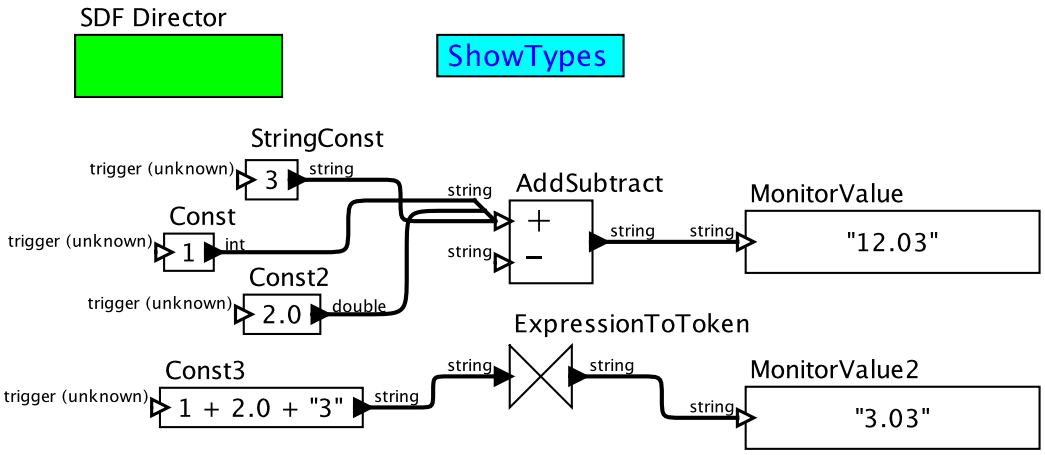


Figure 3.6: Example of premature coercion.[1]

Since coercion and dynamic dispatch are not commutative, it is not always clear which of the two should happen first. If coercion always has precedence, there is the risk of it affecting the behavior of actors in unexpected ways. Consider the following expression: `1 + 2.0 + "3"`. Both in the Ptolemy II expression language as in Java, evaluating this expression yields the result: `"3.03"`. However, as shown in Figure 3.6 we see that the polymorphic actor AddSubtract yields an entirely different result upon adding the input tokens that corresponds with the same expression. The problem is that the default type constraints force the input of AddSubtract up to String, which causes all inputs on the Add port to be automatically converted into a StringToken before being added. If the coercion had not taken place, dynamic dispatch *could* have picked the right add() method, added up the two numbers and then concatenated it with a the string. However, there is no way to enforce a particular ordering to addition of the inputs provided to AddSubtract. Unlike arithmetic addition, string concatenation is nonassociative. This means that even if tokens are not improperly coerced, there is no way to infer from the given model whether the outcome will yield `"33.0"` or `3.03`. The latter is an inherent problem of the AddSubtract as it combines the associative $\sum$-operation and the nonassociative concatenation operation in the same actor.

It must be noted that AddSubtract is not a type safe actor. It does not impose an upper bound on the type of the inputs it can take, but not all types are actually supported; e.g. the subtract operation on a String will yield a run-time error. This is not a result of a token not being compatible with a statically inferred type, but a type not being able to guarantee correct execution. This information is static, but not captured in the subtyping relation. Therefore it can only be detected at run-time.

[1] The blue rectangle represents an attribute that if part of a model shows the types that are assigned to ports.

The problem of improper or premature coercion is limited to the specific case where actors that leverage Java's dynamic dispatch mechanism to implement inclusion polymorphism, and it only comes up when multiple outputs are connected to a single input. If it is up to the type inference process to assign a type to an actor's input port, then this type might correspond to an interface or to a concrete token. Only in the latter case it will cause an actual conversion of incoming tokens. One way to circumvent coercion and expose the original token instances to the implementation of the actor, is the following:

1. An actor must declare its input to be equal to the greatest type that it can handle.
2. (a) The input type must map to a token class that is indeed a common base class for all tokens it supports. If this is not the case, then coercion would be required to offset the instance to the base class that implements the expected interface. As shown in the prior examples, this can be problematic.

   (b) The conversion method of the input type must not do an actual conversion i.e., it merely asserts the subtype relation in terms of interface inheritance.

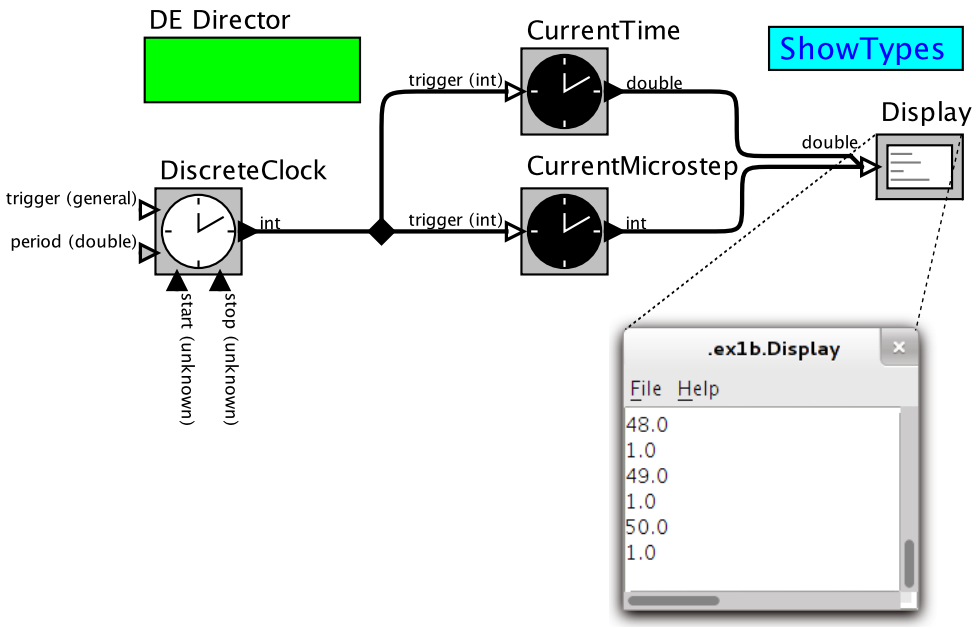


Figure 3.7: Example of improper coercion.

This approach works very well for sink actors i.e., actors that do not have any output ports. Assigning the type General to the Display actor in the model in Figure 3.7, solves the problem. As the display actor merely needs the method toString() to be present in the interface of a token, really any token is acceptable.

The suggested solution, to statically assign an input type that corresponds with the interface all acceptable tokens must implement, is not appropriate for actors of which an output type depends on

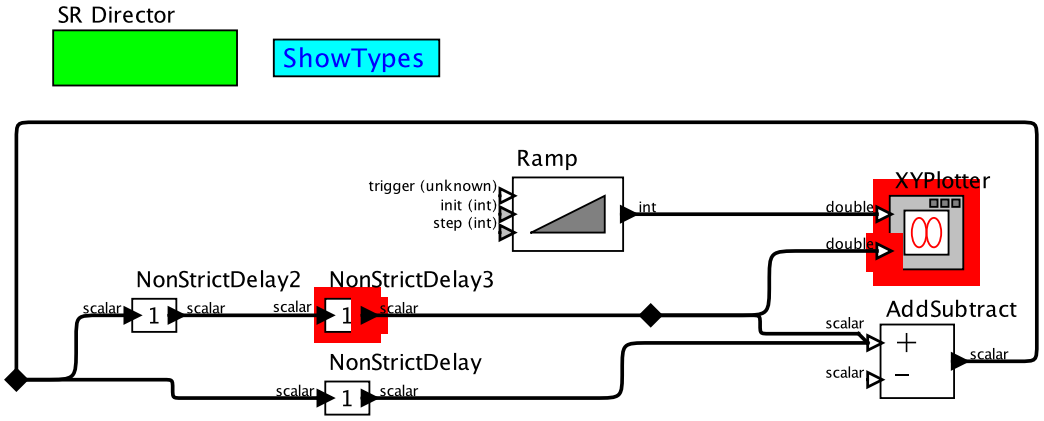


Figure 3.8: A type conflict as a result of declaring the input of AddSubtract to be Scalar.

an input type. If the output type is forced up to an interface type such as Scalar or General, then it becomes impossible to satisfy constraints further downstream that are more specific. The model in Figure 3.8 (model originally created by Chris Shaver) that should plot the Fibonacci sequence, has a downstream plotter that only accepts Double input, which causes a type conflict with the greater type Scalar, even though the actual input of AddSubtract is as small as an Int.

Another, perhaps obvious solution, would be to use different input ports for relations that carry tokens that should not be coerced into a single type before being processed by the actor. The drawback of this solution is that it becomes the responsibility of the user to decide which inputs should be grouped together and which should not. An advantage of this approach is that it enables the user to explicitly specify the order in which the inputs should be processed. This is useful for actors that implement an operation like concatenate, which is nonassociative. Currently, multiports observe the order in which relations were added. So in fact it is already possible to specify an order, but this is not a very visible property of the model. Alternatively, the user could daisy-chain a number of AddSubtract actors to prevent premature coercion or to enforce a specific evaluation order. This can be done with the existing AddSubtract actor, but again, this burdens the user with the task of finding the composition that implements the expected behavior.

3.6 Records, Arrays & Type safety

The way the subtyping order of records is defined, makes that the supertype of two records i.e., the least upper bound of both, is their intersection. Arrays are homogeneous, so they can only contain elements of a single type. Therefore, when an array is instantiated, its contents is typed as the least upper bound of the types of all elements. If an array contains records that do not all share the same label set, then not all labels will appear in the type of the array.

Consider for example the expression {{x = 2}, {y = pi}}, which declares an array containing two records. The type of the contained records are {x = int} and {y = double}, respectively. The supertype of both records is {}, so the array is typed ArrayType({}, 2). In the current implementation, the conversion method of RecordType does not actually do a projection, so the two records are inserted into the array in their original form, retaining all fields. Despite the uniform type of the array, its contents thus may very well be heterogeneous.

If we purely regard types, there is no reason to assume there is anything to be found in a record that is typed as being empty. The concept of lossless convertibility that defines the subtyping relation in Ptolemy II does not regard instances, it applies to types. One could of course always convert an IntToken into a DoubleToken and then do the inverse conversion and end up with the original value, but it is certainly not the case that any DoubleToken could be losslessly converted into an IntToken. The types are part of static analysis and cannot keep track of the conversion history of run-time instances. Therefore, converting {x = 2, y = pi} into {} *is* indeed lossless from a static typing perspective. It is not possible to losslessly convert back into {{x = 2}, {y = pi}} because there is no subtyping relation that allows it.

Not removing fields from a record as it is converted into a record type with fewer fields, can cause unexpected consequences that emerge from polymorphic actors operating on such record. The AddSubtract actor is defined to do an element-wise addition (or concatenation) for records. Fields that do not appear in the corresponding type, are rightfully not considered and will not appear in the output record. This is not established by the automatic conversion mechanism of the type system, but by the actor itself. There is nothing in the implementation of a RecordToken that prevents the actor from accessing undeclared fields, even after being coerced into a supertype. Consequently, the Display actor that simply invokes the toString() method, prints out *all* fields that are present in an incoming RecordToken, regardless of its conversion history or actual type.

Access to record elements that are not declared in the records type, is in principle unsafe. The type system cannot guarantee that undeclared elements are indeed present. Nevertheless, it could be useful to retain certain information in a record without having a guarantee about its presence. It would then be up to the actor to implement the necessary run-time checks to trap errors and deal with them gracefully; it would be better if the run-time type checker could take care of this. An attempt to access an element that is not present in a record's type is similar to a downcast operation in object-oriented programming. A downcast can result in a run-time error. The key insight is that undeclared elements need not be removed, but are ought to be hidden. An explicit downcast would then be required to increase the visibility of a record's fields. This combination of static and dynamic typing, offers the best compromise between type safety and flexibility; types are checked statically where possible, and dynamically if needed.

Chapter 4

Maximally permissive composition

This chapter addresses the problems that are formalized in the previous chapter in a more practical context. It describes the goal of *maximally permissive composition* and explains how by augmenting the type inference mechanism in Ptolemy II with the means for *backward type inference*, we can achieve this goal.

4.1 Composition & Composability

As opposed to layered system designs that reduce complexity "horizontally" by splitting the design into multiple layers, component-based approaches reduce complexity "vertically" by assembling strongly encapsulated design entities called *components* equipped with concise and rigorous interface specifications [Lee and Sangiovanni-vincentelli, 2011]. The idea of *composition* encompasses combining simple components to build more complicated ones. The property of *composability* is the extent to which components are suitable for composition in different environments. Composability facilitates component re-use and broadens an individual component's range of application.

Composability can be maximized by finding the weakest assumptions on the environment sufficient to establish the guarantees on a given component implementation [Lee and Sangiovanni-vincentelli, 2011]. In the context of the Internet of Things (IoT), where components are characterized by their ability to dynamically engage in interaction, composability is of prime importance. Components must be able to function in dynamically established configurations, meaning they must be designed without full knowledge of the shape these configurations may take. Incorporating too much context in the design of a component results in reduced flexibility and compatibility, leading to "stovepipe" solutions limited to isolated vertical conduit.

The behavioral aspects of component composition i.e., the interaction semantics, or *models of computation* are discussed extensively in [Janneck, 2003]. In this work, we merely focus on the typing aspects of composition.

4.1.1 Actors in Ptolemy II

As discussed in Section 2.1.2, Ptolemy II components come in the form of actors[1]. Actors can receive tokens through input ports and sent tokens through output ports. Each input and output

[1]Note that throughout this text the words "component" and "actor" are used interchangeably.

port is associated with a type, and all tokens passed through a port must be compatible with its type, or otherwise a runtime type checking error is thrown. An actor can impose constraints on the types of its ports which limits the types it will accept. The input of an actor is subject to the local requirements of that actor, whereas the output, if left unconstrained by the actor itself, is subject to requirements imposed by downstream actors or the system as a whole.

In terms of types, particularly in the context of subtyping, the requirements with regard to an actor's input are twofold. Firstly, the data needs to be sufficiently specific so that the actor can operate on it in a meaningful way. Second, the data must be complete, meaning that if the data is structured, all elements are present. In Ptolemy II, the former corresponds to depth subtyping and the latter is expressed by width subtyping. In fact, since the width subtyping relation of records is such that a record with more fields is a subtype of a record with fewer fields, the first requirement is a generalization of the second. In both cases, the input is required to be *minimally specific*.

If an input must be minimally specific, then the same holds for the output becomes that input. In the case of subtyping, an instance can always be safely upcasted, and in the case of automatic type conversion, this can be done losslessly. Therefore, in order to obtain actors that allow for *maximally permissive composition*, inputs need to be typed as general as possible and outputs need to be typed as specific as possible. This is not a new idea. In fact, it is very much central to the design of the Ptolemy II type system:

> "Note that even with the convenience provided by the type conversion, actors should still declare the receiving types to be the most general that they can handle and the sending types to be the most specific that includes all tokens they will send. This maximizes their applications." [Xiong, 2002]

The underlying assumption for this to work, is that an actor has prior knowledge about the types of tokens it will send. This is not always a reasonable assumption. Any actor that parses arbitrary data and turns it into a (typed) token, is unable to statically declare or infer a fitting type. Actors like these are very similar to the eval function that is found in some general purpose scripting and programming languages. Not surprisingly, most of these languages are dynamically typed. The question is how to incorporate eval-like actors in a statically typed environment, without breaking type safety and maintaining maximum flexibility.

4.2 How to type untyped data

The most obvious way to type untyped data would be; dynamically. A dynamic type system accommodates untyped data as it interprets values at runtime and assigns them a type accordingly. The problem is that Ptolemy II is *statically* typed, which yields many advantages e.g., early detection of programming mistakes, better coding documentation in the form of type signatures, and more opportunities for optimization. However, static type systems are rigid and lack the flexibility to interact with systems that change unpredictably. Nevertheless, it is desirable to be able to have models interact with sources of untyped data from a file, from the Web, or through interactive components that gather data from the physical world. An actor that mediates such interaction might e.g. take as input a String and outputs whatever it parses out of that String. Whatever that might be, is unknown until the data is actually read, which is supposed to happen when the model executes. But in order to execute the model, the output port of the actor must be typed. This is a chicken-and-egg problem.

Due to the parsing/evaluation operation, the input and output types are completely unrelated, meaning there is no way to infer the output type prior to execution. Deriving a type signature based on the structure of the used data is certainly possible, even in an automated fashion if the actor is able to inspect the data prior to execution, but a caveat of this approach is that the data may be subject to change during the course of execution. This means that a type description might end up to be overspecified, which can cause runtime type checking to fail, making the system needlessly brittle. Ideally, the output should be typed such that if the data adheres to the assigned type, then the model is guaranteed to function correctly. Arguably, this type is maximally permissive if it is no more specific than need be. On the other hand, if the type is too general, type conflicts can occur as downstream actors might enforce stricter requirements on their inputs. The question then is, exactly how specific does the type need to be.

The answer to this question was already hinted at in the previous section. The type of an output port needs to be specific enough to be compatible with downstream input requirements. In other words, if it is unknown what data will be produced, it makes sense to set its type in terms of how the data will be used. It is not particularly straightforward for the user to deduce this type, so ideally this type would be automatically inferred. In the next few sections we investigate the possibilities for leveraging the existing type inference mechanism to deduce maximally permissive types for unknown outputs.

If we address the problem of unknown output types in terms of underdetermined type variables, the notion of which is explained in Section 3.4.3, we observe that in order to get beyond a least fixed point with $\perp$ assignments in it, more constraints must be added. The targeted solution is a fixed point, slightly higher than the original one, which is consistent and has no $\perp$ assignments in it. In addition to being consistent, the assignments for the previously underdetermined type variables is desired to be maximally permissive i.e., no more specific than required by the actors that read the tokens emitted from the ports associated with those type variables.

4.3 Backward type inference

In principle, type inference has no directionality. The constraint solving problem that underlies the type inference mechanism in Ptolemy II is discussed in fine detail in the previous chapter. As was mentioned here, we can recognize two different kinds of type constraints; propagation constraints and limitation constraints. Propagation constraints are the constraints that drive the type inference process, whereas the limitation constraints provide the acceptation criterion for an inferred solution. The notion of directionality comes from the topology of the model as data flows through a model from output ports to input ports between actors, and from input to output port within actors.

The conventional default constraints that are set up along relations and between otherwise unconstrained input and output ports within actors, facilitate *forward propagation*. The forward propagation constraints between actors are of the form $\alpha \leq \tau_{input}$, and within actors they look like $\alpha \leq \tau_{output}$. Forward propagation allows for actors to infer their input type based on the output of upstream actors. Now if we want to do the opposite and infer the type of an output port based on the types of downstream actors' input, what we need is *backward propagation*.

A backward type constraint *between* actors would look like:

$$\alpha \leq \tau_{output} \tag{4.1}$$

The Rehof and Mogensen algorithm iteratively unifies the right-hand variable with the left-hand term, which could be a constant, a monotonic function, or another type variable. As long as the left-hand term eventually yields anything other than $\perp$, the (previously underdetermined) type variable τ_{output} will eventually be pushed up to something higher than $\perp$. Similarly, a backward constraint *within* an actor, is of the form:

$$\alpha \leq \tau_{input} \tag{4.2}$$

Again, the left-hand term is responsible for pushing up the type assignment of τ_{input}. Note that these are the exact same inequalities as used for forward propagation, only their direction is inverted. It is clear that these constraints can be added to influence underdetermined variables, but it is not clear what α should be in order to obtain a valid and consistent, yet maximally permissive solution. This will be investigated in the remainder of this chapter. First we discuss the type constraints that are imposed by the topology of a model, the type constraints between actors. After that, we discuss the type constraints that are imposed by actors themselves.

4.4 Type constraints between actors

The default *forward* type constraints $\tau_{output} \leq \tau_{input}$ that are imposed by the relations between actors, are preserved. These constraints guarantee lossless type convertibility of tokens between two connected ports. They warrant that if a token is compatible with the type of the output port, then it is guaranteed be compatible with the input port on the other end of the relation as well. Therefore, at run-time, right before an actor is about to send a token, the token is checked for type compatibility with the associated output port. The actual conversion is done on the receiving end. This makes sense, because an output port can be connected to many input ports, each of which can have different types.

The *backward* type constraints discussed in this section are additional. These constraints are intended to raise the types of output ports based on the types of input ports they are connected to. This preserves this property that run-time type checking errors are trapped as early as possible, and it does not affect the time at which run-time type conversion takes place. Actual Ptolemy II models are used to provide examples and point out corner cases.

4.4.1 Flat relations

In Ptolemy II it is possible to build multiple levels of hierarchy in a model using composite actors each of which can contain yet another model. The flat relations discussed here, are relations between actors within the same level of hierarchy.

One-to-One

Consider the example model in Figure 4.1 that has a JSONToToken and ExpressionToToken actor in it. Both these actors read a StringToken as input and produce a token that corresponds to an expression encoded in the input string. The input strings are supplied to the model as constant, but not evaluated until the model is executed. If instead of a StringConst actor a FileReader was used, the string would have been retrieved from where ever the FileOrURL-parameter had pointed to.

The XYPlotter declares its inputs to be of type ArrayType(Double), so any output connected to those inputs must be less then or equal to ArrayType(Double), which is exactly what the default forward type constraint expresses. However, this constraint is trivially satisfied by the initial $\bot$ assignment to the output ports of JSONToToken and ExpressionToToken. The maximally permissive assignment would be to have those output ports be typed equal to ArrayType(Double), which can be achieved by inserting the following backward type constraints along each relation:

$$\tau_{outputSink} \leq \tau_{output} \tag{4.3}$$

In this inequality τ_{output} corresponds to the type of an output port and $\tau_{outputSink}$ is the type associated with a port connected to that output port.

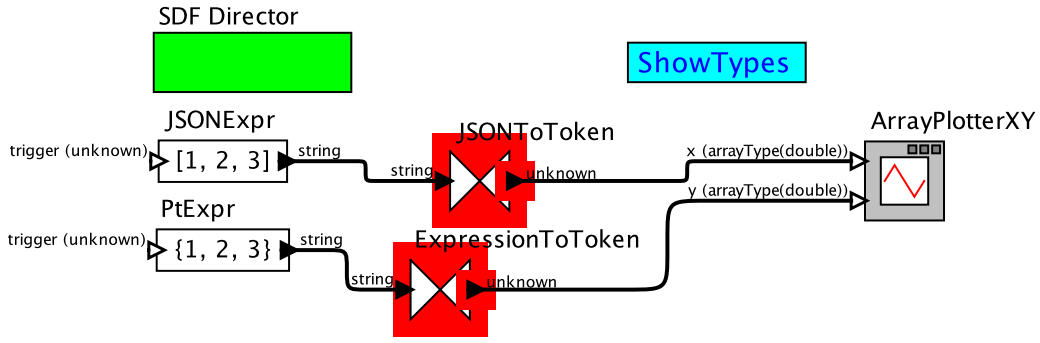


Figure 4.1: Unsuccessful type resolution due to underdetermined output ports.

In particular, the following backward type constraints would be added:

- JSONToToken.input $\leq$ JSONExpr.output
- ExpressionToToken.input $\leq$ PtExpr.output
- ArrayPlotterXY.x $\leq$ JSONToToken.output
- ArrayPlotterXY.y $\leq$ ExpressionToToken.output

In combination with the default forward type constraints that are already in place, what happens is that the types of connected output and input ports are unified. After all, because the subtype ordering is antisymmetric, if $\alpha \leq \beta$ and $\beta \leq \alpha$ then $\alpha = \beta$. The first two backward constraints do not affect any type assignments, but the last two constraints force the previously underdetermined outputs to be of the same type as the input ports they are connected to. The result is shown in Figure 4.2.

One-to-Many

If the output port of an actor is not connected to a single input port, but multiple, then using the simple backward constraint that unifies outputs and inputs, may cause type conflicts. Two backward constraints $\beta \leq \alpha$ and $\gamma \leq \alpha$ constitute $\beta \sqcup \gamma \leq \alpha$, which means that if $\beta < \gamma$ or $\gamma < \beta$, then α can no longer satisfy one of the forward constraints which require that $\alpha \leq \beta$ and $\alpha \leq \gamma$. Instead of expressing the type of an output port in terms of the least upper bound of the types of its destination ports, we should express it in terms of a *lower bound*, because a lower bound is compatible with the

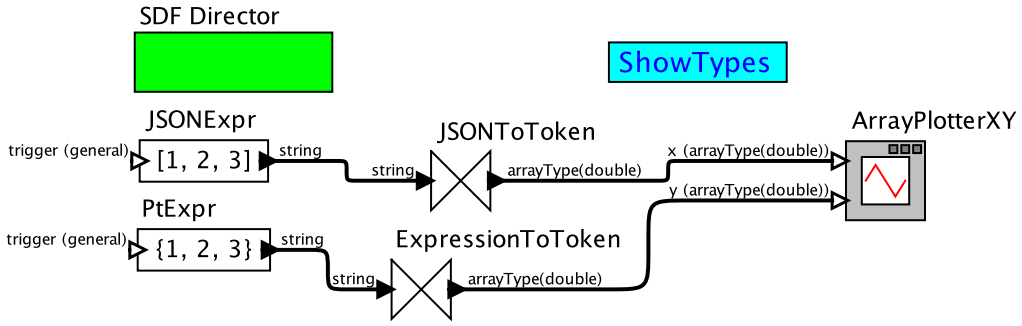


Figure 4.2: Backward inferred types for previously underdetermined output ports.

forward constraints. Because we want the type to be maximally permissive, we would like the type to be equal to the *greatest* lower bound of the types of the output destinations, which is expressed by the following constraint:

$$\sqcap \vec{\mathcal{T}}_{outputSinks} \leq \tau_{output} \tag{4.4}$$

Note that this expression generalizes the inequality given in Equation 4.3 that was used to backward infer types over one-to-one relations; the greatest lower bound of a single argument is just that argument. The example model in Figure 4.3 shows the result of type inference using the backward type constraints that make use the greatest lower bound. The output type of the JSONToToken actor gets inferred successfully even though the input ports of the MonitorValue actors have declared two different, incomparable types. The backward inferred type, ArrayType(int) is the greatest lower bound of the two destination ports, therefore it is compatible with the inputs of both downstream actors.

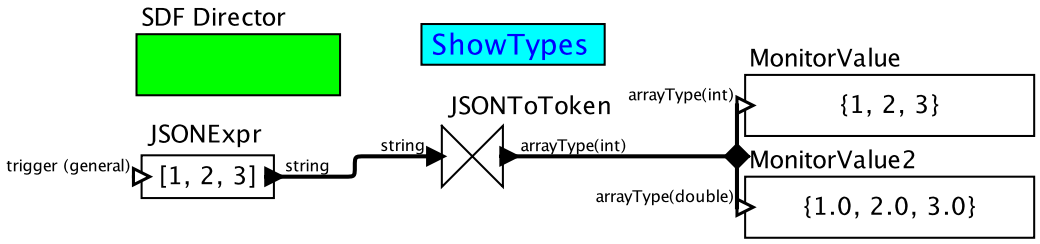


Figure 4.3: Backward inferred output type over a one-to-many relation.

Many-to-One

When two outputs merge into a single input, the only reasonable thing to do is backward infer the same type for both output ports. Again, this can be achieved using the same type constraint presented in Equation 4.4. In the example in Figure 4.4 the JSONToToken and ExpressionToToken outputs are fed into a single Discard actor. To keep things simple, the Discard actor declares its input to be of type arrayType(complex), which gets backward propagated using the backward type constraints:

- Discard.input $\leq$ JSONToToken.output
- Discard.input $\leq$ ExpressionToToken.output

The examples used so far are oversimplified. One could of course imagine much more complex networks of actors that impose requirements on the data that is produced by upstream sources. For these requirements then to be back propagated all the way up to the untyped outputs of these sources, a bit more is needed than just the backward type constraints along flat relations. Next, we discuss the type constraints for backward propagation along hierarchical relations.

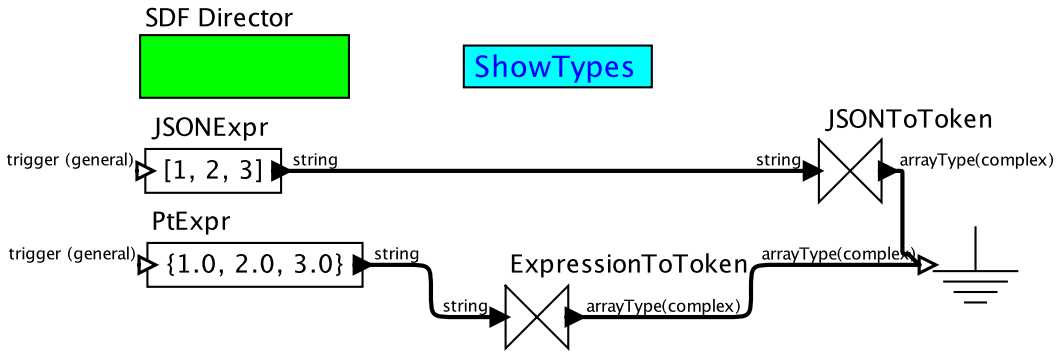


Figure 4.4: Backward inferred output type over a many-to-one relation.

4.4.2 Hierarchical relations

A *composite* actor is a container for a model, that itself can be contained in yet another model. Using composite actors, multiple levels of hierarchy can be constructed. There is always a top level composite that contains the whole model including all of its submodels. Unlike flat relations between actors that always connect outputs to inputs, the inputs of a composite actor can be connected to the inputs of its contained actors. In order to backward infer the type of the input port of a composite actor, the following constraint must be added:

$$\sqcap \vec{\mathcal{T}}_{inputSinks} \leq \tau_{input} \tag{4.5}$$

This is the same constraint as given in Equation 4.4, only this time it is set up between the input of a composite and the inputs of its internally connected contained actors. Note that the relation between the outputs of internal actors and the output of the composite, are already captured by Equation 4.4

TypedCompositeActor

The base class for composite actors is TypedCompositeActor and it is responsible for generating all type constraints imposed by the relations between its contained actors. In summary, given the embedded actors and the set R of relations between them, and the set P of relations between them and the composite itself, the following set of *forward* type constraints are generated:

- $\forall\ (input, sink) \in P\ [\alpha_{input} \leq \alpha_{sink}]$
- $\forall\ (output, sink) \in P \cup R\ [\alpha_{output} \leq \alpha_{sink}]$

, where α can either be a type variable τ, or constant c. These constraints can thus be either propagation of limitation constraints. The propagation constraints with the type variable on the right-hand side are needed to promote the type of the sink port such that it is compatible with the sources that connect to it. The limitation constraints assert that sources shall never have a type greater than the sinks they are connected to.

In order to also do backward type inference, the following set of *backward* type constraints must be generated:

- $\forall\ input \in \{input \mid (input, sink) \in P\}\ [\sqcap \{\alpha_{sink} \mid (input, sink) \in P\} \leq \tau_{input}]$
- $\forall\ output \in \{output \mid (output, sink) \in P \cup R\}\ [\sqcap \{\alpha_{sink} \mid (output, sink) \in P \cup R\} \leq \tau_{output}]$

Again, α can either be a type variable τ, or constant c. It is important to point out that these inequalities are exclusively propagation constraints i.e., they have never a constant but always a type variable on the right-hand side. If this were not the case, then chains of dependent type variables would not be broken up by constants, which almost immediately leads to unnecessary type conflicts. In essence, this would defeat the whole purpose of subtyping, as it would no longer allow for a source port to be assigned a subtype of the type assigned to a connected sink port. The sole purpose of the backward type constraints is to propagate type information upstream, not to enforce inverse type compatibility.

Note that for composite actors that are *opaque* i.e., they are run by their own director, the ports of the composite have a separate internal and external representation, therefore these ports can be typed differently internally and externally. This does not affect the proposed solution, it only means that the type variables associated with these ports exist both in P and R.

4.5 Type constraints within actors

The idea of back propagating type information through the topology of a model by means of backward type constraints, extends to the inner structure of *atomic* (i.e., non-composite) actors as well. For most of these actors, the types of their input and output ports are tightly interdependent. These dependencies can be leveraged to provide upstream actors with the information required to infer a maximally permissive output type. If backward propagation is done both along relations and within atomic actors, type information can travel upstream multiple hops. This way, it is still possible to reach a satisfying solution for the type inference problem, even when the type variables associated with two interconnected ports both start out as $\bot$.

Default type constraints

Similar to the type constraints imposed by relations between actors, the inputs and outputs of an actor that are left completely unconstrained, are captured in a *default* constraint $\tau_{input} \leq \tau_{output}$ that ensures type compatibility of tokens passed from inputs to outputs, and to allow type information to propagate forward. For backward propagation we add the opposite constraint $\tau_{output} \leq \tau_{input}$ for each pair of unconstrained inputs and outputs, which together with the forward type constraint unifies their types; $\tau_{input} = \tau_{output}$.

However, actors often require much more sophisticated type constraints between inputs and outputs. This is because the constraints may depend on particular data transformations that are done inside of the actor. Of course, this also holds for the internal type constraints that need to be generated in order to do backward type inference. In this section we discuss the backward type constraints that are involved with several popular actors in Ptolemy II.

4.5.1 ArrayToSequence

This actor reads an array at the input and writes the array elements as a sequence to the output. This is expressed in terms of types using the following inequality:

$$elementType(\tau_{input}) \leq \tau_{ouput} \tag{4.6}$$

This constraint restricts the ouput to be greater than or equal to the type that the monotonic function $elementType()$ returns, given the type of the input port as an argument. The function $elementType()$, presented in Equation 4.8, simply extracts the element type of the input array. However, if this type cannot be inferred from upstream types, it can be backward inferred by adding the following constraint:

$$arrayTypeFunc(\tau_{output}, length) \leq \tau_{input} \tag{4.7}$$

The monotonic function $arrayTypeFunc()$, shown in Equation 4.9 wraps its argument type into an ArrayType (represented by curly brackets). The length parameter is set as part of the ArrayType only if it has a positive value.

$$elementType(\{\tau\}) = \tau \tag{4.8}$$

$$arrayTypeFunc(\tau, length) = \begin{cases} \{\tau, length\} & \text{if } length > 0 \\ \{\tau\} & \text{otherwise} \end{cases} \tag{4.9}$$

Figure 4.5: Monotonic functions used in type constraints for arrays.

4.5.2 ArrayElement

According to a given index, the ArrayElement actor picks an element out of the array it receives as an input and emits the value on its output port. The index is either set through a parameter named index, or dynamically updated by means of a token received on the input port named index. The token that is used to set or update the parameter is declared to be of type Int (Equation 4.10). By declaring a type variable to be equal to a constant, that symbol is no longer considered a variable and is thus excluded from type inference.

Furthermore, Equation 4.11 constrains the input to be greater than or equal to an ArrayType($\perp$), and just like the ArrayToSequence actor, Equation 4.12 requires the output to be greater than or equal to the element type of the input array. The last inequality, Equation 4.13, is the backward type constraint, identical to the one used for ArrayToSequence.

$$Int = \tau_{index} \tag{4.10}$$
$$\{\bot\} \leq \tau_{input} \tag{4.11}$$
$$elementType(\tau_{input}) \leq \tau_{output} \tag{4.12}$$
$$arrayTypeFunc(\tau_{output}, length) \leq \tau_{input} \tag{4.13}$$

Figure 4.6: Type constraints internal to ArrayElement.

4.5.3 RecordDisassembler

An interesting and slightly more complex example, is the RecordDisassembler actor, which disassembles a RecordToken into multiple outputs. This actor is used by instantiating it, and declaring a number of output ports of which the names correspond to a label in the RecordToken that is expected on the input port. No types need to be declared for the output ports, but it is possible. Undeclared types can be inferred. Essentially, the input record has to contain a field with a corresponding label for every output port, and the type of each output port has to be compatible with the type of that field. This is expressed using the three type constraints in Figure 4.7.

$$constructRecordType(connectedOutputs) \leq \tau_{input} \tag{4.14}$$
$$\tau_{input} \leq \{p = \top \mid p \in connectedOutputs\} \tag{4.15}$$
$$\forall\, p \in connectedOutputs\ [extractRecordType(\tau_{input}, p) \leq \tau_p] \tag{4.16}$$

Figure 4.7: Type constraints internal to RecordDisassembler.

The first type constraint, shown in Equation 4.14, involves the monotonic function *constructRecordType*(P) which constructs a RecordType that has corresponding fields for every port in a list provided by the argument P. The labels match the names of the ports on the list, and the types are equal to the currently inferred or declared types of the respective ports. Only the *connected* output ports are passed on to the function, as others pose no requirement on the contents of the input record.

The first type constraint requires the input type to be a RecordType of which each field is greater than or equal to the type of that field in the RecordType returned by *constructRecordType*(P). However, since width-subtyping for records is defined such that a record with fewer types is greater than a record with more types, this single constraint could be trivially satisfied using an empty record. A second constraint is set up to preclude this trivial solution and ensure that every output port is indeed represented by a field in the input record.

The second type constraint, given in Equation 4.15, sets the input type to be less than or equal to a constant RecordType that contains a label for every output but sets the type for every field to $\top$. Since any type is less than or equal to $\top$, this constraint does not impose any restriction in terms of depth-subtyping. Instead, by means of width-subtyping, it forces the input type to be a RecordType that minimally contains a corresponding label for every connected output port.

$$constructRecordType(P) = \{p = \tau_p \mid (p, \tau_p) \in P\} \tag{4.17}$$

$$extractFieldType(\tau_{Record}, name) = \begin{cases} \bot & \text{if } \tau_{Record} = \bot \\ \tau_{name} & \text{if } \tau_{Record} = \{l : \tau_l \mid l \in L\} \wedge name \in L \\ \top & \text{otherwise} \end{cases} \tag{4.18}$$

Figure 4.8: Monotonic functions used in type constraints for records.

The third type constraint, presented in Equation 4.16, is set up for each output port, and involves a monotonic function $extractFieldType(\tau_{input}, name)$ that extracts the type from the field inside the input record of which the label corresponds to the given port name. The argument τ_{input} represents the type variable associated with the input port of the RecordDisassembler. The name parameter is used to look up the type of the corresponding field inside the RecordType. If a matching label is found, the function returns the associated type, or $\top$ otherwise. The latter requires some explanation. Intuitively, it does not make sense for the function to return $\top$ if no matching label is found. One would expect it to return $\bot$ instead. Since there is no type variable or constant to draw a type from, $\bot$ seems appropriate.

The reason that $extractFieldType(\tau_{input}, name)$ has to return $\top$ if no matching label is found, is to preserve monotonicity. This has everything to do with the width-subtyping relation of structured types. Monotonicity requires that:

$$\tau_1 \leq \tau_2 \Rightarrow extractFieldType(\tau_1, name) \leq extractFieldType(\tau_2, name) \tag{4.19}$$

Assume that $name = b$, $\tau_1 = \{a = Boolean, b : Int\}$, and $\tau_2 = \{a = Boolean\}$. Then $\tau_1 \leq \tau_2$, but since the label b is not present in τ_2, $extractFieldType(\tau_2, name)$ ends up in the second case of its function description, where according to the monotonicity property it must return something that is greater than or equal to $extractFieldType(\tau_1, name)$. This requirement is always satisfied by returning $\top$. In the configuration in which $extractFieldType(\tau_{input}, name)$ is used, however, it will always find a matching label since the inequality in Equation 4.15 ensures that a matching label is present for each output port.

Whereas the last two type constraints are strictly required to assure type safety, the first type constraint serves no other purpose than to back propagate type information from the output ports to the input port. This *backward* type constraint it is responsible for pushing up the types of the fields of the input record to the same level as the types of the output ports during type resolution.

4.5.4 RecordAssembler

The RecordAssembler actor assembles the tokens it receives on its input ports into a record and sends it out on its output port. It is used in a similar way as RecordDisassembler; input ports can be added after instantiating the actor, and the names of the input ports must correspond to the labels associated with the values that are assembled into the output record.

Analogous to RecordDisassembler, the types of the input tokens must match the types of the fields in the record. Not only should the types of the inputs be compatible with (i.e., losslessly convertible into) the types of the fields in the output record, but in order for type information to

back propagate from the output to the inputs, they need to be equal. However, to construct a maximally permissive set of type annotations, there is no need for every input port to be represented by a label in the type of the output record. After all, downstream actors might not require a record with all provided fields; the minimally required label set will suffice to guarantee type safety.

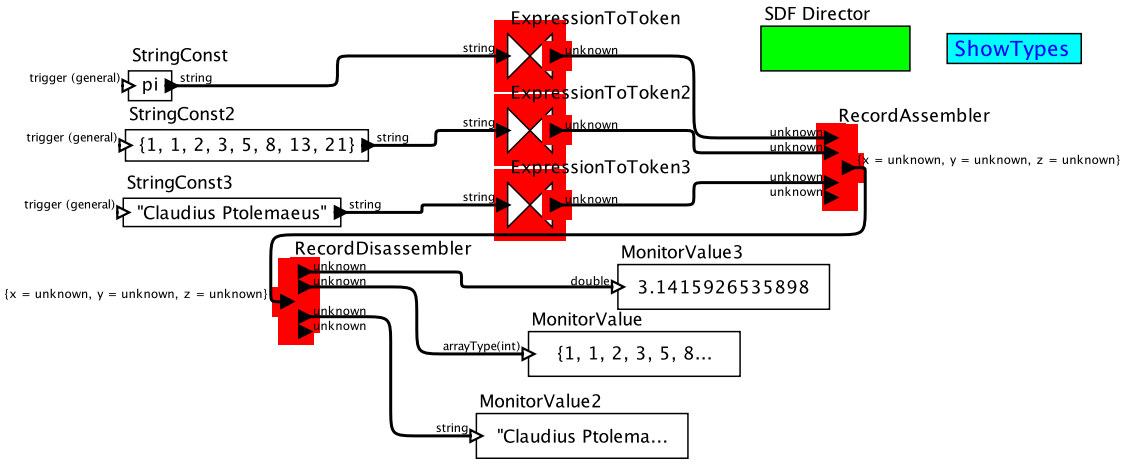


Figure 4.9: Record assemblage and disassemblage without backward type inference.

The type constraints internal to the RecordAssembler, listed in Figure 4.10, make use of the same monotonic functions as the RecordDisassembler that are shown in Figure 4.8.

$$constructRecordType(connectedInputs) \leq \tau_{output} \tag{4.20}$$

$$\forall p \in UndeclaredConnectedInputs \;\; extractFieldType(\tau_{output}, p) \leq \tau_p \tag{4.21}$$

Figure 4.10: Type constraints internal to RecordAssembler.

The first type constraint, given in Equation 4.20, requires the types of the input ports to be compatible with i.e., losslessly convertible into the types of the corresponding fields in the output record. Again, unconnected ports are ignored. This constraint can be viewed as a forward constraint, because during type inference it is responsible for pushing up the type of the output port based on the types of the input ports. This makes type information propagate in a forward direction.

The second type constraint, presented in Equation 4.21, does nothing to ensure type compatibility, but allows type information to propagate in the opposite direction; from the output port to toward the input ports. Together with the first type constraint, it forces the input types to be exactly equal to the types of the corresponding fields in the output record. Because the second constraint is intended to back propagate type information upstream, and not to assure type safety, this constraint is only set up for input ports that are connected, and do not already have a type declared.

The simple model in Figure 4.9 parses a number of expressions, assembles them into a record and then immediately disassembles the record en shows the obtained values. The MonitorValue actors declare a type for their inputs, but the rest of the types are not successfully inferred because ExpressionToToken is unable to determine a type for its output port. Figure 4.11 shows the same model, but now with the additional backward type constraints. This allows the types declared by the MonitorValue actors to back propagate all the way to the ExpressionToToken actors, thereby solving all type errors. Also note that the ports on the RecordAssembler and RecordDisassembler that are left unconnected, are not represented in the inferred record types.

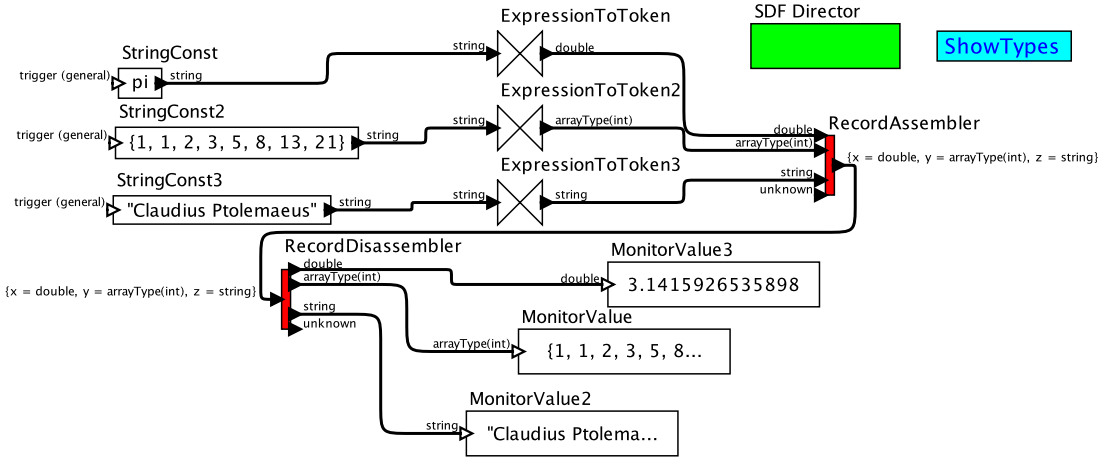


Figure 4.11: Type errors resolved using backward type inference.

4.6 Sink actors & Polymorphism

The actors discussed so far have both inputs and outputs. Backward constraints make the types of inputs to depend on the types of outputs, thereby letting type information propagate upstream during the type inference process. Sink actors do not have outputs. However, the type variable associated with their input port might very well be on the end of a chain of underdetermined type variables that need to be bounded from below. In that case, if the sink actor does not declare a type for its input, there is no constant to bound upstream type variables from below.

If in the context of backward type inference sink actors should the declare an input type, then the question is what type it should be. The discussion in Section 3.5 provides an answer to this question; we use the most general type the actor can handle, provided that:

1. the type maps to a token class that is a common base class for all the tokens it supports;
2. the conversion method of the type does not do an actual conversion.

What this establishes is that the input is limited to tokens that implement the interface associated with the declared type. This is perfectly reasonable for sink actors that rely on Java's subtyping mechanisms to implement polymorphism. In fact, all true sink actors that are currently in the

Ptolemy II actor library, qualify as such. Examples are: Display, Recorder, MonitorValue, Discard, XYPlotter, ArrayPlotter, and so forth. An actor like SetVariable might conceptually look like a sink, but it is type dependent on the variable that it is supposed to set. Therefore, it can impose a backward type constraint to back propagate that type.

Because it is desirable to be able to override the types with a manual type declaration, and also switch between modes where backward type inference is enabled or disabled, the current implementation demands that instead of a declaring the input type using the method setTypeEquals(), a type constraint of the form $c \leq \tau_{input}$ is used to force the input type up to c. This constraint is like a backward type constraint with a constant on the left-hand side instead of a type variable corresponding to a downstream port. Note that this constraint must be combined with the upper bound $\tau input \leq c$ to enforce that no type greater than c is acceptable.

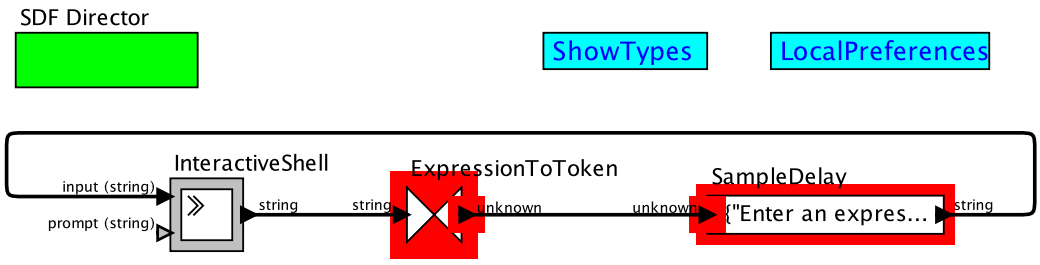


Figure 4.12: An expression evaluator model with underdetermined outputs.

Consider the example model in Figure 4.12 (model originally created by Edward A. Lee). The model has an InteractiveShell that has two input ports; one for specifying a prompt message and another to feed console output into. When the model runs, the InteractiveShell opens up a console and displays a message and a prompt. It captures input entered into the console, and then forwards it to an ExpressionToToken actor. The ExpressionToToken actor attempts to parse the expression it receives, and feeds the result into a SampleDelay, which routes it to back the InteractiveShell for display. The SampleDelay is used to break the dependency cycle in this SDF model, and in addition it provides an initial message to display on the console. The SampleDelay can infer its output type based on the expression given for its initialOutputs parameter, which in this case is a String.

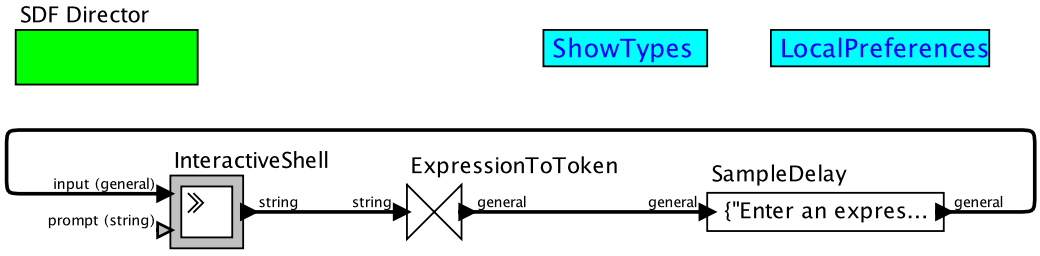


Figure 4.13: An expression evaluator model with backward inferred outputs.

The model has a feedback loop, but type-wise it is broken up in separate parts. The actors like ExpressionToToken that inspired the idea of backward type inference, have an important property in common with sink actors; the types of their inputs and outputs do not correlate. Similarly, the InteractiveShell is both a sink and a source, but there exist no type constraints between its inputs and outputs. The input is displayed on the console and the output is obtained from the user, so any type correlation, if it exists, would be accidental.

With backward type inference, the underdetermined output ports in the expression evaluator model can successfully resolve to a type greater than $\bot$. Aside the default backward constraints along relations, two things are required. First, the SampleDelay actor needs to impose the backward constraint $\tau_{output} \leq \tau_{input}$, which together with the existing constraint $\tau_{input} \leq \tau_{output}$, unifies the types of its ports. Second, in order to achieve a maximally permissive type assignment, the InteractiveShell must declare for its input the most general type it can handle. Because this actor simply displays all input as a String, it merely requires the method toString() to be present, which is the case for all subclasses of Token, therefore $\top$ the most general type InteractiveShell can handle. The type assignments obtained through backward type inference are shown in Figure 4.13.

The fact that in this example all formerly underdetermined type variables resolve to $\top$, might raise the question why the downstream InteractiveShell actor had to declare its input as $\top$, such that the type could then back propagate upstream and push up underdetermined type variables. It would be a lot easier to just have ExpressionToToken declare its output as $\top$ directly. In this particular situation, that approach would have worked, but if downstream inputs would have been constrained to a more specific type than $\top$, a type conflict would have occurred. An example of this is depicted in Figure 3.7. Simply declaring an actors' underdetermined outputs to be of type $\top$, severely limits composability. On the other hand, for sink actors (i.e., actors for which the types of their output ports, if present, do not depend on the types of their inputs), we can declare the input type to be a general as possible without limiting the composability of the actor.

4.6.1 AddSubtract

The AddSubtract actor, and its quirks, have been discussed extensively in Section 3.5. Having already pointed out a number of problems with this actor, together with the fact that it is possibly one of the most used actors in the Ptolemy II library, makes it an interesting case to investigate in the context of backward type inference.

It was already argued that this type of actor cannot statically declare its input type as general as possible, because the type of its output must be greater than or equal to this type, which is likely to cause unnecessary type conflicts further downstream. Because AddSubtract unifies the types of its inputs by taking their least upper bound, there is no need for AddSubtract to backward infer anything if at least one of the inputs can infer a type using forward propagation constraints. Consider the example model in Figure 4.14 (model originally created by Edward A. Lee), which is same the model as in the previous example, but now extended with an AddSubtract actor. Backward type inference is enabled, but the AddSubtract actor does not impose any backward type constraints.

Clearly, the model is well-typed, but a closer look tells that this is not a desirable set of type assignments at all. Whereas the ExpressionToToken actor can in principle parse *any* token out of an input string, and the AddSubtract actor is polymorphic, the backward inferred types do not allow

ExpressionToToken to output tokens that are strictly greater than Int. Entering the value `1.0` into the console does not yield the expected value `2.0`, but results in an error because Double > Int.

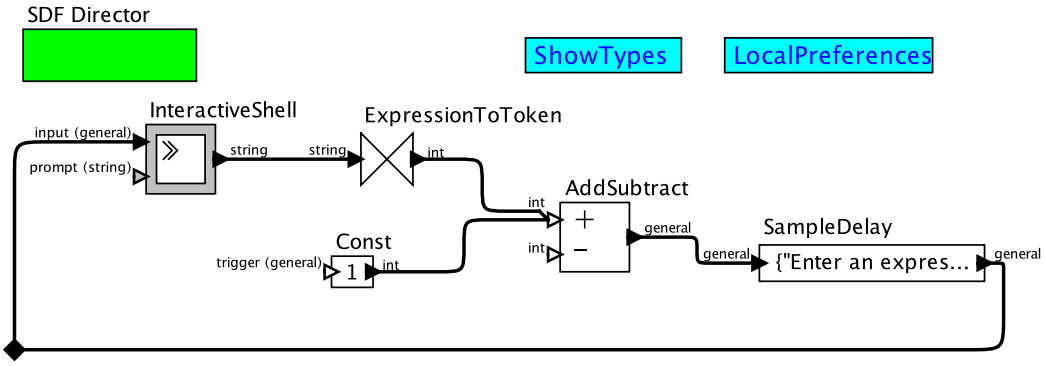


Figure 4.14: An expression evaluator model that is more constrained than necessary.

If we introduce the backward type constraint $\tau_{output} \leq \tau_{plus}$, this problem is resolved; the inferred types are no longer too strict. The result is shown in Figure 4.15. Any expression can be entered into the console, and as long as the AddSubtract actor supports the token, it will operate on it, or otherwise throw an exception. The latter is a peculiar property of AddSubtract; it makes use of the method Token.add() which simply returns an exception with a "not supported" message. Only subclasses of Token that override this method can give it a meaningful implementation. This actually defeats the purpose of static typing, which should establish that if a token is of a certain type, then run-time type errors are guaranteed not to occur. A summary of the type constraints internal to AddSubtract is given in Figure 4.16.

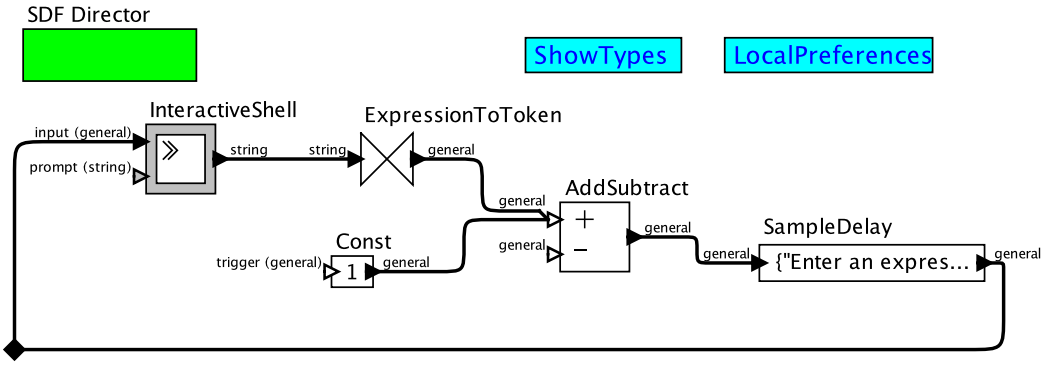


Figure 4.15: An expression evaluator model that is maximally permissive.

The fact that backward type inference gives this model a set of type assignments that let the model work as expected, can not only be attributed to the use of backward type constraints. Of course it also matters what the type is that gets back propagated. In this case, it is the InteractiveShell that declares its input to be $\top$, which bounds upstream type variables from below. But what would happen if a more specific type would propagate upstream?

$$\tau_{plus} = \tau_{minus} \tag{4.22}$$
$$\tau_{plus} \leq \tau_{output} \tag{4.23}$$
$$\tau_{output} \leq \tau_{plus} \tag{4.24}$$

Figure 4.16: Type constraints internal to AddSubtract.

Consider the example model in Figure 4.17 where the InteractiveShell declares its input to be String. Again, the model is well-typed and the type assignments are the least bit surprising. However, if we run the model, the exhibited behavior is rather unexpected.

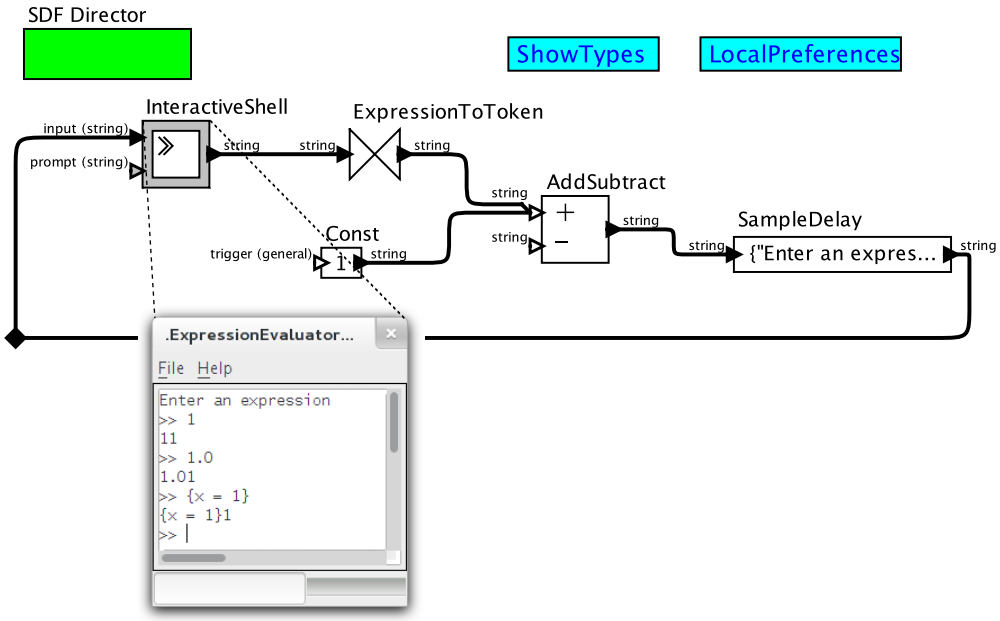


Figure 4.17: An expression evaluator model that shows interference between backward type inference and automatic type conversion.

Declaring the input of InteractiveShell to be of type String is perfectly reasonable, because after all, the received tokens will be displayed as a String in the console. The only expected difference in the behavior of the model, would be that a token that is incompatible with String, would no longer be accepted as input for InteractiveShell, but would trigger a runtime type checking exception instead. Back propagating the type String would only cause the conversion into StringToken to happen further upstream. The problem here, is that the conversion happens too far upstream. As a result, AddSubtract starts doing string concatenation instead of scalar addition. This issue can be explained as a problem with coercion, or a problem with overloading of the AddSubtract actor.

It may be considered impure to combine two entirely different "add" operations in a single actor (or as implementations of a single Java interface, for that matter), but one could argue that it is convenient. Whether the same always holds for automatic type conversion, is not that evident. In [Xiong, 2002] it was argued that coercions happen frequently in programming and therefore it would be convenient to have them automated, but the use of automatic type conversion might be more limited than it seems.

If an actor exhibits parametric polymorphism i.e., its implementation is entirely unaware of types, it has no need for type conversions. Other polymorphic actors, the ones that use Java's interface inheritance and dynamic dispatch mechanism to implement inclusion polymorphism, are not always helped by automatic type conversion either. In fact, in Section 3.5 it was demonstrated that automatic type conversion can interfere with dynamic dispatch which may lead to unexpected behavior.

Automatic type conversion may be needed when a port can resolve to a type that has subtypes in the lattice of which the token representations are not subclasses of the token class corresponding to the inferred type. If the actor relies on an interface that is not implemented by all subclasses of the token it expects on basis of the inferred type, then automatic coercion converts incoming tokens into objects that do offer that interface. In Section 3.5, this mechanism was described as coercive default inheritance. For actors that rely exclusively on a property that is shared by *all* tokens, coercion is superfluous. Examples of the latter are: Display which uses Object.toString(); and AddSubtract which uses Token.add() and Token.subtract().

For actors that do not benefit from automatic type conversion, it makes sense to disable it. If we augment the TypedIOPort class with the methods enableTypeCoercion() and disableTypeCoercion(), actors can individually specify the appropriate behavior for each of their ports. This would solve the problems with improper and premature coercion, including the issue illustrated with the model in Figure 4.17. However, in order to assure that no premature coercion takes place in a setting with backward type inference, it is required that *none* of the upstream actors coerce their inputs into a backward inferred type that is presumably more general than strictly necessary.

Chapter 5

Discussion & Conclusions

The Ptolemy II type system was extended with a type inference mode that assigns static, yet maximally permissive types, to dynamic data. The types for otherwise underdetermined outputs are backward inferred based on type constraints imposed by downstream actors. This is achieved using additional type constraints, leaving the original type resolution algorithm unchanged. The extension has no significant impact on the run-time of type resolution. Type errors are trapped at run-time as early as possible, directly at the source. This allows actors to invoke error handling strategies that improve robustness.

5.1 Using backward type inference

The work described in Chapter 4 has been incorporated in the Ptolemy II development tree and will be available in the next major release. The source code and the nightly build are available from:

http://chess.eecs.berkeley.edu/ptexternal/

Backward type inference is not enabled by default. It can be enabled or disabled on the granularity of an individual composite actor using the parameter enabledBackwardTypeInference. The method isBackwardTypeInferenceEnabled() which is available in both TypedAtomicActor and TypedCompositeActor looks for this parameter by increasing scope until the parameter is found, or the top level composite is reached. The top level composite always has the parameter, and it can be set through the context menu (right click, or CTRL-click on the background of the model) by selecting Customize → Configure, after which the dialog in Figure 5.1 should appear.

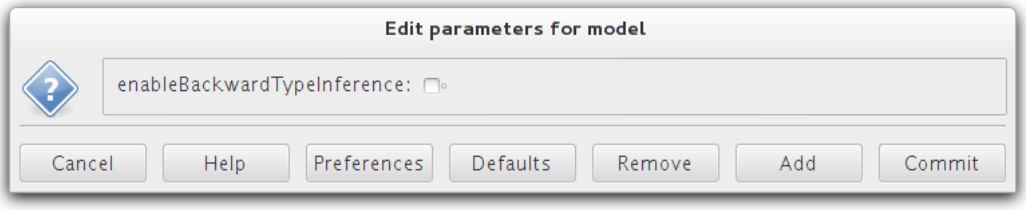


Figure 5.1: Dialog for enabling backward type inference.

Individual actors can be suited with backward type constraints by overriding the method customTypeConstraints() that is inherited from TypedAtomicActor. The method isBackwardTypeInferenceEnabled() can be used to condition the activation of type constraints.

5.2 Limitations & Trade-offs

Backward type inference provides a solution to the problem of handling dynamic data in a statically typed environment. It allows actors that mediate access to untyped data to omit type declarations from their output ports and have maximally permissive types backward inferred based on type constraints that are imposed by downstream actors. However, the solution involves trade-offs and has limitations.

The type resolution algorithm in Ptolemy II is aimed at finding the *most specific* consistent type assignments, which is a valid goal from an implementation cost perspective, particularly when wanting to do code generation. This is at odds with the idea of maximally permissive type assignments, which are supposed to be *least specific*. This is clearly a trade-off between flexibility and performance.

It has been established that the most specific solution coincides with the least fixed point (LFP) of the system of inequality constraints imposed by the configuration of actors. The maximally permissive solution lies beyond the LFP, but it must not be confused with the greatest fixed point (GFP). We use backward type inference instead of targeting the GFP (which if consistent, is arguably maximally permissive), because the GFP is a lot less likely to be consistent. Since the GFP is more general than the LFP, it is less likely to satisfy the limitation constraints. This is illustrated in Figure 3.2.

Many examples were provided to explain how backward type inference establishes maximally permissive types for output ports that otherwise would have been left underdetermined. However, there is one specific case for which the current implementation does not provide a solution. This case is illustrated by Figure 5.2. In this very simple example, a String is meant to be parsed by an ExpressionToToken after which the resulting token is to be sent to two different actors. The ArrayPlotter declares an input type, arrayType(double). The Expression actor declares no types, nor can it infer any because the type of its expression depends on the PortParameter input, the type of which is underdetermined.

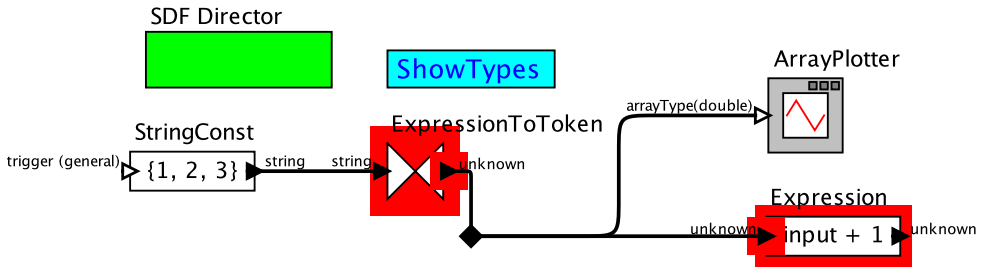


Figure 5.2: Problem with backward type inference over one-to-many relations.

One could argue that the appropriate type assignment for ExpressionToToken.output and Expression.input (and Expression.ouput, for that matter), would be ArrayType(Double). After all, this is the most general type that both ArrayPlotter and Expression would accept. The problem is, the backward constraint ArrayPlotter.input $\sqcap$ Expression.input $\leq$ ExpressionToToken.output yields $\bot$ because Expression.input $= \bot$. In general, the greatest lower bound of any set of type variables of which one or more are equal to $\bot$, yields $\bot$. This means that in any one-to-many configuration where during the type inference process one of the downstream inputs remains underdetermined, backward type inference fails.

To overcome this problem, a more sophisticated approach is required. Presumably, the key to a solution lies within the graph structure of model. In Chapter 4 it was explained that type constraints constitute dependencies between type variables that, directly or by transitivity, depend on type constants. Backward type inference involves adding dependency relations between type variables and replacing type variables with constants. The former is achieved using backward type constraints, and the latter is done by having generic sink actors statically declare the type of their inputs.

A promising idea that needs further investigation is a two stage approach to backward type inference. In the first stage, only the backward type constraints *within* actors are enabled. If after a first run of the Rehof and Mogensen algorithm, the obtained least fixed point yields a solution that is consistent and has no $\bot$ assignments in it, the solution is accepted. If the solution if inconsistent, the solution is rejected. However, if the solution is consistent but does have $\bot$ assignments in it, backward type constraints between actors could be added selectively to build the necessary dependencies to raise underdetermined type variables to a higher type in a second run.

This approach has two potential advantages. Firstly, the graph structure of the model can be taken in consideration to formulate a more expressive backward type constraint. E.g., it would be possible to be specific about which type variables to include in a greatest lower bound, thereby avoiding the problem illustrated in Figure 5.2. Secondly, because backward type constraints are not enabled along every relation, but only along the ones that provide the necessary dependencies to connect an underdetermined variable with a constant, the obtained solution is closer to the original goal of finding the most specific set of type assignments. This yields a solution that is most specific if possible and maximally permissive if needed.

5.3 Other problems

Two independent mechanism are used to achieve polymorphism in Ptolemy II; automatic type conversion and dynamic dispatch. These operations are not commutative, so it matters in what order they occur. Some of the types in the type lattice represent Java interfaces rather than instantiable types, which means their conversion method does not actually convert. Therefore, different type assignments may cause unexpected changes in behavior. Backward type inference exacerbates these problems because it causes coercions to happen further upstream, which flips around the order in which coercion and dispatch takes place.

For parametric polymorphic actors that are entirely unaware of types, automatic type conversion has no purpose. Actors that use Java's interface inheritance and dynamic dispatch mechanism to implement inclusion polymorphism, need to have the original instances of tokens exposed to their implementation, but type conversions discard those. For actors that rely exclusively on a property

that is shared by *all* tokens, it makes sense to disable automatic conversions.

The need for automatic type conversion emerges from the fact that subtype relations in the type lattice are not reflected by the class hierarchy of the corresponding token implementations. For nominal subtypes, coercion would become entirely obsolete if every subtype would represent a Java interface, and a subtype relation in the type lattice would imply an inheritance relation between the associated Java interfaces (i.e., $\tau_1 \leq \tau_2 \Rightarrow I_1 \subseteq I_2$). For structured types, automatic type conversion could be a means to make access to record fields type safe, but the current implementation leaves this opportunity unused, as it omits an actual conversion. The proposed solution is to change the conversion of records such that it hides fields that are not present the type. Hidden fields would still be accessible, but only after an explicit conversion.

5.4 Future work

There are quite a few ideas, recommendations, and questions expressed in this thesis that are worth further exploration. To start with, the two stage variant of the Rehof and Mogensen algorithm sketched out in Section 5.2 is already a prelude to further improvements of the backward type inference mechanism.

Another issue is that in order to make backward type inference generally usable, is that many actors still need to be fitted with backward type constraints such that they back propagate type information upstream. These actors are characterized by having custom type constraints to facilitate forward propagation and downstream compatibility as well.

Error handling strategies were mentioned, but not addressed in great detail. Backward inferred maximally permissive types create the conditions where run-time type checking can be leveraged to execute customized error handling strategies upon type problems with unreliable data sources. It requires more investigation to decide what these strategies should be.

Issues with the type safety of records have been pointed out and a possible solution is given. The implementation of this solution should be straightforward.

Finally, the problems with the interactions between coercion, dynamic dispatch, and type inference demand a solution. The proposed idea to disable automatic type conversion for some actors is a somewhat ad-hoc solution, whereas the idea to match the subtype hierarchies of Ptolemy II types with Java interfaces that corresponding tokens representations must implement, is more structural, but a lot more involved.

Bibliography

[Abadi *et al.*, 1991] Martin Abadi, Luca Cardelli, B Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *... on Programming Languages ...*, 13(2):237–268, 1991.

[Agha and Mason, 1997] G Agha and IA Mason. A foundation for actor computation. *Journal of Functional ...*, (September 2000), 1997.

[Ashton, 2009] Kevin Ashton. That ' Internet of Things' Thing. *RFID Journal*, pages 1–2, 2009.

[Broman and Siek, 2012] David Broman and Jeremy G Siek. Modelyze : a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages. 2012.

[Broman *et al.*, 2006] David Broman, Peter Fritzson, and S Furic. Types in the modelica language. *... of the Fifth International Modelica ...*, 2006.

[Cardelli and Wegner, 1985] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM computing surveys*, 17(4), 1985.

[Cardelli, 1991] Luca Cardelli. Typeful programming. *Formal description of programming concepts*, 1991.

[Cardelli, 1996] Luca Cardelli. Type systems. *ACM Computing Surveys*, pages 1–42, 1996.

[Cartwright and Fagan, 1991] Robert Cartwright and Mike Fagan. Soft Typing. 1991.

[Cisco, 2013] Cisco. Cisco Visual Networking Index : Global Mobile Data Traffic Forecast Update , 2012 2017. Technical report, 2013.

[Cook *et al.*, 1989] WR Cook, Walter Hill, and PS Canning. Inheritance is not subtyping. *Proceedings of the 17th ACM SIGPLAN- ...*, pages 125–135, 1989.

[Davey and Priestley, 2002] BA Davey and HA Priestley. *Introduction to lattices and order*. 2002.

[Dutta and Bilbao-Osorio, 2012] Soumitra Dutta and Beñat Bilbao-Osorio. *The Global Information Technology Report 2012 Living in a Hyperconnected World*. 2012.

[Edwards, 1997] Stephen Anthony Edwards. *The Specification and Execution of Synchronous Reactive Systems*. Phd thesis, University of California, Berkeley, 1997.

[Eker *et al.*, 2003] J. Eker, J.W. Janneck, E.a. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[Foster *et al.*, 2008] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. *2008 Grid Computing Environments Workshop*, pages 1–10, November 2008.

[Hewitt, 1977] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(1977):323–364, 1977.

[Hindley, 1969] R Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[Janneck, 2003] JW Janneck. Actors and their composition. *Formal aspects of computing*, (December), 2003.

[Leavens and Dhara, 2000] GT Leavens and KK Dhara. Concepts of Behavioral Subtyping and a Sketch of their Extension to Component-Based Systems. *Foundations of component-based systems*, 2000.

[Lee and Sangiovanni-vincentelli, 2011] Edward A Lee and Alberto L Sangiovanni-vincentelli. Component-based design for the future. *Design, Automation & Test ...*, 2011.

[Lee *et al.*, 2012] EA Lee, JD Kubiatowicz, Jan M. Rabaey, Alberto L. Sangiovanni-Vincentelli, David Blaauw, Sanjit A. Seshia, and John Wawrzynek. The TerraSwarm Research Center (TSRC) (A White Paper). 2012.

[Lee, 1999] Edward A. Lee. Overview of the Ptolemy project. pages 1–23, 1999.

[Lee, 2008] Edward A. Lee. Cyber Physical Systems: Design Challenges. *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, May 2008.

[Lee, 2011] EA Lee. Heterogeneous actor modeling. *Proceedings of the ninth ACM international conference ...*, pages 3–12, 2011.

[Liskov and Wing, 1994] BH Liskov and JM Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages ...*, 16(6):1811–1841, 1994.

[Macqueen, 2012] David B Macqueen. Quick Introduction to Type Systems, 2012.

[Meijer and Drayton, 2004] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. 2004.

[Milner, 1978] Robin Milner. A Theory of Type Polymorphism in Programming. 375:348–375, 1978.

[Mitchell and Krzysztof, 2002] John C. Mitchell and Apt Krzysztof. *Concepts in Programming Languages*. 2002.

[Mitchell, 1984] JC Mitchell. Coercion and type inference. *Proceedings of the 11th ACM SIGACT-SIGPLAN ...*, pages 175–185, 1984.

[Pierce, 2002] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[Ptolemy.org, 2012] Ptolemy.org. *Claudius Ptolemaeus, Editor, System Design, Modeling, and Simulation*. 2012.

[Rabaey, 2011] JM Rabaey. The swarm at the edge of the cloud-A new perspective on wireless. *VLSI Circuits (VLSIC), 2011 Symposium on*, 3:7–9, 2011.

[Rehof and Mogensen, 1999] Jakob Rehof and Torben Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35:191–221, 1999.

[Siek and Taha, 2007] Jeremy Siek and Walid Taha. Gradual typing for objects. *ECOOP 2007Object-Oriented Programming*, pages 2–27, 2007.

[Syme, 1999] Don Syme. Proving Java type soundness. *Formal Syntax and Semantics of Java*, pages 83–118, 1999.

[Wright and Felleisen, 1994] AK Wright and M Felleisen. A syntactic approach to type soundness. *Information and computation*, 1994.

[Xiong *et al.*, 2005] Y Xiong, E Leet, and X Liu. The design and application of structured types in Ptolemy II. *... Computing, 2005 IEEE ...*, 2005.

[Xiong, 2002] Yuhong Xiong. *An Extensible Type System for Component-Based Design*. PhD thesis, University of California, Berkeley, 2002.

Index