CERIAS Tech Report 2003-20

TYPE MATCHING AND TYPE INFERENCE FOR OBJECT-ORIENTED SYSTEMS

by Tan Zhao

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907

TYPE MATCHING AND TYPE INFERENCE FOR OBJECT-ORIENTED SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Tian Zhao

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2002

To my parents.

ACKNOWLEDGMENTS

I would like to express my most sincere appreciation to my adviser, Professor Jens Palsberg, who has given me patient guidance in conducting research and generous financial support to continue my study for the past three and half years. I could not have finished this thesis without the countless late-night discussions with Professor Jens Palsberg where he shared his invaluable insights with me so generously.

I would like to thank my co-authors whose collaborations made this thesis possible. Chapter 2 of this thesis is based on joint work [PZ01, JPZ02] with Professor Somesh Jha (University of Wisconsin at Madison) and Professor Jens Palsberg. Chapter 3 is based on joint work [PZJ02] with Professor Jens Palsberg and Dr. Trevor Jim (AT&T research lab). Chapter 4 is based on joint work [PZ02] with Professor Jens Palsberg.

I am also thankful to my other committee members, Professor Jan Vitek, Dr. Jakob Rehof (Microsoft Research), Professor Mikhail Atallah, and Professor Antony Hosking for providing valuable suggestions to improve the thesis. I thank the classmates of the Secure Software Systems Lab who make my graduate life much more enjoyable.

Finally, I would like to thank my parents for the lifelong guidance, encouragement, and unconditioned love; and I thank my sister who helps me both financially and spiritually. I would like to thank Yin Wen who is always by my side for the past three years.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Zhao, Tian. Ph.D., Purdue University, August, 2002. Type Matching and Type Inference for Object-Oriented Systems. Major Professor: Jens Palsberg.

Type systems in object-oriented systems are useful tools to ensure correctness, safety, and integration of programs. This thesis studies the matching of recursive interface types for the purpose of software-system integration and type inference for object types to help reduce bulky type information for programs with flexible type systems.

We explore the problem of equality and subtyping of recursive types. Potential applications include automatic generation of bridge code for multi-language systems and type-based retrieval of software modules from libraries. We present efficient decision procedures for a notion of type equality that includes unfolding of recursive types, and associativity and commutativity of product types. Advocated by Auerbach, Barton, and Raghavachari, these properties enable flexible matching of recursive types.

We also present results on type inference for object-oriented languages with flexible type systems including features such as read-only field and record concatenation.

Read-only fields are useful in object calculi, pi calculi, and statically-typed intermediate languages because they admit covariant subtyping, unlike updateable fields. For example, Glew's translation of classes and objects to an intermediate calculus places the method tables of classes into read-only fields; covariant subtyping on the method tables is required to ensure that subclasses are translated to subtypes. In programs that use updateable fields, read-only fields can be either specified or discovered. For both cases, we will show that type inference is equivalent to solving type constraints and computable in polynomial time.

Record concatenation, multiple inheritance, and multiple-object cloning are closely related and part of various language designs. For example, in Cardelli's untyped Obliq

language, a new object can be constructed from several existing objects by cloning followed by concatenation; an error is given in case of field name conflicts. We will present a polynomial-time type inference algorithm for record concatenation, subtyping, and recursive types. Our example language is the Abadi-Cardelli object calculus extended with a concatenation operator. Our algorithm enables efficient type checking of Obliq programs without changing the programs at all.

1. Introduction

This thesis studies type-equivalence and type-inference problems in object-oriented systems in order to enhance the interoperability of software components and to infer types for flexible type systems. Type systems are often integral parts of modern programming languages. A fundamental purpose of a type system is to prevent the occurrences of execution errors. Type systems can also increase the efficiency of program execution, allow easier debugging, enable modular code development, and help improve orthogonality of language features. To take advantage of these benefits, researchers need to study the problems related to formal type systems which include, but are not limited to, type equivalence, type inference, and type soundness.

The integration of multi-language systems becomes increasingly important in the area of software systems where large amounts of legacy code exists. With the help of interface types of objects and functions, we are able to connect software components written in different programming languages. In this thesis, we study the type-matching problem during the integration of multi-language systems. We also present type-inference algorithms for type systems with variance annotations. Types with variance annotations allow more flexible subtyping and they are useful for typing object calculi, mobile processes, and statically-typed intermediate languages. We also deal with annotated types that help support concatenation of objects.

In the rest of this chapter, we first give a brief introduction to type systems in general and to types in object-oriented systems; and we later explain the applicable areas of type equivalence including component retrieval in software libraries and generating bridge code for multi-language systems; and lastly, we explain the applications of type inference for object-oriented systems.

1.1 Type Systems

Type systems promote safety at runtime and help eradicate evasive errors which otherwise may go unnoticed during program execution. Some runtime errors of a program such as dividing by zero can cause the execution to be halted. Other errors such as improper access to memory locations may not be detected and consequently cause the program to behave incorrectly. Type systems are designed to catch some of these evasive errors before runtime by statically checking the consistency between type declarations and their associated programs. When there are errors that cannot be detected statically, dynamic checks are often needed to ensure safety of program execution. Well-designed type systems can eliminate more of the runtime errors, hence, less dynamic checks are needed and programs execute more efficiently. Also, with a large fraction of errors automatically detected by typechecking methods, programmers can be more efficient in debugging their programs.

Besides detecting runtime errors, type systems are also important tools for modular compilation of programs and essential for collaborative programming in large software systems. Type information of software components can be arranged in the form of interfaces. Interface types can characterize much of the interdependencies of components and modules such that they interact with each other only through their interfaces. While maintaining relatively stable interface types, programmers are able to modify, debug, and compile components and modules independently. Moreover, typechecking algorithms can automatically verify whether software components have followed the specifications described in the interface types. Hence, developing software systems may become more efficient with suitable definitions of interface types.

In order for type systems to be useful, types should have precise definitions and proofs of their formal properties. For instance, the proof of type soundness checks the consistency between the type definitions and the semantics of a programming language. The property of type-soundness guarantees that well-typed programs compute without execution errors. Furthermore, type systems should have decidable typechecking algorithms, should make it straightforward for programmers to identify type errors, and type declarations should be checked statically as much as possible.

To formalize a type system, we need to describe the syntax, scoping rules, semantics, and type rules of the associated programming language. The syntax of a languages usually consists of syntax for types and terms. Types set the upper bounds of the ranges of values that program variables can assume during program execution, and terms are expressions and statements in program fragments. The scoping rules of a language associate occurrences of identifiers to the locations where they are declared. These rules specify the way in which free variables in a program fragment are substituted with terms. Semantics of a language relate terms in program fragments to a set of values. Independent of semantics, the type rules of a language identify a term e with a type A in the form $e : A$, relate two types with a subtyping relation in the form $A \leq B$, and associate types that are equivalent in the form $A = B$. Sometimes we have type variables that need to be associated with their definitions. This information is contained in static type environments. For instance, the type equivalence rule $\Gamma \vdash A = B$ has static type environment Γ which may contain type definitions for A, B.

Type equivalence and subtyping can be either by structure or by name. Structural equivalence and subtyping have the advantage of being precise when defined by type rules and being independent of naming schemes. However, structural type equivalence and subtyping become nontrivial when recursion is involved. We will explain in Section 1.3 why determining structural equivalence of types is important for the interoperability of software components.

We can prove a type soundness theorem by showing that if two terms are semantically equivalent, then they have the same type. Thus, if a type system is sound and decidable, then we can use typechecking algorithm to determine whether a program is well-typed and consequently whether it will execute without errors. To formally prove type soundness or show that a program is well-typed, we can employ a formal language of type systems including judgments, type rules, and type derivations. A judgment is an assertion ϕ entailed by static type environment Γ in the form of $\Gamma \vdash \phi$. A typing judgment of the form $\Gamma \vdash e : A$ asserts that in environment Γ the expression e has type A. A type rule asserts the validity of a certain judgment $\Gamma \vdash \phi$ by assuming the validity other judgments $\Gamma_i \vdash \phi_i, i \in \{1, ..n\}$.

Type rules are usually in the form of $\frac{\Gamma_1 \vdash \phi_1, \ldots, \Gamma_n \vdash \phi_n}{\Gamma \vdash \phi}$. A derivation of a judgment is a tree of judgments constructed by applying type rules. The leaves of a judgment tree should be known to be valid without assuming the validity of any other judgments.

A term e is well typed in an environment Γ if there exists a type A such that the judgment $\Gamma \vdash e : A$ can be obtained at the root of a derivation. Thus, typechecking a program is in fact the discovery of type derivations for all the terms in the program. The type-inference problem in this thesis, which may be called typability or type reconstruction in other literature, is the process of finding a type A, and an environment Γ for an untyped term e such that $\Gamma \vdash e = A$ is valid. In section 1.4, we will explain type-inference problems for object-oriented systems. In particular, we consider type systems with variance annotations so that more flexible subtyping and object concatenation are supported. Subtyping is almost ubiquitous in typed object-oriented languages which we briefly discuss in next section.

1.2 Object-Oriented Systems

Object-oriented approaches emulate the properties and behaviors of physical entities. Unlike functions or procedures, software objects are not designed to have a specific functionality; they may contain a collection of methods that operate on themselves and they may contains a collection of fields that describe the properties of the objects . Object-oriented languages have the advantages of being resilient to modifications and being reusable by allowing flexible replacement of objects and methods.

The abstraction of objects allows programmers to factor out implementation of computation to methods and organize object definitions in ways more suitable for application designs. Modifications to a particular implementation are therefore usually localized to some methods and have less effects on the whole organization of the application.

More reusable components can be written in object-oriented languages because objects and methods are more interchangeable than functions and procedures. Replacement of objects and methods do not require exact matching of types or interfaces unlike replacing modules written in procedural languages. This is due to the prevalent use of subsumption

or subtyping in object-oriented systems. For instance, object replacement only requires that the new object has at least the same set of fields and methods as the one to be replaced. Also, methods can be either reused by inheritance or replaced by overriding. The new methods have to conform to the type signatures of the old ones but with some degrees of flexibility. Replacement of methods also leads to objects that have dynamic sets of methods associated with them. In order for a method to access the rest of fields and methods in the host object, often a notion of *self* is used to identify the host object.

Object-oriented languages can be either object-based or class-based such as C++ and Java. Classes serve as object templates, and objects are instantiated from classes via an operator such as **new**. Inheritance of classes creates class hierarchies and also enables reuse of methods. A subclass is a class that inherits from other classes and subclassing usually implies subtyping. For instance, suppose that class c is a subclass of c' and objects o, o' are instances of c and c' respectively. In languages such as C++ and Java, the type of o is a subtype of the type of o'. Also, object o can assume the type of o' as well, which implies that if there is a variable v of the type of o', then we can assign o to v and this is known as subsumption. Because of subsumption and overriding, when we invoke a method on an object in variable v, we cannot be sure which method will be invoked until runtime and this is known as dynamic dispatch.

Object-based languages are less popular than class-based languages, though they can be simpler and more flexible. Object-based languages emulate behaviors in class-based languages with simpler mechanisms. Without classes as object templates, object-based languages create new objects by cloning prototypes. Also method-updates are used in place of method overriding.

The object-type systems that we are concerned with in Section 1.4 are derived from an Abadi-Cardelli object calculus [AC96a], which is an untyped first-order object calculus with subtyping and recursion, and the syntax of which with some variations is shown in Figure 1.1. Object calculi are further decompositions of object-based languages. Only limited features such as method invocation and method update are built-in for the Abadi-Cardelli object calculus. The object types are a collection of fields with distinct labels

A, B	$::=$	types	
	X		type variable
	$[\ell_i : B_i{}^{i\in 1..n}]$		object type (ℓ_i distinct)
	$\mu(X)A$		recursive type
a, b	$::=$	terms	
	x		variable
	$[\ell_i = \varsigma(x_i)b_i{}^{i\in 1..n}]$		object (ℓ_i distinct)
	$a.\ell$		method invocation
	$a.\ell \Leftarrow \varsigma(x)b$		method update.

Figure 1.1. Syntax of an Abadi-Cardelli calculus

and one type for each field. The largest object type, denoted by $[\ ]$, is an object type without any fields. If a type A contains at least the set of fields in type B, then B is the subtype of A, which is called width subtyping. For instance, $[\ell_i : B_i{}^{i\in 1..n+m}]$ is a subtype of $[\ell_i : B_i{}^{i\in 1..n}]$. The type system for the object calculus in Figure 1.1 has only width subtyping and recursive types.

Even though type-checking can help eliminate programs with runtime errors, it can also rule out sound programs as well. In fact, there are some limitations of the type systems such as the one in Figure 1.1 that render some sound programs untypable. We will discuss this problem in Section 1.4. In Section 1.3, we are mainly interested in interface types of object-oriented languages such as Java, where only type signatures of methods and static data types are defined in interfaces.

1.3 Type Matching

Much of the study of type isomorphisms (equivalence) is motivated by the component-retrieval problem and generation of bridge code for multi-language applications.

1.3.1 Component Retrieval in Software Libraries

Types have been used as search keys in retrieving suitable components from a software library. Many functions or components that fit the specification of users do not always have the exact same type as the one the user provided. We therefore need to somehow relate types isomorphic to the query type.

In some cases, function types are good search keys since non-recursive types are easy to compare and most function libraries contains the types of their components.

Consider a type system with the following terms.

$$\tau \equiv \gamma \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2$$

The symbol γ ranges over base types such as integer and boolean. The symbols $\rightarrow$ and $\times$ denote the function type constructor and the product type (or record type) constructor respectively.

A type of a function can be written as $\sigma \rightarrow \tau$, where σ is the type of arguments and τ is the return type of the function. Suppose we are looking for a function *foo* that takes two arguments of types *boolean* and *integer* respectively and returns a pair consisting of a boolean and an integer, and the type of *foo* can be written as

$$(\text{bool} \times \text{int}) \rightarrow (\text{bool} \times \text{int}).$$

So how do we decide that two types are actually matched? We may require the matched function to have exactly the same type, that is, the argument types are in the same order and so are the return types. This is too restrictive as it turns out. Some functions may have similar types which can be converted into the sought type via simple transformations such as argument reordering or currying. For instance, functions with the following types

$$(\text{int} \times \text{bool}) \rightarrow (\text{bool} \times \text{int}) \quad \text{or} \quad \text{bool} \rightarrow (\text{int} \rightarrow (\text{bool} \times \text{int}))$$

can be converted to *foo* by reordering the argument or an uncurry transformation. Furthermore, a function that returns a pair can be translated into two functions that return the components of the pair. The following type may be what we want as well.

$$((\text{int} \times \text{bool}) \rightarrow \text{bool}) \times ((\text{int} \times \text{bool}) \rightarrow \text{int})$$

In fact, we consider these similar types to belong to an equivalence class consisting of types isomorphic to the type of *foo*. Informally, two types σ and τ are isomorphic if there exist conversion from terms of σ to terms of τ and vice versa, where the compositions of the two conversions in both orders are the identity mappings.

Rittri [Rit91] noted the use of types as query keys for searching functional libraries. To allow flexible retrieval of desired functions, he defined a notion of isomorphism via the functions of a $\lambda\beta\eta$-calculus with surjective pairing. He also gave a semantics and an axiomatic characterization of isomorphism with the axiom rules shown in Figure 2.6. Rittri noted that these isomorphisms also hold in all Cartesian Closed Categories. (An introduction to category theory can be found in [Gol79]). This type system was proved complete for models of the simply typed λ-calculus with surjective pairing and terminal objects by Bruce, Di Cosmo and Longo [BCL92]. Di Cosmo [Cos95] gave a more detailed treatment of type isomorphism including systems involving second order types. He also mentioned as future work to incorporate recursive types into the system of isomorphism.

Zaremski and Wing [ZW95] have done similar work in signature matching for retrieving components from an ML-like functional library. Unlike others, rather than reasoning about a complete set of rules, Zaremski and Wing emphasized the flexibility of combining those rules with generalized, specialized or unified matching. They also included user defined type operators such as *list* in the matching.

When users attempt to retrieve specific functions from the library, usually, more generalized functions will suffice as well. They can instantiate the retrieved functions to get what they need. In addition, it is difficult for a user to guess the most general type of a function. Generalized matching allows users to query the most general type with more specific ones. As shown in [NPS93], generalized matching is NP-complete with the isomorphism defined in Figure 2.6.

Once the desired function is retrieved, programmers may want to convert the interface of the retrieved function into a certain preferred form. It is easy to modify the retrieved function so that the type is preserved under isomorphism. However, if the programmers use a language different from the one that the retrieved function is written in, the translation of

the interface becomes an non-trivial task. Preferably, we would like to automatically convert a component interface written in one language into an interface for another language under type isomorphism. This is essential for generating bridge code for multi-language applications.

1.3.2 Generating Bridge Code for Multi-language Systems

Large and complex software applications often contain modules written in different programming languages. This may be due to the need for reusing legacy components or because certain languages are more suitable for particular application areas. In addition, programmers for distributed applications may want to provide interfaces to facilitate interoperability with other programs already written in several different languages. In any case, mechanisms are needed to glue these multi-lingual components together. CORBA [OMG99], PolySpin [BKW96] and Mockingbird [ACC97, ABCCR99], etc. are systems designed to overcome this difficulty.

In multi-language applications, software modules can be considered to be of two kinds, object and client. Objects must include public interfaces to allow access from clients written in different languages. CORBA-style approaches utilize a separate interface definition language called IDL. The objects are wrapped with language-independent interfaces defined in IDL, and the wrappers are translated into interfaces in the languages that clients are using so that clients can invoke methods in these objects via the interfaces. Exact types are preserved as the method invocations cross the language boundaries, because both the client and object adhere to the common interfaces for interaction.

Since interfaces defined in IDL must be able to be translated into many different languages, the type system in IDL has to be the intersection of the type systems of all the programming languages that CORBA supports. As a result, declarations in IDL lack expressive power and may not be convenient for local computation. Client code has to switch between its own type system for local computation and IDL-derived types for remote operation on objects. In addition, common object types are not transparent to software modules. Consequently, program modules implemented in a language with a more flexible type system have to be modified and retrofitted to use the IDL-based interfaces.

The PolySpin and Mockingbird projects offer alternatives to defining interfaces in a common interface language. In both approaches, clients and objects are written within their own type systems and remote operation across language boundary is supported automatically by compiler-generated bridge code or by modifying object method implementations. Because object interfaces are not defined in a common type system, we must be able to convert an object interface into a compatible form in other languages. PolySpin employed an isomorphism framework similar to Zaremski and Wing [ZW95].

In *PolySpin*, interoperability support is divided into four parts: locator, language arbitor, communicator, and type matcher. A locator is the name management component used to locate objects by their language-neutral name. A language arbitor associates language information with objects as part of the name-object binding. A communicator achieves inter-language invocation by automatically modifying the implementation of object methods. Modified methods consult the language arbitor at each invocation and decide whether to make a local method call or a generated inter-language call. Modification done to object methods is dependent on whether the interface type of the object is compatible with interface types defined in other languages. A type matcher checks type compatibility based on a *relaxed* criteria of type isomorphism.

The implementation of PolySpin takes the type definitions of a set of clients and objects written in multiple languages and modifies the type implementations after checking the compatibility of these types. The modified objects then are ready to accept method calls from clients written in other languages. Users have to supply the type definitions that are matched semantically for PolySpin to do any useful work.

A type matcher only considers abstract object types whose properties are captured completely in the signatures of methods. Signatures of methods are matched with operations including renaming, argument reordering and currying. Two object types are considered as matched if either all methods or a subset of methods in the objects are matched in method signatures. It seems that PolySpin did not consider recursive types.

The *Mockingbird* project is similar to CORBA in that they both make use of an intermediate interface language. A key difference is that Mockingbird automatically generates

interfaces, provided some annotations in the source code. Compared with PolySpin, Mockingbird allows more flexible translation of types across languages. Besides abstract data types such as object, non-abstract types such as records, linked lists and arrays are also considered when generating interfaces. As in PolySpin, inter-language method calls does not preserve exact types, that is, data values can be transported across language boundary if the interfaces of the data object are compatible. Two interfaces in different languages are compatible if the types of the interface are inter-convertible, which means that there is an invertible mapping from one type to the other. This *inter-convertibility* of types follows the ideas of structure-based type isomorphism. Because certain information is stored structurally via inter-object references, the types of interfaces could be recursive. Determining whether two recursive types are isomorphic with all the rules in Figure 2.6 turns out to be quite a hard problem. In fact, Mockingbird used some conservative heuristics to determine the inter-convertibility and to find invertible mappings between types.

The Mockingbird system consists of four elements: extractor, analyzer, synthesizer, and emitter. Extractors extract the data-type definitions from annotated source languages. Analyzers generate an intermediate interface in the Mockingbird Signature Language (MockSL) based on the type definitions. The annotations in source code are some formatted type information provided by programmers. Synthesizers interact with the programmer to decide the type compatibilities of interfaces and generate declarations in clients' languages. In the end, the emitters produce marshaling stubs for each language from MockSL.

The improvement of PolySpin and Mockingbird over CORBA largely rests on the ability to use native type systems in defining operations across programming languages. In PolySpin, interoperability of interfaces are judged by the compatibility of abstract types under non-recursive type isomorphism. The Mockingbird project aims to extend definition of type isomorphism to recursive types and more, and thereby to automate the translation of non-abstract types.

The structure-based type isomorphism of the Mockingbird project seems non-trivial to relate to any formal theory however. Auerbach, Barton, and Raghavachari [ABR98] defined a theory of type isomorphism composed of the axioms of a theory of type isomorphism (see

Figure 2.6) and the axioms of a theory of equality for recursive types (see Figure 2.8). The decidability of each in isolation has been proven. Auerbach, Barton and Raghavachari raised the question of whether the combination of the two set of axioms is consistent and decidable. It turns out that the combination is in fact *inconsistent* since all types can be proven equal in this case. Thus, the isomorphism problem of recursive types cannot simply be defined by the union of two systems and we have to set out finding a new definition that is consistent and decidable.

1.3.3 Type Matching Algorithms

In this thesis, we study the problem of matching recursive types with a subset of the isomorphism rules in Figure 2.6. We are interested in equivalence rules that include associativity and commutativity of product types because they are more useful for current software systems. Even though it is straightforward to given an exponential-time algorithm for the matching problem, it is non-trivial to find a polynomial time solution. Using an iterative approach, we discovered an $O(n^2)$ algorithm [PZ01] for matching recursive types with flexible equality rules characterized by a definition of bisimulation. The algorithm depends on a monotone function constructed from the definition of a bisimulation and from an initial relation on potentially equivalent types. Matched types, if any, should be contained in the greatest fixed point of the monotone function. The time complexity of algorithm was further improved to $O(n \log n)$ time [JPZ02] by reducing the fixed-point computation of the monotone function to the problem of finding the coarsest size-stable partition-refinement of a graph. We have implemented the second algorithm in Java and the implementation allows users to compare interface types with the option to exclude some type equalities when multiple matches exist.

1.4 Type Inference

Type inference automatically discovers type information from untyped or partially typed programs. The problems we study in this thesis are type inference for untyped object calculi with variance annotations where both width and depth subtyping are allowed and where object concatenation is supported. For object types of the form $[\ell : B, \ldots]$, there are several design choices. Abadi and Cardelli [AC96a] explain that if the field ℓ can be both read and

$$
\begin{array}{rcl}
\text{Point} & = & [\text{move} = \varsigma(x)x] \\
\text{ColorPoint} & = & [\text{move} = \varsigma(y)y,\ \text{setcolor} = \varsigma(z)z] \\
\text{Circle} & = & [\text{center} = \varsigma(d)\text{Point}] \\
\text{ColorCircle} & = & \text{Circle.center} \Leftarrow \varsigma(e)\text{ColorPoint.move.setcolor} \\
\text{Main} & = & \text{ColorCircle.center.move}
\end{array}
$$

Figure 1.2. Example of an untypable program in the type system of $\mathbf{Ob}_{1<:\mu}$

updated, then ℓ must be *invariant*, that is, if $[\ell : A, \ldots]$ is a subtype of $[\ell : B, \ldots]$, then $A = B$. This is the case for a type system with syntax in Figure 1.1. However, invariant subtyping turns out to too restrictive for some programs.

For example, in Figure 1.2 is a program written in a variant of the Abadi-Cardelli object calculus [AC96a] with syntax shown in Figure 1.1. Each method $\varsigma(x)b$ binds a name x which denotes the smallest enclosing object, much like "this" in Java. We apologize in advance that the methods do not exhibit any useful behavior; the method were chosen to make them difficult to type check. The "Main" program of this example should execute without run-time errors because of the following reductions.

$$
\begin{array}{rl}
 & \text{ColorCircle.center.move} \\
\rightarrow & (\text{(ColorPoint.move).setcolor}).\text{move} \\
\rightarrow & (\text{ColorPoint.setcolor}).\text{move} \\
\rightarrow & \text{ColorPoint.move} \\
\rightarrow & \text{ColorPoint}
\end{array}
$$

Palsberg and Jim [PJ97] noted that this program is *not* typable in Abadi and Cardelli's type system $\mathbf{Ob}_{1<:\mu}$, which has recursive types, width subtyping for object types, and only invariant fields. The key reason for the untypability is that the body of the `ColorCircle`'s

`center` method forces `ColorPoint` to have a type which is *not* a subtype of the type of `Point`, intuitively as follows.

$$\begin{aligned} \text{Point} &: \mu(X)[\text{move} : X] \\ \text{ColorPoint} &: \mu(X)[\text{move}, \text{setcolor} : X] \\ \mu(X)[\text{move}, \text{setcolor} : X] &\not\leq \mu(X)[\text{move} : X] \end{aligned}$$

Moreover, `ColorCircle.center.move` is *not* typable. Nonetheless, the example in Figure 1.2 is typable in more flexible type systems, which we will study in Chapter 3.

1.4.1 Covariant Read-Only Fields

Variance annotations distinguish fields of an object as covariant, contravariant and invariant. A covariant read-only field (CROF) is a field which enjoys covariant subtyping and which cannot be updated. Similarly, a contravariant field is write-only and cannot be read. An invariant field can be either read or updated.

Following Abadi and Cardelli, we use the notation $[\ell^0 : B, \ldots]$ to denote an object type with an invariant field ℓ; and we use the notation $[m^+ : B, \ldots]$ to denote an object type with a covariant field m. Covariance implies that if $[m^+ : A, \ldots]$ is a subtype of $[m^+ : B, \ldots]$, then A is a subtype of B, a weaker condition than $A = B$.

Variant subtyping of object types increases the expressiveness of type system. Some sound programs that fail to type check with only width subtyping and recursive types are typable with variant subtyping. Variance annotations can also be used for typing and subtyping for mobile processes [PS93] to enforce that some communication channels are for input only or for output only. Read-only fields are useful in statically-typed intermediate languages because they admit covariant subtyping and can be used to type method-table fields in the Glew's encoding of classes and objects [Gle00]. The target language used in the encoding has a super-set of the type syntax shown in Figure 1.3 which is slightly modified to be consistent with the notations in the rest of the thesis. Notice that there are two kinds of variance annotations ϕ and φ in Figure 1.3. We first discuss the variance ϕ for annotating fields of a record type. *Notice that we use record type and object type interchangeably here.*

$$
\begin{array}{lllll}
\text{Type} & \tau, \sigma & ::= & \ldots \mid [\ell_i^{\phi_i} : \tau_i \,^{i \in 1..n}]\varphi \mid \ldots \\
\text{Variances} & \phi & ::= & + \mid 0 \\
 & \varphi & ::= & 0 \mid \rightarrow
\end{array}
$$

Figure 1.3. Part of the syntax of Glew's target language for object and class encoding

CROFs play an essential role in Glew's translation of objects and classes to a typed intermediate language. Like most implementations of object-oriented languages, Glew's translation uses method tables. One of Glew's insights is that the method table can conveniently be placed in a CROF. For example, let a and b be two source-language objects such that the type of b is a subtype of the type of a. The type system for the source language supports that b may have *more* methods than a (width subtyping). This means that the method table in the translation of b will be *longer* than the method table in the translation of a:

$$
\begin{array}{rcl}
\text{translation } (a) & = & \ldots [\text{mt} = m_a, \ldots] \ldots \\
\text{translation } (b) & = & \ldots [\text{mt} = m_b, \ldots] \ldots
\end{array}
$$

where mt is the field name for the method table. Glew's translation of b has a subtype of the type of his translation of a; he makes mt a CROF, and he gives the following types to the translations of a and b:

$$
\begin{array}{rcl}
\text{type-of } (\text{translation } (a)) & = & \ldots [\text{mt}^+ : \text{type-of } (m_a), \ldots] \ldots \\
\text{type-of } (\text{translation } (b)) & = & \ldots [\text{mt}^+ : \text{type-of } (m_b), \ldots] \ldots
\end{array}
$$

Glew's translation produces typed intermediate code, including the annotations 0 and $+$.

1.4.2 Type Systems for Record Concatenation

While covariant subtyping increases the flexibility of subtyping for fields of an object, type annotations that restrict width subtyping for some object types are useful for the purpose of record concatenation. In Glew's type system, the variance φ (see Figure 1.3) is used to annotate a record type so that there are two forms of record types. The variance

annotation 0, as in

$$[\ell_i : B_i \ ^{i\in 1..n}]^0,$$

denotes that records of that type *can* be concatenated, and that subtyping *cannot* be used. The variance annotation $\rightarrow$, as in

$$[\ell_i : B_i \ ^{i\in 1..n}]^{\rightarrow},$$

denotes that records of that type *cannot* be concatenated, and that subtyping *can* be used. For example, if we have

$$\begin{aligned} [l : 5, m : true] &: [l : int, m : bool]^0 \\ [n : 7] &: [n : int]^0 \end{aligned}$$

then for the concatenation (denoted by $+$) of the two records we would get

$$\begin{aligned} [l : 5, m : true] + [n : 7] &: [l : int, m : bool]^0 \oplus [n : int]^0 \\ &= [l : int, m : bool, n : int]^0. \end{aligned}$$

where $\oplus$ is the symmetric concatenation operation on record types which is only defined when the labels sets are disjoint and the two types both have the variance annotation 0. The idea is that if an object has type $[l_i : t_i]^0$, then we know exactly which fields are in the object, and hence we know which other fields we can safely add without introducing a field name conflict. The more flexible types $[\ell_i : B_i \ ^{i\in 1..n}]^{\rightarrow}$ can be used to type objects that will not be concatenated with other objects.

This kind of type annotation can be useful for Cardelli's untyped language named *Obliq* [Car95], where the operation

$$clone(a_1, \ldots, a_n)$$

creates a new object that contains the fields and methods of all the argument objects $a_1, \ldots, a_n$. This is done by first cloning each of $a_1, \ldots, a_n$, and then concatenating the clones. An error is given in case of field name conflicts, that is, in case at least two of $a_1, \ldots, a_n$ have a common field. Cardelli notes that useful idioms are:

$$clone(a, \{l : v\})$$

to inherit the fields of a and add a new field l with initial value v, and:

$$clone(a_1, a_2)$$

to multiply inherit from a_1 and a_2. Obliq's multiple-object cloning is an instance of the idea of concatenating two records of data. In a similar fashion, languages such as C++ [Str93] and Borning and Ingalls' [BI82] version of Smalltalk allow multiple inheritance of classes.

For languages such as Obliq, concatenation is a run-time operation and where a field name conflict is considered an error; such concatenation is known as *symmetric concatenation*. There are several ways of handling field name conflicts. One idea is to do run-time checking, and thereby add some overhead to the execution time. Another idea, which we study in this thesis, is to statically detect field name errors by a type system. The main challenge for such a type system is to find out which objects will eventually be concatenated and give them types that support concatenation.

Writing programs with variance annotations adds extra burden to programmers and on some occasions, having types with variance notations is just too bulky for programs such as those written in intermediate languages. It is therefore interesting to find type inference algorithm for an implicitly-typed version of Glew's intermediate language and for languages such as Obliq. Such an algorithm would make it possible to omit bulky type annotations, and to automatically discover the CROFs and support record concatenation.

1.4.3 Type-Inference Algorithms

In this thesis, we study the problem of inferring object types with variance annotations. We have found an algorithm of type inference for the object expressions with covariant read-only fields [PZJ02]. We apply the algorithm to a variation of Abadi-Cardelli object calculus where some of the object fields can be specified as read-only. Type inference is equivalent to solving type constraints, which in turn is P-complete and computable in $O(n^3)$ time. We have also developed a NP-time algorithm for inferring annotated types to support record concatenation. Both type-inference problems deal with type systems with type annotations which either restrict or relax subtyping relations for fields of an object or a record. The proof structures in Chapter 3 and 4 are quite similar and they follow the style

of [JP97]. Therefore, we will emphasize some of the similarities and differences between the reasonings in Chapter 3 and those in Chapter 4.

1.5 Overview of the Thesis

This chapter has provided a brief summary of our research areas and introduced some motivating examples for the problems that we study in this thesis. In Chapter 2, we present solutions to the type matching problem for recursive types with a notion of flexible equality and also briefly discuss our implementation. Chapter 3 gives the solution and implementation for the type inference problem of discovering covariant read-only fields for an untyped object calculus. In Chapter 4 we solve the type inference problem for record concatenation, subtyping, and recursive types. Finally, Chapter 5 summarizes the thesis and discusses some future directions.

2. Efficient and Flexible Matching of Recursive Types

2.1 Introduction

Much of the previous work on type equality focuses on non-recursive types [BCL92, Cos95, NPS93, Rit90, Rit91, Rit93, Sol83, ZW95]. In this chapter we consider equality of recursive types.

2.1.1 Background

Potential applications of flexible type equality include automatic generation of bridge code for multi-language systems [ABR98, BKW96], and type-based retrieval of software modules from libraries [Rit90, Rit91, Rit93, ZW95].

Software engineers often look into a software library to find reusable components for their applications. A large library can be hard to search, however. It may be organized in alphabetical order or coarsely sorted according to some structure. Beyond the structural information of the library, the only thing that we can rely on is the component name to retrieve the code we need. Component names are difficult to guess. So, it makes sense to search by the type of the components. A component that fits the specification of a programmer does not always have the exact same type as the one the user is using as search key. That is why we need a flexible notion of type equality.

Designing and maintaining a multi-language application often calls for bridge code for components written in various programming languages such as C, C++ and Java. The conversion of values of isomorphic (equivalent) types is essential. The foundation of deciding whether a conversion makes sense at all is a flexible notion of type equality. An alternative might be to start with just one type, and then translate it into a type in a different language [Gay94]. Such a translation may be helpful when building a new software component that should be connected to an existing one. However, when faced with connecting two exist-

ing software components, programmers may find type matching and automatic bridge code generation more helpful.

In object-oriented languages such as C++ and Java, many types are recursive. Thus, to be useful for such languages, a flexible notion of type equality should be able to handle recursive types.

2.1.2 The Problem

Equality and subtyping of recursive types have been studied in the 1990s by Amadio and Cardelli [AC93]; Kozen, Palsberg, and Schwartzbach [KPS95]; Brandt and Henglein [BH97]; Jim and Palsberg [JP97]; and others. These papers concentrate on the case where two types are considered equal if their infinite unfoldings are identical. Type equality can be decided in $O(n\alpha(n))$ time, and a notion of subtyping defined by Amadio and Cardelli [AC93] can be decided in $O(n^2)$ time [KPS95].

If we allow a product-type constructor to be associative and commutative, then two recursive types may be considered equal *without* their infinite unfoldings being identical. Alternatively, think of a product type as a multiset, by which associativity and commutativity are obtained for free. Such flexibility has been advocated by Auerbach, Barton, and Raghavachari [ABR98].

Until now, there are no efficient algorithmic techniques for deciding type equality in this case. One approach would be to guess an ordering and a bracketing of all products, and then use a standard polynomial-time method for checking that the infinite unfoldings of the resulting types are identical. For types without infinite products, such an algorithm runs in NP time. One of the inherent problems with allowing the product-type constructor to be associative and commutative is that

$$\begin{aligned} & A \times A \times B \;=\; A \times B \times A, \\ \text{while} \quad & A \times A \times B \;\neq\; A \times B \times B. \end{aligned}$$

Notice the significance of the multiplicity of a type in a product. One could imagine that an algorithm for deciding type equality would begin by determining the multiplicities of all components of product types, or even order the components. However, it seems like

this would have to rely on being able to decide type equality for the component types, and because the types may be recursive, this seems to lead to a chicken-and-egg problem.

2.1.3 Our Result

We have developed an efficient decision procedure for a notion of type equality that includes unfolding of recursive types, and associativity and commutativity of product types, as advocated by Auerbach et al. For two types of size at most n, our algorithm directly based on our definition of type equivalence decides equality in $O(n^2)$ time. The main data structure is a set of type pairs, where each pair consists of two types that potentially are equal. Initially, all pairs of subtrees of the input types are deemed potentially equal. The algorithm iteratively prunes the set of type pairs, and eventually it produces a set of pairs of equal types. The algorithm takes $O(n)$ iterations each of which takes $O(n)$ time, for a total of $O(n^2)$ time.

We also present an $O(n \log n)$ time algorithm for deciding type equivalence. The algorithm works by reducing the type matching problem to the well-understood problem of finding a size-stable partition of a graph [PT87].

2.1.4 Implementation

We have implemented a type-matching tool based on our second algorithm. The tool is for matching Java interfaces. It supports a notion of equality for which interface names and method names do not matter, and for which the order of the methods in an interface and the order of the arguments of a method do not matter. When given two Java interfaces, our tool will determine whether they are equivalent, and if they are, it will present the user with a textual representation of all possible ways of matching them. In case there is more one way of matching the interfaces, the user can input some restrictions, and invoke the matching algorithm again. These restrictions may come from non-structural information known to the user such as the semantics of the methods. In this way, the user can interact with the tool until a unique matching has been found.

2.1.5 Chapter Overview

In the following section we give an overview of our techniques by way of an example. This example will be used later in the chapter for illustrative purposes. In Section 2.3

we summarize related work. Section 2.4 gives an excerpt of the definitions of terms and automata from [KPS95] which we will use in later sections. In Section 2.5 we present the definitions and properties of types and type equivalence and we explain in detail an $O(n^2)$ algorithm for deciding type equivalence based on our definitions. In Section 2.6 we show an extension to intersection and union types. In Section 2.7, we introduce a $O(n \log n)$ algorithm to decide type equivalence. An implementation of the algorithm is discussed in Section 2.8. Subtyping of recursive types is discussed in Section 2.9. Concluding remarks appear in Section 2.10.

2.2 Example

The purpose of this section is to give a gentle introduction to the algorithm and some of the definitions in Section 2.5. We do that by walking through a run of our algorithm on a simple example. While the example does not require all of the sophistication of our algorithm, it may give the reader a taste of what follows in Section 2.5.

This example which will be used throughout the chapter. It is straightforward to map a `Java` type to a recursive type of the form considered in this chapter. A collection of method signatures can be mapped to a product type, a single method signature can be mapped to a function type, and in case a method has more than one argument, the list of arguments can be mapped to a product type. Recursion, direct or indirect, is expressed with the μ operator. This section provides an example of of `Java` interfaces and provides an illustration of our algorithm.

Suppose we are given the two sets of Java interfaces shown in Figures 2.1 and 2.2. We would like to find out whether interface I_1 is structurally equal to interface J_2. We want a notion of equality for which interface names and method names do not matter, and for which the order of the methods in an interface and the order of the arguments of a method do not matter.

Notice that interface I_1 is recursively defined. The method m_1 takes an argument of type I_1 and returns a floating point number.In the following, we use names of interfaces and methods to stand for their type structures. The type of method m_1 can be expressed as $I_1 \rightarrow float$. The symbol $\rightarrow$ stands for the function type constructor. Similarly, the type of

```
interface I1 {                  interface I2 {
    float   m1(I1 a);               I1   m3(float a);
      int   m2(I2 a);               I2   m4(float a);
}                               }
```

Figure 2.1. Interfaces I_1 and I_2

```
interface J1 {                  interface J2 {
    J1   n1(float a);               int   n3(J1 a);
    J2   n2(float a);             float   n4(J2 a);
}                               }
```

Figure 2.2. Interfaces J_1 and J_2

m_2 is $I_2 \to int$. We can then capture the structure of I_1 with conventional μ-notation for recursive types:

$$I_1 = \mu\alpha.(\alpha \to float) \times (I_2 \to int)$$

The symbol α is the type variable bound to the type I_1 by the symbol μ. The interface type I_1 is a product type with the symbol $\times$ as the type constructor. Since we think of the methods of interface I_1 as unordered, we could also write the structure of I_1 as

$$\begin{aligned} I_1 &= \mu\alpha.(I_2 \to int) \times (\alpha \to float)\ , \\ I_2 &= \mu\beta.(float \to I_1) \times (float \to \beta)\ . \end{aligned}$$

In the same way, the structures of the interfaces J_1, J_2 are:

$$\begin{aligned} J_1 &= \mu\alpha'.(float \to \alpha') \times (float \to J_2) \\ J_2 &= \mu\beta'.(J_1 \to int) \times (\beta' \to float). \end{aligned}$$

The unfolding rule for recursive types says that

$$\mu\alpha.\tau = \tau[\alpha := \mu\alpha.\tau],$$

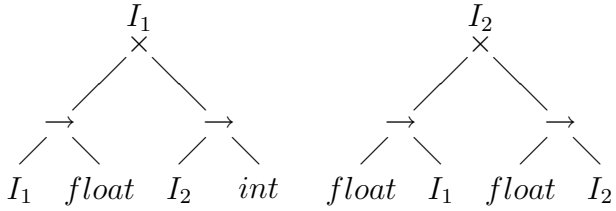


Figure 2.3. Trees for interfaces I_1 and I_2

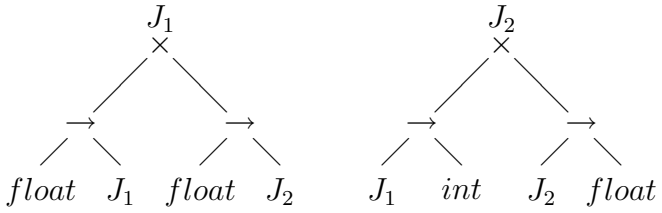


Figure 2.4. Trees for interfaces J_1 and J_2

which means that the recursive type $\mu\alpha.\tau$ is equivalent to τ where every free occurrence of α in τ is replaced by $\mu\alpha.\tau$. Infinite unfolding of a recursive type will result in a regular tree, that is, a tree with a finite number of distinct subtrees.

Trees corresponding to the two types are shown in Figures 2.3 and 2.4. The interface types I_1, J_2 are equal iff there exists a bijection from the methods in I_1 to the methods in J_2 such that each pair of methods in the bijection relation have the same type. The types of two methods are equal iff the types of the arguments and the return types are equal.

The equality of the interface types I_1 and J_2 can be determined by trying out all possible orderings of the methods in each interface and comparing the two types in the form of finite automata. In this case, there are few possible orderings. However, if the number of methods is large and/or some methods take many arguments, the above approach becomes time consuming because the number of possible orderings grows exponentially. An efficient algorithm for determining equality of recursive types will be given later in the chapter.

Our approach is related to the pebbling concept used by Dowling and Gallier [DG84]. We propagate information about inequality from the type pairs known to be unequal toward the ones we are interested in.

We will use the concepts of *bipartite graphs* and *perfect matching*.A bipartite graph is an undirected graph where the vertices can be divided into two sets such that no edge connects vertices in the same set. A perfect matching is a matching, or subset of edges without common vertices, of a graph which touch all vertices exactly once.

We organize the types of interfaces, methods, and base types (such as int) into a bipartite graph (V, W, R), where V represents the types in interfaces I_1, I_2 and W represents the types in interfaces J_1, J_2. That is, $V = \{I_1, I_2, m_1, m_2, m_3, m_4, int, float\}$, and $W = \{J_1, J_2, n_1, n_2, n_3, n_4, int, float\}$. The set of edges R represents "hoped-for" equality of types.

We initialize R as $(V \times W)$, that is, we treat every pair of types as equivalent types at the start. The idea is that by iteration, we remove edges between types that are not equal. When no more edges can be removed, the algorithm stops. The types connected in the final graph are equal.

First, we remove the edges between types that are obviously not equal. For example, an interface type and a method type are not equal; and a base type and a method type are not equal. We remove edges that connect interface types and method types, and edges between method types and base types.

In the iterations that follow, we remove edges between types that are not equal based on the information known from previous iterations. For example, we can determine that the method types m_1 and n_1 are not equal because the argument type of m_1 is I_1 while the argument type of n_1 is $float$, and the edge between I_1 and $float$ is removed in the preceding iteration. Therefore, we remove the edge between m_1 and n_1.

The interesting part is to determine whether the types of two interfaces with n methods each are not equal based on information from previous iterations. This subproblem is equivalent to the perfect matching problem of a bipartite graph (V', W', R'), where V' and W' are the sets of methods in each interface, and there is an edge between two methods iff the types of the two methods have not been determined unequal in the previous iterations. If the set of edges R' is arbitrary, then the complexity of the perfect matching problem is $O(n^{5/2})$ (see [HK73]).

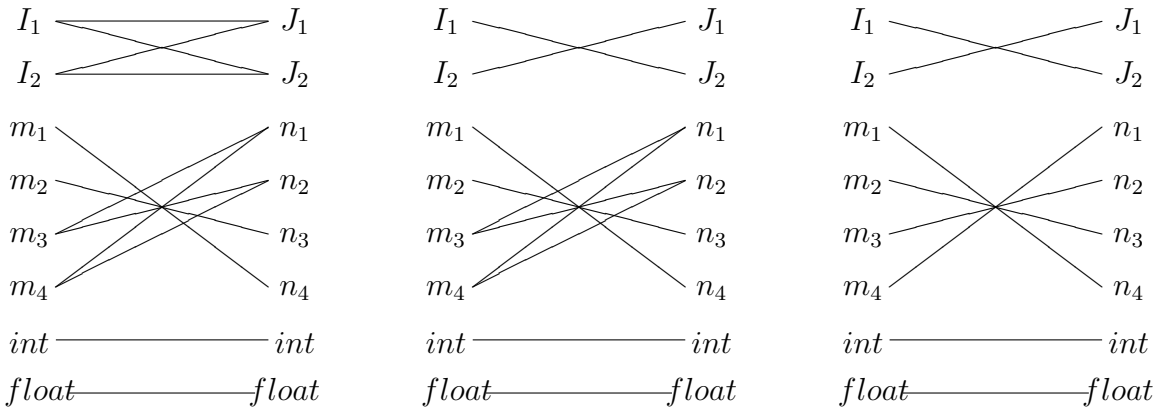


Figure 2.5. Bipartite graphs after the 2nd, 3rd and 4th iterations

However, the graph (V, W, R) has a coherence property: if a vertex in V can reach a vertex in W, then there is an edge between these two vertices. Coherence both enables us to perform each iteration efficiently, and guarantees that the whole algorithm will terminate within $|V| + |W|$ iterations.

The resulting bipartite graphs after the second, the third, and the fourth iterations are given in Figure 2.5. In the third iteration, we examine the edges between interface types and determine whether we should remove some of the edges. For the types of interfaces I_1 and J_1 to be equal, there must exist a bijection from $\{m_1, m_2\}$ to $\{n_1, n_2\}$ such that the pair of methods in the bijection relation are connected in the bipartite graph after the second iteration. It is clear that the types of interface I_1 and J_1 are not equal since there is no edge between m_1, m_2 and n_1, n_2 at all. Thus, the edge between I_1 and J_1 is removed. Similarly, we remove the edge between I_2 and J_2.

By the same steps, we are able to remove the edge between m_3 and n_1, and the edge between m_4 and n_2 in the fourth iteration. After that, we cannot remove any more edges from the graph. Now the algorithm terminates and we can conclude that interface I_1 is equal to interface J_2. If we compare two types that can be represented with two automata each of size at most n, then the above algorithm will spend $O(n)$ time in each iteration and will terminate within $O(n)$ iterations, for a total of $O(n^2)$ time.

The simple example above does not reveal how the coherence property of an edge set can help speed up an iteration. This is because interfaces I_1, I_2, J_1, J_2 only have two methods each. In the Section 2.5 we present an efficient algorithm for the general case.

2.3 Related Work

Problems of type isomorphism can be divided into three categories: word problems, matching problems and unification problems. A word problem is to decide the equality of two types via a theory of isomorphism. The types could be finite or infinite and they may contain types variables. A matching problem is to decide for given a pair (p, s) of types (the pattern and the subject), whether there exists a substitution σ such that $p\sigma$ is equal to s. Similarly, a unification problem is about the existence of σ such that $p\sigma$ and $s\sigma$ are equal. Notice that matching is a generalization of the word problem while a special case of unification. If p and s do not contain type variables, then the matching and unification problems reduce to word problem.

The axiom system T_{CC} in Figure 2.6 gives a sound and complete axiomatization of isomorphism of types in Cartesian Closed categories [Sol83, BCL92]. If we exclude Rules (DISTRIB$\rightarrow$ $\times$), (UNIT), then the remaining axiom system, denoted T_{SMC}, gives a sound and complete axiomatization of isomorphism (called *linear* isomorphism) of types in Symmetric Monoidal Closed categories [Sol93]. Rittri [Rit90, Rit91, Rit93] used both kinds of isomorphism in his work on using types as search keys. The following table summarizes some decidability results for T_{CC} and T_{SMC}.

Axioms	Word problem	Matching problem	Unification problem
T_{CC}	$n^2 \log(n)$ [Con00]	NP-hard, decidable [NPS93]	Undecidable [NPS93]
T_{SMC}	$n \log^2(n)$ [AS97]	NP [NPS93]	NP-complete [NPS93]

One approach to deciding whether two types are isomorphic in T_{CC} is based on reducing both types to normal forms. Bruce, Di Cosmo and Longo defined a notion of normal form and proved its properties. The idea is to repeatedly apply the set of reduction rules $\mathbf{R}$ in Figure 2.7 until it no longer applies. Isomorphism of types in normal form is defined by

$$A \vdash \sigma \times \tau = \tau \times \sigma \quad (\textsc{Com}\times)$$

$$A \vdash \sigma \times (\tau \times \eta) = (\sigma \times \tau) \times \eta \quad (\textsc{Assoc}\times)$$

$$A \vdash (\sigma \times \tau) \to \eta = \sigma \to (\tau \to \eta) \quad (\textsc{Curry})$$

$$A \vdash \sigma \to (\tau \times \eta) = (\sigma \to \tau) \times (\sigma \to \eta) \quad (\textsc{Distrib}\to\ \times)$$

$$A \vdash \sigma \times \mathbf{T} = \sigma \quad (\textsc{Ident}\ \times)$$

$$A \vdash \sigma \to \mathbf{T} = \mathbf{T} \quad (\textsc{Unit})$$

$$A \vdash \mathbf{T} \to \sigma = \sigma \quad (\textsc{Ident}\ \to)$$

$$A \vdash \sigma = \sigma \quad (\textsc{Ref})$$

$$\frac{A \vdash \sigma = \eta \;\; A \vdash \eta = \tau}{A \vdash \sigma = \tau} \quad (\textsc{Trans})$$

$$\frac{A \vdash \sigma = \tau}{A \vdash \tau = \sigma} \quad (\textsc{Sym})$$

$$\frac{A \vdash \sigma_1 = \tau_1 \;\; A \vdash \sigma_2 = \tau_2}{A \vdash \sigma_1 \to \sigma_2 = \tau_1 \to \tau_2} \quad (\textsc{Cong}\to)$$

$$\frac{A \vdash \sigma_1 = \tau_1 \;\; A \vdash \sigma_2 = \tau_2}{A \vdash \sigma_1 \times \sigma_2 = \tau_1 \times \tau_2} \quad (\textsc{Cong}\times)$$

Figure 2.6. T_{CC}

associativity and commutativity of $\times$. Let $\mathsf{nf}(\tau)$ be the normal form of type τ such that

$$\mathsf{nf}(\tau) = \begin{cases} \mathbf{T}, \text{ or a base type, or a function type, or} \\ \tau_1 \times \tau_2 \times \ldots \times \tau_n \end{cases}$$

where the τ_i's are in normal form. We can use the abbreviation $\prod_{i=1}^{n} \tau_i$ for $\tau_1 \times \tau_2 \times \ldots \times \tau_n$ to emphasize that the order of the τ_i's is not important; a product in normal form can be viewed a bag (multi-set) of factors. We can decide equality of two types in normal form with a straightforward recursive algorithm which applies a bag-equality algorithm whenever it encounters a pair of product types. Notice that such an algorithm would not work for recursive types; it would not terminate.

$$\mathbf{R} = \begin{cases} \sigma \to (\tau \to \eta) & \Rightarrow & (\sigma \times \tau) \to \eta \\ \sigma \to (\tau \times \eta) & \Rightarrow & (\sigma \to \tau) \times (\sigma \to \eta) \\ \mathbf{T} \times \tau & \Rightarrow & \tau \\ \tau \times \mathbf{T} & \Rightarrow & \tau \\ \mathbf{T} \to \tau & \Rightarrow & \tau \\ \tau \to \mathbf{T} & \Rightarrow & \mathbf{T} \end{cases}$$

Figure 2.7. Set of rules for reducing non-recursive types into normal forms.

$$A \vdash \mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha] \quad \text{(UNFOLD/FOLD)}$$

$$A, \sigma = \tau, A' \vdash \sigma = \tau \quad \text{(HYP)}$$

$$A \vdash \sigma = \sigma \quad \text{(REF)}$$

$$\frac{A \vdash \sigma = \eta \quad A \vdash \eta = \tau}{A \vdash \sigma = \tau} \quad \text{(TRANS)}$$

$$\frac{A \vdash \sigma = \tau}{A \vdash \tau = \sigma} \quad \text{(SYM)}$$

$$\frac{A, \sigma_1 \to \sigma_2 = \tau_1 \to \tau_2 \vdash \sigma_1 = \tau_1 \quad A, \sigma_1 \to \sigma_2 = \tau_1 \to \tau_2 \vdash \sigma_2 = \tau_2}{A \vdash \sigma_1 \to \sigma_2 = \tau_1 \to \tau_2} \quad \text{(ARROW/FIX)}$$

$$\frac{A, \sigma_1 \times \sigma_2 = \tau_1 \times \tau_2 \vdash \sigma_1 = \tau_1 \quad A, \sigma_1 \times \sigma_2 = \tau_1 \times \tau_2 \vdash \sigma_2 = \tau_2}{A \vdash \sigma_1 \times \sigma_2 = \tau_1 \times \tau_2} \quad \text{(CROSS/FIX)}$$

Figure 2.8. T_R

Equality and subtyping of recursive types have been studied in the 1990s by Amadio and Cardelli [AC93]; Kozen, Palsberg, and Schwartzbach [KPS95]; Brandt and Henglein [BH97]; Jim and Palsberg [JP97]; and others. These papers concentrate on the case where two types are considered equal if and only if their infinite unfoldings are identical. This can be formalized using bisimulation [JP97, Par81]. Sound and complete axiomatizations have been presented by Amadio and Cardelli [AC93], and Brandt and Henglein [BH97]. Related axiomatizations have been presented by Milner [Mil84] and Kozen [Koz94]. This notion of type equality can be decided in $O(n\alpha(n))$ time, and a notion of subtyping defined by Amadio and Cardelli [AC93] can be decided in $O(n^2)$ time [KPS95].

The axiomatization by Brandt and Henglein [BH97], here denoted by T_R (R for Recursive), is shown in Figure 2.8. Auerbach, Barton, and Raghavachari [ABR98], in a quest for a foundation of the Mockingbird system, raised the question of whether $T_{CC} \cup T_R$ is consistent and decidable. They later discovered that this combined system is inconsistent, see also [AF96]. Thus, the isomorphism problem of recursive types cannot simply be defined by $T_{CC} \cup T_R$. Moreover, it seems like reduction by $\mathbf{R}$ may not terminate, for some recursive types.

In the following section we consider a notion of type equality where two types can be equal even if their infinite unfoldings are different. Intuitively, our notion of type equality is

$$T_R \cup \{\ (\textsc{Com}\times), (\textsc{Assoc}\times)\ \}.$$

A related system has been studied by Thatte [Tha96]. We will present several equivalent definitions of type equality, including one based on the axiomatization of Brandt and Henglein [BH97], and one based on the bisimulation approach of Jim and Palsberg [JP97].

2.4 Basic Definitions

In Section 2.5, we will use the notions of terms and term automata defined in [KPS95]. For the convenience of the reader, this section provides an excerpt of the relevant material from [KPS95]. Our algorithm relies on that the types to be matched are represented as term automata.

2.4.1 Terms

Here we give a general definition of (possibly infinite) terms over an arbitrary and finite ranked alphabet Σ. Such terms are essentially labeled trees, which we represent as partial functions labeling strings over ω (the natural numbers) with elements of Σ.

Let Σ_n denote the set of elements of Σ of arity n. Let ω denote the set of natural numbers and let ω^* denote the set of finite-length strings over ω.

A *term* over Σ is a partial function

$$t : \omega^* \quad \rightarrow \quad \Sigma$$

with domain $\mathcal{D}(t)$ satisfying the following properties:

- $\mathcal{D}(t)$ is nonempty and prefix-closed;
- if $t(\alpha) \in \Sigma_n$, then $\{i \mid \alpha i \in \mathcal{D}(t)\} = \{0, 1, \ldots, n-1\}$.

Let t be a term and $\alpha \in \omega^*$. Define the partial function $t\downarrow\alpha : \omega^* \rightarrow \Sigma$ by

$$t\downarrow\alpha(\beta) \quad = \quad t(\alpha\beta)\ .$$

If $t\downarrow\alpha$ has nonempty domain, then it is a term, and is called the *subterm of* t *at position* α.

A term t is said to be *regular* if it has only finitely many distinct subterms; *i.e.*, if $\{t\downarrow\alpha \mid \alpha \in \omega^*\}$ is a finite set.

2.4.2 Term Automata

Every regular term over a finite ranked alphabet Σ has a finite representation in terms of a special type of automaton called a *term automaton*.

Definition 2.4.1. Let Σ be a finite ranked alphabet. A *term automaton* over Σ is a tuple

$$\mathcal{M} \quad = \quad (Q,\ \Sigma,\ q_0,\ \delta,\ \ell)$$

where:

- Q is a finite set of *states*,
- $q_0 \in Q$ is the *start state*,

- $\delta : Q \times \omega \to Q$ is a partial function called the *transition function*, and
- $\ell : Q \to \Sigma$ is a (total) *labeling function*,

such that for any state $q \in Q$, if $\ell(q) \in \Sigma_n$ then

$$\{i \mid \delta(q, i) \text{ is defined}\} \quad = \quad \{0, 1, \ldots, n-1\} \; .$$

We decorate Q, δ, *etc*. with the superscript $\mathcal{M}$ where necessary. □

Let $\mathcal{M}$ be a term automaton as in Definition 2.4.1. The partial function δ extends naturally to a partial function

$$\hat{\delta} : Q \times \omega^* \to Q$$

inductively as follows:

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= q \\ \hat{\delta}(q, \alpha i) &= \delta(\hat{\delta}(q, \alpha), i) \; . \end{aligned}$$

For any $q \in Q$, the domain of the partial function $\lambda\alpha.\hat{\delta}(q, \alpha)$ is nonempty (it always contains ϵ) and prefix-closed. Moreover, because of the condition on the existence of i-successors in Definition 2.4.1, the partial function

$$\lambda\alpha.\ell(\hat{\delta}(q, \alpha))$$

is a term.

Definition 2.4.2. Let $\mathcal{M}$ be a term automaton. The term *represented by* $\mathcal{M}$ is the term

$$t_{\mathcal{M}} \quad = \quad \lambda\alpha.\ell(\hat{\delta}(q_0, \alpha)) \; .$$

A term t is said to be *representable* if $t = t_{\mathcal{M}}$ for some $\mathcal{M}$. □

Intuitively, $t_{\mathcal{M}}(\alpha)$ is determined by starting in the start state q_0 and scanning the input α, following transitions of $\mathcal{M}$ as far as possible. If it is not possible to scan all of α because some i-transition along the way does not exist, then $t_{\mathcal{M}}(\alpha)$ is undefined. If on the other hand $\mathcal{M}$ scans the entire input α and ends up in state q, then $t_{\mathcal{M}}(\alpha) = \ell(q)$.

Lemma 2.4.3. *Let t be a term. The following are equivalent:*

(i) *t is regular;*

(ii) *t is representable;*

(iii) *t is described by a finite set of equations involving the μ operator.*

2.5 Type Equality

In this section, we define a notion of type equality where the product-type constructor is associative and commutative, and we present an efficient decision procedure.

In Section 2.5.1 we define our notion of type, and in Sections 2.5.2 and 2.5.3 we give some preliminaries about bipartite graphs and fixed points needed later. In Section 2.5.4 we present our notion of type equality, in Section 2.5.5 we show a convenient characterization of type equality, and in Section 2.5.6 we present an efficient decision procedure.

2.5.1 Recursive Types

A type is a regular term over the ranked alphabet

$$\Sigma = \Gamma \ \cup \ \{\rightarrow\} \ \cup \ \{\prod^{n}, n \geq 2\},$$

where Γ is a set of base types, $\rightarrow$ is binary, and $\prod^n$ is of arity n. With the notation of Appendix 2.4, the root symbol of a type t is written $t(\epsilon)$.

We impose the restriction that given a type σ and a path α, if $\sigma(\alpha) = \prod^n$, then $\sigma(\alpha i) \in \Gamma \cup \{\rightarrow\}$, for all $i \in \{1..n\}$. The set of types is denoted $\mathcal{T}$. Given a type σ, if $\sigma(\epsilon) = \rightarrow$, $\sigma(0) = \sigma_1$, and $\sigma(1) = \sigma_2$, then we write the type as $\sigma_1 \rightarrow \sigma_2$. If $\sigma(\epsilon) = \prod^n$ and $\sigma(i) = \sigma_{i+1}\ \forall i \in \{0, 1, \ldots, n-1\}$, then we write the type σ as $\prod_{i=1}^{n} \sigma_i$.

Intuitively, our restriction means that products cannot be immediately nested, that is, one cannot form a product one of whose immediate components is again a product. We impose this restriction for two reasons:

i) it effectively rules out infinite products such as $\mu\alpha.(int \times \alpha)$, and

ii) it ensures that types are in a "normal form" with respect to associativity, that is, the issue of associativity is reduced to a matter of the order of the components in a $\prod_{i=1}^{n} \sigma_i$ type.

Currently, we are unable to extend our algorithm to handle infinite products. Types without infinite products can easily be "flattened" to conform to our restriction.

For Java interfaces, our restriction has no impact. We model interfaces using one kind of product-type constructor, we model argument-type lists using *another* kind of product-type constructor, and we model method types using the function-type constructor. The syntax of Java interfaces ensures that a straightforward translation of a Java interface to our representation of types will automatically satisfy our restriction.

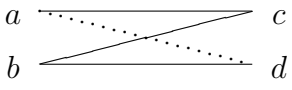
2.5.2 Bipartite Graphs

A bipartite graph (V, W, R) is given by two sets V, W of vertices, and a set $R \subseteq V \times W$ of undirected edges.

For our application, we will only be interested in bipartite graphs where the edge sets are *coherent*. A relation R is coherent iff

$$\text{if } (a, c), (b, c), (b, d) \in R, \text{ then } (a, d) \in R.$$

It can be illustrated by the following picture,



where the edges (a, c), (b, c), and (b, d) imply the existence of the edge (a, d).

Lemma 2.5.1. *Suppose $\mathcal{G} = (V, W, R)$ is a bipartite graph where R is coherent. If $a \in V$ can reach $d \in W$, then $(a, d) \in R$.*

Proof. Suppose $a \in V$ can reach $d \in W$ in k steps. Since all the edges are between V and W, each step will move from one set to the other. Therefore, k must be an odd number and let $k = 2 * n + 1$, $n \geq 0$.

We proceed by induction on n.

$(n = 0)$ We have that a can reach d in one step, so $(a, d) \in R$.

Suppose the Lemma holds for $n = m > 0$

$(n = m+1)$ We have that a can reach d in $2*m+3$ steps. Let c and b be the $(2*m+1)$th and $(2*m+2)$th nodes a reaches along the path to d, then $(b,c),(b,d) \in R$. By the induction hypothesis, $(a,c) \in R$. Consequently, $(a,d) \in R$ by the coherence property of R.

□

Definition 2.5.2. Suppose $\prod_{i=1}^{n} \sigma_i, \prod_{i=1}^{n} \tau_i$ are two types and R is a relation on types. The matching function $match(\prod_{i=1}^{n} \sigma_i, \prod_{i=1}^{n} \tau_i, R)$ is *true* iff there exists a bijection $t : \{1..n\} \rightarrow \{1..n\}$ such that $\forall i$, $(\sigma_i, \tau_{t(i)}) \in R$. □

Lemma 2.5.1 enables a simple algorithm for $match(\prod_{i=1}^{n} \sigma_i, \prod_{i=1}^{n} \tau_i, R)$ where R is coherent and finite. Let V, W be two finite sets such that $\sigma_i \in V$, for all $i \in \{1..n\}$, $\tau_i \in W$, for all $i \in \{1..n\}$, and $R \subseteq V \times W$. Let $N = |V| + |W|$. The bipartite graph (V, W, R) has at most N connected components, $B_1, B_2, \ldots$, and we label them with numbers starting at 1. Thus, all the numbers are in the set $\{1..N\}$.

Define a function $I : (V \cup W) \rightarrow \{1..N\}$, where $I(\sigma) = i$ iff $\sigma \in B_i$. Two types σ and τ are in the same connected component iff σ can reach τ in (V, W, R). Thus, by Lemma 2.5.1, we have $(\sigma, \tau) \in R$ iff $I(\sigma) = I(\tau)$.

Let $[.]$ denotes a multi-set of elements.

Lemma 2.5.3. $match(\prod_{i=1}^{n} \sigma_i, \prod_{i=1}^{n} \tau_i, R)$ *is* true *iff*

$$[I(\sigma_1), I(\sigma_2), .., I(\sigma_n)] = [I(\tau_1), I(\tau_2), .., I(\tau_n)].$$

Proof. If $[I(\sigma_1), I(\sigma_2), .., I(\sigma_n)] = [I(\tau_1), I(\tau_2), .., I(\tau_n)]$, then there exists bijection $t : \{1..n\} \rightarrow \{1..n\}$, such that $\forall i$, $I(\sigma_i) = I(\tau_{t(i)})$. By the definition of I, vertex σ_i can reach vertex $\tau_{t(i)}$; thus, by Lemma 2.5.1, $(\sigma_i, \tau_{t(i)}) \in R, \forall i$. Therefore, $match(\prod_{i=1}^{n} \sigma_i, \prod_{i=1}^{n} \tau_i, R)$ is *true*.

Suppose $match(\prod_{i=1}^{n} \sigma_i, \prod_{i=1}^{n} \tau_i, R)$ is *true*. There exists bijection t such that $(\sigma_i, \tau_{t(i)}) \in R, \forall i$. Thus, $I(\sigma_i) = I(\tau_{t(i)})$ since σ_i and $\tau_{t(i)}$ are connected. Since $[I(\tau_1), I(\tau_2), .., I(\tau_n)] = [I(\tau_{t(1)}), I(\tau_{t(2)}), .., I(\tau_{t(n)})]$, we have $[I(\sigma_1), I(\sigma_2), .., I(\sigma_n)] = [I(\tau_1), I(\tau_2), .., I(\tau_n)]$. □

2.5.3 Monotone Functions and Fixed Points

We now recall the notion of a greatest fixed point of a monotone function, and we prove three basic results about greatest fixed points that will be needed in Section 2.5.5.

Let $\mathcal{P}$ denote the unary operator which maps a set to its power-set. Consider the lattice $(\mathcal{P}(Z), \subseteq)$ and a function

$$F \;:\; \mathcal{P}(Z) \to \mathcal{P}(Z).$$

We say that F is monotone iff if $z_1 \subseteq z_2$, then $F(z_1) \subseteq F(z_2)$. If F is monotone, then Tarski's fixed point theorem [Tar55] gives that F has a greatest fixed point νF given by:

$$\nu F \;=\; \bigcup\{\, X \mid X \subseteq F(X) \,\}.$$

Suppose F is monotone, and $\mathcal{Z} \subseteq Z$. In Section 2.5.5, we will be particularly interested in a case where $\mathcal{Z}$ is finite and Z is infinite. Define

$$\begin{aligned} H &\in \mathcal{P}(\mathcal{Z}) \to \mathcal{P}(\mathcal{Z}) \\ H(X) &= F(X) \cap \mathcal{Z}. \end{aligned}$$

Lemma 2.5.4. $\nu H \subseteq \nu F \cap \mathcal{Z}$.

Proof.

$$\begin{aligned} \nu H &= \bigcup\{\, X \mid X \subseteq H(X) \,\} \\ &= \bigcup\{\, X \mid X \subseteq F(X) \cap \mathcal{Z} \,\} \\ &= (\bigcup\{\, X \mid X \subseteq F(X) \cap \mathcal{Z} \,\}) \cap \mathcal{Z} \\ &\subseteq (\bigcup\{\, X \mid X \subseteq F(X) \,\}) \cap \mathcal{Z} \\ &= \nu F \cap \mathcal{Z}. \end{aligned}$$

□

The converse of Lemma 2.5.4 may be false. For example, consider

$$Z \;=\; \{1, 2\}$$

$$\begin{aligned} \mathcal{Z} &= \{1\} \\ F(\{1,2\}) &= \{1,2\} \\ F(\{1\}) &= F(\{2\}) = F(\emptyset) = \emptyset. \end{aligned}$$

We have that F is monotone, $\nu F = \{1,2\}$, and $\nu H = \emptyset$. We conclude that $\nu F \cap \mathcal{Z} = \{1,2\} \cap \{1\} = \{1\} \not\subseteq \emptyset = \nu H$.

We now give a sufficient condition under which the converse of Lemma 2.5.4 is true.

Lemma 2.5.5. *Suppose that if $X \subseteq F(X)$, then $F(X) \cap \mathcal{Z} \subseteq F(X \cap \mathcal{Z})$. We have $\nu F \cap \mathcal{Z} \subseteq \nu H$.*

Proof. From $X \subseteq F(X)$ we have

$$X \cap \mathcal{Z} \subseteq F(X) \cap \mathcal{Z} \subseteq F(X \cap \mathcal{Z}).$$

Now we can calculate as follows:

$$\begin{aligned} \nu F \cap \mathcal{Z} &= \bigcup\{\, X \mid X \subseteq F(X) \,\} \cap \mathcal{Z} \\ &= \bigcup\{\, Y \mid \exists X : (Y = X \cap \mathcal{Z}) \wedge (X \subseteq F(X)) \,\} \\ &\subseteq \bigcup\{\, Y \mid \exists X : (Y = X \cap \mathcal{Z}) \wedge (X \cap \mathcal{Z} \subseteq F(X \cap \mathcal{Z})) \,\} \\ &= \bigcup\{\, Y \mid \exists X : (Y = X \cap \mathcal{Z}) \wedge (Y \subseteq F(Y)) \,\} \\ &= \bigcup\{\, Y \mid (Y \subseteq \mathcal{Z}) \wedge (Y \subseteq F(Y)) \,\} \\ &= \bigcup\{\, Y \mid Y \subseteq F(Y) \cap \mathcal{Z} \,\} \\ &= \bigcup\{\, Y \mid Y \subseteq H(Y) \,\} \\ &= \nu H. \end{aligned}$$

□

If S is finite, then a well-known characterization of νF is given by:

$$\nu F = \bigcap_{i=0}^{\infty} F^i(S).$$

Lemma 2.5.6. *If H is a monotone function from $(\mathcal{P}(V \times W), \subseteq)$ to itself, where V, W are finite and $N = |V| + |W|$, and if for all non-negative integers i, $H^i(V \times W)$ is coherent, then $\nu H = H^N(V \times W)$.*

Proof. Let $X = (V \times W)$. Since H is monotone, $H^{i+1}(X) \subseteq H^i(X)\ \forall i \geq 0$.

If $H^{i+1}(X) = H^i(X)$, then $H^i(X)$ is a fixed point of H and $H^j(X) = H^i(X), \forall\, j > i$. Otherwise, if $H^{i+1}(X) \subset H^i(X)$, then $H^{i+1}(X) \subset \ldots \subset H^1(X) \subset X$.

Suppose $H^{i+1}(X) \subset H^i(X)$ and $(v, w) \in (H^i(X) \cap \neg H^{i+1}(X))$. We construct the bipartite graph $\mathcal{G}^i = (V, W, H^i(X))$. Each connected component of $\mathcal{G}^i$ corresponds to one or more connected component in $\mathcal{G}^{i+1}$, because any set of vertices that are connected in $\mathcal{G}^{i+1}$ are connected in $\mathcal{G}^i$ as well.

Since $(v, w) \in H^i(X)$, v, w are in the same connected component of $\mathcal{G}^i$. From $(v, w) \in \neg H^{i+1}(X)$ and Lemma 2.5.1, v cannot reach w in $\mathcal{G}^{i+1}$. Therefore, v and w are in separate connected components of $\mathcal{G}^{i+1}$. Consequently, $\mathcal{G}^{i+1}$ has at least one more connected component than $\mathcal{G}^i$.

Consider $\{H^i(X)\}_{i=0}^k$ such that $H^k(X) \subset \ldots \subset H^1(X) \subset X$. Then the bipartite graph $\mathcal{G}^k$ has at least k connected components. However, $\mathcal{G}^k$ can have at most N connected components, which is the case when there is no edge in the graph and each vertex forms a connected component. Thus, $k \leq N$ and $H^N(X) = H^{N+1}(X)$.

We conclude that $\nu H = \bigcap_{i=0}^{\infty} H^i(X) = \bigcap_{i=0}^{N} H^i(X) = H^N(X)$. □

2.5.4 Type Equality

We now give three equivalent definitions of type equality. They will be denoted $\mathbf{EQ}, \mathcal{R}, \nu F$.

The first definition is based on the rule set T_{RAC} (R for Recursive, A for Associative, and C for Commutative) in Figure 2.9. The rule $(\prod/\textsc{Fix})$ entails that the product-type constructor is associative and commutative. Define

$$\mathbf{EQ} \;=\; \{\ (\sigma, \tau)\ \mid\ \emptyset \vdash \sigma = \tau\ \}.$$

The second definition of type equality is based on the idea of bisimilarity. A relation R on types is called a *bisimulation* if it satisfies the following three conditions:

$$A, \sigma = \tau, A' \vdash \sigma = \tau \quad (\textsc{Hyp})$$

$$A \vdash \sigma = \sigma \quad (\textsc{Ref})$$

$$\frac{A, \sigma_1 \to \sigma_2 = \tau_1 \to \tau_2 \vdash \sigma_1 = \tau_1 \;\; A, \sigma_1 \to \sigma_2 = \tau_1 \to \tau_2 \vdash \sigma_2 = \tau_2}{A \vdash \sigma_1 \to \sigma_2 = \tau_1 \to \tau_2} \quad (\to/\textsc{Fix})$$

$$\frac{A, \prod_{i=1}^n \sigma_i = \prod_{i=1}^n \tau_i \vdash \sigma_i = \tau_{t(i)},\ i \in \{1..n\}}{A \vdash \prod_{i=1}^n \sigma_i = \prod_{i=1}^n \tau_i} \quad (\textstyle\prod/\textsc{Fix})$$

where $t : \{1..n\} \to \{1..n\}$ is a bijection

Figure 2.9. T_{RAC}.

(C**)** If $(\sigma, \tau) \in R$, then $\sigma(\epsilon) = \tau(\epsilon)$.

($P1$**)** If $(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) \in R$, then $(\sigma_1, \tau_1) \in R$ and $(\sigma_2, \tau_2) \in R$.

($P2$**)** If $(\prod_{i=1}^n \sigma_i, \prod_{i=1}^n \tau_i) \in R$, then $match(\prod_{i=1}^n \sigma_i, \prod_{i=1}^n \tau_i, R)$ is *true*.

A relation R is said to be *consistent* if it satisfies property C, and it is said to be *closed* if it satisfies $P1, P2$. Bisimulations are closed under union, therefore, there exists a largest bisimulation

$$\mathcal{R} \;=\; \bigcup \{\; R \;\mid\; R \text{ is a bisimulation } \}.$$

The third definition of type equality is based on the notion of greatest fixed points. Define

$$\begin{aligned} F \;&\in\; \mathcal{P}(\mathcal{T} \times \mathcal{T}) \to \mathcal{P}(\mathcal{T} \times \mathcal{T}) \\ F \;&=\; \lambda R.\{\; (\sigma, \tau) \;\mid\; \sigma, \tau \text{ are base types and } \sigma(\epsilon) = \tau(\epsilon) \;\} \\ &\qquad \cup \{\; (\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) \;\mid\; (\sigma_1, \tau_1), (\sigma_2, \tau_2) \in R \;\} \\ &\qquad \cup \{\; (\Pi_{i=1}^n \sigma_i, \Pi_{i=1}^n \tau_i) \;\mid\; match(\Pi_{i=1}^n \sigma_i, \Pi_{i=1}^n \tau_i, R) \;\} \end{aligned}$$

Notice that F is monotone so it has a greatest fixed point νF.

Lemma 2.5.7. *R is a bisimulation iff $R \subseteq F(R)$.*

Proof. Suppose first that R is a bisimulation. For every type pair $(\sigma, \tau) \in R$, if σ, τ are base types, then $\sigma(\epsilon) = \tau(\epsilon)$, so $(\sigma, \tau) \in F(R)$. If $\sigma = \sigma_1 \to \sigma_2$, $\tau = \tau_1 \to \tau_2$, then $(\sigma_1, \tau_1), (\sigma_2, \tau_2) \in R$, so $(\sigma, \tau) \in F(R)$. Similarly for $\sigma = \prod_{i=1}^{n} \sigma_i, \tau = \prod_{i=1}^{n} \tau_i$.

Conversely, suppose that $R \subseteq F(R)$. It is straightforward to prove that R is a bisimulation; we omit the details. □

Theorem 2.5.8. $\mathbf{EQ} = \mathcal{R} = \nu F$.

Proof. For a proof of $\mathbf{EQ} = \mathcal{R}$, see Appendix A.1. From Lemma 2.5.7 we have

$$\begin{aligned} \mathcal{R} &= \bigcup \{\, R \mid R \text{ is a bisimulation} \,\} \\ &= \bigcup \{\, R \mid R \subseteq F(R) \,\} = \nu F. \end{aligned}$$

□

We may apply the principle of *co-induction* to prove that two types are related in $\mathcal{R}$. That is, to show $(\sigma, \tau) \in \mathcal{R}$, it is sufficient to find a bisimulation R such that $(\sigma, \tau) \in R$.

Theorem 2.5.9. *$\mathcal{R}$ is a congruence relation.*

Proof. By co-induction, see appendix A.2. □

Theorem 2.5.9 implies that $\mathcal{R}$ is an equivalence relation. Two types τ_1 and τ_2 are said to be *equivalent* (denoted by $\tau_1 \cong \tau_2$) iff $(\tau_1, \tau_2) \in \mathcal{R}$.

2.5.5 A Characterization of Type Equality

In this section we prove that type equality can be decided by an iterative method (Theorem 2.5.15). To prove this result, we need five lemmas which establish that coherence is preserved by one step of iteration (Lemmas 2.5.10, 2.5.11, 2.5.12), and that it is sufficient to concentrate on the types that are subtrees of the input types (Lemmas 2.5.13, 2.5.14).

Lemma 2.5.10. *If $R \subseteq (\mathcal{T} \times \mathcal{T})$ is coherent, then $F(R)$ is coherent.*

Proof. First, notice that if $(\sigma, \tau) \in F(R)$, then $\sigma(\epsilon) = \tau(\epsilon)$ by the definition of F.

Suppose $(a, c), (b, c), (b, d) \in F(R)$, we want to show that $(a, d) \in F(R)$. There are three cases.

i) $a..d$ are base types. We have $a(\epsilon) = c(\epsilon) = b(\epsilon) = d(\epsilon)$, so $(a, d) \in F(R)$.

ii) $a..d$ are $\rightarrow$ types. Suppose $a = a_1 \rightarrow a_2$, $b = b_1 \rightarrow b_2$, $c = c_1 \rightarrow c_2$, and $d = d_1 \rightarrow d_2$. We have (a_i, c_i), (b_i, c_i) and $(b_i, d_i) \in R$, $i = 1, 2$. Since R is coherent, $(a_i, d_i) \in R$, $i = 1, 2$, which means $(a, d) \in F(R)$.

iii) $a..d$ are product types. Suppose $a = \prod_{i=1}^{n} a_i$, $b = \prod_{i=1}^{n} b_i$, $c = \prod_{i=1}^{n} c_i$, and $d = \prod_{i=1}^{n} d_i$. We have $(a, c) \in R$ and $match(\prod_{i=1}^{n} a_i, \prod_{i=1}^{n} c_i, R)$ is true. The same applies to (b, c) and (b, d). Therefore, $\exists$ bijections s, t, u from $\{1..n\}$ to $\{1..n\}$ such that $(a_i, c_{s(i)})$, $(b_i, c_{t(i)})$, $(b_i, d_{u(i)}) \in R, \forall i$. Let bijection $v = u \circ t^{-1} \circ s$, we have $(a_i, d_{v(i)}) \in R \; \forall i$, since R is coherent. Thus, $match(\prod_{i=1}^{n} a_i, \prod_{i=1}^{n} d_i, R)$ is true. and $(a, d) \in F(R)$.

□

For $\sigma \in \mathcal{T}$, define

$$V_\sigma = \{\ \tau \ \mid\ \tau \text{ is a subterm of } \sigma\ \}.$$

Given σ, τ, define

$$\begin{aligned} H &\in \mathcal{P}(V_\sigma \times V_\tau) \rightarrow \mathcal{P}(V_\sigma \times V_\tau) \\ H &= \lambda R.(F(R) \cap (V_\sigma \times V_\tau)). \end{aligned}$$

Lemma 2.5.11. *If $R \subseteq (V_\sigma \times V_\tau)$ is coherent, then $H(R)$ is coherent.*

Proof. By the definition of H, we have $H(R) = F(R) \cap (V_\sigma \times V_\tau)$.

Since $R \subseteq (V_\sigma \times V_\tau) \subset (\mathcal{T} \times \mathcal{T})$, by Lemma 2.5.10, $F(R)$ is coherent. Thus, if $(a, c), (b, c), (b, d) \in F(R) \cap (V_\sigma \times V_\tau)$, then $(a, d) \in F(R)$ and $(a, d) \in (V_\sigma \times V_\tau)$ because $a \in V_\sigma$ and $d \in V_\tau$. Therefore, $(a, d) \in F(R) \cap (V_\sigma \times V_\tau)$, and $H(R)$ is coherent.

□

Lemma 2.5.12. *For all n, $H^n(V_\sigma \times V_\tau)$ is coherent.*

Proof. We proceed by induction on n.

For $n = 0$, we have $H^0(V_\sigma \times V_\tau) = (V_\sigma \times V_\tau)$. If $(a, c), (b, c), (b, d) \in (V_\sigma \times V_\tau)$, then $(a, d) \in (V_\sigma \times V_\tau)$ since $a \in V_\sigma$ and $d \in V_\tau$.

Suppose $H^n(V_\sigma \times V_\tau)$ is coherent. Since $H^n(V_\sigma \times V_\tau) \subseteq (V_\sigma \times V_\tau)$, we know that $H(\ H^n(V_\sigma \times V_\tau)\ )$ is coherent, by Lemma 2.5.11. □

Lemma 2.5.13. $F(R) \cap (V_\sigma \times V_\tau) \subseteq F(R \cap (V_\sigma \times V_\tau))$.

Proof. Let $\mathcal{Z} = (V_\sigma \times V_\tau)$.

$$
\begin{aligned}
& F(R) \ \cap \ \mathcal{Z} \\
&= \ \{\ (\sigma', \tau') \in \mathcal{Z} \ \mid \ \sigma', \tau' \text{ are base types and } \sigma'(\epsilon) = \tau'(\epsilon)\ \} \\
&\quad \cup \ \{\ (\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) \in \mathcal{Z} \ \mid \ (\sigma_1, \tau_1), (\sigma_2, \tau_2) \in R\ \} \\
&\quad \cup \ \{\ (\Pi_{i=1}^n \sigma_i, \Pi_{i=1}^n \tau_i) \in \mathcal{Z} \ \mid \ match(\Pi_{i=1}^n \sigma_i,\ \Pi_{i=1}^n \tau_i,\ R)\ \} \\
&= \ \{\ (\sigma', \tau') \in \mathcal{Z} \ \mid \ \sigma', \tau' \text{ are base types and } \sigma'(\epsilon) = \tau'(\epsilon)\ \} \\
&\quad \cup \ \{\ (\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) \in \mathcal{Z} \ \mid \ (\sigma_1, \tau_1), (\sigma_2, \tau_2) \in R \cap \mathcal{Z})\ \} \\
&\quad \cup \ \{\ (\Pi_{i=1}^n \sigma_i, \Pi_{i=1}^n \tau_i) \in \mathcal{Z} \ \mid \ match(\Pi_{i=1}^n \sigma_i,\ \Pi_{i=1}^n \tau_i,\ R \cap \mathcal{Z})\ \} \\
&\subseteq \ \{\ (\sigma', \tau') \ \mid \ \sigma', \tau' \text{ are base types and } \sigma'(\epsilon) = \tau'(\epsilon)\ \} \\
&\quad \cup \ \{\ (\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) \ \mid \ (\sigma_1, \tau_1), (\sigma_2, \tau_2) \in R \cap \mathcal{Z})\ \} \\
&\quad \cup \ \{\ (\Pi_{i=1}^n \sigma_i, \Pi_{i=1}^n \tau_i) \ \mid \ match(\Pi_{i=1}^n \sigma_i,\ \Pi_{i=1}^n \tau_i,\ R \cap \mathcal{Z})\ \} \\
&= \ F(R \cap \mathcal{Z});
\end{aligned}
$$

□

Lemma 2.5.14. $\nu H = \nu F \cap (V_\sigma \times V_\tau)$.

Proof. By Lemma 2.5.4, we have $\nu H \subseteq \nu F \cap (V_\sigma \times V_\tau)$. By Lemma 2.5.13, $F(R) \cap (V_\sigma \times V_\tau) \subseteq F(R \cap (V_\sigma \times V_\tau))$. Therefore, by Lemma 2.5.5, we also have $\nu H \supseteq \nu F \cap (V_\sigma \times V_\tau)$.

□

Theorem 2.5.15. $(\sigma, \tau) \in \mathcal{R}$ *iff* $(\sigma, \tau) \in H^N(V_\sigma \times V_\tau)$*, where* $N = |V_\sigma| + |V_\tau|$.

Proof. From $(\sigma, \tau) \in (V_\sigma \times V_\tau)$ we have that $(\sigma, \tau) \in \mathcal{R}$ iff $(\sigma, \tau) \in \mathcal{R} \cap (V_\sigma \times V_\tau)$. Moreover, from Theorem 2.5.8 and Lemma 2.5.14 we have

$$\mathcal{R} \cap (V_\sigma \times V_\tau) = \nu F \cap (V_\sigma \times V_\tau) = \nu H.$$

Finally, Lemma 2.5.12 shows that $H^i(V_\sigma \times V_\tau)$ is coherent for all i, so by Lemma 2.5.6, $\nu H = H^N(V_\sigma \times V_\tau)$. □

2.5.6 Algorithm and Complexity

We can use Theorem 2.5.15 to give an algorithm for deciding type equality. Given a type pair (σ, τ), we can decide $(\sigma, \tau) \in \mathcal{R}$ by deciding $(\sigma, \tau) \in H^N(V_\sigma \times V_\tau)$, where $N = |V_\sigma| + |V_\tau|$. To do this, we need to apply H at most N times. In each round, according to Lemma 2.5.12, H will be applied to a coherent relation R, where $H(R)$ is also coherent. Thus, we only need to represent coherent relations. We will now present such a representation scheme, and we will show that given a representation of R, we can efficiently compute a representation of $H(R)$.

Given a coherent relation R, we represent R by a function

$$I : (V_\sigma \cup V_\tau) \to \{1..N\},$$

where $(\sigma', \tau') \in R$ iff $I(\sigma') = I(\tau')$. The existence of such a representation was established in Section 2.5.2. The abstraction function abs maps a function I to the relation represented by I:

$$abs(I) = \{\ (\sigma', \tau') \in (V_\sigma \times V_\tau)\ \mid\ I(\sigma') = I(\tau')\ \}.$$

Since I represents R, we want to define $\mathcal{H}(I)$ as a representation of $H(R)$. The function $\mathcal{H}$ has the following properties:

$$\begin{aligned} \mathcal{H}(I)(\sigma') &= \mathcal{H}(I)(\tau') \\ &\Leftrightarrow \sigma'(\epsilon) = \tau'(\epsilon) \\ \mathcal{H}(I)(\sigma_1 \to \sigma_2) &= \mathcal{H}(I)(\tau_1 \to \tau_2) \end{aligned}$$

$$\begin{aligned} & \Leftrightarrow \quad I(\sigma_1) = I(\tau_1) \wedge I(\sigma_2) = I(\tau_2) \\ \mathcal{H}(I)(\Pi_{i=1}^n \sigma_i) \quad & = \quad \mathcal{H}(I)(\Pi_{i=1}^n \tau) \\ & \Leftrightarrow \quad [I(\sigma_1), \ldots, I(\sigma_n)] = [I(\tau_1), \ldots, I(\tau_n)], \end{aligned}$$

where σ', τ' are base types.

Any such function $\mathcal{H}$ satisfies the following lemma 2.5.16, which states that we can compute a representation of the result of applying H to the relation represented by I, by computing $\mathcal{H}(I)$.

Lemma 2.5.16. $H(abs(I)) = abs(\mathcal{H}(I))$.

Proof. Suppose $(\sigma', \tau') \in H(abs(I))$. We have $\sigma'(\epsilon) = \tau'(\epsilon)$ by definition of H and F. There are three cases.

i) σ', τ' are base types. Since $\mathcal{H}(I)(\sigma') = \mathcal{H}(I)(\tau') \Leftrightarrow \sigma'(\epsilon) = \tau'(\epsilon)$, we have $(\sigma', \tau') \in abs(\mathcal{H}(I))$.

ii) σ', τ' are $\rightarrow$ types. Suppose that $\sigma' = \sigma_1 \rightarrow \sigma_2$ and $\tau' = \tau_1 \rightarrow \tau_2$. We have $(\sigma_1, \tau_1), (\sigma_2, \tau_2) \in abs(I)$. By the definition of $abs(I)$, $I(\sigma_1) = I(\tau_1)$ and $I(\sigma_2) = I(\tau_2)$. Hence, $\mathcal{H}(I)(\sigma_1 \rightarrow \sigma_2) = \mathcal{H}(I)(\tau_1 \rightarrow \tau_2)$ and $(\sigma', \tau') \in abs(\mathcal{H}(I))$.

iii) σ', τ' are product types. Suppose that $\sigma' = \prod_{i=1}^n \sigma_i$ and $\tau' = \prod_{i=1}^n \tau_i$. We have $match(\sigma', \tau', abs(I))$ true. By Lemma 2.5.3 and the definition of $abs(I)$, we have $[I(\sigma_1), \ldots, I(\sigma_n)] = [I(\tau_1), \ldots, I(\tau_n)]$. Therefore, $\mathcal{H}(I)(\Pi_{i=1}^n \sigma_i) = \mathcal{H}(I)(\Pi_{i=1}^n \tau)$, and $(\sigma', \tau') \in abs(\mathcal{H}(I))$.

Conversely, if $(\sigma', \tau') \in abs(\mathcal{H}(I))$, we have $\mathcal{H}(I)(\sigma') = \mathcal{H}(I)(\tau')$. It is straightforward to show that $(\sigma', \tau') \in H(abs(I))$ by a case analysis as above. We omit the details. □

Here is a particular definition of an $\mathcal{H}$ which satisfies the three properties. Given I, we define $\mathcal{H}(I)$ in three steps:

i) Define $\sqsubseteq$ on $(V_\sigma \cup V_\tau)$ to be the smallest preorder which includes the following definitions. First,

$$\Gamma \sqsubseteq \sigma_1 \to \sigma_2 \sqsubseteq \prod_{i=1}^{n} \tau_i$$

for all base types Γ, all function types $\sigma_1 \to \sigma_2$, and all product types $\prod_{i=1}^{n} \tau_i$. Next, we choose some arbitrary linear ordering of the base types. Finally, we use I to further sort the function types, and to further sort the product types. The idea of the further sorting is to define a lexicographical order based on I. Given a string of k numbers $m_1 \ldots m_k$, the notation $sort(m_1 \ldots m_k)$ denotes a string of the same k numbers but now in increasing order.

$\sigma_1 \to \sigma_2 \sqsubseteq \tau_1 \to \tau_2$ iff $I(\sigma_1)I(\sigma_2)$ is lexicographically less than $I(\tau_1)I(\tau_2)$

$\prod_{i=1}^{n} \sigma_i \sqsubseteq \prod_{i=1}^{n} \tau_i$ iff $sort(I(\sigma_1) \ldots I(\sigma_n))$ is lexicographically less than $sort(I(\tau_1) \ldots I(\tau_n))$.

ii) Notice that $\sqsubseteq$ can be viewed as a directed graph. Number the strongly connected components of $\sqsubseteq$ in ascending order.

iii) Define $\mathcal{H}(I)(\sigma)$ to be the number of the strongly connected component to which σ belongs.

It is straightforward to show that the resulting $\mathcal{H}(I)$ satisfies the three properties listed earlier.

Let us now restate the definition of $\mathcal{H}(I)$ in a more algorithmic style. The main task is to sort the elements of $V_\sigma \cup V_\tau$ by $\sqsubseteq$. This is done in two steps:

i) generate a string of numbers for each element of $V_\sigma \cup V_\tau$:

- for each base type, generate a one-character string;
- for each function type $\sigma_1 \to \sigma_2$, generate $I(\sigma_1)I(\sigma_2)$; and
- for each product type $\prod_{i=1}^{n} \sigma_i$, generate $sort(I(\sigma_1) \ldots I(\sigma_n))$, and

ii) sort the generated strings by lexicographical order.

We will now consider the complexity of computing $\mathcal{H}(I)$.

Let σ be represented by the term automaton

$$\mathcal{M}_\sigma = (V_\sigma,\ \Sigma,\ q_0,\ \delta,\ \ell).$$

Notice that we can construct a directed graph (V_σ, E_σ), where $(q, q') \in E_\sigma$ iff $\delta(q, i) = q'$, for some $i \in \{0, 1, .., n-1\}$ and $\ell(q) \in \Sigma_n$. Similarly, for type τ, we can construct a directed graph (V_τ, E_τ). Let $M = |E_\sigma| + |E_\tau|$.

We now show that we can compute $\mathcal{H}(I)$ in $O(M)$ time.

The size of I and $\mathcal{H}(I)$ is N. For each product type $\prod_{i=1}^{n_k} \sigma_i \in (V_\sigma \cup V_\tau)$, we compute $sort[I(\sigma_1), I(\sigma_2), .., I(\sigma_{n_k})]$ in $O(n_k)$ time using COUNTING SORT [CLR90].

In graph (V_σ, E_σ), the vertex $\prod_{i=1}^{n_k} \sigma_i$ has n_k outgoing edges. Suppose there are K such vertices in the graph, then $\Sigma_{k=1}^{K} n_k \leq |E_\sigma|$. Similarly, for the product types $\prod_{i=1}^{m_k} \tau_i$ in graph (V_τ, E_τ), we have $\Sigma_{k=1}^{K'} m_k \leq |E_\tau|$, where K' is the total number of product types in V_τ. Since $M = |E_\sigma| + |E_\tau|$, the total amount of time for computing $sort(.)$ for all product types is $O(M)$.

To order all the $\rightarrow$ types and products types, we need to lexicographically order strings of numbers. Using RADIX SORT [CLR90], the ordering of all strings can be computed in time linear in the total size of the strings. The size of the string corresponding to type $\prod_{i=1}^{n_k} \sigma_i \in V_\sigma$ is n_k, which is equal to the number of outgoing edges of $\prod_{i=1}^{n_k} \sigma_i$ in (V_σ, E_σ). The size of the string corresponding to $\sigma_1 \rightarrow \sigma_2 \in V_\sigma$ is 2, which is equal to the number of outgoing edges of $\sigma_1 \rightarrow \sigma_2$ in (V_σ, E_σ). Therefore, the total size of strings corresponding to $\rightarrow$ types and product types in V_σ is equal to $|E_\sigma|$. Similarly, the total size of strings corresponding to $\rightarrow$ types and product types in V_τ is equal to $|E_\tau|$. Thus, the lexicographical ordering of all strings costs $O(M)$ time.

In conclusion, our decision procedure for membership in $\mathcal{R}$ is given by $O(N)$ iterations each of which takes $O(M)$ time. Thus, we have shown the following result.

Theorem 2.5.17. *Type equality as defined by $\mathcal{R}$ can be decided in $O(N \times M)$ time.*

2.6 Equality of Intersection and Union Types

Palsberg and Pavlopoulou [PP98] defined a type system with intersection and union types, together with a notion of type equality. An intersection type is written $\wedge_{i=1}^{n}\sigma_i$, and a union type is written $\vee_{i=1}^{n}\sigma_i$. Their notion of equality of intersection types is the same as our notion of equality of product types. Their notion of equality of union types has the distinguishing features that $\sigma \vee \sigma = \sigma$, and that there is a special base type $\perp$ such that $\sigma \vee \perp = \perp \vee \sigma = \sigma$.

The goal of this section is to demonstrate that our framework is sufficiently robust to handle union types with only minor changes to the algorithm and correctness proof. We will present the definitions and theorems in the same order as in Section 2.5. We do not show the proofs; they are similar to the ones in Section 2.5.

Palsberg and Pavlopoulou [PP98] define a set of types, where, intuitively, each type is of one of the forms:

$$\bigvee_{i=1}^{n} \wedge_{k=1}^{n_i}(\sigma_{ik} \rightarrow \sigma'_{ik})$$
$$(\bigvee_{i=1}^{n} \wedge_{k=1}^{n_i}(\sigma_{ik} \rightarrow \sigma'_{ik})) \vee int.$$

In the case where the unions are empty, the first form can be simplified to $\perp$, and the second form can be simplified to int.

A type is a regular term over the ranked alphabet

$$\Sigma = \{int, \perp, \rightarrow\} \cup \{\wedge^n, n \geq 2\} \cup \{\vee^n, n \geq 2\},$$

where $int, \perp$ are nullary, $\rightarrow$ is binary, and $\vee^n, \wedge^n$ are n-ary operators.

Palsberg and Pavlopoulou [PP98] impose the restrictions that given a type σ and a path α, if $\sigma(\alpha) = \vee^n$, then $\sigma(\alpha i) \in \{int, \perp, \rightarrow\} \cup \{\wedge^n, n \geq 2\}$, for all $i \in \{1..n\}$, and if $\sigma(\alpha) = \wedge^n$, then $\sigma(\alpha i) = \rightarrow$, for all $i \in \{1..n\}$. Intuitively, the restrictions mean that neither union types nor intersection types can be immediately nested, that is, one cannot form a union type one of whose immediate components is again a union type, and similarly for intersection types. Moreover, a union type cannot be an immediate component of an intersection type. The set of types is denoted $\hat{\mathcal{T}}$.

Given a type σ, if $\sigma(\epsilon) = \rightarrow$, $\sigma(1) = \sigma_1$, and $\sigma(2) = \sigma_2$, then we write the type as $\sigma_1 \rightarrow \sigma_2$. If $\sigma(\epsilon) = \wedge^n$ and $\sigma(i) = \sigma_i \; \forall i \in \{1, 2, \ldots, n\}$, then we write the type σ as $\wedge_{i=1}^n \sigma_i$. If $\sigma(\epsilon) = \vee^n$ and $\sigma(i) = \sigma_i \; \forall i \in \{1, 2, \ldots, n\}$, then we write the type σ as $\vee_{i=1}^n \sigma_i$. If $\sigma(\epsilon) = \bot$, then we write the type as $\bot$. If $\sigma(\epsilon) = int$, then we write the type as int.

Definition 2.6.1. The function $match(\wedge_{i=1}^n \sigma_i, \wedge_{j=1}^n \tau_j, R)$ is true iff there exists a bijection $t : \{1..n\} \rightarrow \{1..n\}$ such that for all $i \in \{1..n\} : (\sigma_i, \tau_{t(i)}) \in R$. □

Palsberg and Pavlopoulou [PP98] define type equality as follows. A relation R is called a *bisimulation* if it satisfies the following six conditions:

i) If $(\vee_{i=1}^n \sigma_i, \vee_{j=1}^m \tau_j) \in R$, then

- for all $i \in \{1..n\}$, where $\sigma_i(\epsilon) \neq \bot$: there exists $j \in \{1..m\} : (\sigma_i, \tau_j) \in R$, and
- for all $j \in \{1..m\}$, where $\tau_j(\epsilon) \neq \bot$, there exists $i \in \{1..n\} : (\sigma_i, \tau_j) \in R$.

ii) If $\tau(\epsilon) \in \{int, \bot, \rightarrow\} \cup \{\wedge^m, m \geq 2\}$, and $(\vee_{i=1}^n \sigma_i, \tau) \in R$, then,

- for all $i \in \{1..n\}$, where $\sigma_i(\epsilon) \neq \bot$: $(\sigma_i, \tau) \in R$, and
- if $\tau(\epsilon) \neq \bot$, then there exists $i \in \{1..n\} : (\sigma_i, \tau) \in R$.

iii) If $\tau(\epsilon) \in \{int, \bot, \rightarrow\} \cup \{\wedge^m, m \geq 2\}$, and $(\tau, \vee_{i=1}^n \sigma_i) \in R$, then,

- for all $i \in \{1..n\}$, where $\sigma_i(\epsilon) \neq \bot$: $(\tau, \sigma_i) \in R$, and
- if $\tau(\epsilon) \neq \bot$, then there exists $i \in \{1..n\} : (\tau, \sigma_i) \in R$.

iv) If $(\wedge_{i=1}^n \sigma_i, \wedge_{j=1}^n \tau_j) \in R$, then $match(\wedge_{i=1}^n \sigma_i, \wedge_{j=1}^n \tau_j, R)$.

v) If $(\sigma_1 \rightarrow \sigma_2, \tau_1 \rightarrow \tau_2) \in R$, then $(\sigma_1, \tau_1) \in R$ and $(\sigma_2, \tau_2) \in R$.

vi) If $(\sigma, \tau) \in R$, then either

$$\sigma(\epsilon) = \tau(\epsilon) \in \{int, \bot, \rightarrow\} \cup \{\wedge^n, n \geq 2\}, \text{ or}$$
$$\sigma(\epsilon) \in \{\vee^n, n \geq 2\}, \text{ or}$$
$$\tau(\epsilon) \in \{\vee^n, n \geq 2\}.$$

Bisimulations are closed under union, therefore, there exists a largest bisimulation

$$\mathcal{E} = \bigcup\{\ R\ \mid\ R \text{ is a bisimulation}\}.$$

The set $\mathcal{E}$ is Palsberg and Pavlopoulou's notion of type equality. It is straightforward to show, by co-induction, that

$$\sigma \vee \bot\ =\ \bot \vee \sigma\ =\ \sigma \vee \sigma\ =\ \sigma.$$

We now reformulate the above definition of bisimulation to make it better fit the framework of Section 2.5.

Definition 2.6.2. Define $\sigma \simeq_R \tau$ iff

- $\sigma(\epsilon) = \tau(\epsilon) \in \{int, \bot, \rightarrow\} \cup \{\wedge^m, m \geq 2\}$,
- if $\sigma = \sigma_1 \rightarrow \sigma_2$ and $\tau = \tau_1 \rightarrow \tau_2$, then $(\sigma_1, \tau_1) \in R$ and $(\sigma_2, \tau_2) \in R$, and
- if $\sigma = \wedge_{i=1}^n \sigma_i$ and $\tau = \wedge_{i=1}^n \tau_i$, then $match(\wedge_{i=1}^n \sigma_i, \wedge_{j=1}^n \tau_j, R)$.

The function $\widehat{match}(\sigma, \tau, R)$ is true iff

i) if $\sigma = \vee_{i=1}^n \sigma_i$ and $\tau = \vee_{j=1}^m \tau_j$, then
   - for all $i \in \{1..n\}$, where $\sigma_i(\epsilon) \neq \bot$: there exists $j \in \{1..m\} : \sigma_i \simeq_R \tau_j$, and
   - for all $j \in \{1..m\}$, where $\tau_j(\epsilon) \neq \bot$, there exists $i \in \{1..n\} : \sigma_i \simeq_R \tau_j$.

ii) if $\sigma = \vee_{i=1}^n \sigma_i$, and $\tau(\epsilon) \in \{int, \bot, \rightarrow\} \cup \{\wedge^m, m \geq 2\}$, then,
   - for all $i \in \{1..n\}$, where $\sigma_i(\epsilon) \neq \bot$: $\sigma_i \simeq_R \tau$, and
   - if $\tau(\epsilon) \neq \bot$, then there exists $i \in \{1..n\} : \sigma_i \simeq_R \tau$.

iii) if $\tau = \vee_{i=1}^n \tau_i$, and $\sigma(\epsilon) \in \{int, \bot, \rightarrow\} \cup \{\wedge^m, m \geq 2\}$, then,
   - for all $i \in \{1..n\}$, where $\tau_i(\epsilon) \neq \bot$: $\sigma \simeq_R \tau_i$, and
   - if $\sigma(\epsilon) \neq \bot$, then there exists $i \in \{1..n\} : \sigma \simeq_R \tau_i$.

□

Lemma 2.6.3. *If R is a bisimulation and $\sigma(\epsilon), \tau(\epsilon) \neq \vee^n$, where $n \geq 2$, then $(\sigma, \tau) \in R$ iff $\sigma \simeq_R \tau$.*

The following is an equivalent definition of bisimulation. A relation R is called a bisimulation if it satisfies the following four conditions:

i) If $(\sigma, \tau) \in R$, then $\widehat{match}(\sigma, \tau, R)$.

ii) If $(\wedge_{i=1}^n \sigma_i, \wedge_{j=1}^n \tau_j) \in R$, then $match(\wedge_{i=1}^n \sigma_i, \wedge_{j=1}^m \tau_j, R)$.

iii) If $(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) \in R$, then $(\sigma_1, \tau_1) \in R$ and $(\sigma_2, \tau_2) \in R$.

iv) If $(\sigma, \tau) \in R$, then either

$$\sigma(\epsilon) \;=\; \tau(\epsilon) \;\in\; \{int, \bot, \to\} \cup \{\wedge^n, n \geq 2\}, \text{ or}$$
$$\sigma(\epsilon) \in \{\vee^n, n \geq 2\}, \text{ or}$$
$$\tau(\epsilon) \in \{\vee^n, n \geq 2\}.$$

Define

$$\begin{aligned}
\hat{F} \;&\in\; \mathcal{P}(\hat{\mathcal{T}} \times \hat{\mathcal{T}}) \to \mathcal{P}(\hat{\mathcal{T}} \times \hat{\mathcal{T}}) \\
\hat{F} \;&=\; \lambda R.\{\ (\sigma, \tau) \ \mid\ \sigma, \tau \text{ are base types and } \sigma(\epsilon) = \tau(\epsilon)\ \} \\
&\quad \cup\ \{\ (\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) \ \mid\ (\sigma_1, \tau_1), (\sigma_2, \tau_2) \in R\ \} \\
&\quad \cup\ \{\ (\wedge_{i=1}^n \sigma_i, \wedge_{i=1}^n \tau_i) \ \mid\ match(\wedge_{i=1}^n \sigma_i, \wedge_{i=1}^n \tau_i, R)\ \} \\
&\quad \cup\ \{\ (\sigma, \tau) \ \mid\ \widehat{match}(\sigma, \tau, R)\ \}
\end{aligned}$$

Notice that $\hat{F}$ is monotone so it has a greatest fixed point $\nu\hat{F}$.

Theorem 2.6.4. $\mathcal{E} = \nu\hat{F}$.

Theorem 2.6.5. *$\mathcal{E}$ is a congruence relation.*

Given σ, τ, define

$$\begin{aligned} \hat{H} &\in \mathcal{P}(V_\sigma \times V_\tau) \to \mathcal{P}(V_\sigma \times V_\tau) \\ \hat{H} &= \lambda R.(\hat{F}(R) \cap (V_\sigma \times V_\tau)). \end{aligned}$$

Theorem 2.6.6. $(\sigma, \tau) \in \mathcal{E}$ *iff* $(\sigma, \tau) \in \hat{H}^N(V_\sigma \times V_\tau)$, *where* $N = |V_\sigma| + |V_\tau|$.

Given a coherent relation R, we represent R by a function

$$I : (V_\sigma \cup V_\tau) \to \{1..N\},$$

where $(\sigma', \tau') \in R$ iff $I(\sigma') = I(\tau')$.

The abstraction function abs maps a function I to the relation represented by I:

$$abs(I) = \{\ (\sigma', \tau') \in (V_\sigma \times V_\tau)\ \mid\ I(\sigma') = I(\tau')\ \}.$$

If I represents R, then we want to define $\hat{\mathcal{H}}(I)$ as a representation of $\hat{H}(R)$. The function $\hat{\mathcal{H}}$ should have the following properties:

$$\begin{aligned} \hat{\mathcal{H}}(I)(\sigma') &= \hat{\mathcal{H}}(I)(\tau') \\ &\Leftrightarrow \sigma'(\epsilon) = \tau'(\epsilon) \\ \hat{\mathcal{H}}(I)(\sigma_1 \to \sigma_2) &= \hat{\mathcal{H}}(I)(\tau_1 \to \tau_2) \\ &\Leftrightarrow I(\sigma_1) = I(\tau_1) \wedge I(\sigma_2) = I(\tau_2) \\ \hat{\mathcal{H}}(I)(\wedge_{i=1}^n \sigma_i) &= \hat{\mathcal{H}}(I)(\wedge_{i=1}^n \tau_i) \\ &\Leftrightarrow [I(\sigma_1), \ldots, I(\sigma_n)] = [I(\tau_1), \ldots, I(\tau_n)] \\ \hat{\mathcal{H}}(I)(\vee_{i=1}^m \sigma_i) &= \hat{\mathcal{H}}(I)(\vee_{i=1}^n \tau_i) \\ &\Leftrightarrow \{\hat{\mathcal{H}}(I)(\sigma_1), \ldots, \hat{\mathcal{H}}(I)(\sigma_m)\} \setminus \{\hat{\mathcal{H}}(I)(\bot)\} = \\ &\qquad \{\hat{\mathcal{H}}(I)(\tau_1), \ldots, \hat{\mathcal{H}}(I)(\tau_n)\} \setminus \{\hat{\mathcal{H}}(I)(\bot)\}. \\ \hat{\mathcal{H}}(I)(\vee_{i=1}^m \sigma_i) &= \hat{\mathcal{H}}(I)(\tau) \\ &\Leftrightarrow \{\hat{\mathcal{H}}(I)(\sigma_1), \ldots, \hat{\mathcal{H}}(I)(\sigma_m)\} \setminus \{\hat{\mathcal{H}}(I)(\bot)\} = \\ &\qquad \{\hat{\mathcal{H}}(I)(\tau)\} \setminus \{\hat{\mathcal{H}}(I)(\bot)\}. \end{aligned}$$

where σ', τ' are base types, and $\tau(\epsilon) \in \{int, \bot, \to\} \cup \{\wedge^m, m \geq 2\}$.

Any such function $\hat{\mathcal{H}}$ satisfies the following lemma.

Lemma 2.6.7. $\hat{H}(abs(I)) = abs(\hat{\mathcal{H}}(I))$.

We can define the function $\hat{\mathcal{H}}$ much the same way as $\mathcal{H}$ except for the union types. Once $\hat{\mathcal{H}}$ is defined for base types, $\rightarrow$ types, and intersection types, we can define $\hat{\mathcal{H}}$ for union types the following way. We first compute the set $S(\vee_{i=1}^{m}\sigma_i) = \{\hat{\mathcal{H}}(I)(\sigma_1), \ldots, \hat{\mathcal{H}}(I)(\sigma_m)\} \setminus \{\hat{\mathcal{H}}(I)(\bot)\}$ for every union type $\vee_{i=1}^{m}\sigma_i$. If $S(\vee_{i=1}^{m}\sigma_i) = \emptyset$, then we let $\hat{\mathcal{H}}(I)(\vee_{i=1}^{m}\sigma_i) = \hat{\mathcal{H}}(I)(\bot)$. If $S(\vee_{i=1}^{m}\sigma_i) = \{k\}$, then we let $\hat{\mathcal{H}}(I)(\vee_{i=1}^{m}\sigma_i) = k$. We then order the rest of the union types lexicographically by the sets $S(.)$ and assign unused integers to the union types according to their ranking.

Given a type pair (σ, τ), let $N = |V_\sigma| + |V_\tau|$, and $M = |E_\sigma| + |E_\tau|$. It is now straightforward to show, using the techniques that were applied in Section 2.5, that our decision procedure for membership in $\mathcal{E}$ is given by $O(N)$ iterations each of which takes $O(M)$ time. Thus, we have shown the following result.

Theorem 2.6.8. *Type equality as defined by $\mathcal{E}$ can be decided in $O(N \times M)$ time.*

2.7 An Efficient Algorithm for Type Equivalence

In this section, we will present a slightly more efficient algorithm for the type-matching problem. We have shown that matched types can be found by computing the greatest fixed point of a monotone function constructed from a definition of bisimulation and an initial set of type pairs that are potentially equivalent. By reducing the fixed-point computation of the monotone function to the problem of finding the coarsest size-stable partition refinement of a graph, we are able to reduce the time complexity of type matching to $O(n \log n)$.

Assume that we are given two types τ_1 and τ_2 that are represented as two term automata $\mathcal{M}_1$ and $\mathcal{M}_2$. Lemma 2.7.1 proves that $\tau_1 \cong \tau_2$ (or $(\tau_1, \tau_2) \in \mathcal{R}$) if and only if there is a reflexive bismulation C between $\mathcal{M}_1$ and $\mathcal{M}_2$ such that the initial states of the term automata $\mathcal{M}_1$ and $\mathcal{M}_2$ are related by C. Lemma 2.7.3 essentially reduces the problem of finding a reflexive bisimualtion C between $\mathcal{M}_1$ and $\mathcal{M}_2$ to finding a size-stable coarsest partition [PT87]. Theorem 2.7.4 uses the algorithm of Paige and Tarjan to determine in $O(n \log n)$ time (n is the sum of the sizes of the two term automata) whether there exists a reflexive bisimulation C between $\mathcal{M}_1$ and $\mathcal{M}_2$.

Throughout this section, we will use $\mathcal{M}_1, \mathcal{M}_2$ to denote two term automata over the alphabet Σ:

$$\begin{aligned} \mathcal{M}_1 &= (Q_1, \Sigma, q_{01}, \delta_1, \ell_1) \\ \mathcal{M}_2 &= (Q_2, \Sigma, q_{02}, \delta_2, \ell_2). \end{aligned}$$

We assume that $Q_1 \cap Q_2 = \emptyset$. Define $Q = Q_1 \cup Q_2$. Define also $\delta : Q \times \omega \to Q$ where $\delta = \delta_1 \oplus \delta_2$, and $\ell : Q \to \Sigma$, where $\ell = \ell_1 \oplus \ell_2$, where $\oplus$ denotes disjoint union of two functions. We say that $\mathcal{M}_1, \mathcal{M}_2$ are *bisimular* if and only if there exists a relation $C \subseteq Q \times Q$, called a bisimulation between $\mathcal{M}_1$ and $\mathcal{M}_2$, such that:

- if $(q, q') \in C$, then $\ell(q) = \ell(q')$
- if $(q, q') \in C$ and $\ell(q) = \to$, then $(\delta(q, 0), \delta(q', 0)) \in C$ and $(\delta(q, 1), \delta(q', 1)) \in C$
- if $(q, q') \in C$ and $\ell(q) = \prod^n$, then there exists a bijection $t : \{0..n-1\} \to \{0..n-1\}$ such that $\forall i \in \{0..n-1\}$: $(\delta(q, i), \delta(q', t(i))) \in C$.

Notice that the bisimulations between $\mathcal{M}_1$ and $\mathcal{M}_2$ are closed under union, therefore, there exists a largest bisimulation between $\mathcal{M}_1$ and $\mathcal{M}_2$. It is straightforward to show that the identity relation on Q is a bisimulation, and that any reflexive bisimulation is an equivalence relation. Hence, the largest bisimulation is an equivalence relation.

Lemma 2.7.1. *For types τ_1, τ_2 that are represented by the term automata $\mathcal{M}_1, \mathcal{M}_2$, respectively, we have $(\tau_1, \tau_2) \in \mathcal{R}$ if and only if there is a reflexive bisimulation C between $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $(q_{01}, q_{02}) \in C$.*

Proof. Suppose $(\tau_1, \tau_2) \in \mathcal{R}$. Define:

$$C = \{ (q, q') \in Q \times Q \mid (\lambda\alpha.\ell(\hat{\delta}(q, \alpha)), \lambda\alpha.\ell(\hat{\delta}(q', \alpha))) \in \mathcal{R} \}.$$

It is straightforward to show that C is a bisimulation between $\mathcal{M}_1$ and $\mathcal{M}_2$, and that $(q_{01}, q_{02}) \in C$, we omit the details.

Conversely, let C be a reflexive bisimulation between $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $(q_{01}, q_{02}) \in C$. Define:

$$R \;=\; \{\, (\sigma_1, \sigma_2) \;\mid\; (q, q') \in C \;\wedge\; \sigma_1 = \lambda\alpha.\ell(\hat{\delta}(q, \alpha)) \;\wedge\; \sigma_2 = \lambda\alpha.\ell(\hat{\delta}(q', \alpha)) \,\}$$

From $(q_{01}, q_{02}) \in C$, we have $(\tau_1, \tau_2) \in R$. It is straightforward to prove that R is a bisimulation, we omit the details. From $(\tau_1, \tau_2) \in R$ and R being a bisimulation, we conclude that $(\tau_1, \tau_2) \in \mathcal{R}$. □

A *partitioned graph* is a 3-tuple (U, E, P), where U is a set of nodes, $E \subseteq U \times U$ is an edge relation, and P is a *partition* of U. A partition P of U is a set of pairwise disjoint subsets of U whose union is all of U. The elements of P are called its *blocks*. If P and S are partitions of U, then S is a *refinement* of P if and only if every block of S is contained in a block of P.

A partition S of a set U can be characterized by an equivalence relation K on U such that each block of S is an equivalence class of K. If U is a set and K is an equivalence relation on U, then we use U/K to denote the partition of U into equivalence classes for K.

A partition S is *size-stable* with respect to E if and only if for all blocks $B_1, B_2 \in S$, and for all $x, y \in B_1$, we have $|E(x) \cap B_2| = |E(y) \cap B_2|$, where $E(x)$ is the set $\{y | (x, y) \in E\}$. If E is clear from the context, we will simply use size-stable. We will repeatedly use the following characterization of size-stable partitions.

Lemma 2.7.2. *For an equivalence relation K, we have that U/K is size-stable if and only if for all $(u, u') \in K$, there exists a bijection $\pi : E(u) \to E(u')$ such that for all $u_1 \in E(u)$, we have $(u_1, \pi(u_1)) \in K$.*

Proof. Suppose that U/K is size-stable. Let $(u, u') \in K$. Let B_1 be the block of U/K which contains u and u'. For each block B_2 of U/K, we have that $|E(u) \cap B_2| = |E(u') \cap B_2|$. So, for each block B_2 of U/K, we can construct a bijection from $E(u) \cap B_2$ to $E(u') \cap B_2$, such that for all $u_1 \in E(u) \cap B_2$, we have $(u_1, \pi(u_1)) \in K$. These bijections can then be merged to single bijection $\pi : E(u) \to E(u')$ with the desired property.

Conversely, suppose that for all $(u, u') \in K$, there exists a bijection $\pi : E(u) \to E(u')$ such that for all $u_1 \in E(u)$, we have $(u_1, \pi(u_1)) \in K$. Let $B_1, B_2 \in U/K$, and let $x, y \in B_1$. We have that $(x, y) \in K$, so there exists a bijection $\pi : E(x) \to E(y)$ such that for all $u_1 \in E(x)$, we have $(u_1, \pi(u_1)) \in K$. Each element of $E(x) \cap B_2$ is mapped by π to an element of $E(y) \cap B_2$. Moreover, each element of $E(y) \cap B_2$ must be the image under π of an element of $E(x) \cap B_2$. We conclude that π restricted to $E(x) \cap B_2$ is a bijection to $E(y) \cap B_2$, so $|E(x) \cap B_2| = |E(y) \cap B_2|$. □

Given two term automata $\mathcal{M}_1, \mathcal{M}_2$, we define a partitioned graph (U, E, P):

$$
\begin{aligned}
U &= Q \cup \{\ \langle q, i\rangle \mid q \in Q \wedge \delta(q,i) \text{ is defined } \} \\
E &= \{\ (q, \langle q, i\rangle) \mid \delta(q,i) \text{ is defined } \} \\
&\cup \{\ (\langle q, i\rangle, \delta(q,i)) \mid \delta(q,i) \text{ is defined } \} \\
L &= \{\ (q, q') \in Q \times Q \mid \ell(q) = \ell(q') \ \} \\
&\cup \{\ (\langle q, i\rangle, \langle q', i'\rangle) \mid \ell(q) = \ell(q') \text{ and if } \ell(q) = \rightarrow, \text{ then } i = i' \ \} \\
P &= U/L.
\end{aligned}
$$

The graph contains one node for each state and transition in $\mathcal{M}_1, \mathcal{M}_2$. Each transition in $\mathcal{M}_1, \mathcal{M}_2$ is mapped to two edges in the graph. This construction ensures that if a node in the graph corresponds to a state labeled $\prod^n$, then that node will have n distinct successors in the graph. This is convenient when establishing a bijection between the successors of two nodes labeled $\prod^n$.

The equivalence relation L creates a distinction between the two successors of a node that corresponds to a state labeled $\rightarrow$. This is done by ensuring that if $(\langle q, i\rangle, \langle q, i'\rangle) \in L$ and $\ell(q) = \rightarrow$, then $i = i'$. This is convenient when establishing a bijection between the successors of two nodes labeled $\rightarrow$.

Lemma 2.7.3. *There exists a reflexive bisimulation C between $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $(q_{01}, q_{02}) \in C$ if and only if there exists a size-stable refinement S of P such that q_{01} and q_{02} belong to the same block of S.*

Proof. Let $C \subseteq Q \times Q$ be a reflexive bisimulation between $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $(q_{01}, q_{02}) \in C$. Define an equivalence relation $K \subseteq U \times U$ such that:

$$\begin{aligned} K &= C \\ &\cup \{ (\langle q, i\rangle, \langle q', i\rangle) \mid (q, q') \in C \wedge \ell(q) = \ell(q') = \rightarrow \} \\ &\cup \{ (\langle q, i\rangle, \langle q', i'\rangle) \mid (q, q') \in C \wedge (\delta(q, i), \delta(q', i')) \in C \\ &\qquad\qquad \wedge \ell(q) = \ell(q') \wedge \ell(q) \neq \rightarrow \} \\ S &= U/K. \end{aligned}$$

From $(q_{01}, q_{02}) \in C$, we have $(q_{01}, q_{02}) \in K$, so q_{01} and q_{02} belong to the same block of S. We will now show that S is a size-stable refinement of P.

Let $(u, u') \in K$. From Lemma 2.7.2 we have that it is sufficient to show that there exists a bijection $\pi : E(u) \rightarrow E(u')$, such that for all $u_1 \in E(u)$, we have $(u_1, \pi(u_1)) \in K$. There are three cases.

First, suppose $(u, u') \in C$. We have

$$\begin{aligned} E(u) &= \{ \langle u, i\rangle \mid \delta(u, i) \text{ is defined } \} \\ E(u') &= \{ \langle u', i'\rangle \mid \delta(u', i') \text{ is defined } \}. \end{aligned}$$

Let us consider each of the possible cases of u and u'. If $\ell(u) = \ell(u') \in \Gamma$, then $E(u) = E(u') = \emptyset$, and the desired bijection exists trivially. Next, if $\ell(u) = \ell(u') = \rightarrow$, then

$$\begin{aligned} E(u) &= \{ \langle u, 0\rangle, \langle u, 1\rangle \} \\ E(u') &= \{ \langle u', 0\rangle, \langle u', 1\rangle \}, \end{aligned}$$

so the desired bijection is $\pi : E(u) \rightarrow E(u')$, where $\pi(\langle u, 0\rangle) = \langle u', 0\rangle$ and $\pi(\langle u, 1\rangle) = \langle u', 1\rangle$, because $(\langle u, 0\rangle, \langle u', 0\rangle) \in K$ and $(\langle u, 1\rangle, \langle u', 1\rangle) \in K$. Finally, if $\ell(u) = \ell(u') = \prod^n$, then

$$\begin{aligned} E(u) &= \{ \langle u, i\rangle \mid \delta(u, i) \text{ is defined } \} \\ E(u') &= \{ \langle u', i'\rangle \mid \delta(u', i') \text{ is defined } \}. \end{aligned}$$

From $(u, u') \in C$, we have a bijection $t : \{0..n-1\} \to \{0..n-1\}$ such that $\forall i \in \{0..n-1\} : (\delta(u, i), \delta(u', t(i))) \in C$. From that, the desired bijection can be constructed.

Second, suppose $u = \langle q, i\rangle$ and $u' = \langle q', i\rangle$, where $(q, q') \in C$, and $\ell(q) = \ell(q') = \rightarrow$. We have

$$\begin{aligned} E(u) &= \{\ \delta(q, i)\ \} \\ E(u') &= \{\ \delta(q', i)\ \}, \end{aligned}$$

and from $(q, q') \in C$ we have $(\delta(q, i), \delta(q', i)) \in C \subseteq K$, so the desired bijection exists.

Third, suppose $u = \langle q, i\rangle$ and $u' = \langle q', i'\rangle$, where $(q, q') \in C$, $(\delta(q, i), \delta(q', i')) \in C$, $\ell(q) = \ell(q')$, and $\ell(q) \neq \rightarrow$. We have

$$\begin{aligned} E(u) &= \{\ \delta(q, i)\ \} \\ E(u') &= \{\ \delta(q', i')\ \}, \end{aligned}$$

and $(\delta(q, i), \delta(q', i')) \in C \subseteq K$, so the desired bijection exists.

Conversely, let S be a size-stable refinement of P such that q_{01} and q_{02} belong to the same block of S. Define:

$$\begin{aligned} K &= \{\ (u, u') \in U \times U \mid u, u' \text{ belong to the same block of } S\ \} \\ C &= K \cap (Q \times Q). \end{aligned}$$

Notice that $(q_{01}, q_{02}) \in C$ and that C is reflexive. We will now show that C is a bisimulation between $\mathcal{M}$ and $\mathcal{M}'$.

First, suppose $(q, q') \in C$. From S being a refinement of P we have $(q, q') \in L$, so $\ell(q) = \ell(q')$.

Second, suppose $(q, q') \in C$ and $\ell(q) = \rightarrow$. From the definition of E we have

$$\begin{aligned} E(q) &= \{\ \langle q, 0\rangle, \langle q, 1\rangle\ \} \\ E(q') &= \{\ \langle q', 0\rangle, \langle q', 1\rangle\ \}. \end{aligned}$$

From S being size-stable, $(q, q') \in C \subseteq K$, and Lemma 2.7.2 we have that there exists a bijection $\pi : E(q) \to E(q')$ such that for all $u \in E(q)$ we have that $(u, \pi(u)) \in K$. From

$K \subseteq L$ and $\ell(q) = \longrightarrow$ we have that there is only one possible bijection π:

$$\begin{aligned} \pi(\langle q, 0\rangle) &= \langle q', 0\rangle \\ \pi(\langle q, 1\rangle) &= \langle q', 1\rangle, \end{aligned}$$

so $(\langle q, 0\rangle, \langle q', 0\rangle) \in K$ and $(\langle q, 1\rangle, \langle q', 1\rangle) \in K$. From the definition of E we have, for $i \in \{0, 1\}$,

$$\begin{aligned} E(\langle q, i\rangle) &= \delta(q, i) \\ E(\langle q', i\rangle) &= \delta(q', i), \end{aligned}$$

and since S is size-stable, we have, for $i \in \{0, 1\}$, $(\delta(q, i), \delta(q', i)) \in K$. Moreover, for $i \in \{0, 1\}$, $(\delta(q, i), \delta(q', i)) \in Q \times Q$; therefore we conclude, $(\delta(q, i), \delta(q', i)) \in C$.

Third, suppose $(q, q') \in C$ and $\ell(q) = \prod^n$. From the definition of E we have

$$\begin{aligned} E(q) &= \{\ \langle q, i\rangle \mid \delta(q, i) \text{ is defined } \} \\ E(q') &= \{\ \langle q', i\rangle \mid \delta(q', i) \text{ is defined } \}. \end{aligned}$$

Notice that $|E(q)| = |E(q')| = n$. From S being size-stable, $(q, q') \in C \subseteq K$, and Lemma 2.7.2, we have that there exists a bijection $\pi : E(q) \to E(q')$ such that for all $u \in E(q)$ we have that $(u, \pi(u)) \in K$. From π we can derive a bijection $t : \{0..n-1\} \to \{0..n-1\}$ such that $\forall i \in \{0..n-1\}$: $(\langle q, i\rangle, \langle q', t(i)\rangle) \in K$. From the definitions of E and E' we have that for $i \in \{0..n-1\}$,

$$\begin{aligned} E(\langle q, i\rangle) &= \{\ \delta(q, i)\ \} \\ E(\langle q', i\rangle) &= \{\ \delta(q', i)\ \}, \end{aligned}$$

and since S is size-stable, and, for all $i \in \{0..n-1\}$, $(\langle q, i\rangle, \langle q', t(i)\rangle) \in K$, we have $(\delta(q, i), \delta(q', t(i))) \in K$. Moreover, $(\delta(q, i), \delta(q', t(i))) \in Q \times Q$; therefore, we conclude $(\delta(q, i), \delta(q', t(i))) \in C$. □

The *size* of a term automata $\mathcal{M} = (Q, \Sigma, q_0, \delta, l)$ is $|Q| + \mid |\delta|$, i.e., the sum of the number of states and transitions in the automata.

Theorem 2.7.4. *For types τ_1, τ_2 that can be represented by term automata $\mathcal{M}_1, \mathcal{M}_2$ of size at most n, we can decide $(\tau_1, \tau_2) \in \mathcal{R}$ in $O(n \log n)$ time.*

Proof. From Lemma 2.7.1 we have that $(\tau_1, \tau_2) \in \mathcal{R}$ if and only if there is a reflexive bisimulation C between $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $(q_{01}, q_{02}) \in C$. Fromc Lemma 2.7.3 we have that there exists a reflexive bisimulation C between $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $(q_{01}, q_{02}) \in C$ if and only if there exists a size-stable refinement S of P such that q_{01} and q_{02} belong to the same block of S.

Paige and Tarjan [PT87] give an $O(m \log p)$ algorithm to find the coarsest size-stable refinement of P, where m is the size of E and p is the size of the universe U.

Our algorithm first constructs (U, E, P) from $\mathcal{M}_1$ and $\mathcal{M}_2$, then runs the Paige-Tarjan algorithm to find the coarsest size-stable refinement S of P, and finally checks whether q_{01} and q_{02} belong to the same block of S.

If $\mathcal{M}_1$ and $\mathcal{M}_2$ are of size at most n, then the size of E is at most $2n$, and the size of U is at most $2n$, so the total running time of our algorithm is $O(2n \log(2n)) = O(n \log n)$. □

Next, we illustrate how our algorithm determines that equivalence between the types. Details of the algorithm can be found in [PT87]. Consider two types I_1 and J_1 defined in Section 2.2. The set of types corresponding to the two interfaces are:

$$\{I_1, I_2, m_1, m_2, m_3, m_4, \textit{int}, \textit{float}\}$$
$$\{J_1, J_2, n_1, n_2, n_3, n_4, \textit{int}, \textit{float}\}$$

Figure 2.10 shows various steps of our algorithm. For simplicity, the figure only shows the blocks of actual types, but not the blocks of the extra nodes of the form $\langle q, i \rangle$. The blocks in the first row are based on labels, e.g., states labeled with $\times$ are in the same block. In the next step, the block containing the methods are split based on the type of the result of the method, e.g.. methods m_1 and n_4 both return *float*, so they are in the same block. In the next step (corresponding to the third row) the block $\{I_1, I_2, J_1, J_2\}$ are split. The final partition, where block $\{m_3, m_4, n_1, n_2\}$ is split, is shown in the fourth row.

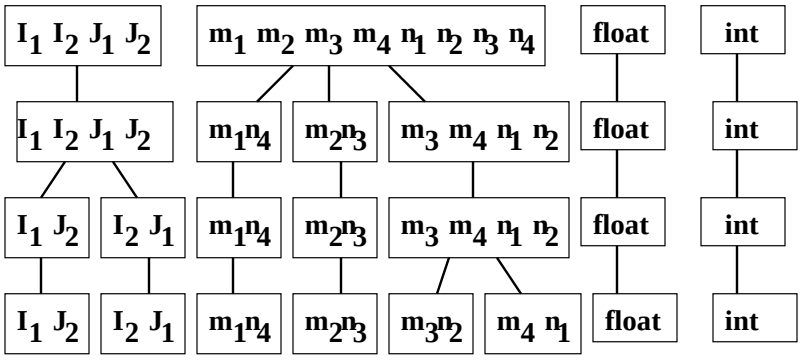


Figure 2.10. Blocks of types

Our algorithm can be tuned to take a specific user needs into account. This is done simply by modifying the definition of the equivalence relation L. For example, suppose a user cares about the order of the arguments to a method. This means that the components of the product type that models the argument list should not be allowed to be shuffled during type matching. We can prevent shuffling by employing the same technique that the current definition of L uses for function types. The idea is to insist that two component types may only be matched when they have the same component index.

Another example of the tunability of our algorithm involves the modifiers in `Java`. Suppose a programmer is developing a product that is multi-threaded. In this case the programmer may only want to match `synchronized` methods with other `synchronized` methods. This can be handled easily in our framework by changing L such that two method types may only be matched when they are both synchronized. On the other hand if the user is working on a single-threaded product, the keyword `synchronized` can be ignored. The same observation applies to other modifiers such as `static`.

2.8 Implementation

We have implemented our algorithm in `Java` and the current version is based on the code written by Wanjun Wang. The implementation has a graphical user interface so that users may input type definitions written in a file and also may specify restrictions on type isomorphism. The implementation and documentation are available at

`http://www.cs.purdue.edu/homes/tzhao/matching/matching.htm`.

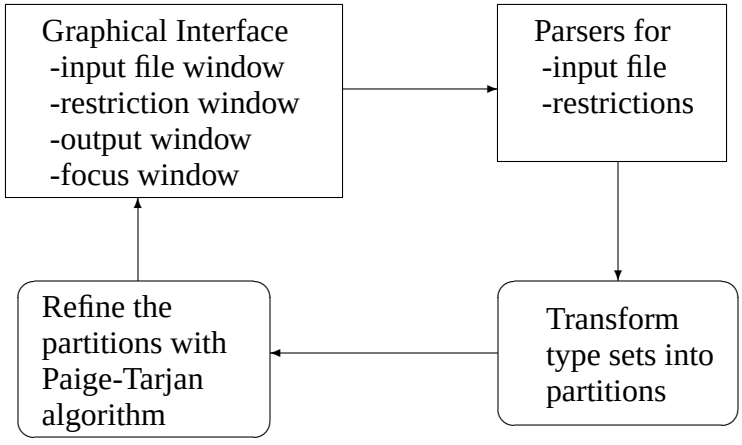


Figure 2.11. Schematic diagram for the implementation

Suppose we are given the following file with four `Java` interfaces.

interface I_1 {
    float m_1 $(I_1\ a, \textit{int}\ b)$;
    int m_2 $(I_2\ a)$;
}

interface I_2 {
    J_2 m_3 $(\textit{float}\ a)$;
    I_1 m_4 $(\textit{float}\ a)$;
}

interface J_1 {
    I_1 n_1 $(\textit{float}\ a)$;
    J_2 n_2 $(\textit{float}\ a)$;
}

interface J_2 {
    int n_3 $(J_1\ a)$;
    float n_4 $(\textit{int}\ a, J_2\ b)$;
}

The implementation, as illustrated in the Figure 2.11, will read and parse the input file and then transform the type definitions into partitions of numbers with each type definition and dummy type assigned a unique number. The partitions will be refined by Paige-Tarjan algorithm until it is *size-stable* as defined in this chapter. Finally, we will be able to read the results from the final partitions. Two types are isomorphic if the numbers assigned to them are in the same partition. The implementation will give the following output:

$$I_1 = J_2 \quad , \quad I_2 = J_1$$

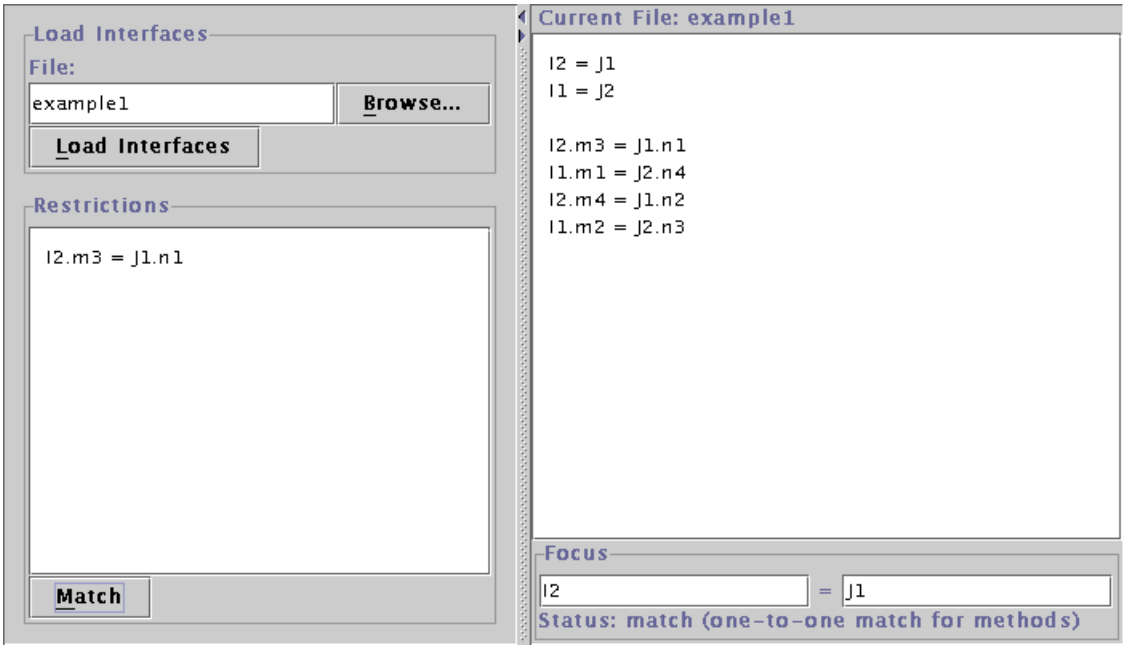


Figure 2.12. Screen shot of the implementation

$$I_2.m_3 = I_2.m_4 \quad = \quad J_1.n_1 = J_1.n_2$$
$$I_1.m_1 = J_2.n_4 \quad , \quad I_1.m_2 = J_2.n_3 \ .$$

We can see that the types of interfaces I_2 and J_1 are isomorphic and moreover, all method types of I_2, J_1 match. Suppose that we have additional information about the method types such that only method m_3 and n_1 should have isomorphic types. We can restrict the type matching by adding $I_2.m_3 = J_1.n_1$ to the *restrictions* window of the user interface. The new matching result is illustrated by the screen shot in figure 2.12.

Note that we are able to focus on the matching of two interface types such as I_2, J_1 as in the *focus* windows of Figure 2.12, where I_2, J_1 are matched and their methods are matched one to one.

2.9 Subtyping of Recursive Types

In this section we discuss subtyping and formalize it using a simulation relation. We also discuss reasons why the algorithm given in Section 2.7 is not applicable to subtyping of of recursive types. Consider the interfaces K_1 and K_2 shown in Figure 2.13, and suppose a

interface K_1 {
    K_1 m (*float* a, *boolean* b);
    boolean p (K_1 j);
}

interface K_2 {
    K_2 m (*int* i, *boolean* b);
}

Figure 2.13. Interfaces K_1 and K_2

user is looking for K_2. The interfaces K_1 and K_2 can be mapped to the following recursive types:

$$\begin{aligned} \tau_1 &= \mu\alpha.((\mathit{float} \times \mathit{boolean}) \rightarrow \alpha) \times (\alpha \rightarrow \mathit{boolean}) \\ \tau_2 &= \mu\beta.(\mathit{int} \times \mathit{boolean}) \rightarrow \beta) \end{aligned}$$

Assuming that int is a subtype of $float$ (we can always coerce integers into floats) we have that τ_1 is a subtype of τ_2. Therefore, the user can use the interface K_1. There are several points to notice from this example. In the context of subtyping, we need two kinds of products: one that models a collection of methods and another that models sequence of parameters. In our example, the user only specified a type corresponding to method m. Therefore, during the subtyping algorithm method p should be ignored. However, the parameters of method m are also modeled using products and none of these can be ignored. Therefore, we consider two types of product type constructors in our type systems and the subtyping rule for these two types of products are different.

As stated before, a type is a regular term, in this case over the ranked alphabet

$$\Sigma = \Gamma \cup \{\rightarrow\} \cup \{\prod^n, n \geq 2\} \cup \{\times^n, n \geq 2\}.$$

Roughly speaking, $\prod^n$ and $\times^n$ will model collection of parameters and methods respectively. Also assume that we are given a subtyping relation on the base types Γ. If τ_1 is a subtype of τ_2, we will write it as $\tau_1 \preceq \tau_2$. A relation $\bar{R}$ is called a *simulation* on types if it satisfies the following conditions:

- if $(\sigma, \tau) \in \bar{R}$ and $\sigma(\epsilon) \in \Gamma$, then $\tau(\epsilon) \in \Gamma$ and $\sigma(\epsilon) \preceq \tau(\epsilon)$.

- if $(\sigma, \tau) \in \bar{R}$ and $\sigma(\epsilon) \in (\{\rightarrow\} \cup \{\prod^n, n \geq 2\})$, then $\sigma(\epsilon) = \tau(\epsilon)$.

- if $(\sigma_1 \rightarrow \sigma_2, \tau_1 \rightarrow \tau_2) \in \bar{R}$, then $(\tau_1, \sigma_1) \in \bar{R}$ and $(\sigma_2, \tau_2) \in \bar{R}$.

- if $(\prod_{i=0}^{n-1} \sigma_i, \prod_{i=0}^{n-1} \tau_i) \in \bar{R}$, then there exists a bijection $t : \{0 \ldots n-1\} \rightarrow \{0 \ldots n-1\}$ such that for all $i \in \{0 \ldots n-1\}$, we have $(\sigma_i, \tau_{t(i)}) \in \bar{R}$.

- Suppose $(\sigma, \tau) \in \bar{R}$, $\sigma(\epsilon) = \times^n$, and $\sigma = \times_{i=0}^{n-1} \sigma_i$. If $\tau(\epsilon) \notin \{\times^m, m \geq 2\}$, then there exists a $j \in \{0 \ldots n-1\}$ such that $(\sigma_j, \tau) \in \bar{R}$. Otherwise, assume that $\tau(\epsilon) = \times^m$, where $m \leq n$ and $\tau = \times_{i=0}^{m-1} \tau_i$. In this case, then there exists an injective function $s : \{0 \ldots m-1\} \rightarrow \{0 \ldots n-1\}$ such that for all $i \in \{0 \ldots m-1\}$, we have $(\sigma_{s(i)}, \tau_i) \in \bar{R}$. Notice that this rule allows ignoring certain components of σ.

As is the case with bisimulations, simulations are closed under union, therefore there exists a largest simulation (denoted by $\bar{\mathcal{R}}$).

Let $\mathcal{M}_1, \mathcal{M}_2$ denote two term automata over Σ:

$$\begin{aligned} \mathcal{M}_1 &= (Q_1, \Sigma, q_{01}, \delta_1, \ell_1) \\ \mathcal{M}_2 &= (Q_2, \Sigma, q_{02}, \delta_2, \ell_2). \end{aligned}$$

We assume that $Q_1 \cap Q_2 = \emptyset$. Define $Q = Q_1 \cup Q_2$. Define also $\delta : Q \times \omega \rightarrow Q$ where $\delta = \delta_1 \oplus \delta_2$, and $\ell : Q \rightarrow \Sigma$, where $\ell = \ell_1 \oplus \ell_2$, where $\oplus$ denotes disjoint union of two functions. We say that $\mathcal{M}_2$ *simulates* $\mathcal{M}_1$ (denoted by $\mathcal{M}_1 \preceq \mathcal{M}_2$) if and only if there exists a relation $D \subseteq Q \times Q$, called a *simulation relation* between $\mathcal{M}_1$ and $\mathcal{M}_2$, such that:

- if $(q, q') \in D$ and $\ell(q) \in \Gamma$, then $\ell(q') \in \Gamma$ and $\ell(q) \preceq \ell(q'))$.

- if $(q, q') \in D$ and $\ell(q) \in (\{\rightarrow\} \cup \{\prod^n, n \geq 2\})$, then $\ell(q) = \ell(q')$.

- if $(q, q'), \in D$ and $\ell(q) = \rightarrow$, then $(\delta(q, 0), \delta(q', 0)) \in D$ and $(\delta(q', 1), \delta(q, 1)) \in D$.

- if $(q, q'), \in D$ and $\ell(q) = \prod^n$, then there exists a bijection $t : \{0 \ldots n-1\} \rightarrow \{0 \ldots n-1\}$ such that for all $i \in \{0 \ldots n-1\}$, we have $(\delta(q, i), \delta(q', t(i)) \in D$.

- Suppose $(q, q') \in D$ and $\ell(q) = \times^n$. If $\ell(q') \notin \{\times^m, m \geq 2\}$, then there exists a $j \in \{0 \ldots n-1\}$ such that $(\delta(q, j), q') \in D$. Otherwise, assume that $\ell(q') = \times^m$, where $m \leq n$ and in this case, there exists an injective function $s : \{0 \ldots m-1\} \rightarrow \{0 \ldots n-1\}$ such that for all $i \in \{0 \cdots m-1\}$, we have $(\delta(q, s(i)), \delta(q', i)) \in D$.

Notice that the simulations between $\mathcal{M}_1$ and $\mathcal{M}_2$ are closed under union, therefore, there exists a largest simulation between $\mathcal{M}_1$ and $\mathcal{M}_2$. The proof of Lemma 2.9.1 is similar to the proof of Lemma 2.7.1 and is omitted.

Lemma 2.9.1. *For types τ_1, τ_2 that are represented by the term automata $\mathcal{M}_1, \mathcal{M}_2$, respectively, we have $(\tau_1, \tau_2) \in \bar{\mathcal{R}}$ if and if only there is a reflexive simulation D between $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $(q_{01}, q_{02}) \in D$.*

The largest simulation between the term automata $\mathcal{M}_1$ and $\mathcal{M}_2$ is given by the following greatest fixed point

$$\nu D.sim(q, q', D).$$

where $D \subseteq Q_1 \times Q_2$ and the predicate $sim(q, q', D)$ is the conjunction of the five conditions which appear in the definition of the simulation relation between two automata. Let n and m be the size of the term automata $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively. Since nm is a bound on the size of D, the number of iterations in computing the greatest fixed point is bounded by nm. In general, the relation D (or for that matter the simulation relation) is not symmetric. On the other hand, the bisimulation relation was an equivalence relation, and so could be represented as a partition on the set $Q_1 \cup Q_2$, or in other words, partitions give us a representation of an equivalence relation that is linear in the sum of the sizes of the set of states Q_1 and Q_2. The Paige-Tarjan algorithm uses the partition representation of the equivalence relation. Since D is not symmetric (and thus not an equivalence relation), it cannot be represented by a partition. This is the crucial reason why our previous algorithm cannot be used for subtyping.

2.10 Conclusion

In this chapter, we addressed the problem of matching recursive types with flexible equality rules. We characterized the equivalence of recursive types by a definition of bisim-

ulation. The decision procedure for type equivalence depends on the computation of the greatest fixed point of a monotone function constructed from the definition of bisimulation and from an initial relation on potentially equivalent types. A straightforward implementation of this approach resulted in an $O(n^2)$ algorithm.

An algorithm with $O(n \log n)$ time complexity was found by reducing the fixed-point computation to the problem of finding the coarsest size-stable partition of a graph. To our knowledge, this is the most efficient algorithm for this problem. Our results are applicable to the problem of matching signatures of software components and to the integration of multi-language systems. We showed an extension of our decision procedure to the intersection and union types and issues related to subtyping of recursive types were also addressed.

Remark: A natural next step is to investigate how to automatically generate bridge code for a multi-language system. We would also like to find out whether our notion of type equality is sound and complete for some class of models of recursive types. On the implementation side, we want to make connections to work on multiset discrimination [CP95] and chaotic fixed-point iteration [Cou81].

When dealing with building bridge code between interfaces, there are interesting equivalences involving currying and uncurrying at the interface level [AC96b, ABR98]. Recall that currying is usually expressed with the rule

$$\sigma_1 \to (\sigma_2 \to \sigma_3) = (\sigma_1 \times \sigma_2) \to \sigma_3.$$

Consider the type

$$\sigma = \mu\alpha.(int \to \alpha).$$

When uncurrying is allowed, σ is equivalent to a number of types containing product types of different sizes, such as:

$$\begin{aligned} \sigma &= \mu\alpha.((int \times int) \to \alpha) \\ &= \mu\alpha.((\prod_{i=1}^{4} \tau_i) \to \alpha) \end{aligned}$$

where, for all $i \in 1..4$, $\tau_i = int$. Notice that σ does not contain any product types, while the second type contains a binary product type, and the third type contains a 4-ary product type. It remains an open problem to decide this notion of type equality.

3. Automatic Discovery of Covariant Read-Only Fields

3.1 Introduction

3.1.1 Background

In the quest for more expressive type systems, subtyping has become a common means for flexible matching of types. For example, in an assignment x=e, we can allow x and e to have different types, provided that the type of e is a subtype of the type of x. This relieves the programmer from having to insert coercions.

Subtyping comes in many flavors. For object types of the form $[\ell : B, \ldots]$, where $\ell : B$ denotes a field ℓ of type B, there are several design choices. Abadi and Cardelli [AC96a] explain that if the field ℓ can be both read and updated, then ℓ must be *invariant* in the subtype ordering, that is, if $[\ell : A, \ldots]$ is a subtype of $[\ell : B, \ldots]$, then $A = B$. Following Abadi and Cardelli, we will use the notation $[\ell^0 : B, \ldots]$ where the superscript 0 denotes that the object type has an invariant field ℓ.

A covariant read-only field (CROF) is a field which enjoys covariant subtyping and which cannot be updated. Again following Abadi and Cardelli, we will use the notation $[m^+ : B, \ldots]$ where the superscript $+$ denotes that the object type has a covariant field m. Covariance means that if $[m^+ : A, \ldots]$ is a subtype of $[m^+ : B, \ldots]$, then A is a subtype of B, a weaker condition than $A = B$.

CROFs are a useful addition to type systems. For example, Glew has given a nice translation of classes and objects into an intermediate calculus in which the method tables of classes are placed in CROFs; covariant subtyping on the method table fields is needed to ensure that subclasses are translated to subtypes. We will discuss Glew's translation in more detail later in the chapter.

First, however, we illustrate some of the central technical difficulties of CROFs with a much simpler, more whimsical example. Our program is written in a variant of an Abadi-

Cardelli object calculus [AC96a]. Each method `@(`x`)`b binds a name x that denotes the smallest enclosing object, much like Java's "this." The program declares four objects, then makes two successive method calls:

```
let
  Man      = [id = @(x)x]
  Batman   = [id = @(y)y  secretid = @(z)z]
  Phone    = [dial = @(d)Man]
  Batphone = update Phone.dial <=
     @(e) call (call Batman.id).secretid
in
  call (call Batphone.dial).id
```

Here is the intuition behind the example. A `Man` has an identity (himself). `Batman` has an identity too, and he also has a secret identity. By dialing the `Phone` you can reach `Man`, and by dialing the `Batphone` you can reach `Batman`: the definition of `Batphone` is equivalent to `[dial = @(e)Batman]`.

Some fairly powerful type systems cannot handle this example: Palsberg and Jim [PJ97] noted that this program is *not* typable in Abadi and Cardelli's type system $\mathbf{Ob}_{1<:\mu}$, which has recursive types, width subtyping for object types, and only invariant fields. The reason is that in this calculus, `Batman` is not a subtype of `Man`:

$$\begin{array}{rcll} \texttt{Man} & : & \mu(X)[\texttt{id}^0 : X] & \text{(from the definition of Man)} \\ \texttt{Batman} & : & \mu(Y)[\texttt{id}^0, \texttt{secretid}^0 : Y] & \text{(from the definition of Batman)} \\ \mu(Y)[\texttt{id}^0, \texttt{secretid}^0 : Y] & \not\leq & \mu(X)[\texttt{id}^0 : X] & \text{(from the definition of subtyping)} \end{array}$$

The `id` field of `Batman` has `Batman`'s type, and the `id` field of `Man` has `Man`'s type, so by invariant subtyping, the two types are not related. The only common type of `Man` and `Batman` is $[\,]$, so we have

$$\begin{array}{rcl} \texttt{Phone} & : & [\texttt{dial}^0 : [\,]] \\ \texttt{Batphone} & : & [\texttt{dial}^0 : [\,]] \end{array}$$

In conclusion, `call (call Batphone.dial).id` is *not* typable.

To increase expressiveness, we can add CROFs to $\mathbf{Ob}_{1<:\mu}$ such that each field can be either invariant or covariant:

$$\mathbf{Ob}_{1<:\mu+} = \mathbf{Ob}_{1<:\mu} \cup \text{CROFs}.$$

This is sufficient to make the above program type check with the following types:

```
Man       :  [id⁰: [ ]]
Batman    :  [id⁰: [secretid⁺: [id⁺: [ ]]]; secretid⁰: [id⁺: [ ]]]
Phone     :  [dial⁰: [id⁺: [ ]]]
Batphone  :  [dial⁰: [id⁺: [ ]]]
```

These types were produced by our implementation of the algorithm presented in this chapter. Notice that the program (the input to our algorithm) does not mention whether a field is read-only, or whether it can both be read and updated. It turns out that this information does not make the problem of type inference any easier, so our algorithm automatically *discovers* which fields should be CROFs.

The types produced by our algorithm are intended to be minimal-shape types, in the sense of [KPS94, PWO97], although proving that is left to future work. The idea of minimal shape is that the inferred types exhibit exactly the amount of structure of objects that the program exploits. For example, we have `Man` : $[\mathtt{id}^0 : [\,]]$ because there is no usage of `Man.id`, so the return type of `Man.id` need not have any features at all. Note that minimal-shape types need not be principal types. If principal types are desired, then the best known approach is to output a representation of the type constraints used by the type inference algorithm, possibly in a simplified form [Pot96]. In the worst case, such a set of constraints has a size that is linear in the size of the program.

One can achieve a degree of modularity by letting the programmer specify types at, say, module boundaries. Those types then become part of the input to the type inference algorithm, so, in effect, the algorithm determines whether there is a typing which is consistent with the specified types. Without any such declared types, the inferred types can be quite

different for two programs that use their objects in slightly different ways. This may not be desirable for large-scale software development. Similarly, slightly different programs may have quite different inferred annotations of the fields of the object types. It remains an open problem to find a convenient mechanism to explain to a programmer why a certain field ended up being a CROF.

The above program is also typable with the so-called simple self-types of Palsberg and Jim [PJ97]. Bugliesi and Pericas-Geertsen [BPG02] observed that any program that can be typed with simple self-types can also be typed in $\mathbf{Ob}_{1<:\mu+}$. Perhaps interestingly, their encoding of simple self-types uses recursive types to type the above program, while our more direct algorithm produces nonrecursive types in this case. More importantly, type inference with simple self-types is NP-complete [PJ97], while our type inference algorithm for the more expressive type system $\mathbf{Ob}_{1<:\mu+}$ runs in polynomial time.

In our implementation, a programmer can specify that some of the fields are read-only. For example, in the above program, a programmer may specify that the `secretid` field is read-only by inserting a `+` annotation:

```
let
  Man      = [id = @(x)x]
  Batman   = [id = @(y)y  secretidˆ+ = @(z)z]
  Phone    = [dial = @(d)Man]
  Batphone = update Phone.dial <=
     @(e) call (call Batman.id).secretid
in
  call (call Batphone.dial).id
```

For this program, our implementation produces the following types:

```
Man       :  [id⁰: [ ]]
Batman    :  [id⁰: [secretid⁺: [id⁺: [ ]]]; secretid⁺: [id⁺: [ ]]]
Phone     :  [dial⁰: [id⁺: [ ]]]
Batphone  :  [dial⁰: [id⁺: [ ]]]
```

Notice that the second occurrence of `secretid` in the type for `Batman` now has the annotation `+` instead of `0`.

If, in addition, the program were changed to make the `dial` method of `Phone` read-only, then our implementation would correctly decide that the resulting program is not typable (because `dial` is updated in `Batphone`).

As we have mentioned, adding annotations, such as the one for `secretid`, seems not to make the type inference problem easier. In this chapter, we show that even if all fields are explicitly specified as either updateable or read-only, the type inference problem is P-complete. If the programmer leaves a field unspecified (or, equivalently, gives the annotation 0), then our algorithm will *discover* whether it is advantageous to make it read-only. The "discovery" process takes place at the type level: a field which was annotated as read-write may be implicitly turned into a read-only field at any time via subsumption.

Glew's type system Glew's translation of objects and classes to a typed intermediate language [Gle00] is an important motivation for our work on CROFs. Like most implementations of object-oriented languages, Glew's translation uses method tables. One of Glew's insights is that the method table can conveniently be placed in a CROF. For example, let a and b be two source-language objects such that the type of b is a subtype of the type of a. The type system for the source language supports that b may have *more* methods than a (width subtyping). This means that the method table in the translation of b will be *longer* than the method table in the translation of a:

$$\begin{aligned} \text{translation } (a) &= \ldots [\text{mt} = m_a, \ldots] \ldots \\ \text{translation } (b) &= \ldots [\text{mt} = m_b, \ldots] \ldots \end{aligned}$$

where mt is the field name for the method table. Glew's translation of b has a subtype of the type of his translation of a; he makes mt a CROF, and he gives the following types to the translations of a and b:

$$\begin{aligned} \text{type-of } (\text{translation } (a)) &= \ldots [\text{mt}^+ : \text{type-of } (m_a), \ldots] \ldots \\ \text{type-of } (\text{translation } (b)) &= \ldots [\text{mt}^+ : \text{type-of } (m_b), \ldots] \ldots \end{aligned}$$

Glew's translation produces typed intermediate code, including the annotations 0 and $+$.

Is type inference possible for an implicitly-typed version of Glew's intermediate language? Our work shows that type inference is possible for a fragment of Glew's type system. Glew's type system also features function types, bounded universal polymorphism, and self types, as well as a special variance annotation for record types which indicates whether subtyping can be applied at all. Our long-term goal is to extend the algorithm to cover a larger fragment of Glew's system. Such an algorithm would make it possible to omit bulky type annotations, and to automatically discover the CROFs.

Constraint solving A $O(n^3)$ time type inference algorithm for Abadi and Cardelli's type system $\mathbf{Ob}_{1<:\mu}$ (only invariant fields) was given by Palsberg [Pal95]; later, an $O(n^2)$ time algorithm for the same problem was given by Henglein [Hen97]. There is a similar $O(n^3)$ type inference algorithm for the calculus with only CROFs (only covariant fields). Surprisingly, there seems to be no easy way to "merge" the two algorithms to obtain an algorithm for the combined type system, $\mathbf{Ob}_{1<:\mu+}$. Both algorithms work by reducing type inference to the problem of solving a set of *constraints*. A constraint is a pair (A, B), where A and B are types that may contain type variables; and the goal is to find a substitution S such that for each constraint (A, B), we have $S(A) \leq S(B)$ where $\leq$ is the subtype order. We will use R to range over sets of constraints; we will often refer to R as a relation on types. A key theorem about both algorithms states:

Theorem A set of constraints is solvable if and only if its closure is consistent.

Here, "closure" means that certain syntactic consequences of the constraints have been added to the constraint set, and "consistent" means that there are no obviously unsatisfiable constraints (e.g., $([\,], [\ell^0 : [\,]])$). Both algorithms construct a solution from a closed, consistent constraint set. This framework has been used for solving subtype constraints for a variety of types [Pal95, KPS94, PWO97, PO95, PS96].

Finding an algorithm thus rests on finding a correct definition of "closure." For the type system with only covariant fields, there are three closure rules, all operating on a constraint set R:

- if $(A, B) \in R$, then $(A, A), (B, B) \in R$ (reflexivity);
- if $(A, B), (B, C) \in R$, then $(A, C) \in R$ (transitivity);
- if $([\ell^+ : B, \ldots], [\ell^+ : B', \ldots]) \in R$, then $(B, B') \in R$ (propagation of subtyping to fields).

Computing the closure takes $O(n^3)$ time. For $\mathbf{Ob}_{1<:\mu}$ (only invariant fields), there are also three closure rules:

- if $(A, B) \in R$, then $(A, A), (B, B) \in R$ (reflexivity);
- if $(A, B), (B, C) \in R$, then $(A, C) \in R$ (transitivity);
- if $(A, [\ell^0 : B, \ldots]), (A, [\ell^0 : B', \ldots]) \in R$, then $(B, B') \in R$ (propagation of subtyping to fields).

Notice that the last rule also can be used to give (B', B), so it actually forces B and B' to be unified. Now, can we solve constraints over the types in $\mathbf{Ob}_{1<:\mu+}$ by taking the union of the two sets of closure rules? As it happens, a notion of closure based on the union of the rules does *not* support the result mentioned above. For example, consider the constraint set that consists of the following two constraints:

$$(V, [\ell^+ : B]) \qquad (V, [\ell^+ : B'])$$

where V is a type variable, $B = [m^0 : [\,]]$, and $B' = [m^0 : [m^0 : [\,]]]$. The key property of B, B' is that they don't have a common lower bound. Apart from reflexivity, this constraint set is closed under the four closure rules above. Moreover, the constraint set is consistent: there are no obviously unsatisfiable constraints. So, if there was a theorem of the form mentioned above, this constraint set should be solvable. However, it is not solvable. To see that, consider the following informal argument. In any solution, V must be assigned a type of the form $[\ell^+ : A, \ldots]$, for some A. (Actually, it might also be possible to annotate ℓ with 0, but that will not help). Now, because ℓ is a CROF, we must be able to satisfy the constraints:

$$(A, B) \qquad (A, B')$$

This is not possible: there is no subtype of both B and B' in this system. With only covariant fields, the set of types form a lattice. Once invariant fields are introduced, not all pairs of types have lower bounds. In our setting with both covariant and invariant fields, all pairs of types have a least upper bound. Conclusion: either there are too few closure rules, or else there is something wrong with the notion of consistency.

The example suggests that in a setting with both covariant and invariant fields, a new technique is called for.

3.1.2 Our Results

We present the design and implementation of a type inference algorithm for $\mathbf{Ob}_{1<:\mu+}$. The algorithm automatically discovers CROFs. It is based on a theorem of the form discussed above, with a new notion of closure and a traditional notion of consistency. Type inference is equivalent to solving type constraints, which in turn is P-complete and computable in $O(n^3)$ time. The novel aspect of our definition of closure is that it keeps track of both subtype relations and which pairs of types must have a lower bound. For the example constraint set above, our closure rules will note that since $[\ell^+ : B]$ and $[\ell^+ : B']$ must have a lower bound, it must also be the case that B and B' have a lower bound, which is obviously false. Our nine closure rules describe the interaction between a set of subtype constraints and a set of lower-bound constraints. In our proof of the main theorem (of the form mentioned above), we use a technique that employs a convenient characterization of the subtyping order (Lemma 3.2.6). The characterization uses notions of subtype-closure and subtype-consistency that are different, yet closely related, to the already-mentioned notions of what we for clarity will call satisfaction-closure and satisfaction-consistency. The result that type inference is P-hard indicates there are no fast NC-class parallel algorithms for the type inference problem, unless NC = P.

Our prototype implementation, already showcased above, works with a version of an Abadi-Cardelli object calculus. The implementation is freely available from:

http://www.cs.purdue.edu/homes/tzhao/type-inference/inference.htm

Future work includes the addition of atomic subtyping [Mit91, Tiu92, HM95, Fre97,

Ben94].

3.1.3 Related Work

One of the first uses of annotations such as $+$, often called *variance annotations*, can be found in Pierce and Sangiorgi's paper [PS93] on typing and subtyping for mobile processes. They used annotations of types in a type system for the π-calculus to enforce that some channels are for input only or for output only.

The variance annotation "$-$" is sometimes used to denote that a field is contravariant and write-only. We know of no easy way of extending the results of this chapter to cover "$-$". The main problem is that for a system with invariant, covariant, and contravariant fields, not all pairs of types have least upper bounds. For example, consider the types

$$\begin{aligned} C &= [l^0 : A] \\ C' &= [l^0 : B] \\ A &= [\,] \\ B &= [m^0 : [\,]]. \end{aligned}$$

Notice that $B \leq A$. Here are two incomparable upper bounds of C, C':

$$\begin{aligned} &[l^+ : A] \\ &[l^- : B]. \end{aligned}$$

However, there is no *least* upper bound of C, C'. So, with both covariant and contravariant fields, the types do *not* form lattice.

Rémy [Rém98] used covariant and contravariant fields in a calculus with object extension, and depth and width subtyping. His language is explicitly typed.

Igarashi and Viroli [IV02] used variance annotations to control subtyping between different instantiations of a generic class, and to specify the visibility of fields and methods. Their example language is explicitly typed.

Igarashi and Kobayashi [IK00] showed how to infer types with annotations about the *uses* of communication channels in concurrent programs. A use is either 0 (never used),

1 (used at most once), or ω (used arbitrarily). The set of uses forms an algebra with operations such as $0 + 1 = 1$. Depending on the use annotations of a record type, the fields may enjoy covariant or contravariant subtyping. Their approach differs from ours in the use of variance variables (called *use variables*), whose value determines whether a channel type is contravariant, covariant or invariant. Because these are variables, structural decomposition (closure) is sometimes be suspended until some of these variables receive values, which requires a form of conditional constraints. Our approach does not use conditional constraints and therefore appear simpler. Another notable difference between their type inference problem and ours is that their type inference problem uses finite types without width subtyping and with all of covariance, contravariance, and invariance, while ours uses recursive types and width subtyping, but only covariance and invariance. It remains to be seen whether it is possible to extend their techniques to handle width subtyping and recursive types.

Tang and Hofmann [TH01] studied type inference for a logic of Abadi and Leino, for the purpose of helping with automatic generation of verification conditions [TH02]. They use a subtyping relation for object types in which fields are invariant and methods are covariant. Thus, in their type inference problem it is explicitly specified what is read-only and what is updateable. The two most notable differences between their type inference problem and ours are that (1) they consider finite types while we study recursive types and (2) we enable automatic discovery of CROFs. Their work was carried out independently of ours; the technical approaches have some basic ideas in common. In particular, most of our nine rules for satisfaction-closure seems to have counterparts in Tang and Hofmann's approach.

Our type system does not contain a bottom type, that is, a least type. Adding a bottom type would make the set of types form a lattice, rather than a semilattice, and it would make more programs type check [WOP95]. Type inference with bottom types have been investigated for various type systems [PWO97, PO95, PS96, BPG02], and in each case type inference can be done in cubic time in the size of the program. Thus, we can expect type inference for our type system extended with a bottom type do be in cubic time. However,

bottom types seem not to be popular. Java does have a top type, that is, a greatest type, called Object, but it does not have a bottom type. Above, we discussed a range of previous work [AC96a, Gle00, PS93, IV02, IK00, TH01, TH02] in which notions of covariant fields were added to type systems in order to type check more programs, and yet *none* of those papers have a bottom type in their type systems. Pierce [Pie02] discusses ways in which adding a bottom type to a type system complicates matters considerably, particularly in systems with bounded quantification. Based on these observations, we feel that even though adding a bottom type simplifies type inference, it is well justified to study a type system without a bottom type,

3.1.4 Examples

We now present two examples that give a taste of the definitions and techniques that are used later in the chapter. We invite the reader to revisit the examples after reading the technical part of the chapter. In the first example we return to the program with Man and Batman. We use the program to illustrate the reduction of the type inference problem to a constraint problem. In the abstract syntax of an Abadi-Cardelli object calculus, we can write the program as follows:

$$\begin{aligned}
\text{Man} &= [\text{id} = \varsigma(x)x] \\
\text{Batman} &= [\text{id} = \varsigma(y)y,\ \text{secretid} = \varsigma(z)z] \\
\text{Phone} &= [\text{dial} = \varsigma(d)\text{Man}] \\
\text{Batphone} &= \text{Phone.dial} \Leftarrow \varsigma(e)\text{Batman.id.secretid} \\
\text{Main} &= \text{Batphone.dial.id}
\end{aligned}$$

We can use the rules in Section 3.4.1 to generate constraints in Figure 3.1. In the left column are all occurrences of subterms in the program; in the right column are the constraints generated for each occurrence. We use $A \equiv B$ to denote the pair of constraints (A, B) and (B, A).

The constraint set in Figure 3.1 is solvable and a solution can be found by running our constraint solving algorithm. The solution that will be generated was displayed earlier in this section; it corresponds to the following type derivation. Define the types P, Q and the

Occurrence	Constraints
x	(U_x, V_x)
Man	$([\text{id}^0 : V_x], V_{Man})$
	$U_x \equiv [\text{id}^0 : V_x]$
y	(U_y, V_y)
z	(U_z, V_z)
Batman	$([\text{id}^0 : V_y,\ \text{secretid}^0 : V_z], V_{Batman})$
	$U_y \equiv [\text{id}^0 : V_y,\ \text{secretid}^0 : V_z]$
	$U_z \equiv [\text{id}^0 : V_y,\ \text{secretid}^0 : V_z]$
Phone	$([\text{dial}^0 : V_{Man}], V_{Phone})$
	$U_d \equiv [\text{dial}^0 : V_{Man}]$
Batphone	$(V_{Phone}, V_{Batphone})$
	$V_{Phone} \equiv U_e$
	$(V_{Phone}, [\text{dial}^0 : V_{Batman.id.secretid}])$
Batman.id	$(V_{Batman}, [\text{id}^+ : V_{Batman.id}])$
Batman.id.secretid	$(V_{Batman.id}, [\text{secretid}^+ : V_{Batman.id.secretid}])$
Batphone.dial	$(V_{Batphone}, [\text{dial}^+ : V_{Batphone.dial}])$
Batphone.dial.id	$(V_{Batphone.dial}, [\text{id}^+ : V_{Batphone.dial.id}])$

Figure 3.1. Constraints for the example program

environments E, F:

$$\begin{aligned} P &= [\text{id}^+ : [\,]] \\ Q &= [\text{id}^0 : [\text{secretid}^+ : P],\ \text{secretid}^0 : P] \\ E &= \emptyset[d : [\text{dial}^0 : P]] \\ F &= \emptyset[e : [\text{dial}^0 : P]]. \end{aligned}$$

We can derive $\emptyset \vdash$ Batphone.dial.id : [ ] as follows, using the type rules to be presented in Section 3. The number to the right of each horizontal line indicates which type rule was used.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{E[x : [\text{id}^0 : [\,]]] \vdash x : [\text{id}^0 : [\,]]}{E[x : [\text{id}^0 : [\,]]] \vdash x : [\,]}(3.5)}{E \vdash \text{Man} : [\text{id}^0 : [\,]]}(3.2)}{E \vdash \text{Man} : P}(3.5)}{\emptyset \vdash \text{Phone} : [\text{dial}^0 : P]}(3.2) \quad \dfrac{}{F \vdash \text{Batman.id.secretid} : P}(3.3)}{\dfrac{\dfrac{\emptyset \vdash \text{Batphone} : [\text{dial}^0 : P]}{\emptyset \vdash \text{Batphone.dial} : P}(3.3)}{\emptyset \vdash \text{Batphone.dial.id} : [\,]}(3.3)}(3.4)$$

Also,

$$\dfrac{\dfrac{\dfrac{F[y : Q] \vdash y : Q}{F[y : Q] \vdash y : [\text{secretid}^+ : P]}(3.5) \quad \dfrac{F[z : Q] \vdash z : Q}{F[z : Q] \vdash z : P}(3.5)}{F \vdash \text{Batman} : Q}(3.2)}{\dfrac{\dfrac{F \vdash \text{Batman} : [\text{id}^+ : [\text{secretid}^+ : P]]}{F \vdash \text{Batman.id} : [\text{secretid}^+ : P]}(3.3)}{F \vdash \text{Batman.id.secretid} : P}(3.3)}(3.5)$$

Notice the five uses of subsumption:

$$\begin{aligned} [\text{id}^0 : [\,]] &\leq [\,] \\ [\text{id}^0 : [\,]] &\leq P \\ Q &\leq [\text{secretid}^+ : P] \\ Q &\leq P \\ Q &\leq [\text{id}^+ : [\text{secretid}^+ : P]]. \end{aligned}$$

Satisfaction-closure of R (excerpt)	Lower-bound relation (excerpt)
$(U \quad , \quad [\ell^+ : [\ell^+ : [m^+ : [\ ]]]])$	$(\ [\ell^+ : [\ell^0 : W]] \quad , \quad [\ell^+ : [\ell^+ : [m^+ : [\ ]]]]\ )$
$(U \quad , \quad [\ell^+ : [\ell^0 : W]])$	$(\ [\ell^0 : W] \quad , \quad [\ell^+ : [m^+ : [\ ]]]\ )$
$(W \quad , \quad [m^+ : [\ ]])$.	

Figure 3.2. The satisfaction-closure (excerpt) of two constraints

Our second example illustrates our algorithm for solving constraints, particularly the role of the closure operation. Let R consist of the following two constraints:

$$(U, [\ell^+ : [\ell^+ : [m^+ : [\ ]]]])$$
$$(U, [\ell^+ : [\ell^0 : W]]),$$

where U, W are type variables and ℓ, m are labels of fields. The satisfaction-closure of R and the accompanying lower-bound relation are shown in Figure 3.2 A pair (A, B) is in the lower-bound relation when it has been deduced that A and B must have a lower bound in the subtype ordering.

Let us now explain how the rules for satisfaction-closure (Definition 3.5.1) generate the constraints in the table above. Since both $(U, [\ell^+ : [\ell^0 : W]])$ and $(U, [\ell^+ : [\ell^+ : [m^+ : [\ ]]]])$ are in the sat-closure of R, we have from Lemma 3.5.3, Property **(C)**, that $([\ell^+ : [\ell^0 : W]], [\ell^+ : [\ell^+ : [m^+ : [\ ]]]])$ is in the lower bound relation, and hence, from sat-closure rule (vii), we have that also $([\ell^0 : W], [\ell^+ : [m^+ : [\ ]]])$ is in the lower-bound relation. Finally, since $([\ell^0 : W], [\ell^+ : [m^+ : [\ ]]])$ is in the lower-bound relation, we have from sat-closure rule (viii) that $(W, [m^+ : [\ ]])$ is in the sat-closure of R. Given the sat-closure of R, call it R', our algorithm checks for satisfaction-inconsistency, that is, subtyping constraints that obviously are unsatisfiable. In this case, the satisfaction-closure is satisfaction-consistent, and our algorithm then constructs the following solution $S_{R'}$:

$$\begin{aligned} S_{R'}(U) &= [\ell^+ : [\ell^0 : [m^+ : [\ ]]]] \\ S_{R'}(W) &= [m^+ : [\ ]]. \end{aligned}$$

One might try to devise a constraint solving algorithm that would be an alternative to ours. The constraint set R is a good benchmark: it seems nontrivial to derive $(W, [m^+ : [\ ]])$ without the help of a lower-bound relation.

Paper overview In Section 3.2, we define types and subtyping, and we give a decision procedure for subtyping. In Section 3.3 we present an extension of an Abadi-Cardelli object calculus, and in Section 3.4 we show that the type inference problem for that calculus is equivalent to a constraint problem. In Section 3.5 we give an $O(n^3)$-time algorithm for solving constraints, and in Section 3.6 we show that the constraint problem is P-hard.

3.2 Types and subtyping

We will work with recursive types, and we choose to represent them by possibly infinite trees.

3.2.1 Defining types as infinite trees

We use U, V to range over the set $\mathcal{TV}$ of type variables; we use k, ℓ, m to range over labels drawn from some possibly infinite set Labels of method names; and we use v to range over the set $\mathsf{Variances} = \{0, +\}$ of variance annotations. Variance annotations are ordered by the partial order $\sqsubseteq$ such that $0 \sqsubseteq +$, $0 \sqsubseteq 0$, and $+ \sqsubseteq +$.

The alphabet Σ of our trees is defined

$$\Sigma = \mathcal{TV} \cup \{\sigma \subseteq \mathsf{Labels} \times \mathsf{Variances} \mid (\ell, v), (\ell, v') \in \sigma \Rightarrow v = v'\}.$$

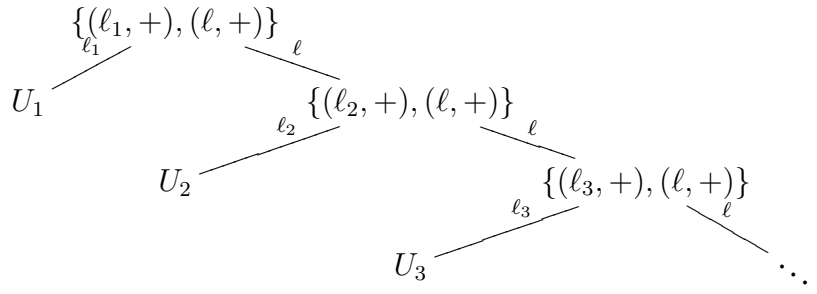
A *path* is a finite sequence $\alpha \in \mathsf{Labels}^*$ of labels, with juxtaposition for concatenation of paths, and ϵ for the empty sequence. A *type* or *tree* A is a partial function from paths into Σ, whose domain is nonempty and prefix closed, and such that $(\ell, v) \in A(\alpha)$ for some $v \in \mathsf{Variances}$ if and only if $A(\alpha\ell)$ is defined. We use A, B, C to range over the set $\mathcal{T}(\Sigma)$ of trees. Notice that types can contain type variables as leaves. We could simplify the development by restricting ourselves to ground types. For example, the notions of subtype-consistency and satisfaction-consistency would then coincide. However, we feel that it is important to show how to handle type variables and therefore we do that in detail.

Note that trees need not be finitely branching or regular. Of course, we will be partic-

ularly interested in two subsets of $\mathcal{T}(\Sigma)$, the finite trees $\mathcal{T}_{\text{fin}}(\Sigma)$ and the finitely branching and regular trees $\mathcal{T}_{\text{reg}}(\Sigma)$. Some definitions, results, and proofs are given in terms of $\mathcal{T}(\Sigma)$, in such a way that they immediately apply to $\mathcal{T}_{\text{fin}}(\Sigma)$ and $\mathcal{T}_{\text{reg}}(\Sigma)$. In particular, we will state conditions on whether one tree is a subtype of another that result in an *algorithm* in case both trees are in $\mathcal{T}_{\text{fin}}(\Sigma)$ or $\mathcal{T}_{\text{reg}}(\Sigma)$.

An example tree is given below, representing the nonregular type

$$[\ell_1^+ : U_1, \ell^+ : [\ell_2^+ : U_2, \ell^+ : [\ell_3^+ : U_3, [\ell^+ : \ldots]]]].$$



We now introduce some convenient notation. We write $A(\alpha) = \uparrow$ if A is undefined on α. If for all $i \in I$, B_i is a tree, ℓ_i is a distinct label, and $v_i \in \mathsf{Variances}$, then $[\ell_i^{v_i} : B_i \quad {}^{i \in I}]$ is the tree A such that

$$A(\alpha) \quad = \quad \begin{cases} \{(\ell_i, v_i) \mid i \in I\} & \text{if } \alpha = \epsilon \\ B_i(\alpha') & \text{if } \alpha = \ell_i \alpha' \text{ for some } i \in I \\ \uparrow & \text{otherwise.} \end{cases}$$

We abuse notation and write U for the tree A such that $A(\epsilon)$ is the type variable U and $A(\alpha) = \uparrow$ for all $\alpha \neq \epsilon$.

3.2.2 Defining subtyping via simulations

Definition 3.2.1. A relation R over $\mathcal{T}(\Sigma)$ is called a *simulation* if, whenever $(A, A') \in R$, we have the following conditions.

- For all U, $A = U$ if and only if $A' = U$.
- For all ℓ, v', B', if $A' = [\ell^{v'} : B', \ldots]$, then there exist v, B such that $A = [\ell^{v} : B, \ldots]$, $v \sqsubseteq v'$, and

- $(B, B') \in R$, and
- $v' = 0$ implies $(B', B) \in R$.

□

For example, the empty relation on $\mathcal{T}(\Sigma)$ and the identity relation on $\mathcal{T}(\Sigma)$ are both simulations. Simulations are closed under unions and intersections, and there is a largest simulation, which we call $\leq$:

$$\leq = \bigcup\{R \mid R \text{ is a simulation}\}.$$

Alternately, $\leq$ can be seen as the maximal fixed point of a monotone function on $\mathcal{P}(\mathcal{T}(\Sigma) \times \mathcal{T}(\Sigma))$. Then we immediately have the following result.

Lemma 3.2.2. *$A \leq A'$ if and only if*

- *For all U, $A = U$ if and only if $A' = U$.*
- *For all ℓ, v', B', if $A' = [\ell^{v'} : B', \ldots]$, then there exist v, B such that $A = [\ell^{v} : B, \ldots]$, $v \sqsubseteq v'$, and*
  - *$B \leq B'$, and*
  - *$v' = 0$ implies $B' \leq B$.*

All of these results are standard in concurrency theory, and have easy proofs, cf. [Mil90]. Similarly, it is easy to show that $\leq$ is a preorder. Our simulations differ from the simulations typically found in concurrency in that they are all anti-symmetric (again, the proof is easy).

Lemma 3.2.3. *$\leq$ is a partial order.*

Proof. See Appendix A.3. □

We may apply the principle of *co-induction* to prove that one type is a subtype of another:

> **Co-induction**: To show $A \leq B$, it is sufficient to find a simulation R such that $(A, B) \in R$.

3.2.3 An algorithm for subtyping

The co-induction principle results in an easy algorithm for subtyping on $\mathcal{T}_{\text{fin}}(\Sigma)$ and $\mathcal{T}_{\text{reg}}(\Sigma)$. Suppose R is a relation on types, and we want to know whether $A \leq B$ for every $(A, B) \in R$. By co-induction this is equivalent to the existence of a simulation containing R. And since simulations are closed under intersection, this is equivalent to the existence of a *smallest* simulation containing R. We can characterize this smallest simulation as follows.

Definition 3.2.4. We say a relation R on types is *subtype-closed* if it satisfies the following two properties.

- If $([\ell^v : B, \ldots], [\ell^{v'} : B', \ldots]) \in R$ and $v \sqsubseteq v'$, then $(B, B') \in R$.
- If $([\ell^0 : B, \ldots], [\ell^0 : B', \ldots]) \in R$, then $(B', B) \in R$.

□

Note that the subtype-closed relations on types are closed under intersection; therefore for any relation R on types, we may define its *subtype-closure* to be the smallest subtype-closed relation containing R. Every simulation is subtype-closed, and subtype-closure is a monotone operation.

Definition 3.2.5. We say a relation R on types is *subtype-inconsistent* if any of the following cases hold.

- $(U, A) \in R$ or $(A, U) \in R$ for some distinct U and A.
- $([\ell^+ : B, \ldots], [\ell^0 : B', \ldots]) \in R$ for some ℓ, B, B'.
- $([\ell_i^{v_i} : B_i \ {}^{i \in I}], [\ell^v : B, \ldots]) \in R$ for some ℓ, v, B, and ℓ_i, v_i, B_i for $i \in I$; and furthermore $\ell \neq \ell_i$ for all $i \in I$.

We say R is *subtype-consistent* iff R is not subtype-inconsistent. □

Note that every simulation is subtype-consistent, and moreover, any subset of a subtype-consistent set is subtype-consistent.

Lemma 3.2.6. *Let R be a relation on types. The following statements are equivalent.*

i) $A \leq B$ *for every* $(A, B) \in R$.

ii) *The subtype-closure of* R *is a simulation.*

iii) *The subtype-closure of* R *is subtype-consistent.*

Proof.

- (ii) $\Rightarrow$ (i): Suppose $(A, B) \in R$ and let R' be the subtype-closure of R. We have $(A, B) \in R \subseteq R'$ and that R' is a simulation, so by co-induction we have that $A \leq B$.

- (i) $\Rightarrow$ (iii): R is a subset of $\leq$, so by monotonicity and the fact that $\leq$ is subtype-closed, the subtype-closure of R is a subset of $\leq$. Then since $\leq$ is subtype-consistent, its subset, the subtype-closure of R, is subtype-consistent.

- (iii) $\Rightarrow$ (ii): Let R' be the subtype-closure of R, and suppose $(A, A') \in R'$.

  If $A = U$, by subtype-consistency $A' = U$; and similarly, if $A' = U$, then $A = U$.

  If $A' = [\ell^{v'} : B', \ldots]$, by subtype-consistency A must be of the form $[\ell^{v} : B, \ldots]$, where $v \sqsubseteq v'$. From $v \sqsubseteq v'$ and R' being subtype-closed, we have $(B, B') \in R'$. If $v' = 0$, then from $v \sqsubseteq v'$ we have $v = 0$, so from R' being subtype-closed, we have $(B', B) \in R'$.

□

This immediately suggests an algorithm for testing whether $A \leq B$ for $A, B \in \mathcal{T}_{\text{reg}}(\Sigma)$: construct the subtype-closure of $\{(A, B)\}$ and test whether it is subtype-consistent. If n is the number of distinct subtrees of A and B, then the subtype-closure of $\{(A, B)\}$ is of size at most the maximum number of pairs of subtrees from A and B, that is, n^2, and can be constructed in n^2 time. Consistency checking is linear, so the algorithm runs in time $O(n^2)$.

Theorem 3.2.7. *Subtyping on $\mathcal{T}_{\text{reg}}(\Sigma)$ is decidable in $O(n^2)$ time.*

In the remainder of the chapter, we only consider types in $\mathcal{T}_{\text{reg}}(\Sigma)$.

3.3 An Abadi-Cardelli Object Calculus

We now present an extension of an Abadi-Cardelli object calculus [AC96a] and a static type system.

We use x, y to range over term variables. Expressions are defined by the following grammar.

a, b, c	$::=$	x	variable
	$\mid$	$[\ell_i^{v_i} = \varsigma(x_i)b_i\ ^{i\in 1..n}]$	object (ℓ_i distinct)
	$\mid$	$a.\ell$	field selection / method invocation
	$\mid$	$(a.\ell \Leftarrow \varsigma(x)b)$	field update / method update

An object $[\ell_i^{v_i} = \varsigma(x_i)b_i\ ^{i\in 1..n}]$ has method names ℓ_i and methods $\varsigma(x_i)b_i$. The order of the methods does not matter. Each method $\varsigma(x)b$ binds a name x which denotes the smallest enclosing object, much like "this" in Java. Those names can be chosen to be different, so within a nesting of objects, one can refer to any enclosing object. Each method name ℓ_i is annotated with a variance annotation $v_i \in \{0, +\}$ which in the case of 0 indicates that the method is both readable/invocable and writable/updateable, while it in the case of + indicates that the method is only readable/invocable. As syntactic sugar, we will allow variance annotations to be omitted, and in such cases the default is 0. With this default, our calculus is an extension of an Abadi-Cardelli calculus [AC96a]: a term in an Abadi-Cardelli calculus is also a term in our calculus, namely one where all variance annotations implicitly are 0. A *value* is of the form $[\ell_i^{v_i} = \varsigma(x_i)b_i\ ^{i\in 1..n}]$. A *program* is a closed expression.

A confluent, small-step operational semantics is defined by the following rules:

- If $a = [\ell_i^{v_i} = \varsigma(x_i)b_i\ ^{i\in 1..n}]$, then, for $j \in 1..n$,
  - $a.\ell_j \rightsquigarrow b_j[x_j := a]$, and
  - if $v_j = 0$, then $(a.\ell_j \Leftarrow \varsigma(y)b) \rightsquigarrow a[\ell_j \leftarrow \varsigma(y)b]$.
- If $b \rightsquigarrow b'$, then $a[b] \rightsquigarrow a[b']$.

Here, $b_j[x_j := a]$ denotes the expression b_j with a substituted for free occurrences of x_j (renaming bound variables to avoid capture); and $a[\ell_j \leftarrow \varsigma(y)b]$ denotes the expression

a with the ℓ_j field replaced by $\varsigma(y)b$, preserving the variance annotation. A *context* is an expression with one hole, and $a[b]$ denotes the term formed by replacing the hole of the context $a[\cdot]$ by the term b (possibly capturing free variables in b).

An expression b is *stuck* if it is not a value and there is no expression b' such that $b \rightsquigarrow b'$. An expression b *goes wrong* if $\exists b' : b \rightsquigarrow^* b'$ and b' is stuck.

A type environment is a partial function with finite domain which maps term variables to types in $\mathcal{T}_{\mathrm{reg}}(\Sigma)$. We use E to range over type environments. We use $E[x : A]$ to denote a partial function which maps x to A, and maps y, where $y \neq x$, to $E(y)$.

The typing rules below allow us to derive judgments of the form $E \vdash a : A$, where E is a type environment, a is an expression, and A is a type in $\mathcal{T}_{\mathrm{reg}}(\Sigma)$.

$$E \vdash x : A \quad (\text{provided } E(x) = A) \tag{3.1}$$

$$\frac{E[x_i : A] \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [\ell_i^{v_i} = \varsigma(x_i)b_i \,^{i \in 1..n}] : A} \quad (\text{where } A = [\ell_i : B_i \,^{i \in 1..n}]) \tag{3.2}$$

$$\frac{E \vdash a : A}{E \vdash a.\ell : B} \quad (\text{where } A \leq [\ell^+ : B]) \tag{3.3}$$

$$\frac{E \vdash a : A \quad E[x : A] \vdash b : B}{E \vdash a.\ell \Leftarrow \varsigma(x)b : A} \quad (\text{where } A \leq [\ell^0 : B]) \tag{3.4}$$

$$\frac{E \vdash a : A}{E \vdash a : B} \quad (\text{where } A \leq B) \tag{3.5}$$

The first four rules express the typing of each of the four constructs in the object calculus and the last rule is the rule of subsumption. We say that a program a is *well-typed* if $\emptyset \vdash a : A$ is derivable for some A. The following result can be proved by a well-known technique [Nie89, WF94].

Theorem 3.3.1. (Type Soundness) *Well-typed programs cannot go wrong.*

The type inference problem for our extension of an Abadi-Cardelli calculus is: given a term a, find a type environment E and a type A such that $E \vdash a : A$, or decide that this is impossible.

3.4 Type Inference is equivalent to Constraint Solving

A *substitution* S is a finite partial function from type variables to types in $\mathcal{T}_{\mathrm{reg}}(\Sigma)$, written $\{U_1 := A_1, \ldots, U_n := A_n\}$. The set $\{U_1, \ldots, U_n\}$ is called the *domain* of the substitution. We identify substitutions with their graphs, and write $(S_1 \cup S_2)$ for the union of two substitutions S_1 and S_2; by convention, we assume that S_1 and S_2 agree on variables in their common domain, so $(S_1 \cup S_2)$ is a substitution. Substitutions are extended to total functions from types to types in the usual way.

A relation R over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$ is solvable if and only if there is a substitution S such that for all $(A, B) \in R$, we have $S(A) \leq S(B)$. In the remainder of the chapter, we use R to range over finite relations on $\mathcal{T}_{\mathrm{reg}}(\Sigma)$. We will refer to each $(A, B) \in R$ as a constraint, and to R as a set of constraints. For a finite relation R on $\mathcal{T}_{\mathrm{reg}}(\Sigma)$, the size of R is the sum of the sizes of occurrences of types in R. The size of a type A is the number of distinct subtrees of A.

We now prove that the type inference problem is logspace-equivalent to solving constraints.

3.4.1 From Type Inference to Constraint Solving

We first prove that the type inference problem is logspace-reducible to solving constraints.

We write $E' \leq E$ if, whenever $E(x) = A$, there is an $A' \leq A$ such that $E'(x) = A'$. Notice that the definition of $E' \leq E$ allows E' to have a larger domain that E. The following standard result can be proved by induction on typings.

Lemma 3.4.1 (Weakening). *If $E \vdash c : C$ and $E' \leq E$, then $E' \vdash c : C$.*

By a simple induction on typing derivations, we obtain the following syntax-directed characterization of typings. The proof uses only the reflexivity and transitivity of $\leq$.

Lemma 3.4.2 (Characterization of Typings). *$E \vdash c : C$ if and only if one of the following cases holds:*

- *$c = x$ and $E(x) \leq C$;*

- $c = a.\ell$, *and for some* A, $E \vdash a : A$, $A \leq [\ell^+ : C]$;
- $c = [\ell_i^{v_i} = \varsigma(x_i)b_i \ ^{i\in 1..n}]$, *and for some* A, *and some* B_i, *for* $i \in 1..n$, $E[x_i : A] \vdash b_i : B_i$, *and* $A = [\ell_i : B_i \ ^{i\in 1..n}] \leq C$; *or*
- $c = (a.\ell \Leftarrow \varsigma(x)b)$, *and for some* A *and* B, $E \vdash a : A$, $E[x : A] \vdash b : B$, $A \leq [\ell^0 : B]$, *and* $A \leq C$.

Definition 3.4.3. Let c be an expression in which all free and bound variables are pairwise distinct. We define X_c, Y_c, E_c, and $\mathcal{C}(c)$ as follows.

- X_c is a set of fresh type variables. It contains a type variable U_x for every term variable x appearing in c.
- Y_c is a set of fresh type variables. It contains a type variable $V_{c'}$ for each occurrence of a subterm c' of c. (If c' occurs more than once in c, then $V_{c'}$ is ambiguous. However, it will always be clear from context which occurrence is meant.)
- E_c is a type environment, defined by

$$E_c = \{x : U_x \mid x \text{ is free in } c\}.$$

- $\mathcal{C}(c)$ is the set of the following constraints over X_c and Y_c:
  - For each occurrence in c of a variable x, the constraint

$$(U_x, V_x). \tag{3.6}$$

  - For each occurrence in c of a subterm of the form $a.\ell$, the constraint

$$(V_a, [\ell^+ : V_{a.\ell}]) \tag{3.7}$$

  - For each occurrence in c of a subterm of the form $[\ell_i^{v_i} = \varsigma(x_i)b_i \ ^{i\in 1..n}]$, the constraint

$$([\ell_i^{v_i} : V_{b_i} \ ^{i\in 1..n}], V_{[\ell_i^{v_i} = \varsigma(x_i)b_i \ ^{i\in 1..n}]}) \tag{3.8}$$

    and for each $j \in 1..n$, the constraints

$$U_{x_j} \equiv [\ell_i^{v_i} : V_{b_i} \ ^{i\in 1..n}]. \tag{3.9}$$

- For each occurrence in c of a subterm of the form $(a.\ell \Leftarrow \varsigma(x)b)$, the constraints

$$(V_a, V_{(a.\ell \Leftarrow \varsigma(x)b)}) \tag{3.10}$$

$$V_a \equiv U_x \tag{3.11}$$

$$(V_a, [\ell^0 : V_b]). \tag{3.12}$$

□

In the definition of $\mathcal{C}(c)$, each equality $A \equiv B$ denotes the two constraints (A, B) and (B, A).

Our definition of constraint generation is "global." One can also specify constraint generation in a local way, using inference rules and using existential quantifiers to represent local ("fresh") types variables, see, e.g., [SMZ99]. However, for the type system in this chapter, there is no serious disadvantage to using the global approach.

Theorem 3.4.4. *$E \vdash c : C$ if and only if there is a solution S of $\mathcal{C}(c)$ such that $S(V_c) = C$ and $S(E_c) \subseteq E$.*

Each direction of the theorem can be proved separately. However, the proofs share a common structure, so for brevity we will prove them together. The two directions follow immediately from the two parts of the next lemma.

Lemma 3.4.5. *Let c_0 be an expression. For every subterm c of c_0,*

i) *if $E \vdash c : C$, then there is a solution S_c of $\mathcal{C}(c)$ such that $S_c(V_c) = C$ and $S_c(E_c) \subseteq E$; and*

ii) *if S is a solution of $\mathcal{C}(c_0)$, then $S(E_c) \vdash c : S(V_c)$.*

Proof. The proof is by induction on the structure of c. In (ii), we will often use the fact that any solution to $\mathcal{C}(c_0)$ (in particular, S) is a solution to $\mathcal{C}(c) \subseteq \mathcal{C}(c_0)$.

- If $c = x$, then $E_c = \{x : U_x\}$ and $\mathcal{C}(c) = \{(U_x, V_x)\}$.

i) Define $S_c = \{U_x := E(x), V_x := C\}$. Then $S_c(V_c) = S_c(V_x) = C$, and $S_c(E_c) = \{x : E(x)\} \subseteq E$.

Furthermore, by Lemma 3.4.2, $E(x) \leq C$, so S_c is a solution to $\mathcal{C}(c)$.

ii) By (3.1), $S(E_c) \vdash c : S(U_x)$.

And since $S(U_x) \leq S(V_x) = S(V_c)$, we have $S(E_c) \vdash c : S(V_c)$ by (3.5).

- If $c = a.\ell$, then $E_c = E_a$ and $\mathcal{C}(c) = \mathcal{C}(a) \cup \{(V_a, [\ell^+ : V_{a.\ell}])\}$.

i) By Lemma 3.4.2, for some A, $E \vdash a : A$, $A \leq [\ell^+ : C]$.

By induction there is a solution S_a of $\mathcal{C}(a)$ such that $S_a(V_a) = A$ and $S_a(E_a) \subseteq E$.

Define $S_c = S_a \cup \{V_{a.\ell} := C\}$. Then S_c solves $\mathcal{C}(c)$, $S_c(V_c) = S_c(V_{a.\ell}) = C$, and $S_c(E_c) = S_a(E_a) \subseteq E$.

ii) By induction, $S(E_a) \vdash a : S(V_a)$.

Since $S(V_a) \leq S([\ell^+ : V_{a.\ell}])$, by (3.5) we have $S(E_a) \vdash a : S([\ell^+ : V_{a.\ell}])$.

Then by (3.3), $S(E_a) \vdash a.\ell : S(V_{a.\ell})$.

Since $S(V_{a.\ell}) = S(V_c)$, we have $S(E_a) \vdash a.\ell : S(V_c)$.

Finally, $E_c = E_a$ and $c = a.\ell$, so $S(E_c) \vdash c : S(V_c)$ as desired.

- If $c = [\ell_i^{v_i} = \varsigma(x_i)b_i \,^{i \in 1..n}]$, then $E_c = \bigcup_{i \in 1..n}(E_{b_i} \backslash x_i)$, and

$$\begin{aligned}\mathcal{C}(c) = \{\ & ([\ell_i^{v_i} : V_{b_i} \,^{i \in 1..n}], V_c)\ \} \\ & \cup \{\ U_{x_j} \equiv [\ell_i^{v_i} : V_{b_i} \,^{i \in 1..n}] \mid j \in 1..n\ \} \\ & \cup (\textstyle\bigcup_{i \in 1..n} \mathcal{C}(b_i)).\end{aligned}$$

i) By Lemma 3.4.2, for some A, and some B_i for $i \in 1..n$, we have $E[x_i : A] \vdash b_i : B_i$ and $A = [\ell_i^{v_i} : B_i \,^{i \in 1..n}] \leq C$.

By induction, for every $i \in 1..n$ there is a substitution S_{b_i} such that S_{b_i} solves $\mathcal{C}(b_i)$, $S_{b_i}(V_{b_i}) = B_i$, and $S_{b_i}(E_{b_i}) \subseteq E[x_i : A]$.

Let $S_c = (\bigcup_{i \in 1..n} S_{b_i}) \cup \{V_c := C\} \cup \{U_{x_i} := A \mid i \in 1..n\}$.

Clearly, if S_c is well-defined, then it is a solution to $\mathcal{C}(c)$, $S_c(V_c) = C$, and $S_c(E_c) \subseteq E$.

To show that S_c is well-defined, we first assume that the domain of any S_{b_i} is $X_{b_i} \cup Y_{b_i}$ (else restrict S_{b_i} to this set).

Then it suffices to show that for any distinct $j, k \in 1..n$, the substitutions S_{b_j} and S_{b_k} agree on all type variables in their common domain. And if U is in the domain of both S_{b_j} and S_{b_k}, it must have the form U_y for some term variable y free in both b_j and b_k.

Then y must be assigned a type by E, so the conditions $S_{b_j}(E_{b_j}) \subseteq E[x_j : A]$ and $S_{b_k}(E_{b_k}) \subseteq E[x_k : A]$ guarantee that $S_{b_j}(U_y) = E(y) = S_{b_k}(U_y)$. Therefore S_c is well-defined, as desired.

ii) By induction, $S(E_{b_j}) \vdash b_j : S(V_{b_j})$ for all $j \in 1..n$.

By weakening, $S(E_c \cup \{x_j : U_{x_j}\}) \vdash b_j : S(V_{b_j})$ for all $j \in 1..n$.

Since S solves $\mathcal{C}(c)$, $S(U_{x_j}) = S([\ell_i^{v_i} : V_{b_i}\ ^{i \in 1..n}])$ for all $j \in 1..n$.

Then by (3.2), $S(E_c) \vdash c : S([\ell_i^{v_i} : V_{b_i}\ ^{i \in 1..n}])$.

Finally $S([\ell_i^{v_i} : V_{b_i}\ ^{i \in 1..n}]) \leq S(V_c)$, so we have $S(E_c) \vdash c : S(V_c)$ by (3.5).

- If $c = (a.\ell \Leftarrow \varsigma(x)b)$, then $E_c = E_a \cup (E_b \backslash x)$, and

$$\mathcal{C}(c) = \mathcal{C}(a) \cup \mathcal{C}(b) \cup \{(V_a, V_c), V_a \equiv U_x, (V_a, [\ell^0 : V_b])\}.$$

i) By Lemma 3.4.2, for some A and B, $E \vdash a : A$, $E[x : A] \vdash b : B$, $A \leq [\ell^0 : B]$, and $A \leq C$.

By induction there is a solution S_a of $\mathcal{C}(a)$ such that $S_a(V_a) = A$ and $S_a(E_a) \subseteq E$, and a solution S_b of $\mathcal{C}(b)$ such that $S_b(V_b) = B$ and $S_b(E_b) \subseteq E[x : A]$.

Let $S_c = S_a \cup S_b \cup \{V_c := C, U_x := A\}$. (We omit a proof that S_c is well-defined; this can be shown just as in the previous case.)

Then S_c is a solution to $\mathcal{C}(c)$, $S_c(V_c) = C$, and $S_c(E_c) \subseteq E$.

ii) By induction $S(E_a) \vdash a : S(V_a)$ and $S(E_b) \vdash b : S(V_b)$.

By weakening, $S(E_c) \vdash a : S(V_a)$ and $S(E_c[x : U_x]) \vdash b : S(V_b)$.

Then by (3.4), $S(E_c) \vdash c : S(V_a)$, and by (3.5), $S(E_c) \vdash c : S(V_c)$.

□

3.4.2 From Constraint Solving to Type Inference

The following result is proved by a method similar to the one used by Palsberg [Pal95] and Palsberg and Jim [PJ97].

Lemma 3.4.6. *Solvability of constraints is logspace-reducible to the type inference problem.*

Proof. It is straightforward to show that any constraint set over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$ can be simplified, in a solution-preserving way, such that each constraint is of the form (W, W') where W and W' are of the forms V or $[\ell_i^{v_i} : V_i \ ^{i\in 1..n}]$, where $v_i \in \{0, +\}$, and where $V, V_1, \ldots, V_n$ are variables. The advantage of this simplification is that object types are not nested syntactically. Let R be such a simplified constraint set. Define

$$
\begin{array}{lll}
a^R = \ [& \ell_V^0 & = \varsigma(x)(x.\ell_V) \\
 & & \qquad \text{for each variable } V \text{ in } R \\
 & \ell_Q^0 & = \varsigma(x)[\ell_i^{v_i} = \varsigma(y)(x.\ell_{V_i}) \ ^{i\in 1..n}] \\
 & & \qquad \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ ^{i\in 1..n}] \\
 & m_{Q,\ell_j}^0 & = \varsigma(x)((x.\ell_{V_j} \Leftarrow \varsigma(y)(x.\ell_Q.\ell_j)).\ell_Q) \\
 & & \qquad \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ ^{i\in 1..n}] \\
 & & \qquad \text{and for each } j \in 1..n \\
 & k_{Q,\ell_j}^0 & = \varsigma(x)((x.\ell_Q).\ell_j \Leftarrow \varsigma(y)(x.\ell_{V_j})) \\
 & & \qquad \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ ^{i\in 1..n}] \\
 & & \qquad \text{and for each } j \in 1..n \text{ with } v_j = 0 \\
 & \ell_{(W,W')}^0 & = \varsigma(x)((x.\ell_{W'} \Leftarrow \varsigma(y)(x.\ell_W)).\ell_W) \\
 & & \qquad \text{for each constraint } (W, W') \in R \\
]
\end{array}
$$

Notice that a^R can be generated in log space.

We first prove that if R is solvable then a^R is typable. Suppose R has solution S. Define

$$
\begin{array}{lllll}
A = & [& \ell^0_V & : S(V) & \text{for each variable } V \text{ in } R \\
 & & \ell^0_Q & : S(Q) & \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ ^{i\in 1..n}] \\
 & & m^0_{Q,\ell_j} & : S(Q) & \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ ^{i\in 1..n}] \\
 & & & & \text{and for each } j \in 1..n \\
 & & k^0_{Q,\ell_j} & : S(Q) & \text{for each } Q \text{ in } R \text{ of the form } [\ell_i^{v_i} : V_i \ ^{i\in 1..n}] \\
 & & & & \text{and for each } j \in 1..n \text{ with } v_j = 0 \\
 & & \ell^0_{(W,W')} & : S(W) & \text{for each constraint } (W, W') \in R \\
 &] & & &
\end{array}
$$

It is straightforward to show that $\emptyset \vdash a^R : A$ is derivable.

We now prove that if a^R is typable, then R is solvable. Suppose a^R is typable. From Theorem 3.4.4 we get a solution S of $\mathcal{C}(a^R)$.

Notice that each method in a^R binds a variable x. Each of these variables corresponds to a distinct type variable in $\mathcal{C}(a^R)$. Since S is a solution of $\mathcal{C}(a^R)$, and $\mathcal{C}(a^R)$ contains constraints of the form $U_x = [\ldots]$ for each method in a^R (from rule (3.9)), all those type variables are mapped by S to the same type. Thus, we can think of all the bound variables of methods of a^R as being related to the same type variable, which we will write as U_x.

Notice also that most methods in a^R bind a variable y. None of these variables are used in a^R, and each of them corresponds to a distinct type variable in $\mathcal{C}(a^R)$. They will not play any role in the rest of the proof. For $A = [\ell^v : B, \ldots]$, we define $A \downarrow \ell = B$.

Define

$$S_R(V) \quad = \quad S(U_x) \downarrow \ell_V \quad \text{for each variable } V \text{ in } R.$$

The definition is justified by Property 1 below.

- **Property 1** If V is a variable in R, then $S(U_x) \downarrow \ell_V$ is defined.
- **Property 2** For each Q in R of the form $[\ell_i^{v_i} : V_i \ ^{i\in 1..n}]$, we have $S(U_x) \downarrow \ell_Q = [\ell_i^{v_i} : (S(U_x) \downarrow \ell_{V_i}) \ ^{i\in 1..n}]$.

We will proceed by first showing the two properties and then showing that R has solution S_R.

To see Property 1, notice that in the body of the method ℓ_V we have the expression $x.\ell_V$. Since S is a solution of $\mathcal{C}(a^R)$, we have from the rules (3.6) and (3.7) that S satisfies

$$(U_x, V_x) \text{ and } (V_x, [\ell_V^+ : V_{x.\ell_V}]),$$

so

$$S(U_x)\downarrow \ell_V \leq S(V_{x.\ell_V}) \tag{3.13}$$

We conclude that since $S(V_{x.\ell_V})$ is defined, also $S(U_x)\downarrow \ell_V$ is defined.

To see Property 2, let Q be an occurrence in R of the form $[\ell_i^{v_i} : V_i \ ^{i\in 1..n}]$. For each $j \in 1..n$, in the body of the method m_{Q,ℓ_j}, we have the expression $x'.\ell_{V_j} \Leftarrow \varsigma(y)(x.\ell_Q.\ell_j)$ where we, for clarity, have written the first occurrence of x as x'. Since S is a solution of $\mathcal{C}(a^R)$, we have from the rules (3.6), (3.7), and (3.12), that S satisfies

$$(U_x \ , \ V_{x'}) \ \text{ and } \ (V_{x'} \ , \ [\ell_{V_j}^0 : V_{x.\ell_Q.\ell_j}]) \tag{3.14}$$

$$(U_x \ , \ V_x) \ \text{ and } \ (V_x \ , \ [\ell_Q^+ : V_{x.\ell_Q}]) \tag{3.15}$$

$$(V_{x.\ell_Q} \ , \ [\ell_j^+ : V_{x.\ell_Q.\ell_j}]) \tag{3.16}$$

Thus,

$$\begin{array}{rcll} S(U_x)\downarrow \ell_Q & \leq & S(V_{x.\ell_Q}) & \text{from (3.15)} \\ & \leq & [\ell_j^+ : S(V_{x.\ell_Q.\ell_j})] & \text{from (3.16)} \\ & = & [\ell_j^+ : (S(U_x)\downarrow \ell_{V_j})] & \text{from (3.14)} \end{array}$$

Therefore,

$$S(U_x)\downarrow \ell_Q \downarrow \ell_j \ \leq \ S(U_x)\downarrow \ell_{V_j} \tag{3.17}$$

In the body of the method ℓ_Q we have the expression $[\ell_i^{v_i} = \varsigma(y)(x.\ell_{V_i}) \ ^{i\in 1..n}]$. Since S is a solution of $\mathcal{C}(a^R)$, we have from the rules (3.8) and (3.9) that S satisfies

$$([\ell_i^{v_i} : V_{x.\ell_{V_i}} \ ^{i\in 1..n}] \ \ , \ \ V_{[\ell_i^{v_i} = \varsigma(y)(x.\ell_{V_i}) \ ^{i\in 1..n}]}) \tag{3.18}$$

$$U_x \ \equiv \ [\ldots \ell_Q^0 : V_{[\ell_i^{v_i} = \varsigma(y)(x.\ell_{V_i}) \ ^{i\in 1..n}]} \ldots] \tag{3.19}$$

Thus, from (3.18) and (3.19) we have

$$[\ell_i^{v_i} : S(V_{x.\ell_{V_i}}) \ ^{i\in 1..n}] \leq S(V_{[\ell_i^{v_i} = \varsigma(y)(x.\ell_{V_i}) \ ^{i\in 1..n}]}) = S(U_x)\downarrow \ell_Q, \tag{3.20}$$

so

$$\begin{array}{rcll} S(U_x)\downarrow \ell_{V_j} & \leq & S(V_{x.\ell_{V_j}}) & \text{from (3.13)} \\ & \leq & S(U_x)\downarrow \ell_Q \downarrow \ell_j & \text{from (3.20).} \end{array}$$

From that and (3.17) we conclude:

$$S(U_x)\downarrow \ell_Q \downarrow \ell_j \;\; = \;\; S(U_x)\downarrow \ell_{V_j}. \tag{3.21}$$

For each $j \in 1..n$ with $v_j = 0$, in the body of the method k_{Q,ℓ_j}, we have the expression $((x'.\ell_Q).\ell_j \Leftarrow \varsigma(y)(x.\ell_{V_j}))$ where we, for clarity, have written the first occurrence of x as x'. Since S is a solution of $\mathcal{C}(a^R)$, we have from the rules (3.6), (3.7), and (3.12) that S satisfies

$$(U_x \;\; , \;\; V_{x'}) \;\; \text{and} \;\; (V_{x'} \;\; , \;\; [\ell_Q^+ : V_{x'.\ell_Q}]) \tag{3.22}$$

$$(V_{x'.\ell_Q} \;\; , \;\; [\ell_j^0 : V_{x.\ell_{V_j}}]) \tag{3.23}$$

Thus,

$$\begin{array}{rcll} S(U_x)\downarrow \ell_Q & \leq & S(V_{x'.\ell_Q}) & \text{from (3.22)} \\ & \leq & [\ell_j^0 : S(V_{x.\ell_{V_j}})] & \text{from (3.23)} \end{array}$$

From that and (3.20) we have that

- for each $j \in 1..n$ with $v_j = 0$, the variance annotation of the ℓ_j field of $S(U_x)\downarrow \ell_Q$ is 0, and
- for each $j \in 1..n$ with $v_j = +$, the variance annotation of the ℓ_j field of $S(U_x)\downarrow \ell_Q$ is +.

Therefore, by (3.21), we have that $S(U_x)\downarrow \ell_Q = [\ell_i^{v_i} : (S(U_x)\downarrow \ell_{V_i})^{\;\; i\in 1..n}]$, that is, Property 2.

We can summarize Property 1 and 2 as follows.

- **Property 3** If W is a left-hand side or a right-hand side of a constraint in R, then $S(U_x)\downarrow \ell_W$ is defined and $S_R(W) = S(U_x)\downarrow \ell_W$.

We will now show that R has solution S_R.

Consider a constraint (W, W') in R. The body of the method $\ell_{(W,W')}$ contains the expression $x'.\ell_{W'} \Leftarrow \varsigma(y)(x.\ell_W)$ where we, for clarity, have written the first occurrence of x as x'. Since S is a solution of $\mathcal{C}(a^R)$, we have from the rules (3.6), (3.12), (3.6) and (3.7) that S satisfies

$$(U_x \ , \ V_{x'}) \ \text{ and } \ (V_{x'} \ , \ [\ell^0_{W'} : V_{x.\ell_W}]) \tag{3.24}$$

$$(U_x \ , \ V_x) \ \text{ and } \ (V_x \ , \ [\ell^+_W : V_{x.\ell_W}]) \tag{3.25}$$

We conclude

$$\begin{array}{rcll} S_R(W) & = & S(U_x)\downarrow \ell_W & \text{from Property 3} \\ & \leq & S(V_{x.\ell_W}) & \text{from (3.25)} \\ & = & S(U_x)\downarrow \ell_{W'} & \text{from (3.24)} \\ & = & S_R(W') & \text{from Property 3} \end{array}$$

□

3.5 Solving Constraints

In this section we present an algorithm for deciding whether a relation R is solvable. We first define the notions of satisfaction-closure (Section 3.5.1) and satisfaction-consistency (Section 3.5.2), and then we prove that a relation R is solvable if and only its satisfaction-closure is satisfaction-consistent (Theorem 3.5.12).

3.5.1 Satisfaction-closure

Definition 3.5.1. If R is a relation on types, we say R is *satisfaction-closed* (abbreviated sat-closed) if there exists relation L on types such that

i) if $(A, B) \in R$, then $(A, A), (B, B) \in R$.

ii) if $(A, B), (B, C) \in R$, then $(A, C) \in R$;

iii) if $(A, B) \in R$, then $(A, B) \in L$;

iv) if $(A, B) \in L$, then $(B, A) \in L$;

v) if $(A, B) \in L$, and $(B, C) \in R$, then $(A, C) \in L$;

vi) if $([\ell^+ : B, \ldots], [\ell^+ : B', \ldots]) \in R$, then $(B, B') \in R$;

vii) if $([\ell^+ : B, \ldots], [\ell^+ : B', \ldots]) \in L$, then $(B, B') \in L$;

viii) if $([\ell^0 : B, \ldots], [\ell^+ : B', \ldots]) \in L$, then $(B, B') \in R$;

ix) if $([\ell^0 : B, \ldots], [\ell^0 : B', \ldots]) \in L$, then $(B, B') \in R$.

□

Notice that for a given relation R, we can construct a sat-closed relation R' that includes R, by letting R' consist of all pairs of subtrees of types that occur in R.

Notice also that the intersection of a family of sat-closed relations is itself sat-closed. From that we have that for a given relation R, there is the smallest sat-closed relation that includes R; we call that sat-closed relation the *sat-closure* of R.

Notice also that for a given sat-closed relation R, there is the smallest relation L such that the sat-closure rules for R and L are satisfied; we call L *the lower-bound relation for* R.

For a relation R, notice that the sat-closure of R and the lower bound relation for the sat-closure of R are the pairwise-$\subseteq$-smallest pair (R', L) that contains $(R, \emptyset)$ such that R' is sat-closed. We can compute R' and L by a straightforward fixed-point computation that uses the sat-closure rules to monotonically add elements to the two relations. We can analyze the complexity of the fixed-computation using a technique and a theorem of McAllester [McA02], as follows. McAllester [McA02, Section 10] presents a technique for encoding record types, such as the ones we use here, without using elipses. The encoding entails a slight reformulation of the rules for sat-closure; however, the transitivity rule for R (sat-closure rule (ii)) remains the key source of complexity. We can then apply a theorem of McAllester [McA02, Theorem 1] to get that the fixed-point computation takes $O(n^3)$ time where n is the size of R.

Define that (R, L) is solvable iff there exists a mapping S such that

- if $(A, B) \in R$, then $S(A) \leq S(B)$, and
- if $(A, B) \in L$, then there exists a type C, such that $C \leq S(A)$ and $C \leq S(B)$.

Lemma 3.5.2. *A relation and its sat-closure have the same set of solutions.*

Proof. Since the sat-closure of a relation R contains R, it follows that any solution of the sat-closure of R is also a solution of R.

To prove the converse, we will prove the following more general property:

> Any solution of R is also a solution of the pairwise-$\subseteq$-smallest pair (R', L) that contains $(R, \emptyset)$ such that R' is sat-closed.

We proceed by induction on the fixed-point computation of (R', L) from $(R, \emptyset)$. Each iteration begins with a pair (R_1, L_1), and at the end of an iteration step, some pairs may have been added to R_1 and L_1. We need to show that after each iteration, any solution of R is also a solution of the resulting pair of relations.

In the base case, we have that any solution of R is also a solution of $(R, \emptyset)$.

In the induction step, suppose (R_1, L_1) has a solution S. For each of the nine sat-closure rules, we need to show that the rules will only add pairs to R_1 and L_1 that have solution S:

- Assume rule (i) has been used. The relation $\leq$ is reflexive, so $S(A) \leq S(A)$ and $S(B) \leq S(B)$, so S is still a solution.
- Assume rule (ii) has been used. From the induction hypothesis, we have $S(A) \leq S(B)$ and $S(B) \leq S(C)$. The relation $\leq$ is transitive, so $S(A) \leq S(C)$, so S is still a solution.
- Assume rule (iii) has been used. From the induction hypothesis, we have $S(A) \leq S(B)$, so a common $\leq$-lower bound for $S(A)$ and $S(B)$ is $S(A)$, so S is still a solution.
- Assume rule (iv) has been used. From the induction hypothesis, we have that $S(A)$ and $S(B)$ have a common $\leq$-lower bound, so $S(B)$ and $S(A)$ have a common $\leq$-lower bound, so S is still a solution.

- Assume rule (v) has been used. From the induction hypothesis, we have that $S(A)$ and $S(B)$ have a common $\leq$-lower bound D, and that $S(B) \leq S(C)$. The relation $\leq$ is transitive, so $D \leq C$, and hence D is a common $\leq$-lower bound for $S(A)$ and $S(C)$; so S is still a solution.

- Assume rule (vi) has been used. From the induction hypothesis, we have that $[\ell^+ : S(B), \ldots] \leq [\ell^+ : S(B')]$. From Lemma 3.2.2 we have $S(B) \leq S(B')$, so S is still a solution.

- Assume rule (vii) has been used. From the induction hypothesis, we have that $[\ell^+ : S(B), \ldots]$ and $[\ell^+ : S(B')]$ have a common $\leq$-lower bound D. From Lemma 3.2.2 we have that $D = [\ell^v : A]$ where $A \leq S(B)$ and $A \leq S(B')$, so A is a common $\leq$-lower bound for $S(B)$ and $S(C)$; so S is still a solution.

- Assume rule (viii) has been used. From the induction hypothesis, we have that $[\ell^0 : S(B), \ldots]$ and $[\ell^+ : S(B')]$ have a common $\leq$-lower bound D. From Lemma 3.2.2 we have that $D = [\ell^0 : A]$ where $A = S(B)$ and $A \leq S(B')$, so $S(B) = A \leq S(B')$; so S is still a solution.

- Assume rule (ix) has been used. From the induction hypothesis, we have that $[\ell^0 : S(B), \ldots]$ and $[\ell^0 : S(B')]$ have a common $\leq$-lower bound D. From Lemma 3.2.2 we have that $D = [\ell^0 : A]$ where $A = S(B)$ and $A = S(B')$, so $S(B) = A = S(B')$, and therefore $S(B) \leq S(B')$; so S is still a solution.

□

A sat-closed relation has the five properties that are expressed in the following lemma.

Lemma 3.5.3. *Suppose R is sat-closed, and let L be the lower-bound relation for R.*

(A) *If $([\ell^v : B, \ldots], [\ell^{v'} : B', \ldots]) \in L$, then $(B, B') \in L$.*

(B) *If $(A, A') \in L$ and $(A, A_1), (A', A_2) \in R$, then then $(A_1, A_2) \in L$.*

(C) *If $(A, A_1), (A, A_2) \in R$, then $(A_1, A_2) \in L$.*

(D) *If* $([\ell^v : B, \ldots], [\ell^{v'} : B', \ldots]) \in R$ *and* $v \sqsubseteq v'$*, then* $(B, B') \in R$*.*

(E) *If* $([\ell^0 : B, \ldots], [\ell^0 : B', \ldots]) \in R$*, then* $(B', B) \in R$*.*

Proof. For Property **(A)** there are four cases. If $v = +, v' = +$, then from sat-closure rule (vii) we have $(B, B') \in L$. If $v = 0, v' = +$, or $v = 0, v' = 0$, then from sat-closure rules (viii) or (ix), we have $(B, B') \in R$, and then from sat-closure rule (iii) we have $(B, B') \in L$. Finally, if $v = +, v' = 0$, then from sat-closure rule (iv) we have $([\ell^{v'} : B', \ldots], [\ell^v : B, \ldots]) \in L$, so from sat-closure rule (viii) we have $(B', B) \in R$, so from sat-closure rule (iii) we have $(B', B) \in L$, and hence from sat-closure rule (iv) we have $(B, B') \in L$. So, in all cases $(B, B') \in L$.

For Property **(B)**, notice that from sat-closure rule (v), $(A, A') \in L$, and $(A', A_2) \in R$, we have $(A, A_2) \in L$. From sat-closure rule (iv) and $(A, A_2) \in L$, we have $(A_2, A) \in L$. From sat-closure rule (v), $(A_2, A) \in L$, and $(A, A_1) \in R$, we have $(A_2, A_1) \in L$, so from sat-closure rule (iv) we have $(A_1, A_2) \in L$.

For Property **(C)**, notice that from sat-closure rule (i) and $(A, A_1) \in R$ we have $(A, A) \in R$, so from sat-closure rule (iii) we have $(A, A) \in L$. From Property **(B)**, $(A, A) \in L$, and $(A, A_1), (A, A_2) \in R$ we conclude that $(A_1, A_2) \in L$.

For Property **(D)** there are three cases. If $v = +, v' = +$, then from sat-closure rule (vi) we have $(B, B') \in R$. If $v = 0, v' = +$, or $v = 0, v' = 0$, then from sat-closure rule (iv) and $([\ell^v : B, \ldots], [\ell^{v'} : B', \ldots]) \in R$, we have $([\ell^v : B, \ldots], [\ell^{v'} : B', \ldots]) \in L$, so from sat-closure rules (viii) or (ix), we have $(B, B') \in R$.

For Property **(E)**, we have from sat-closure rule (iii) that $([\ell^0 : B, \ldots], [\ell^0 : B', \ldots]) \in L$, so from sat-closure rule (iv), we have $([\ell^0 : B', \ldots], [\ell^0 : B, \ldots]) \in L$, so from sat-closure rule (ix), we have $(B', B) \in R$. □

From Lemma 3.5.3, Properties **(D)** and **(E)**, we have that if a relation is sat-closed, then it is also subtype-closed.

3.5.2 Satisfaction-consistency

Definition 3.5.4. We say R is *satisfaction-inconsistent* (abbreviated sat-inconsistent) if any of the following two cases hold.

- $([\ell^+ : B, \ldots], [\ell^0 : B', \ldots]) \in R$ for some ℓ, B, B'.
- $([\ell_i^{v_i} \ : \ B_i \ \ ^{i \in I}], [\ell^v \ : \ B, \ldots]) \ \in \ R$ for some ℓ, v, B, and ℓ_i, v_i, B_i for $i \in I$; and furthermore $\ell \neq \ell_i$ for all $i \in I$.

We say R is *sat-consistent* iff R is not sat-inconsistent. □

Notice that if a relation is subtype-consistent, then it is also sat-consistent.

Lemma 3.5.5. *If R is solvable, then R is sat-consistent.*

Proof. Immediate. □

3.5.3 Main Result

We first list the terminology used in the later definitions. Recall that $\mathcal{T}_{\mathrm{reg}}(\Sigma)$ is the set of recursive types considered in this chapter.

$$\begin{aligned} \mathsf{Types} &= \mathcal{T}_{\mathrm{reg}}(\Sigma) \\ \mathsf{States} &= \mathrm{P}(\mathsf{Types}) \\ \mathsf{RelTypes} &= \mathrm{P}(\mathsf{Types} \times \mathsf{Types}) \\ \mathsf{RelStates} &= \mathrm{P}(\mathsf{States} \times \mathsf{States}) \end{aligned}$$

To define the solution S_R, we will need the following notation. We use g, h to range over sets of types. Then we make the following definitions.

$$\begin{aligned} g.\ell &= \{B \mid \exists A \in g.\ A = [\ell^v : B, \ldots]\}. \\ \mathrm{above}_R(g) &= \{B \mid \exists A \in g.\ (A, B) \in R\}. \\ \mathrm{ABOVE}_R(R') &= \{(\mathrm{above}_R(\{A\}), \mathrm{above}_R(\{B\})) \mid (A, B) \in R'\}. \\ \mathrm{Var}(g, \ell) &= \sqcap\{v \mid \exists A \in g.\ (\ell, v) \in A(\epsilon)\}. \end{aligned}$$

In the last definition, $\sqcap$ is the greatest lower bound of a nonempty set of variances; $\sqcap\emptyset$ is undefined.

The types of the above definitions are

$$\begin{aligned} g.\ell &: \mathsf{States} \to \mathsf{States} \\ \mathrm{above}_R &: \mathsf{States} \to \mathsf{States} \\ \mathrm{ABOVE}_R &: \mathsf{RelTypes} \to \mathsf{RelStates} \\ \mathrm{Var} &: \mathsf{States} \times \mathsf{Labels} \to \mathsf{Variances} \end{aligned}$$

We define $\mathrm{LV} : \mathsf{States} \to \mathcal{P}(\mathsf{Labels} \times \mathsf{Variances})$ such that for any set g of types we have that $\mathrm{LV}(g)$ is the labels and variances implied by g, namely

$$\mathrm{LV}(g) = \{(\ell, v) \mid v = \mathsf{Var}(g, \ell)\}.$$

For a relation R we build an automaton with states consisting of sets of types appearing in R, and the following one-step transition function:

$$\delta_R(g)(\ell) = \begin{cases} \mathrm{above}_R(g.\ell) & \text{if } g.\ell \neq \emptyset \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We write $\mathsf{States}(R)$ for the set of states of the automaton, and use g, h to range over states.

The one-step transition function is extended to a many-step transition function in the usual way.

$$\begin{aligned} \delta^*_R(g)(\epsilon) &= g, \\ \delta^*_R(g)(\ell\alpha) &= \delta^*_R(\delta_R(g)(\ell))(\alpha). \end{aligned}$$

Any state g defines a type, $\mathrm{Type}_R(g)$, and any relation $\mathcal{R}$ on $\mathsf{States}(R)$ defines a relation on types $\mathrm{TYPE}_R(\mathcal{R})$, as follows:

$$\begin{aligned} \mathrm{Type}_R(g)(\alpha) &= \mathrm{LV}(\delta^*_R(g)(\alpha)), \\ \mathrm{TYPE}_R(\mathcal{R}) &= \{(\mathrm{Type}_R(g), \mathrm{Type}_R(h)) \mid (g, h) \in \mathcal{R}\}. \end{aligned}$$

We have that

$$\begin{aligned} \mathrm{Type}_R &: \mathsf{States} \to \mathsf{Types} \\ \mathrm{TYPE}_R &: \mathsf{RelStates} \to \mathsf{RelTypes} \end{aligned}$$

The following lemma expresses a fundamental property of Type_R.

Lemma 3.5.6. $\mathrm{Type}_R(g)(\ell\alpha) = \mathrm{Type}_R(\delta_R(g)(\ell))(\alpha)$.

Proof.

$$\begin{aligned} \mathrm{Type}_R(g)(\ell\alpha) &= \mathrm{LV}(\delta^*_R(g)(\ell\alpha)) && \text{(Definition of } \mathrm{Type}_R\text{)} \\ &= \mathrm{LV}(\delta^*_R(\delta_R(g)(\ell))(\alpha)) && \text{(Definition of } \delta^*_R\text{)} \\ &= \mathrm{Type}_R(\delta_R(g)(\ell))(\alpha) && \text{(Definition of } \mathrm{Type}_R\text{)} \end{aligned}$$

□

For any relation R on types, we define S_R to be the least substitution such that for every U appearing in R we have

$$S_R(U) = \mathrm{Type}_R(\mathrm{above}_R(\{U\})).$$

We claim that if R is sat-closed and sat-consistent, then S_R is a solution to R.

To illustrate the definition of S_R, let us go into detail of the construction of S_R for the sat-closed and sat-consistent constraint set R in Figure 3.2.

From Lemma 3.5.6, we have that if $\mathrm{LV}(g) = \{(\ell, v)\}$, then

$$\mathrm{Type}_R(g) = [\ell^v : \mathrm{Type}_R(\delta_R(g)(\ell))] = [\ell^v : \mathrm{Type}_R(\mathrm{above}_R(g.\ell))]. \tag{3.26}$$

Let us now consider the relation R in Figure 3.2 and first note that we have:

$$\begin{aligned} \mathrm{above}_R(\{U\}) &= \{\ U, [\ell^+ : [\ell^+ : [m^+ : [\ ]]]], [\ell^+ : [\ell^0 : W]]\ \} \\ \mathrm{above}_R(\{W\}) &= \{\ W, [m^+ : [\ ]]\ \}. \end{aligned}$$

Next, let us calculate $S_R(W)$:

$$\begin{aligned} S_R(W) &= \mathrm{Type}_R(\mathrm{above}_R(\{W\})) && \text{(Definition of } S_R\text{)} \\ &= \mathrm{Type}_R(\{\ W, [m^+ : [\ ]]\ \}) && \text{(Definition of } \mathrm{above}_R\text{)} \\ &= [m^+ : [\ ]] && \text{(From 3.26).} \end{aligned}$$

Finally, let us calculate $S_R(U)$:

$$\begin{aligned} S_R(U) &= \mathrm{Type}_R(\mathrm{above}_R(\{U\})) && \text{(Definition of } S_R\text{)} \\ &= \mathrm{Type}_R(\{\ U, [\ell^+ : [\ell^+ : [m^+ : [\ ]]]], [\ell^+ : [\ell^0 : W]]\ \}) && \text{(Definition of } \mathrm{above}_R\text{)} \\ &= [\ell^+ : \mathrm{Type}_R(\mathrm{above}_R(\{\ [\ell^+ : [m^+ : [\ ]]], [\ell^0 : W]\ \}))] && \text{(From 3.26)} \\ &= [\ell^+ : \mathrm{Type}_R(\{\ [\ell^+ : [m^+ : [\ ]]]], [\ell^0 : W]\ \})] && \text{(Definition of } \mathrm{above}_R\text{)} \\ &= [\ell^+ : [\ell^0 : \mathrm{Type}_R(\mathrm{above}_R(\{\ [m^+ : [\ ]], W\ \}))]] && \text{(From 3.26)} \\ &= [\ell^+ : [\ell^0 : \mathrm{Type}_R(\{\ [m^+ : [\ ]], W\ \})]] && \text{(Definition of } \mathrm{above}_R\text{)} \\ &= [\ell^+ : [\ell^0 : [m^+ : [\ ]]]] && \text{(From 3.26)} \end{aligned}$$

The first step in proving that S_R is a solution to R is to develop a connection between subtype-closure and δ. Define the function $\mathcal{A} : \mathsf{RelTypes} \to \mathsf{RelTypes}$ by $(A, B) \in \mathcal{A}(R)$ iff one of the following conditions holds:

- $(A, B) \in R$.
- For some ℓ, v, v', such that $v \sqsubseteq v'$, we have $([\ell^v : A, \ldots], [\ell^{v'} : B, \ldots]) \in R$.
- For some ℓ, we have $([\ell^0 : B, \ldots], [\ell^0 : A, \ldots]) \in R$.

Note, the subtype-closure of a relation R is the least fixed point of $\mathcal{A}$ containing R.

Define the function $\mathcal{B}_R$: RelStates $\rightarrow$ RelStates by $(g, h) \in \mathcal{B}_R(\mathcal{R})$ iff one of the following conditions holds:

- $(g, h) \in \mathcal{R}$.
- For some ℓ and $(g', h') \in \mathcal{R}$, such that Var$(g', \ell) \sqsubseteq$ Var(h', ℓ), we have $g = \delta_R(g')(\ell)$, $h = \delta_R(h')(\ell)$.
- For some ℓ and $(h', g') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$, $h = \delta_R(h')(\ell)$, Var$(g', \ell) = 0$, and Var$(h', \ell) = 0$.

The next two lemmas (Lemma 3.5.7 and Lemma 3.5.8) are key ingredients in the proof of Lemma 3.5.9. Lemma 3.5.7 states fundamental relationship between TYPE_R, $\mathcal{A}$, and $\mathcal{B}_R$. The intuition behind Lemma 3.5.7 is that it doesn't matter whether we propagate information about subtype relationships (using $\mathcal{B}_R$ and $\mathcal{A}$) before or after we collapse the sets of types in each state to single types (using TYPE_R).

Lemma 3.5.7. *The following diagram commutes:*

$$\begin{array}{ccc} \mathsf{RelStates} & \xrightarrow{\mathrm{TYPE}_R} & \mathsf{RelTypes} \\ \downarrow{\scriptstyle \mathcal{B}_R} & & \downarrow{\scriptstyle \mathcal{A}} \\ \mathsf{RelStates} & \xrightarrow{\mathrm{TYPE}_R} & \mathsf{RelTypes} \end{array}$$

Proof. To prove $\mathrm{TYPE}_R \circ \mathcal{B}_R \subseteq \mathcal{A} \circ \mathrm{TYPE}_R$, suppose $\mathcal{R} \in$ RelStates and $(A, B) \in \mathrm{TYPE}_R \circ \mathcal{B}_R(\mathcal{R})$. There must be a pair of states $(g, h) \in \mathcal{B}_R(\mathcal{R})$ such that $A = \mathrm{Type}_R(g)$ and $B = \mathrm{Type}_R(h)$. We reason by cases on how $(g, h) \in \mathcal{B}_R(\mathcal{R})$. From the definition of $\mathcal{B}_R$ we have that there are three cases.

First, suppose $(g, h) \in \mathcal{R}$. We have $(\mathrm{Type}(g), \mathrm{Type}(h)) \in \mathrm{TYPE}_R(\mathcal{R})$, so from the definition of $\mathcal{A}$ we have $(\mathrm{Type}(g), \mathrm{Type}(h)) \in \mathcal{A} \circ \mathrm{TYPE}_R(\mathcal{R})$.

Second, suppose for some ℓ and $(g', h') \in \mathcal{R}$, such that $\text{Var}(g', \ell) \sqsubseteq \text{Var}(h', \ell)$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. From $(g', h') \in \mathcal{R}$, we have $(\text{Type}_R(g'), \text{Type}_R(h')) \in \text{TYPE}_R(\mathcal{R})$. From Lemma 3.5.6 and $g = \delta_R(g')(\ell)$, we have

$$\begin{aligned} \text{Type}_R(g')(\ell\alpha) &= \text{Type}_R(\delta_R(g')(\ell))(\alpha) && \text{(From Lemma 3.5.6)} \\ &= \text{Type}_R(g)(\alpha) && \text{(From the definition of } g\text{)} \\ &= A(\alpha) && \text{(From the definition of } A\text{)} \end{aligned}$$

so there must exist $v_A = \text{Var}(g', \ell)$ such that

$$\text{Type}(g') = [\ell^{v_A} : A, \ldots].$$

Similarly, there must exist $v_B = \text{Var}(h', \ell)$ such that

$$\text{Type}(h') = [\ell^{v_B} : B, \ldots].$$

From the characterizations of $\text{Type}_R(g')$ and $\text{Type}_R(h')$, and from the definition of $\mathcal{A}$, we have $(A, B) \in \mathcal{A} \circ \text{TYPE}_R(\mathcal{R})$.

Third, suppose for some ℓ and $(h', g') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$, $h = \delta_R(h')(\ell)$, $\text{Var}(g', \ell) = 0$, and $\text{Var}(h', \ell) = 0$. From $(h', g') \in \mathcal{R}$, we have $(\text{Type}_R(h'), \text{Type}_R(g')) \in \text{TYPE}_R(\mathcal{R})$. From the definition of Type_R we have $\text{Type}_R(g') = [\ell^0 : A, \ldots]$ and $\text{Type}_R(h') = [\ell^0 : B, \ldots]$, so, by the definition of $\mathcal{A}$, we have $(A, B) \in \mathcal{A} \circ \text{TYPE}_R(\mathcal{R})$.

To prove $\mathcal{A} \circ \text{TYPE}_R \subseteq \text{TYPE}_R \circ \mathcal{B}_R$, suppose $\mathcal{R} \in \mathsf{RelStates}$ and $(A, B) \in \mathcal{A} \circ \text{TYPE}_R(\mathcal{R})$. We reason by cases on how $(A, B) \in \mathcal{A} \circ \text{TYPE}_R(\mathcal{R})$. From the definition of $\mathcal{A}$ we have that there are three cases.

First, suppose $(A, B) \in \text{TYPE}_R(\mathcal{R})$. There must exist g and h such that $A = \text{Type}_R(g)$, $B = \text{Type}_R(h)$, and $(g, h) \in \mathcal{R}$. From $(g, h) \in \mathcal{R}$ and the definition of $\mathcal{B}_R$, we have that $(g, h) \in \mathcal{B}_R(\mathcal{R})$, so $(A, B) \in \text{TYPE}_R \circ \mathcal{B}_R$.

Second, suppose for some ℓ, v, and v', such that $v \sqsubseteq v'$, we have $([\ell^v : A, \ldots], [\ell^{v'} : B, \ldots]) \in \text{TYPE}_R(\mathcal{R})$. There must exist g' and h' such that $\text{Type}_R(g') = [\ell^v : A, \ldots]$, $\text{Type}_R(h') = [\ell^{v'} : B, \ldots]$, $(g', h') \in \mathcal{R}$, $\text{Var}(g', \ell) = v$, and $\text{Var}(h', \ell) = v'$. Then $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$ are well defined, and $(g, h) \in \mathcal{B}_R(\mathcal{R})$ by the definition

of $\mathcal{B}_R$. And by the definition of Type_R, $A = \mathrm{Type}_R(g)$ and $B = \mathrm{Type}_R(h)$, so $(A, B) \in \mathrm{TYPE}_R \circ \mathcal{B}_R(\mathcal{R})$ as desired.

Third, suppose for some ℓ, we have $([\ell^0 : B, \ldots], [\ell^0 : A, \ldots]) \in \mathrm{TYPE}_R(\mathcal{R})$. There must exist g' and h' such that $\mathrm{Type}_R(g') = [\ell^0 : A, \ldots]$, $\mathrm{Type}_R(h') = [\ell^0 : B, \ldots]$, and $(h', g') \in \mathcal{R}$. By the definition of Type_R, $\mathrm{Var}(g', \ell) = 0$ and $\mathrm{Var}(h', \ell) = 0$. Then $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$ are well defined, and $(g, h) \in \mathcal{B}_R(\mathcal{R})$ by the definition of $\mathcal{B}_R$. And by the definition of Type_R, $A = \mathrm{Type}_R(g)$ and $B = \mathrm{Type}_R(h)$, so $(A, B) \in \mathrm{TYPE}_R \circ \mathcal{B}_R(\mathcal{R})$ as desired. □

Lemma 3.5.8. *Suppose R is sat-closed. For all n, we have that if $(g, h) \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$, then $g \supseteq h$.*

Proof. Let L be the lower-bound relation for R. We will prove the following more general property:

For all n,

if $(g, h) \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$, then

- *$g \supseteq h$, and*
- *if $A_1, A_2 \in g$, then $(A_1, A_2) \in L$.*

We proceed by induction on n. In the base case of $n = 0$, suppose $(g, h) \in \mathrm{ABOVE}_R(R)$. From the definition of ABOVE_R we have that we can choose A, B such that $(A, B) \in R$, $g = \mathrm{above}_R(\{A\})$, and $h = \mathrm{above}_R(\{B\})$. To prove $g \supseteq h$, suppose $C \in h$. We have $(B, C) \in R$, and together with $(A, B) \in R$ and transitivity of R (sat-closure rule (ii)), we have $(A, C) \in R$, so $C \in g$, and hence $g \supseteq h$. Suppose $A_1, A_2 \in g$. We have $(A, A_1), (A, A_2) \in R$, so from Lemma 3.5.3, Property **(C)**, we have that $(A_1, A_2) \in L$.

In the induction step, suppose $(g, h) \in \mathcal{B}_R^{n+1} \circ \mathrm{ABOVE}_R(R)$. From the definition of $\mathcal{B}_R$ we have that there are three cases. First, suppose $(g, h) \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$. From the induction hypothesis we have that the two desired properties are satisfied.

Second, suppose for some ℓ and $(g', h') \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$, such that $\mathrm{Var}(g', \ell) \sqsubseteq \mathrm{Var}(h', \ell)$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. From the induction hypothesis we

have $g' \supseteq h'$. From $g' \supseteq h'$ and the definition on δ_R it is immediate that $g \supseteq h$. Suppose $A_1, A_2 \in g$. From $g = \delta_R(g')(\ell)$ and the definition of δ_R, we have that there exists $[\ell^v : A'_1, \ldots], [\ell^{v'} : A'_2, \ldots] \in g'$, and $(A'_1, A_1), (A'_2, A_2) \in R$. From the induction hypothesis we have $([\ell^v : A'_1, \ldots], [\ell^{v'} : A'_2, \ldots]) \in L$, so from Lemma 3.5.3, property **(A)**, we have that $(A'_1, A'_2) \in L$. From Lemma 3.5.3, Property **(B)**, $(A'_1, A'_2) \in L$, $(A'_1, A_1), (A'_2, A_2) \in R$ we have $(A_1, A_2) \in L$.

Third, suppose for some ℓ and $(h', g') \in \mathcal{B}^n_R \circ \mathrm{ABOVE}_R(R)$, we have $g = \delta_R(g')(\ell)$, $h = \delta_R(h')(\ell)$, $\mathrm{Var}(g', \ell) = 0$, and $\mathrm{Var}(h', \ell) = 0$. From the definition of δ_R and $\mathrm{Var}(g', \ell) = 0$, we have that there exists at least one type $[\ell^0 : A, \ldots] \in g'$. From the induction hypothesis we have $h' \supseteq g'$. Thus, $[\ell^0 : A, \ldots] \in h'$. To prove $g \supseteq h$, suppose $B \in h$. By the definition of δ_R and $\mathrm{Var}(h', \ell) = 0$, there must exist a type $[\ell^0 : B', \ldots] \in h'$ such that $(B', B) \in R$. From the induction hypothesis and $[\ell^0 : A, \ldots], [\ell^0 : B', \ldots] \in h'$, we have $([\ell^0 : A, \ldots], [\ell^0 : B', \ldots]) \in L$. From sat-closure rule (ix), we have $(A, B') \in R$. Therefore, from the transitivity of R (sat-closure rule (ii)) and $(A, B'), (B', B) \in R$, we have $(A, B) \in R$, so $B \in g$, and hence $g \supseteq h$. The property that "if $A_1, A_2 \in g$, then $(A_1, A_2) \in L$" can be proved in the same way as in the previous case.

□

Lemma 3.5.9. *If R is sat-closed, then the subtype-closure of $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$ is subtype-consistent.*

Proof.

$$
\begin{aligned}
&\text{The subtype-closure of } \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R) \\
&= \bigcup_{0 \le n < \infty} \mathcal{A}^n \circ \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R) \qquad \text{(Definition of subtype-closure)} \\
&= \bigcup_{0 \le n < \infty} \mathrm{TYPE}_R \circ \mathcal{B}^n_R \circ \mathrm{ABOVE}_R(R) \qquad \text{(Lemma 3.5.7)} \\
&= \bigcup_{0 \le n < \infty} \bigcup_{(g,h) \in \mathcal{B}^n_R \circ \mathrm{ABOVE}_R(R)} \{(\mathrm{Type}_R(g), \mathrm{Type}_R(h))\} \qquad \text{(Definition of } \mathrm{TYPE}_R\text{)}.
\end{aligned}
$$

From Lemma 3.5.8 we have that if $(g, h) \in \mathcal{B}^n_R \circ \mathrm{ABOVE}_R(R)$, then $g \supseteq h$. If $g \supseteq h$, then it is immediate from the definition of Type_R that $\{(\mathrm{Type}_R(g), \mathrm{Type}_R(h))\}$ is subtype-consistent. Thus, the subtype-closure of $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$ is the union of a family

of subtype-consistent relations. Since the union of a possibly infinite family of subtype-consistent relations is itself subtype-consistent, we conclude that the subtype-closure of $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$ is subtype-consistent. □

The following lemma is a key ingredient in the proof of Lemma 3.5.11. The two lemmas 3.5.10 and 3.5.11 are the two places where it is needed that a relation is satisfaction-consistent.

Lemma 3.5.10. *If $A = [\ell^v : B, \ldots]$ appears in R and R is sat-closed and sat-consistent, then*

$$\mathrm{above}_R((\mathrm{above}_R(\{A\})).\ell) = \mathrm{above}_R(\{B\}).$$

Proof. To prove the direction $\supseteq$, notice that from sat-closure rule (i) and A appearing in R, we have $(A, A) \in R$, so $A \in \mathrm{above}_R(\{A\})$, hence $B \in (\mathrm{above}_R(\{A\})).\ell$, and thus

$$\mathrm{above}_R((\mathrm{above}_R(\{A\})).\ell) \supseteq \mathrm{above}_R(\{B\}).$$

To prove the direction $\subseteq$, suppose $C \in \mathrm{above}_R((\mathrm{above}_R(\{A\})).\ell)$. From that we have there exists $C' \in (\mathrm{above}_R(\{A\})).\ell$ such that $(C', C) \in R$. From $C' \in (\mathrm{above}_R(\{A\})).\ell$ we have that there exists $[\ell^{v'} : C', \ldots]$ such that $(A, [\ell^{v'} : C', \ldots]) \in R$. From sat-consistency and $(A, [\ell^{v'} : C', \ldots]) \in R$, we have that $v \sqsubseteq v'$. From Lemma 3.5.3, Property **(D)**, $(A, [\ell^{v'} : C', \ldots]) \in R$, and $v \sqsubseteq v'$, we have that $(B, C') \in R$. From transitivity of R (sat-closure rule (ii)) and $(B, C'), (C', C) \in R$, we have $(B, C) \in R$, so $C \in \mathrm{above}_R(\{B\})$. □

Recall that for any relation R on types, we have defined S_R to be the least substitution such that for every U appearing in R, we have $S_R(U) = \mathrm{Type}_R(\mathrm{above}_R(\{U\}))$.

Lemma 3.5.11. *If R is sat-closed and sat-consistent, then*

i) *for any type A appearing in R, $S_R(A) = \mathrm{Type}_R \circ \mathrm{above}_R(\{A\})$; and*

ii) $S_R(R) = \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$.

Proof. The second property is an immediate consequence of the first property.

To prove the first property, we will, by induction on α, show that for all α, for all A appearing in R, $S_R(A)(\alpha) = \mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\alpha)$.

If $\alpha = \epsilon$ and A is a type variable, the result follows by definition of S_R.

If $\alpha = \epsilon$ and $A = [\ell_i^{v_i} : B_i^{1..n}]$, then $S_R(A)(\alpha) = \{(\ell_i, v_i) \mid i \in 1..n\}$ and $\mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\alpha) = \mathrm{LV}(\mathrm{above}_R(\{A\}))$. From sat-closure rule (i) and A appearing in R, we have $(A, A) \in R$, so $A \in \mathrm{above}_R(\{A\})$. From $A \in \mathrm{above}_R(\{A\})$ and sat-consistency, we have $\mathrm{LV}(\mathrm{above}_R(\{A\})) = \mathrm{LV}(\{A\}) = \{(\ell_i, v_i) \mid i \in 1..n\}$, as desired.

If $\alpha = \ell\alpha'$ and A is a type variable, the result follows by definition of S_R.

If $\alpha = \ell\alpha'$ and $A = [\ell^v : B, \ldots]$, then

$$
\begin{aligned}
& S_R(A)(\alpha) \\
&= S_R(B)(\alpha') && \text{(Definition of } S_R\text{)} \\
&= \mathrm{Type}_R \circ \mathrm{above}_R(\{B\})(\alpha') && \text{(Induction hypothesis)} \\
&= \mathrm{LV}(\delta_R^*(\mathrm{above}_R(\{B\}))(\alpha')) && \text{(Definition of } \mathrm{Type}_R\text{)} \\
&= \mathrm{LV}(\delta_R^*(\mathrm{above}_R((\mathrm{above}_R(\{A\})).\ell))(\alpha')) && \text{(Lemma 3.5.10)} \\
&= \mathrm{LV}(\delta_R^*(\delta_R(\mathrm{above}_R(\{A\}))(\ell))(\alpha')) && \text{(Definition of } \delta_R\text{)} \\
&= \mathrm{LV}(\delta_R^*(\mathrm{above}_R(\{A\}))(\ell\alpha')) && \text{(Definition of } \delta_R^*\text{)} \\
&= \mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\alpha) && \text{(Definition of } \mathrm{Type}_R \text{ and } \alpha = \ell\alpha'\text{)}.
\end{aligned}
$$

If $\alpha = \ell\alpha'$ and A is a record without an ℓ field, then $S_R(A)(\alpha)$ is undefined. By sat-consistency, no $C \in \mathrm{above}_R(\{A\})$ has an ℓ field, so from the definition of Type_R we have that $\mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\ell\alpha')$ is undefined, as desired. □

We are now ready to prove the main result of this section.

Theorem 3.5.12. *R is solvable if and only if its sat-closure is sat-consistent.*

Proof. If R is solvable, then we have from Lemma 3.5.2 that the sat-closure of R is solvable, so from Lemma 3.5.5 we have the sat-closure of R is sat-consistent.

Conversely, let R' be the sat-closure of R, and assume that R' is sat-consistent. From Lemma 3.5.9 and Lemma 3.5.11, we have that the subtype-closure of $S_{R'}(R')$ is subtype-consistent, so from Lemma 3.2.6, we have that R' has solution $S_{R'}$, and so from Lemma 3.5.2 we have that R has solution $S_{R'}$. □

In summary, to solve a constraint set R, we proceed as follows. First, we compute the sat-closure R' of R (this takes $O(n^3)$ time where n is the size of R). Next, the R' is checked for sat-consistency (this takes $O(n)$ time). If R' is sat-inconsistent, then R is not solvable, otherwise $S_{R'}$ is an example of a solution of R. Thus, we have shown the following result.

Corollary 3.5.13. *Satisfiability of a constraint set is decidable in $O(n^3)$ time.*

3.6 P-hardness

Theorem 3.6.1. *Solvability of constraints is P-hard.*

Proof. An SC-system (simple constraint set) is a finite set of constraints of the forms

$$\begin{aligned} V &\equiv V' \\ (V &, [\ell_i^0 : V_i \ ^{i\in 1..n}]). \end{aligned}$$

Notice that any SC-system is a constraint set. Thus it is sufficient to prove that solvability of SC-systems over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$ is P-hard. We will do that by reducing a closely related P-hard problem to this problem.

Let $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$ be the subset of $\mathcal{T}_{\mathrm{reg}}(\Sigma)$ where all variance annotations are 0. In [Pal95] it is proved that solvability of SC-systems over $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$ is P-hard.

Let R be an SC-system. It is sufficient to prove that R is solvable over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$ if and only if it is solvable over $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$.

It is immediate that if R is solvable over $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$ then it is solvable over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$.

Conversely, suppose that R is solvable over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$, and let R' be the sat-closure of R. We have that $S_{R'}$ is a solution of R', and therefore, by Lemma 3.5.2, $S_{R'}$ is also a solution of R. By Lemma 3.6.2, $S_{R'}$ maps all variables in R to elements of $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$, □

Lemma 3.6.2. *Let R be an SC-system, and let R' be the sat-closure of R. If R is solvable over $\mathcal{T}_{\mathrm{reg}}(\Sigma)$, then $S_{R'}(V) \in \mathcal{T}^0_{\mathrm{reg}}(\Sigma)$ for all V in R.*

Proof. From the sat-closure rules, it follows that R' is an SC-system as well. It is clear from the definition that $\mathrm{above}_{R'}$ produces only types in $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$. Also by definition, $\mathrm{Type}_{R'}(g)(\alpha) = \mathrm{LV}(\delta^*_{R'}(g)(\alpha))$, and, by the definition of $\delta_{R'}$, we have that $\delta^*_{R'}(g)(\alpha)$ is a set of types in $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$ for any type set g and path α. Thus, the function $\mathrm{Type}_{R'}$ produces only types in $\mathcal{T}^0_{\mathrm{reg}}(\Sigma)$. Since $S_{R'}(V) = \mathrm{Type}_{R'}(\mathrm{above}_{R'}(\{V\}))$, we have that $S_{R'}(V) \in \mathcal{T}^0_{\mathrm{reg}}(\Sigma)$. □

3.7 Conclusion

We can do type inference in polynomial time for objects with both covariant and invariant fields. Covariant read-only fields can be either explicitly specified or discovered by our algorithm. Perhaps surprisingly, specifying read-only fields explicitly seems not to make type inference easier.

It may be possible to represent types as feature trees and thereby build a connection to solving feature constraints [MNP00]; this is left to future work.

An interesting idea for an alternative approach to type inference that may be able to handle all variances $\{0, +, -\}$ was suggested by one of the anonymous reviewers. The idea is to define a metric $d(A, B)$ on the types, and show that it is a complete metric space and all the constructions are contractive (or non-expansive). Then with the Banach Fixpoint Theorem, we may be able show that there is a solution if the closure is consistent. This approach is similar to [MPS86, AW93] and is an interesting idea for future work.

4. Efficient Type Inference for Record Concatenation and Subtyping

4.1 Introduction

4.1.1 Background

In Cardelli's untyped Obliq language [Car95], the operation

$$clone(a_1, \ldots, a_n)$$

creates a new object that contains the fields and methods of all the argument objects $a_1, \ldots, a_n$. This is done by first cloning each of $a_1, \ldots, a_n$, and then concatenating the clones. An error is given in case of field name conflicts, that is, in case at least two of $a_1, \ldots, a_n$ have a common field. Cardelli notes that useful idioms are:

$$clone(a, \{l : v\})$$

to inherit the fields of a and add a new field l with initial value v, and:

$$clone(a_1, a_2)$$

to multiply inherit from a_1 and a_2.

Obliq's multiple-object cloning is an instance of the idea of concatenating two records of data. In a similar fashion, languages such as C++ [Str93] and Borning and Ingalls' [BI82] version of Smalltalk allow multiple inheritance of classes.

In this chapter, we focus on languages such as Obliq where concatenation is a run-time operation and where a field name conflict is considered an error; such concatenation is known as *symmetric concatenation*. There are several ways of handling field name conflicts. One idea is to do run-time checking, and thereby add some overhead to the execution time. Another idea, which we pursue here, is to statically detect field name errors by a type system. The main challenge for such a type system is to find out which objects will eventually be concatenated and give them types that support concatenation.

Type systems for record concatenation have been studied by Wand [Wan91], Harper and Pierce [HP91], Remy [Rem92], Shields and Meijer [SM01], Tsuiki [Tsu94], Zwanenburg [Zwa95, Zwa96] and others. These type systems use ideas such as row variables, present-fields and absent-fields, type-indexed rows, second-order types, and intersection types. More recently, Sulzmann [Sul97] and Pottier [Pot00] have studied type inference with the combination of record concatenation and subtyping. None of these algorithms are, as far as we are aware, known to run in polynomial time.

In this chapter, we investigate the idea of using variance annotations [PS93, AC96a] together with subtyping and recursive types as the basis for typing record concatenation. Following Glew [Gle00], we will use two forms of record types. The variance annotation 0, as in

$$[\ell_i : B_i \,^{i\in 1..n}]^0,$$

denotes that records of that type *can* be concatenated, and that subtyping *cannot* be used. The variance annotation $\rightarrow$, as in

$$[\ell_i : B_i \,^{i\in 1..n}]^{\rightarrow},$$

denotes that records of that type *cannot* be concatenated, and that subtyping *can* be used. For example, if we have

$$\begin{aligned} [l : 5, m : true] &: [l : int, m : boolean]^0 \\ [n : 7] &: [n : int]^0 \end{aligned}$$

then for the concatenation (denoted by $+$) of the two records we would get

$$\begin{aligned} [l : 5, m : true] + [n : 7] &: [l : int, m : boolean]^0 \oplus [n : int]^0 \\ &= [l : int, m : boolean, n : int]^0. \end{aligned}$$

where $\oplus$ is the symmetric concatenation operation on record types which is only defined when the labels sets are disjoint and the two types both have the variance annotation 0. The idea is that if an object has type $[l_i : t_i]^0$, then we know exactly which fields are in the object, and hence we know which other fields we can safely add without introducing a field

name conflict. The more flexible types $[\ell_i : B_i \ ^{i\in 1..n}]^{\rightarrow}$ can be used to type objects that will not be concatenated with other objects.

We restrict our attention to width-subtyping for types with variance annotation $\rightarrow$, and we allow subtyping from variance annotation 0 to $\rightarrow$. Going from 0 to $\rightarrow$ is in effect to forget that a record of that type can be concatenated with other records. Our type system is simpler and less expressive than some previous type systems for record concatenation. Our goal is to analyze the computational complexity of type inference. That complexity may well be less than the complexity of type inference for some of the more expressive type systems.

4.1.2 Our Result

We present the design of a type inference algorithm for the Abadi-Cardelli object calculus extended with a concatenation operator. The type system supports subtyping and recursive types. Our algorithm enables type checking of Obliq programs without changing the programs at all; extending our results to Obliq is left for future work. We prove that the type inference problem is NP-complete.

Our NP algorithm works by reducing type inference to the problem of solving a set of *constraints*. A constraint is a pair (A, B), where A and B are types that may contain type variables and the concatenation operator $\oplus$; and the goal is to find a substitution S such that for each constraint (A, B), we have $S(A) \leq S(B)$ where $\leq$ is the subtype order. We will use R to range over sets of constraints; we will often refer to R as a relation on types. A key theorem states:

> **Theorem** A set of constraints is solvable if and only there exists a closed superset which is consistent.

Here, "closure" means that certain syntactic consequences of the constraints have been added to the constraint set, and "consistent" means that there are no obviously unsatisfiable constraints (e.g., $([m : V]^0, [l : U]^0)$). The algorithm constructs a solution from a closed, consistent constraint set. To solve a constraint set R generated from a program a, we first guess a superset R' of R. Next we check that R' is closed and consistent; this can be

done in polynomial time. This framework has been used for solving subtype constraints for a variety of types [Pal95, KPS94, PWO97, PO95, PS96, PZJ02]. A key difference from these papers is that our constraint problem does not admit a *smallest* closed superset which is consistent. As a reflection of that, the algorithms in [Pal95, KPS94, PWO97, PO95, PS96, PZJ02] all run in polynomial time, while the type inference problem considered here is NP-complete. This is because in the referenced papers, the smallest closed superset of a given constraint set can be computed in polynomial time, while our algorithm has to guess a closed superset.

All type-inference algorithms based on this framework, including the one in this chapter, can be viewed as whole-program analyses because they use a constraint set generated from the whole program. A whole-program analysis can be made modular in several ways [CC02]. For example, we can generalize to type inference with respect to a fixed (non-empty) typing environment. One would start the algorithm with an initial set of constraints for program variables, derived from that fixed environment. Thus, one could collect (or constrain) the substitution provided by a run of the algorithm as an interface to a further program fragment that uses the first one as a library.

Our algorithm uses a new notion of closure and a traditional notion of consistency. Our seven closure rules capture various aspects of the subtyping order. For example, one closure rule ensures that if

$$(V \oplus V', [l : U]^{\rightarrow})$$

is a constraint, then either V or V' must be forced to have an l-field, as illustrated in the example below. That closure rule highlights why the type inference problem is NP-complete: there is a choice which possibly later has to be undone.

In our proof of the main theorem we use the technique of Palsberg, Zhao, and Jim [PZJ02] that employs a convenient characterization of the subtyping order (Lemma 4.2.6). The characterization uses notions of subtype-closure and subtype-consistency that are different, yet closely related, to the already-mentioned notions of what we for clarity will call satisfaction-closure and satisfaction-consistency. The chapter 3 (see also [PZJ02]) concerns type inference with both covariant and invariant fields, and for types that all allow

width-subtyping. In the present chapter, all fields are invariant, but some types (those with the variance-annotation 0) do not admit non-trivial subtyping. While the type inference algorithms reported in the two chapters are entirely different, their correctness proofs have the same basic structure.

4.1.3 Example

We now present an example that gives a taste of the definitions and techniques that are used later in the chapter. Our example program a has two methods l and m:

$$a \;=\; [l = \varsigma(x)(x.l + x.m).k,\ m = \varsigma(y)y.m].$$

When running our type inference algorithm by hand on this program, the result is that a is typable with type

$$a \;:\; [l : \mu\alpha.[k : \alpha]^0, m : [\,]^0]^0.$$

The goal of this section is to illustrate how the algorithm arrives at that conclusion.

We can use the rules in Section 4.4 to generate the following set of constraints, called R. In the left column are all occurrences of subterms in the program; in the right column are the constraints generated for each occurrence. We use $A \equiv B$ to denote the pair of

constraints (A, B) and (B, A).

Occurrence	Constraints
x	(U_x, V_x)
y	(U_y, V_y)
a	$([l : V_{(x.l+x.m).k},\ m : V_{y.m}]^0, V_a)$
	$U_x \equiv [l : V_{(x.l+x.m).k},\ m : V_{y.m}]^0$
	$U_y \equiv [l : V_{(x.l+x.m).k},\ m : V_{y.m}]^0$
$(x.l + x.m).k$	$(V_{x.l+x.m}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$
	$(U_{(x.l+x.m).k}, V_{(x.l+x.m).k})$
$x.l + x.m$	$(V_{x.l} \oplus V_{x.m}, V_{x.l+x.m})$
$x.l$	$(V_x, [l : U_{x.l}]^{\rightarrow})$
	$(U_{x.l}, V_{x.l})$
$x.m$	$(V_x, [m : U_{x.m}]^{\rightarrow})$
	$(U_{x.m}, V_{x.m})$
$y.m$	$(V_y, [m : U_{y.m}]^{\rightarrow})$
	$(U_{y.m}, V_{y.m})$

Notice that, for each bound variable x, we have a type variable U_x. Moreover, for each occurrence of x, we have a type variable V_x. Intuitively, U_x stands for the type of x in the type environment, while V_x stands for the type of an occurrence of x after subtyping. Similarly, $U_{x.l}$ stands for the type of $x.l$ before subtyping, while $V_{x.l}$ stands for the type of $x.l$ after subtyping.

Next, our type inference algorithm will guess a so-called satisfaction-closed superset R' of R. We will here display and motivate some of the interesting constraints in a particular R'. First, from the constraints

$$(U_x, V_x)$$
$$(V_x, [l : U_{x.l}]^{\rightarrow})$$

and transitivity, we have

$$(U_x, [l : U_{x.l}]^{\rightarrow})$$

in R'. Second, from that constraint and

$$U_x \equiv [l : V_{(x.l+x.m).k},\ m : V_{y.m}]^0$$

and the observation that fields have invariant subtyping, we have

$$(V_{(x.l+x.m).k}, U_{x.l})$$

in R'. Third, from the constraints

$$(V_{x.l} \oplus V_{x.m}, V_{x.l+x.m})$$
$$(V_{x.l+x.m}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$$

and transitivity, we have

$$(V_{x.l} \oplus V_{x.m}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$$

in R'. At this point there is a choice. We can force either $V_{x.l}$ or $V_{x.m}$ to be mapped to a type with a k-field. Since there are no other significant constraints on either $V_{x.l}$ or $V_{x.m}$, both choices will be fine. Our algorithm chooses the first one, and so we have the constraint

$$(V_{x.l}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$$

in R'. After this constraint has been added, we can apply transitivity three times to:

$$(U_{(x.l+x.m).k}, V_{(x.l+x.m).k})$$
$$(V_{(x.l+x.m).k}, U_{x.l})$$
$$(U_{x.l}, V_{x.l})$$
$$(V_{x.l}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$$

so we have

$$(U_{(x.l+x.m).k}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$$

in R'. The last constraint makes it apparent that recursive types are needed to solve the constraint system and therefore to type the example program.

Note that the choice we made in applying closure rules to $(V_{x.l} \oplus V_{x.m}, [k : U_{(x.l+x.m).k}]^{\rightarrow})$ implies that sometimes there is no unique solution to our type-inference problem.

Thus, if we want to do type inference for a program fragment without an initial type environment, the best we can do is to generate the constraints, perhaps simplify them [Pot96], and delay solving them until the constraints for the other program fragments become available.

Once our type inference algorithm has guessed a sat-closed R', it checks whether R' is sat-consistent, that is, whether there is at least one constraint which obviously is unsolvable, e.g., $([m : V]^0, [l : U]^0)$. If R' is not sat-consistent, then R has no solution. In the case of the example program, R' is sat-consistent, and our type inference algorithm then derives the following solution from R'. Define

$$\begin{aligned} P &\equiv \mu\alpha.[k : \alpha]^0 \\ Q &\equiv [l : P, m : [\,]^0]^0 \\ E &\equiv \emptyset[x : Q] \\ F &\equiv \emptyset[y : Q], \end{aligned}$$

where P, Q are types, and E, F are type environments. Note that we use so-called equi-recursive types that satisfy a certain equation, rather than the kind of recursive types that have to be explicitly folded and unfolded.

We can derive $\emptyset \vdash a : Q$ as follows.

$$\dfrac{\dfrac{\dfrac{E \vdash (x.l + x.m) : [k : P]^0}{E \vdash (x.l + x.m) : [k : P]^{\rightarrow}}}{E \vdash (x.l + x.m).k : P} \quad \dfrac{\dfrac{F \vdash y : Q}{F \vdash y : [m : [\,]^0]^{\rightarrow}}}{F \vdash y.m : [\,]^0}}{\emptyset \vdash a : Q}$$

Notice the two uses of subsumption:

$$\begin{aligned} [k : P]^0 &\leq [k : P]^{\rightarrow} \\ Q &\leq [m : [\,]^0]^{\rightarrow}. \end{aligned}$$

We can derive $E \vdash (x.l + x.m) : [k : P]^0$ as follows. Notice that $[k : P]^0 = [k : P]^0 \oplus [\,]^0$.

$$\dfrac{\dfrac{\dfrac{E \vdash x : Q}{E \vdash x : [l : [k : P]^0]^{\to}}}{E \vdash x.l : [k : P]^0} \quad \dfrac{\dfrac{E \vdash x : Q}{E \vdash x : [m : [\,]^0]^{\to}}}{E \vdash x.m : [\,]^0}}{E \vdash (x.l + x.m) : [k : P]^0}$$

Notice the two uses of subsumption:

$$\begin{aligned} Q &\leq [l : [k : P]^0]^{\to} \\ Q &\leq [m : [\,]^0]^{\to}. \end{aligned}$$

We derive the first of these inequalities using the unfolding rule for recursive types to get

$$P = \mu\alpha.[k : \alpha]^0 = [k : \mu\alpha.[k : \alpha]^0]^0 = [k : P]^0,$$

and therefore

$$Q = [l : P, m : [\,]^0]^0 = [l : [k : P]^0, m : [\,]^0]^0.$$

Here is an alternative typing, which arises from forcing $V_{x.m}$ to be mapped to a type with a k-field:

$$\emptyset \vdash a : [l : [\,]^0, m : [k : [\,]^0]^0]^0.$$

4.2 Types and Subtyping

We will work with recursive types, and we choose to represent them by possibly infinite trees.

4.2.1 Defining types as infinite trees

We use U, V to range over the set $\mathcal{TV}$ of type variables; we use k, ℓ, m to range over labels drawn from some possibly infinite set Labels of method names; and we use v to range over the set $\mathsf{Variances} = \{0, \to\}$ of variance annotations. Variance annotations are ordered by the smallest partial order $\sqsubseteq$ such that $0 \sqsubseteq \to$.

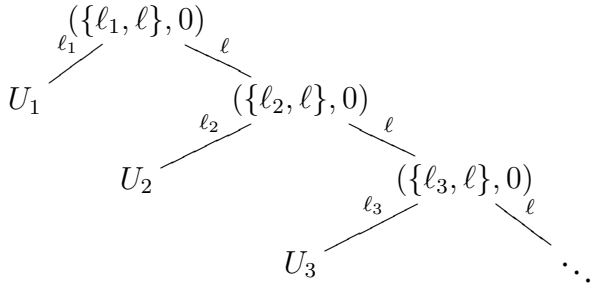
The alphabet Σ of our trees is defined

$$\Sigma = \mathcal{TV} \cup (\mathcal{P}(\mathsf{Labels}) \times \mathsf{Variances}).$$

A *path* is a finite sequence $\alpha \in \mathsf{Labels}^*$ of labels, with juxtaposition for concatenation of paths, and ϵ for the empty sequence. A *type* or *tree* A is a partial function from paths into Σ, whose domain is nonempty and prefix closed, and such that $A(\alpha) = (\{\ell_i \mid i \in I\}, v)$ if and only if $\forall i,\ A(\alpha\ell_i)$ is defined. We use A, B, C to range over the set $\mathcal{T}(\Sigma)$ of trees.

Note that trees need not be finitely branching or regular. A regular tree has finitely many distinct subtrees [Cou83]. Of course, we will be particularly interested in two subsets of $\mathcal{T}(\Sigma)$, the finite trees $\mathcal{T}_{\text{fin}}(\Sigma)$ and the finitely branching and regular trees $\mathcal{T}_{\text{reg}}(\Sigma)$. Some definitions, results, and proofs are given in terms of $\mathcal{T}(\Sigma)$, in such a way that they immediately apply to $\mathcal{T}_{\text{fin}}(\Sigma)$ and $\mathcal{T}_{\text{reg}}(\Sigma)$.

An example tree is given below.



We now introduce some convenient notation. We write $A(\alpha) = \uparrow$ if A is undefined on α. If for all $i \in I$, B_i is a tree, ℓ_i is a distinct label, and $v \in \mathsf{Variances}$, then $[\ell_i : B_i \ {}^{i \in I}]^v$ is the tree A such that

$$A(\alpha) \quad = \quad \begin{cases} (\{\ell_i \mid i \in I\}, v) & \text{if } \alpha = \epsilon \\ B_i(\alpha') & \text{if } \alpha = \ell_i\alpha' \text{ for some } i \in I \\ \uparrow & \text{otherwise.} \end{cases}$$

We abuse notation and write U for the tree A such that $A(\epsilon)$ is the type variable U and $A(\alpha) = \uparrow$ for all $\alpha \neq \epsilon$.

Recursive types are regular trees, and they can be presented by μ-expressions [Cou83, AC93] generated by the following grammar:

$$\begin{array}{rcll} A, B & ::= & U, V & \text{type variable} \\ & \mid & [\ell_i : B_i \ {}^{i \in 1..n}]^{\phi} & \text{object type } (\ell_i\text{'s distinct, } \phi ::= 0 \mid \rightarrow) \\ & \mid & \mu U.A & \text{recursive type} \end{array}$$

We can now define the concatenation operator $\oplus$. If

$$\begin{aligned} A &= [\ell_i : B_i \ ^{i\in I}]^0 \\ A' &= [\ell_i : B_i \ ^{i\in I'}]^0 \end{aligned}$$

and $I \cap I' = \emptyset$, then

$$A \oplus A' = [\ell_i : B_i \ ^{i\in I\cup I'}]^0,$$

and otherwise $A \oplus A'$ is undefined.

4.2.2 Defining Subtyping via Simulations

Our subtyping order supports width subtyping but not depth subtyping.

Definition 4.2.1. A relation R over $\mathcal{T}(\Sigma)$ is called a *simulation* if for all $(A, A') \in R$, we have the following conditions.

- For all U, $A = U$ if and only if $A' = U$.
- For all ℓ_i, $i \in I'$, B'_i, if $A' = [\ell_i : B'_i \ ^{i\in I'}]^{\phi'}$, then there exist B_i such that

$$\begin{gathered} A = [\ell_i : B_i \ ^{i\in I}]^{\phi}, \quad I' \subseteq I, \quad \phi' \sqsupseteq \phi \\ (B_i, B'_i), (B'_i, B_i) \in R, \quad \phi' = 0 \Rightarrow I' = I. \end{gathered}$$

□

Notice that a simulation can contain pairs such as $([\ldots]^0, [\ldots]^{\rightarrow})$, but not $([\ldots]^{\rightarrow}, [\ldots]^0)$. Notice also that the last line of Definition 4.2.1 enforces no depth subtyping.

For example, the empty relation on $\mathcal{T}(\Sigma)$ and the identity relation on $\mathcal{T}(\Sigma)$ are both simulations. Simulations are closed under unions and intersections, and there is a largest simulation, which we call $\leq$ and use as our subtyping order:

$$\leq \ = \bigcup\{R \mid R \text{ is a simulation}\}. \tag{4.1}$$

Alternately, $\leq$ can be seen as the maximal fixed point of a monotone function on $\mathcal{P}(\mathcal{T}(\Sigma) \times \mathcal{T}(\Sigma))$. Then we immediately have the following result.

Lemma 4.2.2. *$A \leq A'$ if and only if*

- *For all U, $A = U$ if and only if $A' = U$.*
- *For all ℓ_i, B'_i, $i \in I'$, and ϕ', if $A' = [\ell_i : B'_i \ ^{i \in I'}]^{\phi'}$, then there exist B_i, such that*

$$A = [\ell_i : B_i \ ^{i \in I}]^{\phi}, \quad I' \subseteq I, \quad \phi' \sqsupseteq \phi, \text{ and}$$
$$\forall i \in I, B_i = B'_i, \quad \phi' = 0 \Rightarrow I' = I.$$

All of these results are standard in concurrency theory, and have easy proofs, c.f. [Mil90]. Similarly, it is easy to show that $\leq$ is a preorder. Our simulations differ from the simulations typically found in concurrency in that they are all anti-symmetric (again, the proof is easy).

Lemma 4.2.3. *$\leq$ is a partial order.*

Proof. See Appendix A.3. □

We may apply the principle of *co-induction* to prove that one type is a subtype of another:

> **Co-induction**: To show $A \leq B$, it is sufficient to find a simulation R such that $(A, B) \in R$.

4.2.3 A characterization of subtyping

We now give a characterization of subtyping (Lemma 4.2.6) which will be used in the proof of the main theorem (Theorem 4.5.15). Suppose R is a relation on types, and we want to know whether $A \leq B$ for every $(A, B) \in R$. By co-induction this is equivalent to the existence of a simulation containing R. And since simulations are closed under intersection, this is equivalent to the existence of a *smallest* simulation containing R. We can characterize this smallest simulation as follows.

Definition 4.2.4. We say a relation R on types is subtype-closed if $([\ell : B, \ldots]^{\phi}, [\ell : B', \ldots]^{\phi'}) \in R$ implies $(B, B'), (B', B) \in R$. □

Note that the subtype-closed relations on types are closed under intersection; therefore for any relation R on types, we may define its *subtype-closure* to be the smallest subtype-closed relation containing R. Every simulation is subtype-closed, and subtype-closure is a monotone operation.

Definition 4.2.5. We say a relation R on types is subtype-consistent if $[\ell_i : B_i \ ^{i \in I}]^{\phi}, \quad [\ell_i : B'_i \ ^{i \in I'}]^{\phi'}) \in R$, implies

- if $\phi' = 0$, then $\phi = 0$ and $I = I'$,
- if $\phi' = \rightarrow$, then $I \supseteq I'$.

□

Note that every simulation is subtype-consistent, and moreover, any subset of an subtype-consistent set is subtype-consistent.

Lemma 4.2.6. *Let R be a relation on types. The following statements are equivalent.*

i) $A \leq B$ *for every* $(A, B) \in R$.

ii) *The subtype-closure of R is a simulation.*

iii) *The subtype-closure of R is subtype-consistent.*

Proof.

- (ii) $\Rightarrow$ (i): Immediate by co-induction.
- (i) $\Rightarrow$ (iii): R is a subset of $\leq$, so by monotonicity and the fact that $\leq$ is subtype-closed, the subtype-closure of R is a subset of $\leq$. Then since $\leq$ is subtype-consistent, its subset, the subtype-closure of R, is subtype-consistent.
- (iii) $\Rightarrow$ (ii): Let R' be the subtype-closure of R, and suppose $(A, A') \in R'$.

  If $A = U$, by subtype-consistency $A' = U$; and similarly, if $A' = U$, then $A = U$.

  If $A' = [\ell_i : B'_i \ ^{i \in I'}]^{\phi'}$, by subtype-consistency A must be of the form $[\ell_i : B_i \ ^{i \in I}]^{\phi}$, where $\phi \sqsubseteq \phi'$. And since R' is subtype-closed, $(B_i, B'_i), (B'_i, B_i) \in R'$ and $I' \subseteq I$, and $\phi' = 0 \Rightarrow I' = I$, as desired.

□

4.3 The Abadi-Cardelli Object Calculus

We now present an extension of the Abadi-Cardelli object calculus [AC96a] and a type system. The types are recursive types as defined in the previous section.

We use x, y to range over term variables. Expressions are defined by the following grammar.

a, b, c	$::=$	x	variable
	$\mid$	$[\ell_i = \varsigma(x_i)b_i{}^{i\in 1..n}]$	object (ℓ_i distinct)
	$\mid$	$a.\ell$	field selection / method invocation
	$\mid$	$(a.\ell \Leftarrow \varsigma(x)b)$	field update / method update
	$\mid$	$a_1 + a_2$	object concatenation

An object $[\ell_i = \varsigma(x_i)b_i{}^{i\in 1..n}]$ has method names ℓ_i and methods $\varsigma(x_i)b_i$. The order of the methods does not matter. Each method binds a name x which denotes the smallest enclosing object, much like "this" in Java. Those names can be chosen to be different, so within a nesting of objects, one can refer to any enclosing object. A *value* is of the form $[\ell_i = \varsigma(x_i)b_i{}^{i\in 1..n}]$. A *program* is a closed expression.

A small-step operational semantics is defined by the following rules:

- If $a \equiv [\ell_i = \varsigma(x_i)b_i{}^{i\in 1..n}]$, then, for $j \in 1..n$,
  - $a.\ell_j \ \leadsto \ b_j[x_j := a]$,
  - $(a.\ell_j \Leftarrow \varsigma(y)b) \ \leadsto \ a[\ell_j \leftarrow \varsigma(y)b]$.
- If $a_1 \equiv [\ell_i = \varsigma(x_i)b_i{}^{i\in I_1}]$, $a_2 \equiv [\ell_i = \varsigma(x_i)b_i{}^{i\in I_2}]$, and $I_1 \cap I_2 = \emptyset$, then

$$a_1 + a_2 \ \leadsto \ [\ell_i = \varsigma(x_i)b_i{}^{i\in I_1\cup I_2}].$$

- If $b \leadsto b'$ then $a[b] \leadsto a[b']$.

Here, $b_j[x_j := a]$ denotes the ς-term b_j with a substituted for free occurrences of x_j (renaming bound variables to avoid capture); and $a[\ell_j \leftarrow \varsigma(y)b]$ denotes the expression a with

the ℓ_j field replaced by $\varsigma(y)b$. A *context* is an expression with one hole, and $a[b]$ denotes the term formed by replacing the hole of the context $a[\cdot]$ by the term b (possibly capturing free variables in b).

An expression b is *stuck* if it is not a value and there is no expression b' such that $b \rightsquigarrow b'$. An expression b *goes wrong* if $\exists b' : b \rightsquigarrow^* b'$ and b' is stuck.

A type environment is a partial function with finite domain which maps term variables to types in $\mathcal{T}_{\mathrm{reg}}(\Sigma)$. We use E to range over type environments. We use $E[x : A]$ to denote a partial function which maps x to A, and maps y, where $y \neq x$, to $E(y)$.

The typing rules below allow us to derive judgments of the form $E \vdash a : A$, where E is a type environment, a is an expression, and A is a type in $\mathcal{T}_{\mathrm{reg}}(\Sigma)$.

$$E \vdash x : A \quad (\text{provided } E(x) = A) \tag{4.2}$$

$$\frac{E[x_i : A] \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [\ell_i = \varsigma(x_i)b_i{}^{\,i\in 1..n}] : A} \quad (\text{where } A = [\ell_i : B_i{}^{\,i\in 1..n}]^0) \tag{4.3}$$

$$\frac{E \vdash a : A}{E \vdash a.\ell : B} \quad (\text{where } A \leq [\ell : B]^{\rightarrow}) \tag{4.4}$$

$$\frac{E \vdash a : A \quad E[x : A] \vdash b : B}{E \vdash a.\ell \Leftarrow \varsigma(x)b : A} \quad (\text{where } A \leq [\ell : B]^{\rightarrow}) \tag{4.5}$$

$$\frac{E \vdash a_1 : A_1 \quad E \vdash a_2 : A_2}{E \vdash a_1 + a_2 : A_1 \oplus A_2} \tag{4.6}$$

$$\frac{E \vdash a : A}{E \vdash a : B} \quad (\text{where } A \leq B) \tag{4.7}$$

The first five rules express the typing of each of the four constructs in the object calculus and the last rule is the rule of subsumption. We say that a term a is *well-typed* if $E \vdash a : A$ is derivable for some E and A. The following result can be proved by a well-known technique [Nie89, WF94].

Theorem 4.3.1. (Type Soundness) *Well-typed programs cannot go wrong.*

The type inference problem for our extension of the Abadi-Cardelli calculus is: given a term a, find a type environment E and a type A such that $E \vdash a : A$, or decide that this is impossible.

4.4 From Type Inference to Constraint Solving

A *substitution* S is a finite partial function from type variables to types in $\mathcal{T}_{\mathrm{reg}}(\Sigma)$, written $\{U_1 := A_1, \ldots, U_n := A_n\}$. The set $\{U_1, \ldots, U_n\}$ is called the *domain* of the substitution. We identify substitutions with their graphs, and write $(S_1 \cup S_2)$ for the union of two substitutions S_1 and S_2; by convention, we assume that S_1 and S_2 agree on variables in their common domain, so $(S_1 \cup S_2)$ is a substitution. Substitutions are extended to total functions from types to types in the usual way.

Definition 4.4.1. A relation R is solvable if and only if there is a substitution S such that for all $(A, B) \in R$, we have $S(A) \leq S(B)$. □

Definition 4.4.2. We will here focus on so-called C-relations (which we also refer to as *constraint sets*) which contain only pairs (A, B), where A, B are of the forms

- $[\ell : V, \ldots]^{\phi}$,
- V, or
- $V_1 \oplus V_2$,

where V, V_1, V_2 are type variables, and $\phi \in \{0. \rightarrow\}$. □

While $V_1 \oplus V_2$ is not a type, it will become a type once we apply a substitution and get $S(V_1) \oplus S(V_2)$, provided the concatenation is defined. Note that if $V_1 \oplus V_2$ is in R, and R is solvable, then the solution, say S, must make $S(V_1) \oplus S(V_2)$ well-defined. To avoid introducing special terminology for the left-hand sides and right-hand sides of constraints, we will abuse the word type and call $V_1 \oplus V_2$ a type in the remainder of the chapter.

We now prove that the type inference problem is equivalent to solving constraints in the form of C-relations.

We write $E' \leq E$ if, whenever $E(x) = A$, there is an $A' \leq A$ such that $E'(x) = A'$. The following standard result can be proved by induction on typings.

Lemma 4.4.3 (Weakening). *If $E \vdash c : C$ and $E' \leq E$, then $E' \vdash c : C$.*

By a simple induction on typing derivations, we obtain the following syntax-directed characterization of typings. The proof uses only the reflexivity and transitivity of $\leq$ which can be derived from Lemma 4.2.2.

Lemma 4.4.4 (Characterization of Typings). *$E \vdash c : C$ if and only if one of the following cases holds:*

- $c = x$ *and* $E(x) \leq C$*;*
- $c = a.\ell$*, and for some* A *and* B*,* $E \vdash a : A$*,* $A \leq [\ell : B]^{\rightarrow}$*, and* $B \leq C$*;*
- $c = [\ell_i = \varsigma(x_i)b_i{}^{i\in 1..n}]$*, and for some* A*, and some* B_i *for* $i \in 1..n$*,* $E[x_i : A] \vdash b_i : B_i$*, and* $A = [\ell_i : B_i{}^{i\in 1..n}]^0 \leq C$*; or*
- $c = (a.\ell \Leftarrow \varsigma(x)b)$*, and for some* A *and* B*,* $E \vdash a : A$*,* $E[x : A] \vdash b : B$*,* $A \leq [\ell : B]^{\rightarrow}$*, and* $A \leq C$*.*
- $c = a_1 + a_2$*, and for some* A_1, A_2*,* $E \vdash a_1 : A_1$*,* $E \vdash a_2 : A_2$*, and* $A_1 \oplus A_2 \leq C$*.*

We now show how to generate a C-relation from a given program.

Definition 4.4.5. Let c be a ς-term in which all free and bound variables are pairwise distinct. We define X_c, Y_c, E_c, and $\mathcal{C}(c)$ as follows.

- X_c is a set of fresh type variables. It consists of a type variable U_x for every term variable x appearing in c.
- Y_c is a set of fresh type variables. It consists of a type variable $V_{c'}$ for each occurrence of a subterm c' of c, and a type variable $U_{c'}$ for each occurrence of a select subterm $c' = a.\ell$ of c. (If c' occurs more than once in c, then $U_{c'}$ and $V_{c'}$ are ambiguous. However, it will always be clear from context which occurrence is meant.)
- E_c is a type environment, defined by

$$E_c = \{x : U_x \mid x \text{ is free in } c\}.$$

- $\mathcal{C}(c)$ is the set of the following constraints over X_c and Y_c:
  - For each occurrence in c of a variable x, the constraint
  $$(U_x, V_x). \tag{4.8}$$
  - For each occurrence in c of a subterm of the form $a.\ell$, the two constraints
  $$(V_a, [\ell : U_{a.\ell}]^{\rightarrow}) \tag{4.9}$$
  $$(U_{a.\ell}, V_{a.\ell}). \tag{4.10}$$
  - For each occurrence in c of a subterm of the form $[\ell_i = \varsigma(x_i)b_i{}^{\,i\in 1..n}]$, the constraint
  $$([\ell_i : V_{b_i}{}^{\,i\in 1..n}]^0, V_{[\ell_i=\varsigma(x_i)b_i{}^{\,i\in 1..n}]}) \tag{4.11}$$
  and for each $j \in 1..n$, the constraints
  $$U_{x_j} \equiv [\ell_i : V_{b_i}{}^{\,i\in 1..n}]^0. \tag{4.12}$$
  - For each occurrence in c of a subterm of the form $(a.\ell \Leftarrow \varsigma(x)b)$, the constraints
  $$(V_a, V_{(a.\ell\Leftarrow\varsigma(x)b)}) \tag{4.13}$$
  $$V_a \equiv U_x \tag{4.14}$$
  $$(V_a, [\ell : V_b]^{\rightarrow}). \tag{4.15}$$
  - For each occurrence in c of a subterm of the form $(a_1 + a_2)$, the constraint
  $$(V_{a_1} \oplus V_{a_2}, V_{(a_1+a_2)}), \tag{4.16}$$

□

In the definition of $\mathcal{C}(c)$, each equality $A \equiv B$ denotes the two inequalities (A, B) and (B, A).

Theorem 4.4.6. *$E \vdash c : C$ if and only if there is a solution S of $\mathcal{C}(c)$ such that $S(V_c) = C$ and $S(E_c) \subseteq E$.*

Each direction of the theorem can be proved separately. However, the proofs share a common structure, so for brevity we will prove them together. The two directions follow immediately from the two parts of the next lemma.

Lemma 4.4.7. *Let c_0 be a ς-term. For every subterm c of c_0,*

i) *if $E \vdash c : C$, then there is a solution S_c of $\mathcal{C}(c)$ such that $S_c(V_c) = C$ and $S_c(E_c) \subseteq E$; and*

ii) *if S is a solution of $\mathcal{C}(c_0)$, then $S(E_c) \vdash c : S(V_c)$.*

Proof. The proof is by induction on the structure of c. In (2), we will often use the fact that any solution to $\mathcal{C}(c_0)$ (in particular, S) is a solution to $\mathcal{C}(c) \subseteq \mathcal{C}(c_0)$.

- If $c = x$, then $E_c = \{x : U_x\}$ and $\mathcal{C}(c) = \{(U_x, V_x)\}$.

  i) Define $S_c = \{U_x := E(x), V_x := C\}$. Then $S_c(V_c) = S_c(V_x) = C$, and $S_c(E_c) = \{x : E(x)\} \subseteq E$.

  Furthermore, by Lemma 4.4.4, $E(x) \leq C$, so S_c is a solution to $\mathcal{C}(c)$.

  ii) By (4.2), $S(E_c) \vdash c : S(U_x)$.

  And since $S(U_x) \leq S(V_x) = S(V_c)$, we have $S(E_c) \vdash c : S(V_c)$ by (4.7).

- If $c = a.\ell$, then $E_c = E_a$ and $\mathcal{C}(c) = \mathcal{C}(a) \cup \{(V_a, [\ell : U_{a.\ell}]^{\rightarrow}),\ (U_{a.\ell}, V_{a.\ell})\}$.

  i) By Lemma 4.4.4, for some A and B, $E \vdash a : A$, $A \leq [\ell : B]^{\rightarrow}$, and $B \leq C$.

  By induction there is a solution S_a of $\mathcal{C}(a)$ such that $S_a(V_a) = A$ and $S_a(E_a) \subseteq E$.

  Define $S_c = S_a \cup \{U_{a.\ell} := B, V_{a.\ell} := C\}$. Then S_c solves $\mathcal{C}(c)$, $S_c(V_c) = S_c(V_{a.\ell}) = C$, and $S_c(E_c) = S_a(E_a) \subseteq E$.

  ii) By induction, $S(E_a) \vdash a : S(V_a)$.

  Since $S(V_a) \leq S([\ell : U_{a.\ell}]^{\rightarrow})$, by (4.7) we have $S(E_a) \vdash a : S([\ell : U_{a.\ell}]^{\rightarrow})$.

  Then by (4.4), $S(E_a) \vdash a.\ell : S(U_{a.\ell})$.

Since $S(U_{a.\ell}) \leq S(V_{a.\ell}) = S(V_c)$, by (4.7) we have $S(E_a) \vdash a.\ell : S(V_c)$.

Finally, $E_c = E_a$ and $c = a.\ell$, so $S(E_c) \vdash c : S(V_c)$ as desired.

- If $c = [\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$, then $E_c = \bigcup_{i\in 1..n}(E_{b_i} \backslash x_i)$, and

$$\begin{aligned}\mathcal{C}(c) = \{\ ([\ell_i : V_{b_i} \ ^{i\in 1..n}]^0, V_c\ )\}\\ \cup \{\ U_{x_j} \equiv [\ell_i : V_{b_i} \ ^{i\in 1..n}]^0 \mid j \in 1..n\ \}\\ \cup (\textstyle\bigcup_{i\in 1..n} \mathcal{C}(b_i)).\end{aligned}$$

  i) By Lemma 4.4.4, for some A, and some B_i for $i \in 1..n$, we have $E[x_i : A] \vdash b_i : B_i$ and $A = [\ell_i : B_i \ ^{i\in 1..n}]^0 \leq C$.

  By induction, for every $i \in 1..n$ there is a substitution S_{b_i} such that S_{b_i} solves $\mathcal{C}(b_i)$, $S_{b_i}(V_{b_i}) = B_i$, and $S_{b_i}(E_{b_i}) \subseteq E[x_i : A]$.

  We first assume that the domain of any S_{b_i} is $X_{b_i} \cup Y_{b_i}$ (else restrict S_{b_i} to this set). Let $S_c = (\bigcup_{i\in 1..n} S_{b_i}) \cup \{V_c := C\}$

  Clearly, if S_c is well-defined, then it is a solution to $\mathcal{C}(c)$, $S_c(V_c) = C$, and $S_c(E_c) \subseteq E$.

  To show that S_c is well-defined, it suffices to show that for any distinct $j, k \in 1..n$, the substitutions S_{b_j} and S_{b_k} agree on all type variables in their common domain. And if U is in the domain of both S_{b_j} and S_{b_k}, it must have the form U_y for some term variable y free in both b_j and b_k.

  Then y must be assigned a type by E, so the conditions $S_{b_j}(E_{b_j}) \subseteq E[x_j : A]$ and $S_{b_k}(E_{b_k}) \subseteq E[x_k : A]$ guarantee that $S_{b_j}(U_y) = E(y) = S_{b_k}(U_y)$. Therefore S_c is well-defined, as desired.

  ii) By induction, $S(E_{b_j}) \vdash b_j : S(V_{b_j})$ for all $j \in 1..n$.

  By weakening, $S(E_c[x_j : U_{x_j}]) \vdash b_j : S(V_{b_j})$ for all $j \in 1..n$.

  Since S solves $\mathcal{C}(c)$, $S(U_{x_j}) = S([\ell_i : V_{b_i} \ ^{i\in 1..n}]^0)$ for all $j \in 1..n$.

  Then by (4.3), $S(E_c) \vdash c : S([\ell_i : V_{b_i} \ ^{i\in 1..n}]^0)$.

  Finally, since S solves $\mathcal{C}(c)$, $S([\ell_i : V_{b_i} \ ^{i\in 1..n}]^0) \leq S(V_c)$, so we have $S(E_c) \vdash c : S(V_c)$ by (4.7).

- If $c = (a.\ell \Leftarrow \varsigma(x)b)$, then $E_c = E_a \cup (E_b \backslash x)$, and

$$\mathcal{C}(c) = \mathcal{C}(a) \cup \mathcal{C}(b) \cup \{(V_a, V_c), V_a \equiv U_x, (V_a, [\ell : V_b]^{\rightarrow})\}.$$

  i) By Lemma 4.4.4, for some A and B, $E \vdash a : A$, $E[x : A] \vdash b : B$, $A \leq [\ell : B]^{\rightarrow}$, and $A \leq C$.

  By induction there is a solution S_a of $\mathcal{C}(a)$ such that $S_a(V_a) = A$ and $S_a(E_a) \subseteq E$, and a solution S_b of $\mathcal{C}(b)$ such that $S_b(V_b) = B$ and $S_b(E_b) \subseteq E[x : A]$.

  Let $S_c = S_a \cup S_b \cup \{V_c := C, U_x := A\}$. (We omit a proof that S_c is well-defined; this can be shown just as in the previous case.)

  Then S_c is a solution to $\mathcal{C}(c)$, $S_c(V_c) = C$, and $S_c(E_c) \subseteq E$.

  ii) Since S solves $\mathcal{C}(c)$, $S(V_a) \leq S[l : V_b]^{\rightarrow}$. By induction $S(E_a) \vdash a : S(V_a)$ and $S(E_b) \vdash b : S(V_b)$.

  By weakening, $S(E_c) \vdash a : S(V_a)$ and $S(E_c[x : U_x]) \vdash b : S(V_b)$.

  Then by (4.5), $S(E_c) \vdash c : S(V_a)$, and by (4.7), $S(E_c) \vdash c : S(V_c)$.

- If $c = (a_1 + a_2)$, then $E_c = E_{a_1} \cup E_{a_2}$ and

$$\mathcal{C}(c) = \mathcal{C}(a_1) \cup \mathcal{C}(a_2) \cup \{(V_{a_1} \oplus V_{a_2}, V_c)\}.$$

  i) By Lemma 4.4.4, for some A_1 and A_2, $E \vdash a_1 : A_1$, $E \vdash a_2 : A_2$, and $A_1 \oplus A_2 \leq C$.

  By induction there is a solution S_{a_i} of $\mathcal{C}(a_i)$ such that $S_{a_i}(V_{a_i}) = A_i$, and $S_{a_i}(E_{a_i}) \subseteq E$, for $i = 1, 2$.

  Let $S_c = S_{a_1} \cup S_{a_2} \cup \{V_c := C\}$. (We omit a proof that S_c is well-typed; this can be shown as above.) Then S_c is a solution to $\mathcal{C}(c)$, $S_c(V_c) = C$, and $S_c(E_c) \subseteq E$.

  ii) By induction $S(E_{a_1}) \vdash a_1 : S(V_{a_1})$ and $S(E_{a_2}) \vdash a_2 : S(V_{a_2})$.

  By weakening, $S(E_c) \vdash a_1 : S(V_{a_1})$ and $S(E_c) \vdash a_2 : S(V_{a_2})$.

  Then by (4.6), $S(E_c) \vdash c : S(V_{a_1}) \oplus S(V_{a_2})$, and by (4.7), $S(E_c) \vdash c : S(V_c)$.

□

4.5 Solving Constraints

In this section we present an algorithm for deciding whether a C-relation R is solvable. We first list the terminology used in the later definitions.

$$\begin{aligned} \mathsf{Types} &= \text{the set of types} \\ \mathsf{States} &= \mathrm{P}(\mathsf{Types}) \\ \mathsf{RelTypes} &= \mathrm{P}(\mathsf{Types} \times \mathsf{Types}) \\ \mathsf{RelStates} &= \mathrm{P}(\mathsf{States} \times \mathsf{States}) \end{aligned}$$

We use $\mathcal{T}$ to range over sets of types. For any type A such that $A(\epsilon) = (S, \phi)$, we write $labs(A) = S$. For any type A and label ℓ, $A.\ell$ is B if $A = [\ell : B \ldots]^{\phi}$, and is undefined otherwise. Notice that $A(\ell\alpha) = (A.\ell)(\alpha)$. We also make the following definitions.

$$\begin{aligned} \mathcal{T}.\ell &= \{B \mid \exists A \in \mathcal{T}.\ A = [\ell : B, \ldots]^{\phi}\}. \\ \mathrm{above}_R(\mathcal{T}) &= \{B \mid \exists A \in \mathcal{T}.\ (A, B) \in R\}. \\ \mathrm{ABOVE}_R(R') &= \{(\mathrm{above}_R(\{A\}), \mathrm{above}_R(\{B\})) \mid (A, B) \in R'\} \end{aligned}$$

We define function Var_R such that

- if type A is of the form $[\ldots]^{\phi}$, then $\mathrm{Var}_R(A) = \phi$;
- $\mathrm{Var}_R(V \oplus V') = 0$;
- if $V \oplus V'$ or $V' \oplus V$ is in R, then $\mathrm{Var}_R(V) = 0$; and
- $\mathrm{Var}_R(\mathcal{T}) = \sqcap\{\mathrm{Var}_R(A) \mid A \in \mathcal{T}\}$,

where $\sqcap$ is the greatest lower bound of a nonempty set of variances; $\sqcap\emptyset$ is undefined.

The types of the above definitions are

$$\begin{aligned} \mathcal{T}.\ell &: \mathsf{States} \to \mathsf{States} \\ \mathrm{above}_R &: \mathsf{States} \to \mathsf{States} \\ \mathrm{ABOVE}_R &: \mathsf{RelTypes} \to \mathsf{RelStates} \\ \mathrm{Var}_R &: \mathsf{States} \to \mathsf{Variances} \end{aligned}$$

For any set $\mathcal{T}$ of types we define $\mathrm{LV} : \mathsf{States} \to \mathcal{P}(\mathsf{Labels})$, the labels implied by $\mathcal{T}$, by

$$\mathrm{LV}(\mathcal{T}) = \bigcup_{A \in \mathcal{T}} labs(A(\epsilon))$$

In the rest of the section, we first define the notions of satisfaction-closure (Section 4.5.1) and satisfaction-consistency (Section 4.5.2), and we then prove that a C-relation R is solvable if and only if there exists a satisfaction-closed superset which is satisfaction-consistent (Theorem 4.5.15).

4.5.1 Satisfaction-closure

Definition 4.5.1. A C-relation R on types is satisfaction-closed (abbreviated sat-closed) if and only if the following are true:

0 if type A of the form $[\ell : U, \ldots]^{\phi}$ is in R, then $(A, [\ell : U]^{\rightarrow}) \in R$.

A if $(A, B), (B, C) \in R$, then $(A, C) \in R$;

B if $(A, B) \in R$, then $(A, A), (B, B) \in R$;

C if $(A, B) \in R$, and $\mathrm{Var}_R(B) = 0$, then $(B, A) \in R$;

D if $(A, [\ell : U]^{\rightarrow}), (A, [\ell : U']^{\rightarrow}) \in R$, then $(U, U') \in R$;

E if $(V, [\ell : U]^{\rightarrow}) \in R$ and $V \oplus V'$ is in R, then $(V \oplus V', [\ell : U]^{\rightarrow}) \in R$.

F for all $(V \oplus V', [\ell : U]^{\rightarrow}) \in R$, we have either $(V, [\ell : U]^{\rightarrow})$ or $(V', [\ell : U]^{\rightarrow})$ in R.

□

Notice that rule **D** is symmetric in the two hypotheses.

Lemma 4.5.2. *For every solvable C-relation R, there exists a solvable, sat-closed superset R' of R.*

Proof. For a substitution S, define a function

$$G_S : \mathsf{RelTypes} \to \mathsf{RelTypes} \tag{4.17}$$

$$G_S(R) = R \tag{4.18}$$

$$\cup \{ (A, [\ell : U]^{\rightarrow}) \mid \text{type } A \text{ of the form } [\ell : U, \ldots]^{\phi} \text{ is in } R \} \tag{4.19}$$

$$\cup \{ (A, C) \mid (A, B), (B, C) \in R \} \tag{4.20}$$

$$\cup \{ (A, A), (B, B) \mid (A, B) \in R \} \tag{4.21}$$

$$\cup \{ (B, A) \mid (A, B) \in R \wedge \text{Var}_R(B) = 0 \} \tag{4.22}$$

$$\cup \{ (U, U') \mid (A, [\ell : U]^{\rightarrow}), (A, [\ell : U']^{\rightarrow}) \in R \} \tag{4.23}$$

$$\cup \{ (V \oplus V', [\ell : U]^{\rightarrow}) \mid (V, [\ell : U]^{\rightarrow}) \in R \ \wedge \ V \oplus V' \text{ is in } R \} \tag{4.24}$$

$$\cup \{ (V, [\ell : U]^{\rightarrow}) \mid (V \oplus V', [\ell : U]^{\rightarrow}) \in R \ \wedge \ S(V) \text{ has an } \ell\text{-field} \} \tag{4.25}$$

$$\cup \{ (V', [\ell : U]^{\rightarrow}) \mid (V \oplus V', [\ell : U]^{\rightarrow}) \in R \ \wedge \ S(V') \text{ has an } \ell\text{-field} \} \tag{4.26}$$

Given a C-relation R with solution S, define R' as follows:

$$R' \ = \ \bigcup_{n=0}^{\infty} G_S^n(R).$$

It is straightforward to show that $R \subseteq R'$ and that R' is sat-closed. It remains to be shown that R' is solvable. It is sufficient to show that $G_S^n(R)$ has solution S, for all n. We proceed by induction on n. In the base of $n = 0$, we have $G_S^0(R) = R$ and that R has solution S by assumption.

In the induction step, suppose $G_S^n(R)$ has solution S. We will now show that $G_S^{n+1}(R) = G_S(G_S^n(R))$ has solution S. We proceed by case analysis on the definition of G_S.

Let $R_n = G_S^n(R)$ and $R_{n+1} = G_S^{n+1}(R)$. We have from the definition of G_S that the constraints in $R_{n+1} \backslash R_n$ belongs to the union of the sets (4.19) to (4.26). For each of the sets, we need to show that the constraints in it preserve that S is a solution. In each case, S is preserved because:

(4.19) Straightforward from the definition of $\leq$.

(4.20) If $(A, B), (B, C) \in R_n$, then by induction hypothesis, we have $S(A) \leq S(B) \leq S(C)$ and since the $\leq$ is transitive, we have $S(A) \leq S(C)$. Hence, S is a solution to $\{(A, C)\}$.

(4.21) Since the $\leq$ is reflexive, we have $S(A) \leq S(A)$ and $S(B) \leq S(B)$. Hence, S is a solution to $\{(A, A), (B, B)\}$.

(4.22) If $(A, B) \in R_n$ and $\text{Var}_{R_n}(B) = 0$, then by induction hypothesis, $S(A) \leq S(B)$ and by definition of $\leq$, we have $S(A) = S(B)$ as well, which implies $S(B) \leq S(A)$. Hence, S is a solution to $\{(B, A)\}$.

(4.23) If $(A, [\ell : U]^{\rightarrow}), (A, [\ell : U']^{\rightarrow}) \in R_n$, then by induction hypothesis, $S(A) \leq S([\ell : U]^{\rightarrow})$ and $S(A) \leq S([\ell : U']^{\rightarrow})$. By definition of $\leq$, $\exists B$, such that $S(A) = [\ell : B, \ldots]^{\phi}$ and $B = S(U) = S(U')$, which implies $S(U) \leq S(U')$. Hence, S is a solution to $\{(U, U')\}$.

(4.24) If $(V, [\ell : U]^{\rightarrow}) \in R_n$, then by induction hypothesis, $S(V) \leq S([\ell : U]^{\rightarrow})$. From the definition of $V \oplus V'$, we have $S(V \oplus V').\ell_i = S(V).\ell_i, \forall \ell_i \in \text{LV}(S(V))$. Since $S(V) \leq S([\ell : U]^{\rightarrow})$, we have $S(V \oplus V') \leq S([\ell : U]^{\rightarrow})$. Hence, S is a solution to $\{(V \oplus V', [\ell : U]^{\rightarrow})\}$.

(4.25) Since $\ell \in \text{LV}(S(V))$, there exists B such that $S(V) = [\ell : B, \ldots]^0$. By definition of $\leq$ and $S(V) \oplus S(V') \leq [\ell : S(U)]^{\rightarrow}$, we have that $B = S(U)$ and $S(V) \leq [\ell : S(U)]^{\rightarrow}$. Therefore, S is a solution to $\{(V, [\ell : U]^{\rightarrow})\}$.

(4.26) The proof is similar to the previous case.

□

4.5.2 Satisfaction-consistency

Definition 4.5.3. A C-relation R on types is satisfaction-consistent (abbreviated sat-consistent) if and only if the following are true:

i) if $([\ell_i : U_i \ ^{i \in I}]^{\phi}, [\ell_i : U'_i \ ^{i \in I'}]^{\phi'}) \in R$, then $I \supseteq I'$ and $\phi \sqsubseteq \phi'$;

ii) if $([\ell : U, \ldots]^{\phi}, V) \in R$, and $V \oplus V'$ is in R, then $\phi = 0$;

iii) if $V \oplus V'$ is in R, then $\text{LV}(\text{above}_R(\{V\})) \cap \text{LV}(\text{above}_R(\{V'\})) = \emptyset$;

□

Lemma 4.5.4. *If a C-relation R is solvable, then R is sat-consistent.*

Proof. Immediate. □

4.5.3 Main Result

In this section, we will show that if a C-relation is sat-closed and sat-consistent, then it is solvable.

For a C-relation R we build an automaton with states consisting of sets of types appearing in R, and the following one-step transition function:

$$\delta_R(\mathcal{T})(\ell) = \begin{cases} \text{above}_R(\mathcal{T}.\ell) & \text{if } \mathcal{T}.\ell \neq \emptyset \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We write States(R) for the set of states of the automaton, and use g, h to range over states.

The one-step transition function is extended to a many-step transition function in the usual way.

$$\begin{aligned} \delta^*_R(g)(\epsilon) &= g, \\ \delta^*_R(g)(\ell\alpha) &= \delta^*_R(\delta_R(g)(\ell))(\alpha). \end{aligned}$$

Any g defines a type, $\text{Type}_R(g)$, and any relation $\mathcal{R}$ on States(R) defines a constraint set on types $\text{TYPE}_R(\mathcal{R})$, as follows:

$$\begin{aligned} \text{Type}_R(g)(\alpha) &= (\text{LV}, \text{Var}_R)(\delta^*_R(g)(\alpha)), \\ \text{TYPE}_R(\mathcal{R}) &= \{(\text{Type}_R(g), \text{Type}_R(h)) \mid (g,h) \in \mathcal{R}\} \end{aligned}$$

Notice that we use $(\text{LV}, \text{Var}_R)(g)$ to denote $(\text{LV}(g), \text{Var}_R(g))$. We have that

$$\begin{aligned} \text{Type}_R &: \mathsf{States} \to \mathsf{Types} \\ \text{TYPE}_R &: \mathsf{RelStates} \to \mathsf{RelTypes} \end{aligned}$$

Lemma 4.5.5. *If $g = \delta_R(g')(\ell)$, then $\text{Type}_R(g) = \text{Type}_R(g').\ell$.*

Proof.

$$(\text{Type}_R(g').\ell)(\alpha) = \text{Type}_R(g')(\ell\alpha)$$

$$\begin{aligned} &= (\mathrm{LV}, \mathrm{Var}_R)(\delta_R^*(g')(\ell\alpha)) \\ &= (\mathrm{LV}, \mathrm{Var}_R)(\delta_R^*(\delta_R(g')(\ell))(\alpha)) \\ &= (\mathrm{LV}, \mathrm{Var}_R)(\delta_R^*(g)(\alpha)) \\ &= \mathrm{Type}_R(g)(\alpha). \end{aligned}$$

□

Definition 4.5.6. For any C-relation R on types, we define S_R to be the least substitution such that for every U appearing in R we have

$$S_R(U) = \mathrm{Type}_R(\mathrm{above}_R(\{U\})).$$

Note that if $A = [\ell : U, \ldots]^\phi$, then $S_R(A) = [\ell : S_R(U), \ldots]^\phi$. □

We claim that if R is sat-closed and sat-consistent, then S_R is a solution to R.

To prove this claim, the first step is to develop a connection between subtype-closure and δ. Define the function $\mathcal{A} : \mathsf{RelTypes} \to \mathsf{RelTypes}$ by $(A, B) \in \mathcal{A}(R)$ if and only if one of the following conditions holds:

- $(A, B) \in R$.
- For some ℓ, ϕ, and ϕ', we have $([\ell : A, \ldots]^\phi, [\ell : B, \ldots]^{\phi'}) \in R$, or $([\ell : B, \ldots]^{\phi'}, [\ell : A, \ldots]^\phi) \in R$.

Note, the subtype-closure (Definition 4.2.4) of a C-relation R is the least fixed point of $\mathcal{A}$ containing R.

Define the function $\mathcal{B}_R : \mathsf{RelStates} \to \mathsf{RelStates}$ by $(g, h) \in \mathcal{B}_R(\mathcal{R})$, where $g, h \neq \emptyset$, if and only if one of the following conditions holds:

- $(g, h) \in \mathcal{R}$.
- For some ℓ and (g', h') or $(h', g') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$, $h = \delta_R(h')(\ell)$.

The next four lemmas (Lemma 4.5.7, 4.5.8, 4.5.10, and 4.5.11) are key ingredients in the proof of Lemma 4.5.12. Lemma 4.5.7 states the fundamental relationship between TYPE_R, $\mathcal{A}$, and $\mathcal{B}_R$. We will use the notation

$$f \circ g(x) = f(g(x)).$$

Lemma 4.5.7. *The following diagram commutes:*

$$\begin{array}{ccc} \mathsf{RelStates} & \xrightarrow{\mathrm{TYPE}_R} & \mathsf{RelTypes} \\ \downarrow{\scriptstyle \mathcal{B}_R} & & \downarrow{\scriptstyle \mathcal{A}} \\ \mathsf{RelStates} & \xrightarrow{\mathrm{TYPE}_R} & \mathsf{RelTypes} \end{array}$$

Proof. Suppose $\mathcal{R} \in \mathsf{RelStates}$. To prove $\mathrm{TYPE}_R \circ \mathcal{B}_R \subseteq \mathcal{A} \circ \mathrm{TYPE}_R$, suppose $(A, B) \in \mathrm{TYPE}_R \circ \mathcal{B}_R(\mathcal{R})$. There must be a pair of states $(g, h) \in \mathcal{B}_R(\mathcal{R})$ such that $A = \mathrm{Type}_R(g)$ and $B = \mathrm{Type}_R(h)$. We reason by cases on how $(g, h) \in \mathcal{B}_R(\mathcal{R})$. From the definition of $\mathcal{B}_R$ we have that there are three cases.

i) suppose $(g, h) \in \mathcal{R}$. We have $(\mathrm{Type}_R(g), \mathrm{Type}_R(h)) \in \mathrm{TYPE}_R(\mathcal{R})$, so from the definition of $\mathcal{A}$ we have $(\mathrm{Type}_R(g), \mathrm{Type}_R(h)) \in \mathcal{A} \circ \mathrm{TYPE}_R(\mathcal{R})$.

ii) suppose for some ℓ and $(g', h') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. From $(g', h') \in \mathcal{R}$, we have $(\mathrm{Type}_R(g'), \mathrm{Type}_R(h')) \in \mathrm{TYPE}_R(\mathcal{R})$. We have, from Lemma 4.5.5,

$$(\mathrm{Type}_R(g').\ell)(\alpha) = \mathrm{Type}_R(g)(\alpha) = A(\alpha),$$

so $\mathrm{Type}_R(g').\ell = A$. Similarly, $\mathrm{Type}_R(h').\ell = B$. From these two observations, and $(\mathrm{Type}_R(g'), \mathrm{Type}_R(h')) \in \mathrm{TYPE}_R(\mathcal{R})$, and the definition of $\mathcal{A}$, we conclude $(A, B) \in \mathcal{A} \circ \mathrm{TYPE}_R(\mathcal{R})$.

iii) Suppose for some ℓ and $(h', g') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. The proof is similar to the previous case.

To prove $\mathcal{A} \circ \mathrm{TYPE}_R \subseteq \mathrm{TYPE}_R \circ \mathcal{B}_R$, suppose $(A, B) \in \mathcal{A} \circ \mathrm{TYPE}_R(\mathcal{R})$. We reason by cases on how $(A, B) \in \mathcal{A} \circ \mathrm{TYPE}_R(\mathcal{R})$. From the definition of $\mathcal{A}$ we have that there are three cases.

i) suppose $(A, B) \in \mathrm{TYPE}_R(\mathcal{R})$. There must exist g and h such that $A = \mathrm{Type}_R(g)$, $B = \mathrm{Type}_R(h)$, and $(g, h) \in \mathcal{R}$. From $(g, h) \in \mathcal{R}$ and the definition of $\mathcal{B}_R$, we have that $(g, h) \in \mathcal{B}_R(\mathcal{R})$, so $(A, B) \in \mathrm{TYPE}_R \circ \mathcal{B}_R$.

ii) suppose for some ℓ, ϕ, ϕ', we have $([\ell : A, \ldots]^{\phi}, [\ell : B, \ldots]^{\phi'}) \in \mathrm{TYPE}_R(\mathcal{R})$. There must exist g' and h' such that $\mathrm{Type}_R(g') = [\ell : A, \ldots]^{\phi}$, $\mathrm{Type}_R(h') = [\ell : B, \ldots]^{\phi'}$, and $(g', h') \in \mathcal{R}$. Then $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$ are well defined, and $(g, h) \in \mathcal{B}_R(\mathcal{R})$ by the definition of $\mathcal{B}_R$. From $\mathrm{Type}_R(g') = [\ell : A, \ldots]^{\phi}$, $g = \delta_R(g')(\ell)$, and Lemma 4.5.5, we have $\mathrm{Type}_R(g) = \mathrm{Type}_R(g').\ell = A$. Similarly, $\mathrm{Type}_R(h) = B$, so $(A, B) \in \mathrm{TYPE}_R \circ \mathcal{B}_R(\mathcal{R})$ as desired.

iii) Suppose for some ℓ and $(h', g') \in \mathcal{R}$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. The proof is similar to the previous case.

□

Lemma 4.5.8. *Suppose R is sat-closed. If $(g, h) \in \mathrm{ABOVE}_R(R)$, then $g \supseteq h$.*

Proof. Suppose $(g, h) \in \mathrm{ABOVE}_R(R)$. From the definition of ABOVE_R we have that we can choose A, B such that $(A, B) \in R$, $g = \mathrm{above}_R(\{A\})$, and $h = \mathrm{above}_R(\{B\})$. To prove $g \supseteq h$, suppose $C \in h$. We have $(B, C), (A, B) \in R$. Since R is sat-closed and by closure Rule **A**, we have $(A, C) \in R$ and $C \in g$. Hence, $g \supseteq h$. □

The following lemma reflects that $\leq$ does not support depth subtyping. As a consequence, we have designed the sat-closure rules such that, intuitively, if $(A', B') \in R$ and R is sat-closed, then the types constructed from $\{A'\}$ and $\{B'\}$ have the same ℓ field type.

Lemma 4.5.9. *If R is sat-closed, $(A', B') \in R$, and $\mathrm{above}_R(\mathrm{above}_R(\{B'\}).\ell) \neq \emptyset$, then $\mathrm{above}_R(\mathrm{above}_R(\{A'\}).\ell) = \mathrm{above}_R(\mathrm{above}_R(\{B'\}).\ell)$.*

Proof. From $(A', B') \in R$ and Lemma 4.5.8, we have $\mathrm{above}_R(\{A'\}) \supseteq \mathrm{above}_R(\{B'\})$, so $\mathrm{above}_R(\mathrm{above}_R(\{A'\}).\ell) \supseteq \mathrm{above}_R(\mathrm{above}_R(\{B'\}).\ell)$.

To prove $\mathrm{above}_R(\mathrm{above}_R(\{A'\}).\ell) \subseteq \mathrm{above}_R(\mathrm{above}_R(\{B'\}).\ell)$, suppose $A \in \mathrm{above}_R(\mathrm{above}_R(\{A'\}).$ So, there exists $[\ell : U_1, \ldots]^{\phi_1}$ such that

$$\begin{aligned}(A', [\ell : U_1, \ldots]^{\phi_1}) &\in R \\ (U_1, A) &\in R.\end{aligned}$$

From $\mathrm{above}_R(\mathrm{above}_R(\{B'\}).\ell) \neq \emptyset$, we have $B \in \mathrm{above}_R(\mathrm{above}_R(\{B'\}).\ell)$. So, there exists $[\ell : U_2, \ldots]^{\phi_2}$ such that

$$\begin{aligned}(B', [\ell : U_2, \ldots]^{\phi_2}) &\in R \\ (U_2, B) &\in R.\end{aligned}$$

From $(A', B'), (B', [\ell : U_2, \ldots]^{\phi_2}) \in R$, and closure rule A (transitivity), we have $(A', [\ell : U_2, \ldots]^{\phi_2}) \in R$. From

$$\begin{aligned}(A', [\ell : U_1, \ldots]^{\phi_1}) &\in R \\ (A', [\ell : U_2, \ldots]^{\phi_2}) &\in R,\end{aligned}$$

and closure rule 0,A,D, we have $(U_2, U_1) \in R$. From $(U_2, U_1), (U_1, A) \in R$ and closure rule A (transitivity), we have $(U_2, A) \in R$. From

$$\begin{aligned}(B', [\ell : U_2, \ldots]^{\phi_2}) &\in R \\ (U_2, A) &\in R,\end{aligned}$$

we have $A \in \mathrm{above}_R(\mathrm{above}_R(\{B'\}).\ell)$.

□

Lemma 4.5.10. *If $(g, h) \in (\mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R))\backslash\mathrm{ABOVE}_R(R)$, then $g = h$, $\forall n \geq 1$, where R is sat-closed.*

Proof. We proceed by induction on n.

In the base case of $n = 1$, suppose $(g, h) \in (\mathcal{B}_R^1 \circ \mathrm{ABOVE}_R(R))\backslash\mathrm{ABOVE}_R(R)$. From the definition of $\mathcal{B}_R$, there are two cases.

- Suppose for some ℓ and $(g', h') \in \mathrm{ABOVE}_R(R)$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. By the definition of ABOVE_R, there exist types A', B' such that $g' = \mathrm{above}_R(\{A'\})$, $h' = \mathrm{above}_R(\{B'\})$, and $(A', B') \in R$. We have

$$\begin{aligned} \mathrm{above}_R(\mathrm{above}_R(\{B'\}).\ell) &= \delta_R(\mathrm{above}_R(\{B'\}))(\ell) \\ &= \delta_R(h')(\ell) \\ &= h, \end{aligned}$$

  and from $(g, h) \in (\mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R))$, and the definition of $\mathcal{B}_R$, we have $h \neq \emptyset$. From $(A', B') \in R$, $\mathrm{above}_R(\mathrm{above}_R(\{B'\}).\ell) \neq \emptyset$, and Lemma 4.5.9, we have

$$\begin{aligned} g &= \mathrm{above}_R(\mathrm{above}_R(\{A'\}).\ell) \\ &= \mathrm{above}_R(\mathrm{above}_R(\{B'\}).\ell) = h. \end{aligned}$$

- Suppose for some ℓ and $(h', g') \in \mathrm{ABOVE}_R(R)$, we have $g = \delta_R(g')(\ell)$ and $h = \delta_R(h')(\ell)$. The proof is similar as in the previous case.

In the induction step, suppose

$$(g, h) \in (\mathcal{B}_R^{n+1} \circ \mathrm{ABOVE}_R(R)) \backslash \mathrm{ABOVE}_R(R).$$

From the definition of $\mathcal{B}_R$, there exist ℓ such that
(g', h') or $(h', g') \in (\mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)) \backslash \mathrm{ABOVE}_R(R)$ and $g = \delta_R(g')(\ell), h = \delta_R(h')(\ell)$. From the induction hypothesis, we have $g' = h'$. From the definition of δ_R, it is immediate that $g = h$. □

Lemma 4.5.11. *Suppose R is sat-closed. If $(g, h) \in \mathrm{ABOVE}_R(R)$, then* $\mathrm{Var}_R(h) = 0 \Rightarrow \mathrm{LV}(g) = \mathrm{LV}(h)$.

Proof. Suppose $(g, h) \in \mathrm{ABOVE}_R(R)$. From the definition of ABOVE_R, $\exists A, B$ such that $g = \mathrm{above}_R(\{A\}), h = \mathrm{above}_R(\{B\})$ and $(A, B) \in R$. Therefore, $\forall A' \in g, B' \in h$, we have $(A, A'), (A, B') \in R$. Since $\mathrm{Var}_R(h) = 0$, there exists a type $B'' \in h$ such that $\mathrm{Var}_R(B'') = 0$. From closure rule **A**, we have that $\mathrm{LV}(\mathrm{above}_R\{A'\}) \subseteq \mathrm{LV}(\mathrm{above}_R\{A\})$;

and from closure rule **C**, we have that $\mathrm{LV}(\mathrm{above}_R\{A\}) \subseteq \mathrm{LV}(\mathrm{above}_R\{B''\})$. Hence, $\mathrm{LV}(g) \subseteq \mathrm{LV}(\mathrm{above}_R(\{B''\})) \subseteq \mathrm{LV}(h)$.

From Lemma 4.5.8, we have $g \supseteq h$ which implies that $\mathrm{LV}(g) \supseteq \mathrm{LV}(h)$. Therefore, $\mathrm{LV}(g) = \mathrm{LV}(h)$. □

Lemma 4.5.12. *If R is sat-closed, then the subtype-closure of $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$ is subtype-consistent.*

Proof.

The subtype-closure of $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$

$$= \bigcup_{0 \le n < \infty} \mathcal{A}^n \circ \mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R) \quad \text{(Definition of subtype-closure)}$$

$$= \bigcup_{0 \le n < \infty} \mathrm{TYPE}_R \circ \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R) \quad \text{(Lemma 4.5.7)}$$

$$= \bigcup_{0 \le n < \infty} \bigcup_{(g,h) \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)} \{(\mathrm{Type}_R(g), \mathrm{Type}_R(h))\} \text{ (Definition of } \mathrm{TYPE}_R).$$

Suppose $(g, h) \in \mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$. From Lemma 4.5.8 and Lemma 4.5.10, and a case analysis on why (g, h) is in $\mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$, we have that $g \supseteq h$. From Lemma 4.5.11 and Lemma 4.5.10, and a case analysis on why (g, h) is in $\mathcal{B}_R^n \circ \mathrm{ABOVE}_R(R)$, we have that $\mathrm{Var}_R(h) = 0 \Rightarrow \mathrm{LV}(g) = \mathrm{LV}(h)$. Thus, it is immediate from the definition of Type_R that $\{(\mathrm{Type}_R(g), \mathrm{Type}_R(h))\}$ is subtype-consistent.

Thus, the subtype-closure of $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$ is the union of a family of subtype-consistent C-relations. Since the union of a family of subtype-consistent C-relations is itself subtype-consistent, we conclude that the subtype-closure of $\mathrm{TYPE}_R \circ \mathrm{ABOVE}_R(R)$ is subtype-consistent. □

The following lemma is a key ingredient in the proof of Lemma 4.5.14. Lemma 4.5.14 is the place where it is needed that a relation is satisfaction-consistent.

Lemma 4.5.13. *If A of the form $[\ell : B, \ldots]^\phi$ is in R and R is sat-closed, then*

$$\mathrm{above}_R((\mathrm{above}_R(\{A\})).\ell) = \mathrm{above}_R(\{B\}).$$

Proof. To prove the direction $\supseteq$, notice that from sat-closure rule **B** and A appearing in R, we have $(A, A) \in R$, so $A \in \text{above}_R\{A\}$, hence $B \in (\text{above}_R(\{A\})).\ell$, and thus $\text{above}_R((\text{above}_R(\{A\})).\ell) \supseteq \text{above}_R(\{B\})$.

To prove the direction $\subseteq$, suppose $C \in \text{above}_R((\text{above}_R(\{A\})).\ell)$. From that we have there exists $C' \in (\text{above}_R(\{A\})).\ell$ such that $(C', C) \in R$. From $C' \in (\text{above}_R(\{A\})).\ell$ we have that there exists type D of the form $[\ell : C', \ldots]^{\phi'}$ such that $(A, D) \in R$. Together with closure rule **0**, **A**, **B**, and **D**, we have that $(B, C') \in R$. From transitivity of R (sat-closure rule **A**) and $(B, C'), (C', C) \in R$, we have $(B, C) \in R$, and $C \in \text{above}_R(\{B\})$. □

Lemma 4.5.14. *If R is sat-closed and sat-consistent, then*

i) *for any type A appearing in R, $S_R(A) = \text{Type}_R \circ \text{above}_R(\{A\})$; and*

ii) $S_R(R) = \text{TYPE}_R \circ \text{ABOVE}_R(R)$.

Proof. The second property is an immediate consequence of the first property.

To prove the first property, we will, by induction on α, show that for all α, for all A appearing in R, $S_R(A)(\alpha) = \text{Type}_R \circ \text{above}_R(\{A\})(\alpha)$.

If $\alpha = \epsilon$ and A is an ordinary type variable, the result follows by definition of S_R.

If $\alpha = \epsilon$ and A is of the form $V \oplus V'$, $S_R(V) = [\ell_i : B_i'\ ^{i \in I}]^0, S_R(V') = [\ell_i : B_i'\ ^{i \in I'}]^0$, and $\text{Type}_R \circ \text{above}_R(\{A\}) = [\ell_i : B_i\ ^{i \in J}]^0$, we need to show that $J = I \cup I'$, $B_i = B_i', \forall i \in J$, and $I \cap I' = \emptyset$. From R being sat-closed and closure rules **0**, **E**, we have $\text{LV}(\text{above}_R(\{V, V'\})) \subseteq \text{LV}(\text{above}_R(\{A\}))$. From R being sat-closed and closure rules **0**, **F**, we have $\text{LV}(\text{above}_R(\{A\})) \subseteq \text{LV}(\text{above}_R(\{V, V'\}))$. We conclude $\text{LV}(\text{above}_R(\{A\})) = \text{LV}(\text{above}_R(\{V, V'\}))$. Thus, $J = I \cup I'$ and by sat-consistency rule 3, we have $I \cap I' = \emptyset$. Because of closure rules **0**, **D**, **E**, and **F**, we have that $B_i = B_i', \forall i \in J$.

If $\alpha = \epsilon$ and $A = [\ell_i : B_i\ ^{i \in \{1..n\}}]^\phi$, then $S_R(A)(\alpha) = (\{\ell_i \mid i \in 1..n\}, \phi)$ and $\text{Type}_R \circ \text{above}_R(\{A\})(\alpha) = (\text{LV}(\text{above}_R(\{A\})), \phi)$. From closure rule **B** and A appearing in R, we have $(A, A) \in R$, so $A \in \text{above}_R(\{A\})$. From $A \in \text{above}_R(\{A\})$, we have $\text{LV}(\{A\}) \subseteq \text{LV}(\text{above}_R(\{A\}))$. From $A \in \text{above}_R(\{A\})$ and sat-consistency rules 1 and

2, we have $\mathrm{LV}(\mathrm{above}_R(\{A\})) \subseteq \mathrm{LV}(\{A\})$. We conclude $\mathrm{LV}(\{A\}) = \mathrm{LV}(\mathrm{above}_R(\{A\}))$. From the definition of Var_R, we have that $\mathrm{Var}_R(A) = \phi$. By sat-consistency rule 1, we have $\mathrm{Var}_R(\mathrm{above}_R(\{A\}) = \phi$, as desired.

If $\alpha = \ell\alpha'$ and A is a type variable, the result follows by definition of S_R.

If $\alpha = \ell\alpha'$ and A is of the form $V \oplus V'$, then either $S_R(V)$ or $S_R(V')$ has an ℓ field. Suppose it is $S_R(V)$ that has an ℓ field:

$$\begin{aligned}
S_R(A)(\alpha) &= (S_R(V) \oplus S_R(V'))(\alpha) && \text{(Definition of } S_R\text{)} \\
&= S_R(V)(\alpha) && (S_R(V) \text{ has an } \ell \text{ field)} \\
&= \mathrm{Type}_R \circ \mathrm{above}_R(\{V\})(\alpha) && \text{(Definition of } S_R\text{)} \\
&= \mathrm{Type}_R \circ \mathrm{above}_R(\{V, V'\})(\alpha) && (S_R(V') \text{ has no } \ell \text{ field)} \\
&= \mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\alpha). && \text{(from the proof of the base case)}
\end{aligned}$$

The case where it is $S_R(V')$ that has an ℓ field is similar, we omit the details.

If $\alpha = \ell\alpha'$ and $A = [\ell : B, \ldots]^\phi$, then

$$\begin{aligned}
& S_R(A)(\alpha) \\
&= S_R(B)(\alpha') && \text{(Definition of } S_R\text{)} \\
&= \mathrm{Type}_R \circ \mathrm{above}_R(\{B\})(\alpha') && \text{(Induction hypothesis)} \\
&= (\mathrm{LV}, \mathrm{Var}_R)(\delta_R^*(\mathrm{above}_R(\{B\}))(\alpha')) && \text{(Definition of } \mathrm{Type}_R\text{)} \\
&= (\mathrm{LV}, \mathrm{Var}_R)(\delta_R^*(\mathrm{above}_R((\mathrm{above}_R(\{A\})).\ell))(\alpha')) && \text{(Lemma 4.5.13)} \\
&= (\mathrm{LV}, \mathrm{Var}_R)(\delta_R^*(\delta_R(\mathrm{above}_R(\{A\}))(\ell))(\alpha')) && \text{(Definition of } \delta_R\text{)} \\
&= (\mathrm{LV}, \mathrm{Var}_R)(\delta_R^*(\mathrm{above}_R(\{A\}))(\ell\alpha')) && \text{(Definition of } \delta_R^*\text{)} \\
&= \mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\alpha) && \text{(Definition of } \mathrm{Type}_R \text{ and } \alpha = \ell\alpha'\text{)}.
\end{aligned}$$

If $\alpha = \ell\alpha'$ and A is a record without an ℓ field, then $S_R(A)(\alpha)$ is undefined. By sat-consistency rule 1, no $C \in \mathrm{above}_R(\{A\})$ has an ℓ field, so from the definition of Type_R we have that $\mathrm{Type}_R \circ \mathrm{above}_R(\{A\})(\ell\alpha')$ is undefined, as desired.

□

Theorem 4.5.15. *R is solvable if and only if there exists a sat-closed superset R' of R, such that R' is sat-consistent.*

Proof. If R is solvable, then we have from Lemma 4.5.2 that there exists solvable, sat-closed superset R' of R, so from Lemma 4.5.4, we have that R' is sat-consistent.

Conversely, let R' be a sat-closed superset of R, and assume that R' is sat-consistent. From Lemma 4.5.12 and Lemma 4.5.14, we have that the subtype-closure of $S_{R'}(R')$ is subtype-consistent. From the subtype-closure of $S_{R'}(R')$ being subtype-consistent and Lemma 4.2.6, we have $A \leq B$ for every $(A, B) \in S_{R'}(R')$, so $S_{R'}(A') \leq S_{R'}(B')$ for every $(A', B') \in R'$, and hence R' has solution $S_{R'}$. From $R \subseteq R'$ and that R' is solvable, we have that R is solvable. □

Theorem 4.5.16. *The type inference problem is in NP.*

Proof. From Theorem 4.4.6 we have the type inference problem is polynomial-time reducible to the constraint problem. To solve a constraint set R generated from a program a, we first guess a superset R' of R. Notice that we only need to consider an R' which has a size which is polynomial in the size of a. Next we check that R' is sat-closed and sat-consistent. It is straightforward to see that this can be done in polynomial time. □

4.6 NP-hardness

In this section we prove that the type inference problem is NP-hard. We do this in two steps. First we prove that solvability of so-called simple constraints can be reduced to the type inference problem, and then we prove that solving simple constraints is NP-hard.

4.6.1 From Constraints to Types

For any ς-term c, the the constraint set $\mathcal{C}(c)$ is defined as follows.

Definition 4.6.1. Given a denumerable set of variables, a *simple* constraint set is a finite set of constraints of the forms

$$(V \quad , \quad [\ell_i : V_i \ ^{i \in 1..n}]^0)$$
$$(V \oplus V' \quad , \quad [\ell_i : V_i \ ^{i \in 1..n}]^0)$$

where $V, V', V_1, \ldots, V_n$ are variables. □

Lemma 4.6.2. *Solvability of simple constraint sets is polynomial-time reducible to the type inference problem.*

Proof. Let $\mathcal{C}$ be a simple constraint set. Define

$$
\begin{array}{lll}
a^{\mathcal{C}} = \quad [\quad \ell_V & = \varsigma(x)(x.\ell_V) & \\
 & \qquad \text{for each variable } V \text{ in } \mathcal{C} & \\
\ell_Q & = \varsigma(x)[\ell_i = \varsigma(y)(x.\ell_{V_i})^{\ i\in 1..n}] & \\
 & \qquad \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i^{\ i\in 1..n}]^0 & \\
m_{Q,\ell_j} & = \varsigma(x)((x.\ell_{V_j} \Leftarrow \varsigma(y)(x.\ell_Q.\ell_j)).\ell_Q) & \\
 & \qquad \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i^{\ i\in 1..n}]^0 & \\
 & \qquad \text{and for each } j \in 1..n & \\
k_Q & = \varsigma(x)(x.\ell_Q + [\,]) & \\
 & \qquad \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i^{\ i\in 1..n}]^0 & \\
\ell_{(V,Q)} & = \varsigma(x)((x.\ell_Q \Leftarrow \varsigma(y)(x.\ell_V)).\ell_V) & \\
 & \qquad \text{for each constraint } (V, Q) \text{ in } \mathcal{C} & \\
 & \qquad \text{where } Q \text{ is of the form } [\ell_i : V_i^{\ i\in 1..n}]^0 & \\
\ell_{(V\oplus V',Q)} & = \varsigma(x)((x.\ell_Q \Leftarrow \varsigma(y)(x.\ell_V + x.\ell_{V'})).\ell_Q) & \\
 & \qquad \text{for each constraint } (V \oplus V', Q) \text{ in } \mathcal{C} & \\
 & \qquad \text{where } Q \text{ is of the forms } [\ell_i : V_i^{\ i\in 1..n}]^0 & \\
\quad] & &
\end{array}
$$

Notice that $a^{\mathcal{C}}$ can be generated in polynomial time.

We first prove that if $\mathcal{C}$ is solvable then $a^{\mathcal{C}}$ is typable. Suppose $\mathcal{C}$ has solution S. Define

$$
\begin{array}{lllll}
A = & [& \ell_V & : S(V) & \text{for each variable } V \text{ in } \mathcal{C} \\
 & & \ell_Q & : S(Q) & \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i \ ^{i\in 1..n}]^0 \\
 & & m_{Q,\ell_j} & : S(Q) & \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i \ ^{i\in 1..n}]^0 \\
 & & & & \text{and for each } j \in 1..n \\
 & & k_Q & : S(Q) & \text{for each } Q \text{ in } \mathcal{C} \text{ of the form } [\ell_i : V_i \ ^{i\in 1..n}]^0 \\
 & & \ell_{V\leq Q} & : S(V) & \text{for each constraint } (V, Q) \text{ in } \mathcal{C} \\
 & & & & \text{where } Q \text{ is of the form } [\ell_i : V_i \ ^{i\in 1..n}]^0 \\
 & & \ell_{V\oplus V'\leq Q} & : S(Q) & \text{for each constraint } (V \oplus V', Q) \text{ in } \mathcal{C} \\
 & & & & \text{where } Q \text{ is of the form } [\ell_i : V_i \ ^{i\in 1..n}]^0 \\
 &]^0 & & &
\end{array}
$$

Clearly $\emptyset \vdash a^{\mathcal{C}} : A$ is derivable.

We now prove that if $a^{\mathcal{C}}$ is typable, then $\mathcal{C}$ is solvable. Suppose $a^{\mathcal{C}}$ is typable. From Theorem 4.4.6 we get a solution S of $\mathcal{C}(a^{\mathcal{C}})$. Notice that each method in $a^{\mathcal{C}}$ binds a variable x. Each of these variables corresponds to a distinct type variable in $\mathcal{C}(a^{\mathcal{C}})$. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, and $\mathcal{C}(a^{\mathcal{C}})$ contains constraints of the form $U_x = [\ldots]^0$ for each method in $a^{\mathcal{C}}$ (from rule (4.12)), all those type variables are mapped by S to the same type. Thus, we can think of all the bound variables of methods of $a^{\mathcal{C}}$ as being related to the same type variable, which we will write as U_x.

The solution S has the following two properties.

- **Property 1** If V is a variables in $\mathcal{C}$, then $S(U_x) \downarrow \ell_V$ is defined.
- **Property 2** For each Q in $\mathcal{C}$ of the form $[\ell_i : V_i \ ^{i\in 1..n}]^0$, we have $S(U_x) \downarrow \ell_Q = [\ell_i : (S(U_x) \downarrow \ell_{V_i}) \ ^{i\in 1..n}]^0$.

To see Property 1, notice that in the body of the method ℓ_V we have the expression $x.\ell_V$. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (4.8) and (4.9) that S satisfies

$$(U_x, V_x) \text{ and } (V_x, [\ell_V : U_{x.\ell_V}]).$$

We conclude that $S(U_x) \downarrow \ell_V = S(U_{x.\ell_V})$ is defined.

To see Property 2, let Q be an occurrence in $\mathcal{C}$ of the form $[\ell_i : V_i \ ^{i\in 1..n}]^0$. For each $j \in 1..n$, in the body of the method m_{Q,ℓ_j}, we have the expression $x'.\ell_{V_j} \Leftarrow \varsigma(y)(x.\ell_Q.\ell_j)$

where we, for clarity, have written the first occurrence of x as x'. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (4.8), (4.15), (4.8), (4.9), (4.10), (4.9), and (4.10) that S satisfies

$$(U_x \quad , \quad V_{x'}) \text{ and } (V_{x'},\ [\ell_{V_j} : V_{x.\ell_Q.\ell_j}]^{\rightarrow}) \tag{4.27}$$

$$(U_x \quad , \quad V_x) \text{ and } (V_x,\ \ [\ell_Q : U_{x.\ell_Q}]^{\rightarrow}) \tag{4.28}$$

$$(U_{x.\ell_Q} \quad , \quad V_{x.\ell_Q}) \tag{4.29}$$

$$(V_{x.\ell_Q} \quad , \quad [\ell_j : U_{x.\ell_Q.\ell_j}]^{\rightarrow}) \tag{4.30}$$

$$(U_{x.\ell_Q.\ell_j} \quad , \quad V_{x.\ell_Q.\ell_j}) \tag{4.31}$$

Thus,

$$\begin{array}{rcll} S(U_x)\downarrow\ell_Q & = & S(U_{x.\ell_Q}) & \text{from (4.28) and Lemma 4.2.2} \\ & \leq & S(V_{x.\ell_Q}) & \text{from (4.29)} \\ & \leq & [\ell_j : S(U_{x.\ell_Q.\ell_j})]^{\rightarrow} & \text{from (4.30)} \\ S(U_x)\downarrow\ell_Q\downarrow\ell_j & = & S(U_{x.\ell_Q.\ell_j}) & \text{from Lemma 4.2.2} \\ & \leq & S(V_{x.\ell_Q.\ell_j}) & \text{from (4.31)} \\ & = & S(U_x\downarrow\ell_{V_j}) & \text{from (4.27) and Lemma 4.2.2} \end{array}$$

In the body of the method k_Q, we have the expression $(x.\ell_Q + [\,])$. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (4.8), (4.9), (4.10), and (4.16) that S satisfies

$$(U_x \quad , \quad V_x) \text{ and } (V_x,\ [\ell_Q : U_{x.\ell_Q}]^{\rightarrow}) \tag{4.32}$$

$$(U_{x.\ell_Q} \quad , \quad V_{x.\ell_Q}) \tag{4.33}$$

$$(V_{x.\ell_Q} \oplus V_{[\,]} \quad , \quad V_{x.\ell_Q+[\,]}) \tag{4.34}$$

Thus, from (4.32), Lemma 4.2.2, (4.33), (4.34) and the definition of $\oplus$, we have

$$S(U_x)\downarrow\ell_Q = S(U_{x.\ell_Q}) \leq S(V_{x.\ell_Q}) = [\ldots]^0. \tag{4.35}$$

In the body of the method ℓ_Q we have the expression $[\ell_i = \varsigma(y)(x.\ell_{V_i})^{\ i\in 1..n}]$. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (4.8), (4.9), (4.10), (4.11) and (4.12) that S

satisfies

$$\forall j \in 1..n,\ \ (U_x \quad , \quad V_x) \text{ and } (V_x,\ [\ell_{V_j} : U_{x.\ell_{V_j}}]^{\rightarrow}) \tag{4.36}$$

$$(U_{x.\ell_{V_j}} \quad , \quad V_{x.\ell_{V_j}}) \tag{4.37}$$

$$([\ell_i^0 : V_{x.\ell_{V_i}}\ {}^{i\in 1..n}]^0 \quad , \quad V_{[\ell_i = \varsigma(y)(x.\ell_{V_i})\ \ i\in 1..n]}) \tag{4.38}$$

$$U_x \ \equiv\ [\ldots \ell_Q : V_{[\ell_i = \varsigma(y)(x.\ell_{V_i})\ \ i\in 1..n]} \ldots]^0 \tag{4.39}$$

Thus, from (4.38) and (4.39), we have

$$[\ell_i : S(V_{x.\ell_{V_i}})\ {}^{i\in 1..n}]^0 \leq S(V_{[\ell_i = \varsigma(y)(x.\ell_{V_i})\ \ i\in 1..n]}) = S(U_x)\downarrow \ell_Q$$

and together with (4.36), Lemma 4.2.2 and (4.37), we have

$$\forall j \in 1..n,\ \ S(U_x)\downarrow \ell_{V_j} = S(U_{x.\ell_{V_j}}) \leq S(V_{x.\ell_{V_j}}) = S(U_x)\downarrow \ell_Q \downarrow \ell_j.$$

Since we have both

$$\begin{aligned} S(U_x)\downarrow \ell_Q \downarrow \ell_j \ &\leq\ S(U_x)\downarrow \ell_{V_j} \text{ and} \\ S(U_x)\downarrow \ell_Q \downarrow \ell_j \ &\geq\ S(U_x)\downarrow \ell_{V_j}, \end{aligned}$$

we have

$$S(U_x)\downarrow \ell_Q \downarrow \ell_j \ = \ S(U_x)\downarrow \ell_{V_j} \tag{4.40}$$

and together (4.40) and (4.35) give that $S(U_x) \downarrow \ell_Q = [\ell_i : S(U_x) \downarrow \ell_{V_j}^{\ i\in 1..n}]^0$, that is, Property 2.

From Property 1 we have that we can define

$$S_{\mathcal{C}}(V) \ = \ S(U_x)\downarrow \ell_V \quad \text{for each variable } V \text{ in } \mathcal{C}. \tag{4.41}$$

With this definition, we can restate Property 2 as

$$S_{\mathcal{C}}(Q) \ = \ S(U_x)\downarrow \ell_Q \quad \text{where } Q = [\ell_i : V_i\ {}^{i\in 1..n}]^0. \tag{4.42}$$

We will now show that $\mathcal{C}$ has solution $S_{\mathcal{C}}$.

Consider first a constraint (V, Q) in $\mathcal{C}$, where $Q = [\ell_i : V_i \ ^{i\in 1..n}]^0$. The body of the method $\ell_{(V,Q)}$ contains the expression $x'.\ell_Q \Leftarrow \varsigma(y)(x.\ell_V)$ where we, for clarity, have written the first occurrence of x as x'. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (4.8), (4.15), (4.8), (4.9), and (4.10) that S satisfies

$$(U_x \quad , \quad V_{x'}) \text{ and } (V_{x'}, \ [\ell_Q : V_{x.\ell_V}]^{\rightarrow}) \tag{4.43}$$

$$(U_x \quad , \quad V_x) \text{ and } (V_x, \ [\ell_V : U_{x.\ell_V}]^{\rightarrow}) \tag{4.44}$$

$$(U_{x.\ell_V} \quad , \quad V_{x.\ell_V}) \tag{4.45}$$

We conclude

$$\begin{array}{rcll} S_{\mathcal{C}}(V) & = & S(U_x)\downarrow \ell_V & \text{from (4.41)} \\ & = & S(U_{x.\ell_V}) & \text{from (4.44) and Lemma 4.2.2} \\ & \leq & S(V_{x.\ell_V}) & \text{from (4.45)} \\ & = & S(U_x)\downarrow \ell_Q & \text{from (4.43) and Lemma 4.2.2} \\ & = & S_{\mathcal{C}}(Q) & \text{from (4.42).} \end{array}$$

Consider next a constraint $(V \oplus V', Q)$ in $\mathcal{C}$, where Q is of the form $[\ell_i : V_i \ ^{i\in 1..n}]^0$. The body of the method $\ell_{(V\oplus V',Q)}$ contains the expression $x'.\ell_Q \Leftarrow \varsigma(y)(x.\ell_V + x.\ell_{V'})$ where we, for clarity, have written the first occurrence of x as x'. Since S is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (4.8), (4.15), (4.8), (4.9), (4.10), and (4.16) that S satisfies

$$(U_x \quad , \quad V_{x'}) \text{ and } (V_{x'}, \ [\ell_Q : V_{x.\ell_V + x.\ell_{V'}}]^{\rightarrow}) \tag{4.46}$$

$$(U_x \quad , \quad V_x) \text{ and } (V_x, \ [\ell_V : U_{x.\ell_V}]^{\rightarrow}) \tag{4.47}$$

$$(U_x \quad , \quad V_x) \text{ and } (V_x, \ [\ell_{V'} : U_{x.\ell_{V'}}]^{\rightarrow}) \tag{4.48}$$

$$(U_{x.\ell_V} \quad , \quad V_{x.\ell_V}) \tag{4.49}$$

$$(U_{x.\ell_{V'}} \quad , \quad V_{x.\ell_{V'}}) \tag{4.50}$$

$$(V_{x.\ell_V} \oplus V_{x.\ell_{V'}} \quad , \quad V_{x.\ell_V + x.\ell_{V'}}) \tag{4.51}$$

We conclude

$$
\begin{array}{rcll}
S_{\mathcal{C}}(V) & = & S(U_x)\downarrow \ell_V & \text{from (4.41)} \\
 & = & S(U_{x.\ell_V}) & \text{from (4.47) and Lemma 4.2.2} \\
 & = & S(V_{x.\ell_V}) & \text{from (4.49) and (4.51)} \\
S_{\mathcal{C}}(V') & = & S(U_x)\downarrow \ell_{V'} & \text{from (4.41)} \\
 & = & S(U_{x.\ell_{V'}}) & \text{from (4.48) and Lemma 4.2.2} \\
 & = & S(V_{x.\ell_{V'}}) & \text{from (4.50) and (4.51)} \\
S_{\mathcal{C}}(V) \oplus S_{\mathcal{C}}(V') & = & S(V_{x.\ell_V}) \oplus S(V_{x.\ell_{V'}}) & \text{from above} \\
 & \leq & S(V_{x.\ell_V + x.\ell_{V'}}) & \text{from (4.51)} \\
 & = & S(U_x)\downarrow \ell_Q & \text{from (4.46) and Lemma 4.2.2} \\
 & = & S_{\mathcal{C}}(Q) & \text{from (4.42).}
\end{array}
$$

□

4.6.2 Solving Simple Constraints is NP-hard

In this section we show that solving simple constraint systems is NP-hard.

Suppose we are given a Boolean expression

$$\psi = \ \wedge_{i=1}^{n}(l_{i1} \vee l_{i2} \vee l_{i3})$$

where X_ψ is the set of variables occurring in ψ, and each literal l_{ij} is of the form x or $\bar{x}$, where $x \in X_\psi$. We will use the notation I_x for the set of positions (ij) for which $l_{ij} = x$ or $l_{ij} = \bar{x}$. Furthermore, if $l_{ij} = x$ or $l_{ij} = \bar{x}$, then we use I_{ij} to denote I_x. We will use the abbreviations

$$\mathsf{False} = [\,]^0 \qquad \mathsf{True} = [q : [\,]^0]^0.$$

Their only significance is that $\mathsf{False} \neq \mathsf{True}$. We will construct a simple constraint system $\mathcal{C}_\psi$ over the variables

$$
\begin{array}{rl}
 & \{\ U_x, U_{\bar{x}}, V_x, V_{\bar{x}}, T_x, T_{\bar{x}}, R_x \ \mid \ x \in X_\psi\ \} \\
\cup & \{\ P_{ij} \ \mid \ i \in 1..n, j \in 1..3\} \\
\cup & \{\ A_{ij} \ \mid \ i \in 1..n, j \in 0..3\}.
\end{array}
$$

The constraint system $\mathcal{C}_\psi$ consists of:

- for each $x \in X_\psi$, the constraints

$$(U_x \oplus U_{\bar{x}} \quad , \quad [k : R_x]^0) \tag{4.52}$$

$$(U_x \oplus T_x \quad , \quad [k : V_x]^0) \tag{4.53}$$

$$(U_{\bar{x}} \oplus T_{\bar{x}} \quad , \quad [k : V_{\bar{x}}]^0) \tag{4.54}$$

$$(R_x \quad , \quad [m_{ij} : A_{ij}^{(ij)\in I_x}]^0) \tag{4.55}$$

$$(V_x \oplus V_{\bar{x}} \quad , \quad [m_{ij} : A_{ij}^{(ij)\in I_x}]^0) \tag{4.56}$$

- for all $i \in 1..n$ and for all $j \in 1..3$, the constraints:

$$(V_{l_{ij}} \oplus P_{ij} \quad , \quad [m_{ij} : A_{i(j-1)}, \ \ m_{i'j'} : A_{i'j'}^{(i'j')\in I_{ij}\setminus(ij)}]^0) \tag{4.57}$$

- for all $i \in 1..n$, the constraints:

$$(A_{i0} \quad , \quad \mathsf{False})$$

$$(A_{i3} \quad , \quad \mathsf{True}).$$

In the last constraint, we use the abbreviation (A_{i3}, True) to denote the two constraints $(A_{i3}, [q : B]^0)$, $(B, [\,]^0)$, where B is a fresh variable.

Lemma 4.6.3. *Solving simple constraint systems is NP-hard.*

Proof. Given that 3-SAT is NP-hard, it is sufficient to show that ψ is satisfiable if and only if $\mathcal{C}_\psi$ is solvable.

Suppose first that ψ has solution f. Here is a mapping S_f from the variables of $\mathcal{C}_\psi$ to types. If $f(x)$ is true, then we have:

v	$S_f(v)$
U_x	$[\,]^0$
$U_{\bar{x}}$	$[k : S_f(R_x)]^0$
V_x	$[\,]^0$
$V_{\bar{x}}$	$S_f(R_x)$
T_x	$[k : [\,]^0]^0$
$T_{\bar{x}}$	$[\,]^0$
R_x	$[m_{ij} : S_f(A_{ij})^{\ (ij)\in I_x}]^0$.

If $f(x)$ is false, then we have:

v	$S_f(v)$
U_x	$[k : S_f(R_x)]^0$
$U_{\bar{x}}$	$[\,]^0$
V_x	$S_f(R_x)$
$V_{\bar{x}}$	$[\,]^0$
T_x	$[\,]^0$
$T_{\bar{x}}$	$[k : [\,]^0]^0$
R_x	$[m_{ij} : S_f(A_{ij})\ \ ^{(ij)\in I_x}]^0$.

For $i \in 1..n$ and $j \in 1..3$, define

$$S_f(P_{ij}) = \begin{cases} [m_{ij} : S_f(A_{i(j-1)}),\ \ m_{i'j'} : S_f(A_{i'j'})\ \ ^{(i'j')\in I_{ij}\setminus(ij)}]^0 & f(l_{ij}) \text{ is true} \\ [\,]^0 & \text{otherwise.} \end{cases}$$

Define the function g from Booleans to $\{\mathsf{False}, \mathsf{True}\}$ such that $g(\text{false}) = \mathsf{False}$ and $g(\text{true}) = \mathsf{True}$. For $i \in 1..n$,

v	$S_f(v)$
A_{i0}	False
A_{i1}	$g \circ f(l_{i1})$
A_{i2}	$g \circ f(l_{i1} \vee l_{i2})$
A_{i3}	True.

It is straightforward to check that S_f is a solution to the constraints in $\mathcal{C}_\psi$ of the forms (4.52)–(4.56), we omit the details. Here we will focus on showing that S_f is a solution to the constraints in $\mathcal{C}_\psi$ of the form (4.57). Suppose we are given $i \in 1..n$ and $j \in 1..3$. There are two cases. First, if $f(l_{ij})$ is true, then $S_f(P_{ij}) = [m_{ij} : S_f(A_{i(j-1)}),\ \ m_{i'j'} : S_f(A_{i'j'})\ \ ^{(i'j')\in I_{ij}\setminus(ij)}]^0$ and $S_f(V_{l_{ij}}) = [\,]^0$. Hence, the constraint (4.57) is satisfied.

Second, if $f(l_{ij})$ is false, then $S_f(P_{ij}) = [\,]^0$ and $S_f(V_{l_{ij}}) = S_f(R_{l_{ij}})$. Hence, we must show that $S_f(A_{ij}) = S_f(A_{i(j-1)})$. There are three cases.

- If $j = 1$, then $S_f(A_{i1}) = g \circ f(l_{i1}) = g(\text{false}) = \mathsf{False} = S_f(A_{i0})$.

- If $j = 2$, then $S_f(A_{i2}) = g \circ f(l_{i1} \vee l_{i2}) = g \circ (f(l_{i1}) \vee f(l_{i2})) = g \circ (f(l_{i1}) \vee \text{false}) = g \circ f(l_{i1}) = S_f(A_{i1})$.

- If $j = 3$, then $S_f(A_{i3}) = \mathsf{True}$. Since ψ is satisfiable and $f(l_{i3})$ is false, we have that $f(l_{i1} \vee l_{i2})$ is true, so $S_f(A_{i2}) = g \circ f(l_{i1} \vee l_{i2}) = g(\text{true}) = \mathsf{True}$. We conclude that $S_f(A_{i3}) = \mathsf{True} = S_f(A_{i2})$.

Conversely, suppose S is a solution to $\mathcal{C}_\psi$.

Property 1: For every $x \in X_\psi$, we have either $S(V_x) = S(R_x)$ and $S(V_{\bar{x}}) = [\,]^0$, or we have $S(V_x) = [\,]^0$ and $S(V_{\bar{x}}) = S(R_x)$.

To prove Property 1, notice that from (4.52) we have exactly one of $S(U_x) = [k : S(R_x)]^{\rightarrow}$ or $S(U_{\bar{x}}) = [k : S(R_x)]^{\rightarrow}$. From that and (4.53)–(4.54), we have that either $S(V_x) = S(R_x)$ or $S(V_{\bar{x}}) = S(R_x)$. From that and (4.56) we get Property 1.

Define

$$f_S(x) = \begin{cases} \text{false} & S(V_x) = S(R_x) \\ \text{true} & \text{otherwise.} \end{cases}$$

Going for a contradiction, let us suppose that f_S does not satisfy ψ. That means that must exist i such that, for all $j \in 1..3$, $f_S(l_{ij}) = \text{false}$. From the definition of f_S and Property 1 we have that, for $j \in 1..3$, there is a variable x such that $(ij) \in I_x$ and $S(V_{l_{ij}}) = S(R_x)$. From that and (4.55) and (4.57), we conclude

$$\mathsf{False} = S(A_{i0}) = S(A_{i1}) = S(A_{i2}) = S(A_{i3}) = \mathsf{True},$$

a contradiction. □

Theorem 4.6.4. *The type inference problem is NP-complete.*

Proof. We have that type inference is in NP from Theorem 4.5.16. NP-hardness follows from Lemma 4.6.2 and Lemma 4.6.3. □

4.7 Conclusion

Type inference with record concatenation, subtyping, and recursive types is NP-complete. Future work includes implementing the algorithm for a language such as Obliq, and to attempt to combine our technique with our algorithm for type inference with both covariant and invariant fields [PZJ02].

The construction used in our NP-hardness proof may be applicable to other types systems. In particular, our notion of simple constraint systems may be reducible to even more restrictive type inference problems than the one we have considered.

5. Summary and Future Work

5.1 Summary

In this thesis, we have studied type-matching and type-inference problems for object-oriented systems. In the introductory chapter, we briefly explained formal type systems and their applications; we also discussed class-based, object-based languages and object calculi. We surveyed the applicable areas of type matching which include retrieval of software components and generating bridge code for multi-language systems. Moreover, we listed some of the motivations of type inference for object-type systems with variance annotations.

In Chapter 2, we presented the solutions to the type matching problem for object interface types with flexible equality rules. Our solution is based on a definition of bisimulation. A straightforward implementation of the solution results in an $O(n^2)$ time algorithm and the time complexity was further improved to $O(n \log n)$ by a reduction to the problem of finding the coarsest size-stable refinement of a graph. We implemented the last algorithm in Java. The implementation allows users to input definitions of two interfaces and generates matched types if any exist.

In Chapter 3, we developed type inference algorithm for automatically discovering covariant read-only fields for an invariant of Abadi-Cardelli object calculus. Covariant read-only fields are applicable to object calculi, mobile processes and typed intermediate languages. In Chapter 4, we developed algorithm of type inference for type systems with record concatenations which are useful in untyped object-based languages such as Obliq.

5.2 Future Work

We have demonstrated that type matching and type inference can be valuable methods for improving software integration and programming efficiency. More work can be done to

make these methods more practical. For instance, it may be interesting to find out whether one could infer principal types based on the type equality rules used in type matching. The existence and decidability of such principal types in type assignment systems for objects or functions could help component-software writers to omit some of the interface types in their applications. One other future direction is to support interoperability by automatically generating a converter from data in one language to data of an equivalent type in another language. Since data conversion may require types such as linked lists and arrays, a union type operator denoted by $+$ may be needed in the set of type-equality rules that we consider. Since $\times$ operator is distributive over $+$, the existing approaches may no longer apply.

To make our type-inference approach more practical, we would like to find out whether it is possible to combine the type-inference algorithms in Chapter 3 and 4 to cover a larger fragment of the type system in [Gle00]. Such an algorithm can be helpful for enabling the omission of excessive type annotations. We are also interested in finding a type-inference algorithm for objects with both read-only and write-only fields.

LIST OF REFERENCES

LIST OF REFERENCES

[ABCCR99] Joshua Auerbach, Charles Barton, Mark Chu-Carroll, and Mukund Raghavachari. Mockingbird: Flexible stub compilation from pairs of declarations. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 393–402, June 1999.

[ABR98] Joshua Auerbach, Charles Barton, and Mukund Raghavachari. Type isomorphisms with recursive types. Research report RC 21247, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York, August 1998.

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Also in Proceedings of the 19th ACM Symposium on Principles of Programming Languages.

[AC96a] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AC96b] Maria-Virginia Aponte and Roberto Di Cosmo. Type isomorphisms for module signatures. In *Proceedings of Eighth International Symposium on Programming Languages, Implementations, Logics, and Programs*, pages 334–346. Springer-Verlag (*LNCS* 1140), 1996.

[ACC97] Joshua Auerbach and Mark C. Chu-Carroll. The mockingbird system: A compiler-based approach to maximally interoperable distributed programming. Research report RC 20718, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York, February 1997.

[AF96] Martín Abadi and Marcelo P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 242–252, 1996.

[AS97] A. Andreev and S. Soloviev. A deciding algorithm for linear isomorphism of types with complexity $O(n \log^2(n))$. *Lecture Notes in Computer Science*, 1290:197ff, 1997.

[AW93] Alexander Aiken and Edward Wimmers. Type inclusion constraints and type inference. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[BCL92] Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

[Ben94] Marcin Benke. Efficient type reconstruction in the presence of inheritance. Manuscript, 1994.

[BH97] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications*, 1997.

[BI82] Alan H. Borning and Daniel H. H. Ingalls. Multiple inheritance in Smalltalk80. In *Proceedings of the Second National Conference on Artificial Intelligence*, 1982.

[BKW96] Daniel J. Barrett, Alan Kaplan, and Jack C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *Proceedings of the Fourth Symposium on the Foundations of Software Engineering*, San Francisco, California, October 1996.

[BPG02] Michele Bugliesi and Santiago Pericas-Geertsen. Type inference for variant object types. *Information and Computation*, 2002.

[Car95] Luca Cardelli. A language with distributed scope. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, 1995.

[CC02] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Proceedings of CC'02, International Conference on Compiler Construction*, pages 159–178. Springer-Verlag (*LNCS* 2304), 2002.

[CLR90] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts, 1990.

[Con00] Jeffrey Considine. Deciding isomorphisms of simple types in polynomial time. Manuscript, 2000.

[Cos95] Roberto Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Birkhäuser, 1995.

[Cou81] Patrick Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Cou83] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(1):95–169, 1983.

[CP95] Jiazhen Cai and Robert Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science*, 145(1–2)(1–2):189–228, 1995.

[DG84] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Progamming*, 1(3):267–84, October 1984.

[Fre97] Alexandre Frey. Satisfying systems of subtype inequalities in polynomial space. In *Proceedings of the Fourth International Static Analysis Symposium*. Springer-Verlag (*LNCS*), 1997.

[Gay94] David E. Gay. Interface definition language conversions: Recursive types. *ACM SIGPLAN Notices*, 29(8):101–110, August 1994.

[Gle00] Neal Glew. An efficient class and object encoding. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 311–324, Minneapolis, Minnesota, October 2000.

[Gol79] Robert Goldblatt. *Topoi: The Categorical Analysis of Logic*, volume 98 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, Amsterdam, 1979.

[Hen97] Fritz Henglein. Breaking through the n^3 barrier: Faster object type inference. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, 1997. http://www.cs.williams.edu/ kim/FOOL/index.html.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[HM95] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Proceedings of the 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 176–185, 1995.

[HP91] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages, Orlando*, pages 131–142. ACM, 1991.

[IK00] Atsushi Igarashi and Naoki Kobayashi. Type reconstruction for linear pi-calculus with I/O subtyping. *Journal of Information and Computation*, 161(1):1–44, 2000. An extended abstract appeared in the *Proceedings of SAS '97*, LNCS 1302.

[IV02] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *Proceedings of ECOOP'02, 16th European Conference on Object-Oriented Programming*, Spain, June 10-14 2002.

[JP97] Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, 1997.

[JPZ02] Somesh Jha, Jens Palsberg, and Tian Zhao. Efficient type matching. In *Proceedings of the Fifth Foundations of Software Science and Computation Structures*, Grenoble, France, April 2002.

[Koz94] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1 May 1994.

[KPS94] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994. Preliminary version in Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.

[KPS95] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, 1995. Preliminary version in Proceedings of the 20th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 419–428, Charleston, South Carolina, January 1993.

[McA02] David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[Mil84] Robin Milner. A complete inference system for a class of regular behaviors. *Journal of Computer and System Sciences*, 28(3):439–466, June 1984.

[Mil90] Robin Milner. Operational and algebraic semantics of concurrent processes. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 19, pages 1201–1242. The MIT Press, New York, 1990.

[Mit91] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1:245–285, 1991.

[MNP00] Martin Müller, Joachim Niehren, and Andreas Podelski. Ordering constraints over feature trees. *Constraints, an International Journal*, 5(1-2):7–42, January 2000.

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Computation*, 71(1/2):95–130, October/November 1986.

[Nie89] Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of the Second International Parallel Architectures and Languages Europe Conference*, pages 357–373, April 1989.

[NPS93] Paliath Narendran, Frank Pfenning, and Richard Statman. On the unification problem for Cartesian closed categories. In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 57–63. IEEE Computer Society Press, 1993.

[OMG99] OMG. The common object request broker: Architecture and specification. Technical report, Object Management Group, 1999. Version 2.3.1.

[Pal95] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. Preliminary version in Proceedings of the Nineth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

[Par81] D.M.R. Park. Concurrency and automata on infinite sequences. In *Proceedings of the Fifth GI Conference*, pages 15–32. Springer-Verlag (*LNCS* 104), 1981.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PJ97] Jens Palsberg and Trevor Jim. Type inference with simple selftypes is NP-complete. *Nordic Journal of Computing*, 4(3):259–286, Fall 1997.

[PO95] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995. Preliminary version in Proceedings of the 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 367–378, San Francisco, California, January 1995.

[Pot96] Francois Pottier. Simplifying subtyping constraints. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 1996.

[Pot00] Francois Pottier. A 3-part type inference engine. In Gert Smolka, editor, *Proceedings of the 2000 European Symposium on Programming*, volume 1782, pages 320–335. Springer Verlag, 2000.

[PP98] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In *Proceedings of the 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 197–208, San Diego, California, January 1998.

[PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science*, pages 376–385, 1993.

[PS96] Jens Palsberg and Scott Smith. Constrained types and their expressiveness. *ACM Transactions on Programming Languages and Systems*, 18(5):519–527, September 1996.

[PT87] Robert Paige and Robert Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.

[PWO97] Jens Palsberg, Mitchell Wand, and Patrick M. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

[PZ01] Jens Palsberg and Tian Zhao. Efficient and flexible matching of recursive types. *Information and Computation*, 171:364–387, 2001. Preliminary version in Proceedings of the Fifteenth Annual IEEE Symposium on Logic in Computer Science, pages 388–398, Santa Barbara, California, June 2000.

[PZ02] Jens Palsberg and Tian Zhao. Efficient type inference for record concatenation and subtyping. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, Copenhagen, Denmark, July 2002.

[PZJ02] Jens Palsberg, Tian Zhao, and Trevor Jim. Automatic discovery of covariant read-only fields. In *Proceedings of the Ninth International Workshop on Foundations of Object-Oriented Languages*, Portland, Oregon, January 2002.

[Rem92] Didier Remy. Typing record concatenation for free. In *ACM Symposium on Principles of Programming Languages*, pages 166–176, 1992.

[Rém98] Didier Rémy. From classes to objects via subtyping. In *European Symposium On Programming*. Springer-Verlag (*LNCS* 1381), March 1998.

[Rit90] Mikael Rittri. Retrieving library identifiers via equational matching of types. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *LNAI*, pages 603–617, Kaiserslautern, FRG, July 1990. Springer Verlag.

[Rit91] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

[Rit93] Mikael Rittri. Retrieving library functions by unifying types modulo linear isomorphism. *RAIRO Theoretical Informatics and Applications*, 27(6):523–540, 1993.

[SM01] Mark Shields and Erik Meijer. Type-indexed rows. In *Symposium on Principles of Programming Languages*, pages 261–275, 2001.

[SMZ99] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Technical report, University of South Australia, 1999. Technical Report ACRC-99-009.

[Sol83] Sergei V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22:1387–1400, 1983.

[Sol93] Sergei Soloviev. A complete axiom system for isomorphism of types in closed categories. pages 380–392. Springer-Verlag (*LNAI* 698), 1993.

[Str93] Bjarne Stroustrup. *A History of C++: 1979–1991*. 1993. Manuscript.

[Sul97] Martin Sulzmann. Designing Record Systems. Research Report YALEU/DCS/RR-1128, Yale University, Department of Computer Science, April 1997.

[Tar55] Alfred Tarski. A lattice-theoretical fixed point theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TH01] Francis Tang and Martin Hofmann. Type inference for objects with base types. Draft, 2001.

[TH02] Francis Tang and Martin Hofmann. Generation of verification conditions for abadi and leino's logic of objects. In *Proceedings of the Ninth International Workshop on Foundations of Object-Oriented Languages*, Portland, Oregon, January 2002.

[Tha96] Satish Thatte. Automated synthesis of interface adapters for reusable classes. In *Proceedings of the 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 174–185, 1996.

[Tiu92] Jerzy Tiuryn. Subtype inequalities. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 308–315, 1992.

[Tsu94] Hideki Tsuiki. On typed calculi with a merge operator. In *Foundations of Software Technology and Theoretical Computer Science*, pages 101–112, 1994.

[Wan91] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation,* 93(1):1–15, 1991.

[WF94] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WOP95] Mitchell Wand, Patrick M. O'Keefe, and Jens Palsberg. Strong normalization with non-structural subtyping. *Mathematical Structures in Computer Science*, 5(3):419–430, 1995.

[ZW95] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering Methodology*, 4(2):146–170, April 1995.

[Zwa95] Jan Zwanenburg. Record concatenation with intersection types. Technical Report 95–34, Eindhoven University of Technology, 1995.

[Zwa96] Jan Zwanenburg. A type system for record concatenation and subtyping. In Kim Bruce and Giuseppe Longo, editors, *Informal proceedings of Third Workshop on Foundations of Object-Oriented Languages,* 1996.

APPENDIX

APPENDIX

A.1 Proof of the first half of Theorem 2.5.8

Theorem A.1.1. $\mathbf{EQ} = \mathcal{R}$.

Proof. First we prove $\mathbf{EQ} \subseteq \mathcal{R}$ (*soundness*). Suppose Δ is a derivation tree for $\emptyset \vdash \sigma = \tau$. Let R be the set of type pairs that are found in Δ on the right-hand side of $\vdash$, except for applications of the rule (HYP). It is straightforward to see that all other type pairs in Δ are elements of R. Notice that $(\sigma, \tau) \in R$. It is straightforward to show that $R \subseteq F(R)$. From that and Lemma 2.5.7 we have that R is a bisimulation, so, by co-induction, $(\sigma, \tau) \in \mathcal{R}$.

Next we prove $\mathcal{R} \subseteq \mathbf{EQ}$ (*completeness*). Suppose $(\sigma, \tau) \in \mathcal{R}$. Choose a bisimulation R' such that $(\sigma, \tau) \in R'$. Define $R = R' \cap (V_\sigma \times V_\tau)$. Notice that R is a finite set, and $(\sigma, \tau) \in R$. Let us show that R is a bisimulation. First, from R' being a bisimulation and Lemma 2.5.7, $R' \subseteq F(R')$. It follows that $R' \cap (V_\sigma \times V_\tau) \subseteq F(R') \cap (V_\sigma \times V_\tau)$. ¿From Lemma 2.5.13 we have $F(R') \cap (V_\sigma \times V_\tau) \subseteq F(R' \cap (V_\sigma \times V_\tau))$, so $R' \cap (V_\sigma \times V_\tau) \subseteq F(R' \cap (V_\sigma \times V_\tau))$, that is, $R \subseteq F(R)$. Thus, by Lemma 2.5.7, R is a bisimulation.

From R, we can now construct a derivation tree for $\emptyset \vdash \sigma = \tau$. The function $\mathcal{S}$, see below, is a recursive function that takes as inputs (1) an environment A, and (2) a type pair (σ, τ). The call $\mathcal{S}(A, (\sigma, \tau))$ returns a suggestion for a derivation tree for $A \vdash \sigma = \tau$.

$\mathcal{S}\ (A, (\sigma, \tau)) =$

- If σ, τ are base types, then return $A \vdash \sigma = \tau$
- If $(\sigma, \tau) \in A,$ then return $A \vdash \sigma = \tau$
- If $\sigma = \sigma_1 \to \sigma_2, \tau = \tau_1 \to \tau_2,$ then return
$$\frac{\mathcal{S}((A, \sigma = \tau), (\sigma_i, \tau_i)) \quad \forall i \in \{1, 2\}}{A \vdash \sigma = \tau}$$
- If $\sigma = \Pi_{i=1}^n \sigma_i, \tau = \Pi_{i=1}^n \tau_i,$ then return

$$\frac{\mathcal{S}((A, \sigma = \tau), (\sigma_i, \tau_{t(i)})) \quad \forall i \in \{1..n\}}{A \vdash \sigma = \tau}$$

where $(\sigma_i, \tau_{t(i)}) \in R$ and

t is a bijection from $\{1..n\}$ to $\{1..n\}$.

Consider the call $\mathcal{S}(\emptyset, (\sigma, \tau))$. It is straightforward to see that in every recursive call to $\mathcal{S}$, all type pairs in the arguments are elements of R. Since R is a bisimulation, this ensures that the rules in $EQ_{\rightarrow \prod}$ apply. Moreover, every time $\mathcal{S}$ is called, the size of A will increase by one, since otherwise we could use the second case in the definition of $\mathcal{S}$ to avoid further recursive calls. This limits the depth of the recursion to the number of elements of R. Since R is finite, we conclude that $\mathcal{S}(\emptyset, (\sigma, \tau))$ has a finite depth of recursion and that the size of the resulting derivation tree for $\emptyset \vdash \sigma = \tau$ is finite. □

A.2 Proof of Theorem 2.5.9

Theorem A.2.1. *$\mathcal{R}$ is a congruence relation.*

Proof. We will show that $\mathcal{R}$ is reflexive, symmetric, transitive, and a congruence in the $\rightarrow$ and $\prod$ constructors.

(Reflexivity) Suppose γ is a base type. Construct the relation

$$R = \{\ (\sigma, \sigma)\ \mid\ \sigma \text{ is a base type}\ \}.$$

We have $(\gamma, \gamma) \in R$, and R is closed and consistent. Hence, R is a bisimulation, and, by co-induction, $(\gamma, \gamma) \in \mathcal{R}$.

(Symmetry) Suppose $(\sigma, \tau) \in \mathcal{R}$. Choose a bisimulation R such that $(\sigma, \tau) \in R$, and construct from R the relation:

$$R' = \{\ (\sigma, \sigma')\ \mid\ (\sigma', \sigma) \in R\ \}.$$

From $(\sigma, \tau) \in R$, we have $(\tau, \sigma) \in R'$. R' is bisimulation because the conditions for being a bisimulation are symmetric with respect to the two components of a type pair. So, by co-induction, $(\tau, \sigma) \in \mathcal{R}$.

(Transitivity) Suppose $(\sigma, \delta), (\delta, \tau) \in \mathcal{R}$. Choose bisimulations R_1, R_2 such that $(\sigma, \delta) \in R_1$, $(\delta, \tau) \in R_2$, and construct from R the relation

$$R = \{\ (\sigma_1, \sigma_3)\ \mid\ (\sigma_1, \sigma_2) \in R_1, (\sigma_2, \sigma_3) \in R_2\ \}.$$

From $(\sigma, \delta) \in R_1,\ (\delta, \tau) \in R_2$, we have $(\sigma, \tau) \in R$.

For any $(\sigma_1, \sigma_2) \in R_1, (\sigma_2, \sigma_3) \in R_2$, we have $\sigma_1(\epsilon) = \sigma_2(\epsilon), \sigma_2(\epsilon) = \sigma_3(\epsilon)$, so $\sigma_1(\epsilon) = \sigma_3(\epsilon)$, and therefore R is consistent.

If $\sigma = \sigma_1 \to \sigma_2$, $\delta = \delta_1 \to \delta_2$, and $\tau = \tau_1 \to \tau_2$, then, for every $i \in \{1, 2\}$, we have $(\sigma_i, \delta_i) \in R_1, (\delta_i, \tau_i) \in R_2$, so $(\sigma_i, \tau_i) \in R$, and therefore R is closed under condition $P1$.

If $\sigma = \prod_{i=1}^{n} \sigma_i$, $\delta = \prod_{i=1}^{n} \delta_i$, and $\tau = \prod_{i=1}^{n} \tau_i$, then there exist bijections, u, v such that, for every $i \in \{1..n\}$, we have $(\sigma_{u(i)}, \delta_i) \in R_1, (\delta_{v(i)}, \tau_i) \in R_2$, so $(\sigma_{t(i)}, \tau_i) \in R$, where, $t = u \circ v$, and therefore R is closed under condition $P2$.

We conclude that R is a bisimulation, and, by co-induction, $(\sigma, \tau) \in \mathcal{R}$.

(Congruence in $\to$) Suppose $(\sigma_1, \tau_1), (\sigma_2, \tau_2) \in \mathcal{R}$, and $\sigma = \sigma_1 \to \sigma_2, \tau = \tau_1 \to \tau_2$, Choose bisimulations R_1, R_2 such that $(\sigma_1, \tau_1) \in R_1$, $(\sigma_2, \tau_2) \in R_2$, and construct from R_1, R_2 the relation

$$R = \{(\sigma, \tau)\} \cup R_1 \cup R_2.$$

We have $(\sigma, \tau) \in R$ by construction.

Since bisimulation is closed under union, $R_1 \cup R_2$ is a bisimulation. Moreover, $\sigma(\epsilon) = \tau(\epsilon) = \to$, and $(\sigma_1, \tau_1), (\sigma_2, \tau_2) \in R$, so R is a bisimulation, and, by co-induction, $(\sigma, \tau) \in \mathcal{R}$.

(Congruence in $\prod$) Suppose, for every $i \in \{1..n\}$, that $(\sigma_i, \tau_{t_i}) \in \mathcal{R}$, where t is a bijection from $\{1..n\}$ to $\{1..n\}$, and $\sigma = \prod_{i=1}^{n} \sigma_i, \tau = \prod_{i=1}^{n} \tau_i$. For each $i \in \{1..n\}$, choose a bisimulation R_i such that $(\sigma_i, \tau_{t(i)}) \in R_i$, and construct the relation

$$R = \{(\sigma, \tau)\} \cup (\bigcup_{i=1}^{n} R_i).$$

We have $(\sigma, \tau) \in R$ by construction.

Since bisimulation is closed under union, $\bigcup_{i=1}^{n} R_i$ is a bisimulation. Moreover, $\sigma(\epsilon) = \prod^n = \tau(\epsilon)$, and for every $i \in \{1..n\}$, we have $(\sigma_i, \tau_{t_i}) \in R$, so R is a bisimulation, and, by co-induction, $(\sigma, \tau) \in \mathcal{R}$.

□

A.3 Proof of Lemma 3.2.3

Here we give a full proof that $\leq$ is a partial order. First, $\leq$ is reflexive because the identity on $\mathcal{T}(\Sigma)$ is a simulation. The composition $(R \circ R')$ of binary relations R and R' over a set X is defined in the usual way:

$$(x, x') \in (R \circ R') \Leftrightarrow \exists x'' \in X.\ (x, x'') \in R, (x'', x') \in R'.$$

Lemma A.3.1. *If R is a reflexive simulation, then $(R \circ R)$ is a simulation.*

Proof. Suppose $(A, A') \in (R \circ R)$. Thus, $\exists A''$ such that $(A, A''), (A'', A') \in R$.

- If $A' = U$, then $A'' = U$ since $(A'', A') \in R$; and then $A = U$ since $(A, A'') \in R$.
- Similarly, if $A = U$, then $A' = U$.
- Otherwise $A' = [\ell^{v'} : B', \ldots]$. Then since R is a simulation, we have
  - $A'' = [\ell^{v''} : B'', \ldots]$,
  - $A = [\ell^{v} : B, \ldots]$,
  - $v \sqsubseteq v'' \sqsubseteq v'$,
  - $(B'', B') \in R$,
  - $v' = 0 \Rightarrow (B', B'') \in R$,
  - $(B, B'') \in R$, and
  - $v'' = 0 \Rightarrow (B'', B) \in R$.

  Since $\sqsubseteq$ is transitive we have $v \sqsubseteq v'$. We have $(B, B'') \in R$, $(B'', B') \in R$; that is, $(B, B') \in (R \circ R)$. If $v' = 0$, then from $v'' \sqsubseteq v'$ we have $v'' = 0$, and so $(B', B'') \in R$, $(B'', B) \in R$, that is, $(B', B) \in (R \circ R)$, as desired.

□

Corollary A.3.2. $\leq$ *is transitive.*

Proof. Just note that $\leq$ is reflexive, and $\leq \supseteq (\leq \circ \leq)$ because $\leq$ is the largest simulation. □

Lemma A.3.3. *Every simulation is antisymmetric.*

Proof. Let R be a simulation. We prove the following statement by induction on α:

If $(A, A') \in R$ and $(A', A) \in R$, then $A = A'$, that is, $A(\alpha) = A'(\alpha)$ for every α.

- If $\alpha = \epsilon$, we show $A(\alpha) = A'(\alpha)$ by cases on the structure of A.
  - If $A = U$, then by the definition of simulation, $A' = U$. Therefore $A(\alpha) = U = A'(\alpha)$.
  - If A is a record type, then by the definition of simulation and the antisymmetry of $\sqsubseteq$, A' is a record type with exactly the same labels and variances; that is, $A = [\ell_i^{v_i} : B_i \ ^{i \in 1..n}]$ and $A' = [\ell_i^{v_i} : B_i' \ ^{i \in 1..n}]$. Therefore $A(\alpha) = \{\ell_i^{v_i} : i \in 1..n\} = A'(\alpha)$ as desired.
- If $\alpha = \ell\alpha'$, we consider two cases.
  - If $A(\ell)$ is undefined, then either $A = U$ for some U, or A is a record type with no ℓ field. In the first case, $A' = U$ because $(A', A) \in R$. In the second case, A' has no ℓ field (otherwise $(A, A') \in R$ would imply A has an ℓ field, contradiction). In either case, $A'(\ell)$ is undefined, so both $A(\alpha)$ and $A'(\alpha)$ are undefined.
  - If $A = [\ell^v : B, \ldots]$, then by the definition of simulation and the antisymmetry of $\sqsubseteq$, we have $A' = [\ell^v : B', \ldots]$ and $(B, B'), (B', B) \in R$. Then by induction, $B(\alpha') = B'(\alpha')$. So $A(\alpha) = B(\alpha') = B'(\alpha') = A'(\alpha)$ as desired.

□

A.4 Notation for Chapter 2

$\rightarrow, \prod^n$	function type constructor and product type constructor of arity n
α, β	type variable
δ	transition function
Γ	a set of base types
ℓ	labeling function
ω	the set of natural number
Σ	finite ranked alphabet
σ, τ, η	types
A	static type environment
B	blocks in a partition of a graph
abs	function that maps integer function such as I to relation on types
C, D	relations on states
$\mathcal{E}, \mathcal{R}$	the largest bisimulations on types
E	relation on nodes of a graph
F, H	functions on power-sets
$\mathcal{H}$	function on integer functions such as I
I	function that maps types to integers
i	index
K, L	equivalence relations on nodes of graph
$match$	matching function
$\mathcal{P}$	unary operator that maps a set to its power-set
P, S	partitions of a graph
Q	finite set of states
q	state
R	relation on types
$\bar{\mathcal{R}}$	the largest simulation on types
$\mathcal{M}$	term automaton

$\mathbf{T}$	Unit type
t, π	bijections
$\mathcal{T}, V, W$	sets of types
U	set of nodes of a graph
$X, Y, Z, \mathcal{Z}$	generic sets

A.5 Notation for Chapters 3 and 4

$0, +$	variances
$\leq$	largest simulation
α	path (element of Labels^*)
ϵ	the empty path
δ_R	one-step transition function
Σ	the alphabet of trees
σ	element of Σ
A, B, C, W	types
$A(\alpha)$	symbol of Σ at path α in type A
$A \downarrow \alpha$	subtree of type A at path α
$A[U := B]$	type A with U replaced by B
$\mathcal{A}$	function from $\mathsf{RelTypes}$ to $\mathsf{RelTypes}$
$\mathcal{B}_R$	function from $\mathsf{RelStates}$ to $\mathsf{RelStates}$
a, b, c	terms
$a[\cdot]$	context with one hole
$a[b]$	context $a[\cdot]$ with hole filled by term b
$a[\ell \leftarrow \varsigma(x)b]$	update field ℓ of a with $\varsigma(x)b$
$a[x := b]$	term a with x replaced by b
a^R	term whose typability is equivalent to R
$a \rightsquigarrow b$	term a rewrites to term b
above_R	function from States to States
ABOVE_R	function from $\mathsf{RelTypes}$ to $\mathsf{RelStates}$
$\mathcal{C}(a)$, E_a, X_a, Y_a	system equivalent to typability of term a
E	type environment
$E[x : A]$	type environment with updated binding for x
$E \backslash x$	type environment with binding for x removed
g, h	states or sets of types

I	index set
i, j	indices
k, ℓ, m	labels
Labels	the set of labels
LV	function from States to $\mathcal{P}(\mathsf{Labels} \times \mathsf{Variances})$
n	index bound
$\mathcal{P}$	power set
Q	elements of constraints
R	relation on types also known as set of constraints
$\mathcal{R}$	relation on sets of types
RelTypes	relations of types
RelStates	relations of sets of types
S	substitution
States	sets of types
$\mathcal{T}$	set of types
$\mathcal{T}.\ell$	function from States to States
$\mathcal{TV}$	set of type variables
$\mathcal{T}(\Sigma)$, $\mathcal{T}_{\text{fin}}(\Sigma)$, $\mathcal{T}_{\text{reg}}(\Sigma)$	type tree, finite type tree, and regular type tree
Type_R	function from States to Types
TYPE_R	function from RelStates to RelTypes
Types	the set of types
U, V	type variables
Var	function from States $\times$ Labels to Variances
Variances	the set of variances
v	variance (element of Variances)
X	generic set
x, y	term variables
$\leq$	our subtyping relation
$\sqsubseteq$, $\sqcup$	ordering and lub on Variances

VITA

VITA

Tian Zhao was born in JingDeZhen, JiangXi Province, P.R. China, in 1974. He received a Bachelor of Science in international trade and finance from ZhongShan University in July 1994. After working for a year in a direct-investment firm, he decided to go to graduate school. He first went to Louisiana State University to study economics in 1995. One year later, he transferred to Purdue University to continue his study in economics. During the period from 1996 to 1999, he had taken courses in a variety of areas including computer science and electrical engineering. In January 1999, he formally transferred to the computer science department and started to pursue Ph.D. degree in computer science. He received a Master of Science in economics in May 1998, a Master of Science in computer science in May 1999, and a Master of Science in Electrical and Computer Engineering in May 2001. He received the degree of Doctor of Philosophy in computer science in August 2002.