This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.
Natural Type Inference

Eirini Vlassi Pandi

A dissertation submitted to the University of Edinburgh towards the degree of Doctor of Philosophy

August 2023
Abstract

Recently, dynamic language users have started to recognize the value of types in their code. To fulfil this need, many popular dynamic languages have adopted extensions that support type annotations. A prominent example is that of TypeScript which offers a module system, classes, interfaces, and an optional type system on top of JavaScript.

However, providing usable (not too verbose, or complex) types via traditional type inference is more challenging in optional type systems. Motivated by this, we redefine the goal of type inference for optionally typed languages as: infer the maximally natural and sound type, instead of the most general one. By the maximally natural and sound, we refer to a type that (1) is derivable in the type system, and (2) maximally reflects the intention of the programmer with respect to a learnt model.

We formally devise a type inference problem that aids the inference of the maximally natural type. Towards this goal, our problem asks to combine information derived from two sources: (1) from algorithmic type systems using deductive logic-based techniques; and (2) from the source code text using inductive machine learning techniques.

To tackle our formulated problem, we develop two frameworks that combine the two sources of information using mathematical optimization. In the first framework, we formulate the inference problem as a problem in numerical optimization. In the second framework, we map the inference problem into popular problems in discrete optimization: maximum satisfiability (MaxSAT) and Integer Linear Programming (ILP). Both frameworks are built to be consistent with information derived from the different sources. Moreover, through formal proofs, we validate the soundness and completeness of the developed framework for a core \(\lambda\)-calculus with named types.

To assess the efficacy of the developed frameworks, we implement them in a tool named Optyper that realizes natural type inference for TypeScript. We evaluate Optyper on TypeScript programs obtained from real world projects. By evaluating our theoretical frameworks we show that, in practice, the combination of logical and natural constraints yields a large improvement in performance over either kind of information individually. Further, we demonstrate that our frameworks out-perform state-of-the-art techniques in type inference to produce natural and sound types.
Acknowledgements

The present dissertation lies in the intersection of programming languages, software engineering, and machine learning. To work in an interdisciplinary field is not an easy task, but I got extremely lucky to get a team of exceptional supervisors for each one of these fields. I am extremely grateful and equally thankful to all of them: Andy Gordon, Earl Barr, and Charles Sutton, for patiently sharing their knowledge with all of us; to helping me growing as a researcher and supporting myself in every step and difficulty during my studies.

Thanks to the Edinburgh Microsoft Research Joint Initiative in Informatics, which has kindly and generously supported this Ph.D. through its scholarship program. The scholarship has funded my PhD studies for the first three years. It also funded my travel expenses to conferences at amazing places all over the world.

I am also grateful to Don Sannella, who supported me as an additional co-advisor, physically, in Edinburgh and together with Gordon Plotkin, served on my Ph.D. committee and gave me useful feedback in our annual meetings.

I would like to thank all my colleagues and friends from the Informatics Forum in Edinburgh, where most of this research was undertaken, and give credit to Iordanis Chatzinikolaidis who helped me gain a deeper understanding of continuous optimization.

I would also like to express my gratitude to Viktor Vafeiadis for hosting me during my internship and beyond that at the Max Plank Institute for Software Systems in Kaiserslautern. During this period I had the opportunity to work with Michalis Kolokogiannakis, who introduced me to the world of model checking, and Rajarshi Roy, with whom we had many fruitful conversations regarding discrete optimization and how it can be applied to support the thesis presented here. Overall, my ten-month stay at MPI was a transforming experience for my career, mostly because of the exceptionally friendly research community there.

Lastly, I want to thank my mother and my grandmother for supporting me throughout my life and reminding that there is always a way forward.
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Eirini Vlassi Pandi
Στὶς ἀγαπημένες μου
## Contents

List of Symbols ix

1 Introduction 1
   1.1 Context ........................................... 1
   1.2 Thesis ........................................... 4
   1.3 Contributions and Dissertation Outline ............... 6

2 Related Work 9
   2.1 Type Systems ....................................... 9
   2.2 Learning-based Programming .......................... 12
   2.3 Learning-based Type Inference ....................... 13

3 A Calculus of Named Types 17
   3.1 Problem Statement ................................... 18
   3.2 Types, Type Structures, and Type Equations .......... 21
   3.3 Definition of Type Equality ........................ 22
   3.4 Expressions, Values, and Operational Semantics .... 25
   3.5 Declarative Type System ................................ 28
   3.6 The Type Inference Problem for Function Definitions .34

4 Type Inference by Accumulating Logical Constraints 35
   4.1 Type Variables, Logical Constraints, Type Variable Valuation 36
   4.2 Algorithmic Type System ................................ 41
   4.3 Formal Properties: Termination, Soundness, and Completeness 44

5 Defining Natural Type Inference 65
   5.1 Formalizing Textual Information as Natural Constraints 65
   5.2 The Natural Type Inference Problem for Function Definitions 68
   5.3 High-level Overview of the Algorithm for Natural Type Inference 69

6 Natural Type Inference using Continuous Optimization 71
   6.1 Background: Fuzzy Logic ............................ 72
### List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>set of type variables</td>
</tr>
<tr>
<td>$\Gamma^o$</td>
<td>algorithmic type environment</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>type variable</td>
</tr>
<tr>
<td>$\text{Bin}$</td>
<td>binary matrix transformation</td>
</tr>
<tr>
<td>$C$</td>
<td>logical constraints</td>
</tr>
<tr>
<td>$E$</td>
<td>expression</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>type environment</td>
</tr>
<tr>
<td>$N$</td>
<td>total number of natural constraints</td>
</tr>
<tr>
<td>$M$</td>
<td>type valuation matrix</td>
</tr>
<tr>
<td>$\mu$</td>
<td>variable valuation</td>
</tr>
<tr>
<td>$N$</td>
<td>natural constraints</td>
</tr>
<tr>
<td>$\text{NatConstr}$</td>
<td>function that generates natural constraints</td>
</tr>
<tr>
<td>$\uparrow$</td>
<td>natural type valuation</td>
</tr>
<tr>
<td>$\text{NatVal}$</td>
<td>natural value</td>
</tr>
<tr>
<td>$\text{NC}$</td>
<td>a function that returns a probability distribution from texts to name types</td>
</tr>
<tr>
<td>$P$</td>
<td>probability matrix</td>
</tr>
<tr>
<td>$S$</td>
<td>type structure</td>
</tr>
<tr>
<td>$C$</td>
<td>set of all logical constraints</td>
</tr>
<tr>
<td>$\mathcal{V}$</td>
<td>set of identifiers</td>
</tr>
<tr>
<td>$\mathcal{M}$</td>
<td>set of all type valuation matrices</td>
</tr>
</tbody>
</table>
\( \mathcal{P} \) set of all typing probability matrices
\( \mathcal{V} \) set of (identifier, type variable) pairs
\( \Sigma \) type signature

\( \mathcal{T} \) total number of type names
\( t \) type name

\( \mathcal{V} \) total number of type variables
Introduction

To code is to name. While coding, developers must name parameters, local variables, functions, modules, and programs themselves. Often these names are meaningful. For instance, variables named `username` or `password` have different connotations, although both should have all the attributes of a type `String`. In this dissertation we explore how to extend the traditional type inference problem to one that soundly takes into account names of identifiers.

In this chapter, Section 1.1 gives the overall context of the dissertation. Section 1.2 states the thesis of the dissertation. Finally, Section 1.3 outlines each chapter along with the contributions of this dissertation.

1.1 Context

Programming languages share properties with natural languages related to their purpose as a means of communication and having a syntactic form separate from their semantics. Following this observation, the software engineering community has recently adopted the term *natural source code*, the idea of thinking of source code as natural written by humans and meant to be understood by other humans (Knuth, 1992). The structured and highly composable nature of source code data provides fertile ground for using and creating machine learning models that exploit the probabilistic reasoning capabilities that these models can offer and allow us to view existing problems in a new perspective.

Recently machine learning techniques have been applied to build practical programming tools that enhance the development process by leveraging “Big Code” (Big-Code, 2017). Just like vast amounts of data on the web enabled “Big Data” applications and altered a
number of areas, such as natural language processing and computer vision, the increased amount of publicly available source code, through code repositories, like GitHub (2020), enables a new class of applications that leverage these large codebases. People want to reuse code, and machine learning could provide the perfect framework to do that. There is already a growing interest in predictive models of source code that build on methods from machine learning and statistical natural language processing to predict the source code text that programmers will write next. By contrast, relatively little work has applied probabilistic models from machine learning to infer semantic properties of programs.

One of the numerous potential fields of application is aiming towards automatic enhancement of type inference. Although different languages use types differently, types are an important factor in virtually all programming languages. Whether a language is going to be statically or dynamically typed is an important implementation decision (Meyerovich and Rabkin, 2012), which involves a trade-off.

On one end of the spectrum, we have statically typed languages that aim to enforce correctness and safety properties on programs by guaranteeing constraints on program behaviour using types. A large scale user-study suggests that programmers benefit from type safety (Hanenberg et al., 2014); use of types has also been shown to prevent field bugs (Gao et al., 2017). However, type safety comes at a cost: these languages often require explicit type annotations, which imposes the burden of declaring and maintaining these annotations on the programmer. Strongly statically-typed, usually functional languages, like Haskell or ML, offer type inference procedures that reduce the cost of explicitly writing types but come with a steep learning curve (Tirronen et al., 2015).

On the other end of the spectrum, we have dynamically typed languages, which either lack or do not require type annotations. This attribute makes dynamic languages a popular choice over statically typed languages (Meyerovich and Rabkin, 2012). Initially designed for quick scripting or rapid prototyping, these languages have begun reaching the limits of what can be achieved without the help of type annotations, as witnessed by the heavy industrial investment in and proliferation of static type systems for these languages (TypeScript (Microsoft, 2020) and Flow (Facebook, 2019) are just two examples). Retrofit for dynamic languages, these type systems include gradual (Siek and Taha, 2006) and optional type systems (Bracha, 2004). Like classical type systems, these type systems require annotations to provide benefits.

For dynamic languages, however, the task of type inference is far more challenging. Firstly, dynamic languages have complex type features such as subtyping, structural types, and union types that contribute to the complexity of the problem. Moreover, programmers
are allowed to write more expressive programs and use libraries that cannot be precisely type checked using standard type systems. In addition, like classical type systems, these type systems require annotations to provide benefits, but this process itself turns out to be quite burdensome for the programmer. Conventional type systems aim to infer most general types, but unfortunately, in the presence of the above circumstances, the generated types are often verbose, difficult to understand, and may not reflect the programmer’s intention.

As indicated earlier, the machine learning community has identified this challenging topic and by seeing it in a new light has proposed learning-based type inference techniques. Specifically, for type inference, machine learning allows us to develop less strict type inference systems that learn to predict types from uncertain information, such as comments, names, and lexical context, even when traditional type inference procedures fail to infer a useful type.

The classic literature on conventional type systems takes great care to demonstrate that type inference only suggests sound types (Milner, 1978; Pierce, 2002). Learning-based type inference is not in conflict with classical type inference but complements it. There are settings, like TypeScript, where correct type inference is too imprecise. In these settings, learning-based type inference helps the human in the loop to move a partially typed codebase—one lacking so many type annotations that classical type inference can make little progress—to a sufficiently annotated state that classical type inference can take over and finish the job.

The closest related works to this end are JSNice (Raychev et al., 2015) and NL2Type (Malik et al., 2019), which use probabilistic graphical models to statistically infer types of identifiers in programs written in JavaScript, along with the DeepTyper tool (Hellendoorn et al., 2018), which targets TypeScript. These approaches all use machine learning models to capture the structural similarities between typed and untyped source code and extract a statistical model for the text. All of these tools are using various sources of knowledge that are typically ignored by type inference algorithms, but at the same time discard certain information arising from classical type inference procedures.

Wei et al. (2020) propose a different approach to probabilistic type inference for TypeScript. LambdaNet is based on graph neural networks (Allamanis et al., 2017), which learn type dependency graphs extracted by static code analysis on the training data. Their type dependency graph has edges that are either logical or contextual, corresponding to what we call logical or natural constraints. All the aforementioned methods are based on pure learning techniques without validating their predictions against the corresponding type
checkers for the language targeted.

TypeWriter (Pradel et al., 2020) trains a neural model to predict types based on code context and natural language information in Python source code, but on top of that validates its predictions by employing a combinatorial search strategy against a gradual type checker that helps them to rule out any inconsistent predictions. Kazerounian et al. (2021) proposed SimTyper, a type inference system for Ruby that soundly combines standard type inference with heuristics.

Recently, Allamanis et al. (2020) introduced Typilus, which removes the constraint of predicting types within a fixed size vocabulary and enables prediction for user-defined and rare types. Mir et al. (2022) with their tool Type4Py proposed an innovative $k$-nearest neighbour search that enables to scale the size of the type vocabulary up to 40k types. (See Section 2.3 for an extensive discussion of learning-based type inference systems.)

### 1.2 Thesis

As discussed, there is a great amount of interest towards a new kind of type inference procedure that aims to combine information from conventional type systems and learning procedures. Undoubtedly, inference procedures that are learning-based but at the same time produce sound typings are more useful for the programmer. In this dissertation we
focus our attention on these type inference procedures, which we refer to as natural type inference. The aim is now to infer the maximally natural type, in the sense that is both sound and maximally influenced by a learning procedure. This dissertation formalizes this problem for a small λ calculus, proposes novel solutions to solve it via optimization methods, provide theoretical guarantees, and evaluates our approach.

Overall, we present the following thesis:

“Conventional type systems can be formally extended to take into account statistical information in a sound way and infer the maximally natural type. The natural type inference problem is an NP-hard problem, that can be solved optimally by reducing it to an integer linear programming (ILP) problem or any other equivalent problem, such as maximum satisfiability (MaxSAT).”

To substantiate this thesis, we present a novel algorithm for type inference that assigns sound and natural types to identifiers. To ensure that our types are both sound and natural, we propose exploiting two types of constraints on the type variables. The first type, which we refer to as logical constraints, is the constraints deduced by classical type inference algorithms. In particular, they are obtained by accumulating and propagating constraints on the type variables, in the spirit of Milner’s Algorithm W (Milner, 1978). The other type, which we refer to as natural constraints, is the information produced by machine learning procedures on the identifiers. This type of information refers generically to indirect, statistical constraints about types. In this work, we focus our attention on a specific kind of natural constraint, where each identifier name maps to a probability vector over nominal types (Section 5.1).

We define natural type inference as the problem of inferring a type assignment that satisfies all the logical constraints and is as natural as possible based on the natural constraints. To do so, we describe an algorithm to extract both logical and natural constraints from the input program, and then to combine them to produce the maximally natural type assignment. To effectively combine the two types of constraints, we encode the inference problem as an optimization problem. We explore different ways to define and solve this optimization problem, both in the continuous and the discrete space. Our formulation allows us to use off the shelf algorithms to effectively solve our problem. We prove that our proposed discrete solution is optimal. We evaluate our approach in a real-world scenario, by implementing an end-to-end application, called Optyper, which aims to suggest missing types for TypeScript files.

Next, we show in a minimal formal setting how natural type inference can help to provide more intuitive types to the programmer.
1.3 Contributions and Dissertation Outline

The main contributions of this dissertation is to define and formalize the problem of natural type inference, present two algorithms to solve it, and prove termination, correctness, and optimality for both of them. Our specific algorithm deals with ambiguities arising from overloading, dot-notation, and structural equality of type names.

Here is the outline of the dissertation

Chapter 2 details on the related work.

Section 7.3 defines an exemplary type inference task as finding a type signature for an untyped function definition within a λ-calculus, whose types are defined by a global set of equations between type names and scalar, record, and function types. The operational semantics and type system satisfy preservation and progress properties: Theorem 3.2 and Theorem 3.1.

Chapter 4 presents a new algorithmic type system that given an expression yields logical constraints (Section 4.1). The algorithm is terminating (Theorem 4.1) and the logical constraints are sound and complete with respect to the declarative type system (Theorem 4.2). Corollary 4.1 is that our overall task, finding a type signature for an untyped function definition, is equivalent to satisfying the logical constraint extracted from the function definition.

Chapter 5 describes a procedure of exploiting deep learning to extract natural language information from source code, or the so-called natural constraints (Section 5.1). This is followed by the formal presentation of the problem of natural type inference (Problem 5.1) that asks to infer a type that satisfies the logical constraints and maximizes the “value” of the satisfied natural constraints.

Chapter 6 shows how to combine a relaxation of the logical constraints (Section 6.2) with probability distributions over the library of types to form a continuous constrained optimization problem (Section 6.4). Theorem 6.1 relates the logical semantics and its relaxations. By Theorem 6.2, if the optimizer terminates successfully, we get the optimal solution to the natural constraints that satisfies the logical constraints.

Chapter 7 presents an efficient reduction of the natural type inference problem into two popular problems in discrete optimization, namely maximum satisfiability (MaxSAT) and Integer Linear Programming (ILP). Such reductions allow us to leverage industrial MaxSAT and ILP solvers to infer sound types. We prove that both reductions
provide us with sound and complete algorithms for type inference in Theorem 7.1 and Theorem 7.2. Finally, we prove that our reductions are optimal, by proving the \( \text{NP} \)-hardness of the natural type inference problem using Theorem 7.3.

Chapter 8 presents a realization of our algorithms for natural type inference for TypeScript, named Optyper. We investigate its ability to infer types on several real world programs obtained from GitHub repositories. We deduce that combining logical and natural constraints yields a large improvement in performance over either natural or logical constraints individually. Further, by comparing against state-of-the-art techniques LamdaNet and TypeWriter, we assert Optyper’s ability to derive sound and natural types within reasonable time.

Chapter 9 summarises the conclusions that we draw from this scientific study and suggests possible next steps for future work.

For the reader’s convenience we provide relevant background knowledge within the appropriate chapter.
2

Related Work

Natural type inference is a new approach on learning-based type inference that optimizes over both logical and natural constraints. Related work spans across classical, deterministic type inference Section 2.1, some key works on the more general topic of machine learning for source code. Section 2.3, and earlier machine learning approaches Section 2.3.

2.1 Type Systems

Although different languages use types differently, types are an important factor in virtually all programming languages. The main purpose of types and of their corresponding type system is to verify and enforce a level of correctness in programs by guaranteeing certain constraints on the program’s behavior. Types have other purposes as well, for instance enabling enhanced compiler optimization and warnings, providing forms of documentation, decompilation or deobfuscation and improving code readability. Whether a language is going to be statically or dynamically typed (type-checked at compile time or run time respectively) is an important implementation decision, which involves a trade-off. Although statically typed languages (such as C and Java) are type-safe, their requirement of explicit types annotations comes with an additional cost for the programmer, who has to write and maintain them. It is perhaps partly for this reason that several popular languages, like Python and JavaScript, do not include static type annotations as part of the language. This lack of static type enforcement means programmers can write code that is verbose and flexible, and that takes advantage of highly dynamic features such as metaprogramming. As a result, these type of languages were used initially for rapid prototyping and development of programs. Type inference, that is the automatic deduction of the type of an expression at compile time, can provide the best of both worlds. This feature is an
inherent trait of strongly, statically-typed, functional languages (like Haskell or ML). In this direction, some procedural languages attempt to include type inference as a feature. For instance, in C++ programmers can use the auto keyword to avoid writing the type in the definition of a variable with an explicit initialization, while in C# (starting with version 3) the var keyword can be used as a convenient syntactic sugar for shorter local variable declarations. Nevertheless, C# is still a statically typed language. These enhancements are implemented via compiler tricks and thus are considered as a small step towards a world of static typing where possible, and dynamic typing when needed.

The quest for more modular and extensible static analysis techniques has resulted in the development of richer type systems. Refinement types, that is subsets of types that satisfy a logical predicate (like boolean expression), constrain the set of values described by the type, and hence allow the use of modern logic solvers (such as SAT and SMT engines) to dramatically extend the scope of invariants that can be statically verified. An implementation of this concept comes with Logically Qualified Data Types, abbreviated to Liquid Types. DSOLVE is an early application of liquid type inference in OCAML (Rondon et al., 2008). A type-checking algorithm, which relies on an SMT solver to compute subtyping efficiently for a core, first order functional language enhanced with refinement types (Bierman et al., 2012), provides a different approach. LiquidHaskell (Vazou et al., 2014) is a static verifier of Haskell based on Liquid Types via SMT and predicate abstraction. The expressive type system of functional languages made the task of adding refinement types easier to achieve. For instance, these languages take as primitive the useful idea of data tagged with data constructors by providing kinds, that is, a type of a type, and algebraic types as built-in notion. Although modern scripting languages have popularized the use of higher-order constructs, attempts to apply refinement typing to scripts have mostly proven to be impractical (Chugh et al., 2012). Unfortunately, as each programming language has developed its own characteristics, all of the above solutions are tied to a specific language. However, richer type annotation holds the promise of a more precise, modular and extensible analysis, and, as the need of building programs that conform the specifications emerges we should therefore search for novel and universally applicable solutions in integrating static and dynamic typing and thereby combining the benefits of both typing disciplines.

**Gradual vs. Optional Type Systems**

Towards this pursuit, recently dynamic languages have also started to pay more attention to typings. This, has resulted in new kinds of type systems. The most two prominent
examples are gradual and optional type systems, which we discuss briefly below.

Several JavaScript extensions (like Closure Compiler (Google, 2019) and TypeScript (Microsoft, 2020)) add optional type annotations to program variables. In JavaScript, these annotations are provided by specially formatted comments known as JSDoc. However, these extensions often fail to scale to realistic programs that make use of dynamic evaluation and complex libraries (for example jQuery), which cannot be analyzed precisely (Jensen et al., 2009). In line with this approach we have seen extensions for other popular scripting languages, for instance Python community introduced PEP 484 (Rossum et al., 2014), which adds optional static typing to Python 3.5 and newer, or RuboCop (Bastov, 2018b), which serves as a static analyzer for Ruby by enforcing many of the guidelines outlined in the community Ruby Style Guide (Bastov, 2018a) and performing various check types known as cops.

A gradual language is one that obeys the gradual guarantee. Informally, the gradual guarantee states that, if we change the types in a gradual program to be “more precise” that is, we replace an any type with an integer or function signature — the changed program either has the same behaviour as the original or raises a dynamic type error. Gradual languages accomplish this by injecting runtime type casts where an untyped, ‘any’ value may flow into a typed variable (Siek et al., 2015). In other words, gradual languages recover type soundness at runtime with casts. These casts are costly, as chronicled in “Is Sound Gradual Typing Dead?” (Takikawa et al., 2016). In response, academic research has centred on compiler optimizations aimed at removing or consolidating casts.

Optional types are industry’s response to the cost of graduality. TypeScript is a pioneer and perhaps remains the most prominent example. An optionally typed language can be viewed as a gradual language that skips runtime type casts. By design, optionally typed languages are unsound: they do not prevent operations be applied to operands with incorrect types. Instead, optional languages provide the static benefits of gradual typing, that is, improved navigation, documentation, and erroring on local type inconsistencies, without its runtime cost.

Perhaps because of its industrial origin, the fact that optionally typed languages intentionally violate soundness, and the concomitant lack of publications, academic computer science conflates gradual and optional typing. Indeed, Siek et al. (2015) proposed the gradual guarantee in an effort to clarify the meaning of gradual typing. Apparently, these type systems gain popularity among the programmers (Di Grazia and Pradel, 2022), so we believe that exploring this topic further is a promising research direction.
2.2 Learning-based Programming

In this section, we will outline the area of learning-based source code models aiming to give an intuition along with a rough overview of the whole area.

Programming languages share properties with natural language related to their purpose as a means of communication and having a syntactic form separate from its semantics. Based on this observation Hindle et al. (2012) proposed the naturalness hypothesis for software. "Programming languages, in theory, are complex, flexible and powerful, but the programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks."

In line with this observation, the software engineering community has recently adopted the term natural source code, that is the idea of thinking of source code as natural written by humans and meant to be understood by other humans (Hindle et al., 2012). The structured and highly composable nature of source code data provides fertile ground for using and creating machine learning models that exploit the probabilistic reasoning capabilities that these models can offer and allow us to view existing problems in a new perspective.

Although the interdisciplinary field between machine learning and programming languages is quite young, some complete reviews of this emerging area are already available. Allamanis et al. (2018) in their survey give an extensive synopsis of works that model source code in a probabilistic way by containing a learning component and using complex representations of the underlying code. A detailed description of the area is also given by Vechev and Yahav (2016) in their related article Programming with “Big Code”. While Gottschlich et al. (2018), in their position paper, examine the research area by categorising the class of the challenges involved in three main, overlapping pillars: intention, invention, and adaptation. Additionally, there has been an effort from the community to gather resources, datasets and code in a single website (http://learnbigcode.github.io/).

A major subfield of this area concerns about predictive models of source code that build on methods from machine learning and statistical natural language processing (NLP) aiming to predict the source code text that programmers will write next. Program Synthesis, could be considered as taking the problem of predicting code to the extreme. Although this is a rather hard problem, some of the most popular machine learning applications sprung from this idea. For example, the FlashFill by Gulwani (2011) tool is robust enough to being shipped as part of a commercial product (for example Excel) and thus being used
daily from thousands of users. More recently, the machine learning community has been exploring similar ideas to a broader context by willing to capture more general program structures. Pursuant to this Graves et al. (2014) worked to encode differentiable Turing Machines while Cai et al. (2017) tried to incorporate the notion of recursion directly into the neural network. By combining the two lines of research, the DeepCoder tool uses input-output examples as a training data to a deep network that learns to predict properties of the intermediate program (Balog et al., 2016). Predictive models of source code have been applied for other tasks such as prediction or autocomplete of next token or a phrase in a program, to suggesting names that better follow naming conventions (Allamanis et al., 2014), and JSNice for deminifying JavaScript (Raychev et al., 2015). Most recently (Bavishi et al., 2017) improved the results of JSNice by combining a lightweight static analysis, an auto-encoder neural network and a RNN into a single tool.

A different subfield of learning-based programming aims different kind of program analysis tasks. Program analysis aims to provide guarantees about properties of the code such as correctness or safety. For such tasks most commonly formal methods are being recruited to generate a set of constraints that has to be respected by the program. These hard constraints emerge new challenges for the probabilistic world of machine learning; as there is not a trivial way to incorporate them along with the logical rules that they should follow to a probabilistic model. However, the task of predicting semantic properties, such as types, could alleviate the programmer from the burden of explicitly annotating the code while also support program verification and bug finding at no additional cost. The aforementioned task is indeed the main goal of this dissertation, as discussed in Chapter 1. In the following section, we survey closely related works in more detail.

2.3 Learning-based Type Inference

Learning-based type inference is a flourishing research area for at least two reasons. First, recent breakthroughs in machine learning (ML) have enabled researchers to apply ML techniques to effectively predict type annotations for dynamic programming languages, whose dynamism has stymied traditional type inference. Second, popular dynamic languages are adopting optional type annotations, generating the data ML needs in abundance in the form of huge, publicly available repositories of code\(^1\). For instance, Python introduced optional typing in PEP 484 (Rossum et al., 2014), or equivalent TypeScript (Microsoft, 2020), which adds optional type definitions to JavaScript.

\(^1\)These repositories are the oil fields of the digital economy, if we believe that “Data is the new oil.”, as Clive Humby observed (Arthur, 2013).
This section surveys work in learning-based type inference starting with the pioneering work using graphical models, before considering sequence-to-sequence, including the latest hierarchical model, and graphical neural networks. It discusses sound approaches, then closes by observing the lack of a formal treatment, a gap we fill in this work.

The pioneering work in this area builds probabilistic graphical models from structures extracted from source code. JSNice (Raychev et al., 2015) takes JavaScript code, extracts its abstract syntax tree, and converts the AST into a conditional random field (CRF) (Sutton et al., 2012). JSNice employs Maximum a Posteriori (MAP) inference over its CRF to predict both names and types. Its predictions are unsound. Indeed, the authors state “where soundness is required, the approach presented here will have value as part of a guess-and-check loop”. Xu et al. also construct a graphical model, in the form of a factor graph (Xu et al., 2016). This work requires heuristically chosen weights in the factors that integrate logical and natural constraints. Both works formalize the construction of their model and validate their predictions empirically by reporting precision and recall over a corpus.

Given sufficient training data, neural approaches learn features themselves, obviating manual feature identification and extraction as well as the realization of heuristics to process them. DeepTyper (Hellendoorn et al., 2018) was the first to use a sequence-to-sequence model to predict types for TypeScript. Its core idea is to train a neural model on an aligned corpus of TypeScript and JavaScript code. DeepTyper consumes its training data as a raw token stream. As such, this stream implicitly combines the logical and natural constraints embedded within it; DeepTyper itself must learn to distinguish and exploit both. DeepTyper formalizes its neural architecture and is unsound, sometimes predicting multiple types for a variable across its uses in a single scope. Like its predecessors, DeepTyper is empirically compared to JSNice using accuracy from information retrieval. NL2Type is trained solely on function signatures and JSDoc comments (Malik et al., 2019). NL2Type formally defines its features and how it constructs training data to expose them to its neural network; its evaluation is empirical, reported using information retrieval measures and focused on how it can complement JSNice. Type4Py employs an innovative $k$-nearest neighbour search that enables to scale the size of the type vocabulary up to 40k types. Its training data includes existing type annotations and, taking a page from Typilus below, it employs triplet loss to advance the state of the art (Mir et al., 2022). Type4Py tersely describes its model and validates its performance empirically. Like DeepTyper, NL2Type and Type4Py are both unsound.

Because source code is inherently graph-structured, graph neural networks (GNN) are a
natural architecture for learning-based type inference (Gilmer et al., 2017; Allamanis et al., 2018). Targeting TypeScript, LambdaNet was the first to use a GNN for type inference; it applies static analysis to its training data to build its model’s graph. LambdaNet combines logical and contextual (which includes natural) constraints. It is the first approach to effectively predict user-defined types not encountered during training by using a pointer-network-like architecture (Vinyals et al., 2015; Allamanis et al., 2016) over an open vocabulary. LambdaNet’s architecture explicitly models a fixed set of type constraints as hypergraphs. Typilus (Allamanis et al., 2020) is a GNN that employs metalearning using triplet loss to predict an open vocabulary of types for Python, including rare, even unseen-during-training, user-defined types. Both GNN models use an iterative computation called message passing to compute predictions, which is closely related to the sum-product algorithm that Xu et al. (2016) use. Even LambdaNet, which explicitly models some type constraints, does not enforce them over its predictions. Indeed, in practice, we observe that LambdaNet, despite explicitly modeling logical type constraints, produces annotations that do not respect the learnt logical relationships. In short, both LambdaNet and Typilus are unsound. Both formally detail their models, which are their core contributions. As is conventional in this space, both are evaluated empirically over code corpora and the results are reported using accuracy.

Williams et al. (2020) present an algorithm to infer unit types for numbers in spreadsheets cells. They first generate logical constraints—sets of equations—on unit types by analyzing formulas and format information (such as currencies or percentages). Relying on a method due to Orchard et al. (2015), they transform the constraints to linear equations and solve by matrix reduction, to obtain a set of unconstrained critical variables, which amount to the most general unit typing. Rather than present spreadsheet users with unknown variables, they use textual information such as column headers or labels on cells together with a pre-trained language model to predict the most likely concrete units for the critical variables (and hence the numeric cells in the workbook). They formalize, but do not state or prove theorems about, their algorithm. Their evaluation is empirical, reported in terms of the usual suspects of information retrieval measures.

None of the approaches covered so far, whether graphical or neural, explicitly model the underlying type inference rules, so their predictions miss useful type constraints. Recognising this problem, researchers proposed TypeWriter and SimTyper. TypeWriter (Pradel et al., 2020) realizes the guess-and-check integration of prediction with type checking proposed by JSNice. Targeting Python, TypeWriter enumerates the top-ranked predictions from a neural type predictor, then invokes Python’s gradual type checker mypy (The-Mypy-Project, 2014) to filter out those that do not type-check. As the authors claim, TypeWriter
is soundy (Livshits et al., 2015) up to a fixed type context (as usual, changing a module’s context may invalidate a correct prediction made in another context). Because of its reliance on mypy, TypeWriter is not, however, sound by construction. The authors formalize TypeWriter’s model and present its guess-check-and-backtrack procedure in pseudocode and report its performance using information retrieval measures. SimTyper (Kazerounian et al., 2021) builds on a previous system, InferDL (Kazerounian et al., 2020), which adds heuristics to DRuby (Furr et al., 2009), a type inference system for Ruby, based on type inclusion constraints (Aiken and Wimmers, 1993). Following InferDL, SimTyper uses a guess-check-and-backtrack approach to find readable nominal types instead of hard-to-read structural types. It extracts constraints from its input program, and simplifies them. If any type variable is assigned a complex structural type, SimTyper attempts to generalize to a more readable nominal type, while maintaining soundness. Candidate nominal types are all classes that subsume the inferred structural type. SimTyper uses a transformer model to rank these candidates. For each candidate in order, it adds the equality constraint that equates the candidate and the inferred complex structural type to its constraint set. If the resulting constraints are not sound, SimTyper backtracks and repeats, up to a bound. SimTyper’s results are reported for various relaxed matches using precision and recall.

In short, both TypeWriter and SimTyper achieve natural type inference in phases that separately handle logical and natural constraints. This is less efficient than the approach we present here that combines logical constraints and natural constraints into a single joint optimization problem. Their separate handling of natural and logical constraints also means that they do not formalize their approaches from the ground up, and prove their correctness and termination, as we do in this dissertation (see Section 5.2).
A Calculus of Named Types

This chapter paves the way for the remainder of the dissertation by introducing a \(\lambda\)-calculus with named types to formalize and verify natural type inference. We describe its syntax, operational semantics, and type system, and state its basic metatheory. Given these definitions, we can frame the type inference problem to be solved by algorithms in later chapters.

Section 3.1 introduces the problem, along with some examples and shows that natural type inference resolves ambiguities arising from dot-notation, from overloading of arithmetic operators, and from type aliases for structurally equivalent types. Our formalism represents each of these ambiguities.

Section 3.2 defines the structure of types and definitional type equations. A type \(t\) is simply a name drawn from a finite set \(\mathcal{T}\) of type names (such as \(\text{Int}\) or \(\text{IntArray}\)). Each type name \(t\) is defined by a type equation \(\text{type } t = S\) where \(S\) is a type structure, which could be either a base, record, or function type (see Definition 3.1). For instance, the type equation \(\text{type } \text{Range} = \{\text{length} : \text{Int}, \text{breadth} : \text{Int}\}\) defines the type \(\text{Range}\) to have the type structure \(\{\text{length} : \text{Int}, \text{breadth} : \text{Int}\}\). Recursive types may be defined by recursive type equations.

Section 3.3 defines equivalence between types as a bisimulation relation, following Brandt and Henglein (1998). Proposition 3.1 is that type equivalence is decidable, while Definition 3.4 defines type equality between a type name \(t\) and a type structure \(S\).

Section 3.4 defines the syntax of expressions and values, along with the operational semantics of the core language.

The core judgment of our type system is a type assignment relation that determines when
an expression $E$ may be assigned a type name $t$. Unusually for a $\lambda$-calculus, our formalism assigns type names $t$ but not type structures $S$ to expressions, which allows us to easily limit the solutions to our constraints to a finite universe of named types. Still, this aspect of our formalism does not limit which expressions may be typed, because we can introduce a new type name for any desired type structure.

Section 3.5 characterises the type assignment via two separate inductive definitions that yield the same relation. The two definitions differ in how they introduce type equality. In the first, type equality appears only in a single retyping rule (akin to a subsumption rule in a system with subtyping). For this first system, we state and prove progress (Theorem 3.1) and preservation (Theorem 3.2) theorems, along with standard . In the second syntax-directed system, type equality appears in each of the rules. Proposition 3.3 shows the two definitions are equivalent.

Finally, Section 3.6 states the type inference problem for the calculus.

### 3.1 Problem Statement

Names can also mislead, when poorly chosen or when the program has evolved to use them quite differently to when they were first given. This is why analyses often ignore names and treat them just as nonces that tie a definition to a set of uses. This approach can, however, be too conservative.

Consider the standard problem of type inference for a dynamic language (such as JavaScript or Python). Specifically, consider the definition of a function $f$ that has formal arguments $x_1, \ldots, x_n$, and whose result is determined by the untyped expression $E$ with $fv(E) \subseteq x_1, \ldots, x_n$:

$$\text{function } f(x_1, \ldots, x_n) \text{ return } E$$

Although $E$ is untyped (like JavaScript), we assume that there is a type assignment relation for untyped expressions (like the one for TypeScript (Microsoft, 2020)), and a library of types $t_i$ available for typing expressions. Given this input, type inference is the task of computing a type signature for $f$, that is, a tuple of pairs $(x_1 : t_1, \ldots, x_n : t_n, f : t)$, such that the following is derivable in the type assignment relation for expressions:

$$x_1 : t_1, \ldots, x_n : t_n \vdash E : t$$

There may be many valid type signatures for the same untyped input, creating a challenge for type inference: which valid signature to return? An algorithm for natural type inference
is one that exploits natural language information, such as the identifiers $x_1, \ldots, x_n, f$ and other lexical information in $E$, when selecting which valid type signature to return.

**Three Example Functions Each With Multiple Type Signatures**

Here are three examples of untyped function definitions, to illustrate some of the sources of ambiguity that can be resolved by natural type inference.

```plaintext
function uppercase(str) return // 1st
  {length = str.length;
   items = \(i\) let char = str.items(i) in char < 97 ? char : char < 123 ? char − 32 : char}
function diffRange(range1, range2) return range1.length − range2.length // 2nd
function intEqual3(int1, int2, int3) return int1 == int2 ? int2 == int3 : false // 3rd
```

Consider the following library of type definitions.

```plaintext
type Char = Int
type String = \{length : Int, items : Int → Char\}
type IntArray = \{length : Int, items : Int → Int\}
type Range = \{length : Int, breadth : Int\}
```

We work in a $\lambda$-calculus with scalar types $Bool$ and $Int$, record, and function types, and a type system where equivalence of named types is determined by structure, as in TypeScript, and not by name. For instance, although the types $String$ and $IntArray$ are syntactically different (because they are different type names), they are structurally equivalent (because they have the same structure, if we ignore the names). Note that the above type library is a shorthand for the full type library properly defined in Definition 3.2.

A function definition may have multiple type signatures for several reasons:

(1) Two different named types may actually be structurally equivalent. For example, $String$ and $IntArray$ are structurally equivalent. A function such as $uppercase$ can be typed using either of these types, resulting in syntactically distinct but structurally equivalent type signatures: for example,

```plaintext
(str : String, uppercase : String)
versus (str : IntArray, uppercase : IntArray)
versus (str : IntArray, uppercase : String).
```
(2) Differently structured types may share the same field. For example, \textit{String}, \textit{IntArray}, and \textit{Range} share the field name \textit{length}. A function such as \textit{diffRange} can be typed using any of these types, resulting in syntactically and structurally distinct type signatures: for example,

\[
\text{(range1 : Range, range2 : Range, diffRange : Int)}
\]

\textit{versus}

\[
\text{(range1 : String, range2 : IntArray, diffRange : Int)}
\]

\textit{versus}

\[
\text{(range1 : IntArray, range2 : IntArray, diffRange : Char)}.
\]

(3) A primitive operator such as equality == is overloaded over multiple primitive types, such as \textit{Bool} and \textit{Int}. A function such as \textit{intEqual3} can be typed using either of these types, resulting in syntactically and structurally distinct type signatures: for example,

\[
\text{(int1 : Int, int2 : Int, int3 : Int, intEqual3 : Bool)}
\]

\textit{versus}

\[
\text{(int1 : Bool, int2 : Bool, int3 : Bool, intEqual3 : Bool)}
\]

\textit{versus}

\[
\text{(int1 : Char, int2 : Char, int3 : Char, intEqual3 : Bool)}.
\]

These examples of type ambiguity are not purely academic: similar issues arise directly in TypeScript. For example, when given the second example, TypeScript infers the uninformative intersection \textit{string | any[]} for both \textit{range1} and \textit{range2} and, when given the third example, TypeScript infers \textit{any} for each parameter, which does not help neither the programmer nor the type system.

In many previous papers, the goal of type inference is to find a unique most general type, perhaps polymorphic, for an untyped program. For example, if we supported subtyping and the declared \textbf{type} \textit{HasIntLength} = \{\textit{length : Int}\}, the \textit{diffRange} example could be assigned the signature \textit{(range1 : HasIntLength, range2 : HasIntLength, diffRange : Int)}. Curried function types provide a way of defining and using functions that accept multiple arguments, but only return a single value. Thinking of signatures as such, this signature is a supertype of either \textit{(range1 : Range, range2 : Range, diffRange : Int)} or \textit{(range1 : String, range2 : IntArray, diffRange : Int)}, the two signatures mentioned for \textit{diffRange}.

Our perspective is that sometimes a programmer will prefer a less general but more natural type; for example, because they want to hide the implementation details of how a length is maintained, and instead wish to use domain-relevant words like \textit{Range} in their signatures.

For example, given \textit{diffRange} (see above) as input, what is the type signature \textit{(range1 : t1, range2 : t2, diffRange : t)} to output?
Our type inference algorithm generates the following logical and natural constraints.

- The logical constraints, after simplification, are that \((\text{range1} : t_1, \text{range2} : t_2, \text{diffRange} : t)\) is a type signature for \(\text{diffRange}\) if and only if (1) \(t_1 = \text{String}\) or \(t_1 = \text{IntArray}\) or \(t_1 = \text{Range}\), (2) \(t_2 = \text{String}\) or \(t_2 = \text{IntArray}\) or \(t_2 = \text{Range}\), and (3) \(t = \text{Int}\). A detailed description of how we generate these constraints, can be found in Section 4.3.

- In a simplified form the natural constraints eventually map each of the identifiers \(\{\text{range1}, \text{range2}, \text{diffRange}\}\) to a probability vector over possible types. The formal definition can be found in Section 5.1.

In the context of the library types \(\text{Char}, \text{String}, \text{IntArray},\) and \(\text{Range}\), and the builtin types \(\text{Int}\) and \(\text{Bool}\), the natural constraints bias the choice of both \(t_1\) and \(t_2\) to be \(\text{Range}\), consistent with logical constraints (1) and (2). The natural constraint \((\text{diffRange}, t)\) biases the choice of \(t\) to be \(\text{Range}\), but this violates the logical constraint (3), which takes priority.

In all, we get that:

- The algorithm assigns \(\text{diffRange}\) the type signature

  \((\text{range1} : \text{Range}, \text{range2} : \text{Range}, \text{diffRange} : \text{Int})\).

Similarly, on our other examples we get as the maximally natural type signature:

- The algorithm assigns \(\text{uppercase}\) the type signature

  \((\text{str} : \text{String}, \text{uppercase} : \text{String})\).

- The algorithm assigns \(\text{intEqual3}\) the type signature

  \((\text{int1} : \text{Int}, \text{int2} : \text{Int}, \text{int3} : \text{Int}, \text{intEqual3} : \text{Bool})\).

### 3.2 Types, Type Structures, and Type Equations

In this section we give the formal definitions of types \(t\) and type structures \(S\).

**Definition 3.1 (Types).**

\[
\begin{align*}
  t & \quad \text{type (drawn from a finite set of names, including \text{Bool} and \text{Int})} \\
  t & ::= \text{Bool} \mid \text{Int} \quad \text{base type} \\
  \ell & \quad \text{label for field in a record}
\end{align*}
\]
Each type \( t \) has a unique type equation: \textbf{type} \( t = S \)

In particular, each base type \( \iota \) has the type equation: \textbf{type} \( \iota = \iota \)

Our core syntax does not allow for a type alias \textbf{type} \( t_1 = t_2 \), and nor does it allow nested type structures such as \( S_1 \rightarrow S_2 \) or \( \{ \ell_i : S_i \}_{i \in 1..n} \). However, we can interpret these as shorthands for the core syntax. Given the definition \textbf{type} \( t_2 = S \) for \( t_2 \), we can interpret the alias \textbf{type} \( t_1 = t_2 \) as being short for \textbf{type} \( t_1 = S \). We can interpret \textbf{type} \( t = S_1 \rightarrow S_2 \) as meaning \textbf{type} \( t = t_1 \rightarrow t_2 \) where \( t_1 \) and \( t_2 \) are fresh type names defined by \textbf{type} \( t_1 = S_1 \) and \textbf{type} \( t_2 = S_2 \).

Here is the library in the core syntax, where we have introduced the \texttt{Int2Char} and \texttt{Int2Int} as names for the function types used in our motivating examples (Section 3.1).

\begin{verbatim}
Definition 3.2 (Type Library).

\textbf{type} \( \texttt{Bool} = \texttt{Bool} \)
\textbf{type} \( \texttt{Int} = \texttt{Int} \)
\textbf{type} \( \texttt{Char} = \texttt{Int} \)
\textbf{type} \( \texttt{Int2Char} = \texttt{Int} \rightarrow \texttt{Char} \)
\textbf{type} \( \texttt{Int2Int} = \texttt{Int} \rightarrow \texttt{Int} \)
\textbf{type} \( \texttt{String} = \{ \text{length} : \texttt{Int}, \text{items} : \texttt{Int2Char} \} \)
\textbf{type} \( \texttt{IntArray} = \{ \text{length} : \texttt{Int}, \text{items} : \texttt{Int2Int} \} \)
\textbf{type} \( \texttt{Range} = \{ \text{length} : \texttt{Int}, \text{breadth} : \texttt{Int} \} \)
\end{verbatim}

3.3 Definition of Type Equality

This section explains our standard construction of type equality, and also illustrates that our formalism supports recursive types. Although recursive types are not needed for the motivating examples in Section 3.1, they arise naturally from our formalism and in practice.

Let \textit{type equality}, \( t \leftrightarrow t' \), hold between two types \( t \) and \( t' \) that have the same structure, disregarding names. We define type equality as a bisimulation relation, following Brandt
and Henglein (1998). It is a decidable equivalence relation. Amadio and Cardelli (1993) define recursive types as types defined by recursive type equations. Here is an example from their paper, the recursive type \texttt{Cell} of an integer-containing memory cell:

\begin{verbatim}
  type Unit = {}
  type Read = Unit \rightarrow Int
  type Write = Int \rightarrow Cell
  type Add = Cell \rightarrow Cell
  type Cell = {read : Read, write : Write, add : Add}
\end{verbatim}

Type equality in our formalism is by structure rather than by name. According to Amadio and Cardelli (1993), Algol 68 was the first language based on structural type equality in the presence of recursive types. Intuitively, the abstract structure of a type is a potentially infinite tree induced by the nested unfolding of its definition. Two types are equal if their abstract structures are equal.

Building on the literature on subtyping recursive types, we formalize equality of two types’ abstract structure as a co-inductive bisimulation relation (Milner, 1989; Gordon, 1994), following Brandt and Henglein (1998).

**Definition 3.3 (Simulation, Bisimulation, and Type Equality).**

- A binary relation on types $R$ is a \textit{simulation} if and only if:
  1. whenever $t R t'$ and \texttt{type} $t' = \iota$, then \texttt{type} $t = \iota$;
  2. whenever $t R t'$ and \texttt{type} $t' = \{\ell_i : t'_i \in 1..n\}$, there are $t_i$ with \texttt{type} $t = \{\ell_i : t_i \in 1..n\}$ and $t_i R t'_i$ for each $i \in 1..n$;
  3. whenever $t R t'$ and \texttt{type} $t' = t'_1 \rightarrow t'_2$ there are $t_1$, $t_2$ such that \texttt{type} $t = t_1 \rightarrow t_2$ and $t'_1 R t_1$ and $t_2 R t'_2$.

- A relation $R$ is a \textit{bisimulation} if and only if both $R$ and its converse $R^{-1}$ are simulations.

- The type equality relation $<:>$ is the union of all bisimulations.

The following holds by standard constructions (Milner, 1989).

**Lemma 3.1.** Type equality is reflexive, symmetric, and transitive, and is the largest bisimulation.

\begin{proof}
Using standard arguments by constructing bisimulations.
\end{proof}

**Proposition 3.1.** Type equality $t <:> t'$ is decidable.
Proof. As a corollary of Lemma 3.1, it follows that $t <: t'$ if and only if there is a bisimulation $R$ such that $t R t'$. Since there is a finite number of types, there is a finite number of bisimulations. Hence, equality of two types $t$ and $t'$ can be decided in principle by enumerating all the bisimulations in search of the pair $(t, t')$. \hfill \Box

To check type equality in practice, we can adapt existing algorithms for typing and subtyping recursive types to our system (Amadio and Cardelli, 1993; Brandt and Henglein, 1998).

To exemplify reasoning with bisimulations, we show that the types from Chapter 1 partition into five equivalence classes: \{\text{Bool}\}, \{\text{Int, Char}\}, \{\text{Int2Char, Int2Int}\}, \{\text{String, IntArray}\}, and \{\text{Range}\}.

To check that the pairs of types in these classes are equal, consider this relation.

$$R \triangleq \{(\text{Int, Char}), (\text{Int2Char, Int2Int}), (\text{String, IntArray})\}$$

We can see that both $R$ and $R^{-1}$ are simulations, and therefore that $R$ is a bisimulation. By Lemma 3.1, $<:>$ is the largest bisimulation and therefore $R \subseteq <:>$. It follows that $\text{Int} <: \text{Char}, \text{Int2Char} <: \text{Int2Int}$ and $\text{String} <: \text{IntArray}$.

Conversely, we can easily check that types from each of the five equivalence classes cannot be in a bisimulation with types from any of the others, and therefore cannot be equal.

Most presentations of $\lambda$-calculi with recursive types use the notation $\mu t.S$ for the recursive type defined by the equation $\text{type } t = S$. Patrignani et al. (2021) include a comprehensive survey. We do not use the $\mu t.S$ notation because our goal is an explicit formalism of named types defined by type equations.

Up to type equality, our example types partition into five type equivalence classes: \{\text{Bool}\}, \{\text{Int, Char}\}, \{\text{Int2Char, Int2Int}\}, \{\text{String, IntArray}\}, and \{\text{Range}\}.

Additionally, we adopt some type-specific notations for constraints. To do so, we introduce a relation $t <: S$ meaning that the type $t$ equals any other with the structure $S$.

**Definition 3.4 (Type Equality).** Let the relation $t <: S$ between type $t$ and type structure $S$ hold as follows:

1. $t <: \iota$, means type $t = \iota$;
2. $t <: \{\ell_i : t_i \in 1..n\}$ means there are $t_i'$ with type $t = \{\ell_i : t_i' \in 1..n\}$ and $t_i <: t_i'$ for all $i \in 1..n$;
Our core type system does not include subtyping, and so the constraints we need are mostly type equality constraints; a type system with subtyping would use type inclusion constraints (Aiken and Wimmers, 1993; Aiken et al., 1998).

### 3.4 Expressions, Values, and Operational Semantics

#### Expressions and Values

To distinguish between the variables denoting values and functions, we conventionally use the metavariable $x$ for values and metavariable $f$ for functions. Moreover, variables can occur either free or bound in an expression. Separately, we have a set of record labels used to name the fields of a record. We consider labels to be a separate name space from identifiers; labels cannot be bound.

**Definition 3.5** (Syntax of Expressions and Values).

\[
\begin{align*}
  v &::= \text{variable identifiers} \\
  x &::= \text{(value) variable} \\
  f &::= \text{function variable} \\
  \ell &::= \text{record label} \\
  E &::= \text{expression} \\
  x &::= \text{variable} \\
  b &::= \text{boolean literal ($b \in \{\text{true, false}\}$)} \\
  c &::= \text{integer literal ($c \in \mathbb{Z}$)} \\
  E_1 \oplus E_2 &::= \{\oplus \in \{-, <, ==\}\} \text{selection of binary operators} \\
  \{\ell_i = E_i \mid i \in 1..n\} &::= \text{record ($n \geq 0$)} \\
  E.\ell &::= \text{projection} \\
  E_1 ? E_2 : E_3 &::= \text{conditional expression} \\
  \text{let } x = E_1 \text{ in } E_2 &::= \text{let-expression} \\
  \lambda(x)E &::= \text{lambda abstraction} \\
  E_1 (E_2) &::= \text{application} \\
  V &::= \text{value} \\
  b | c | \{\ell_i = V_i \mid i \in 1..n\} &::= \text{at most } x \text{ free in } E \\
  \lambda(x)E &::= \text{at most } x \text{ free in } E
\end{align*}
\]

Two of the expression forms are variable binders. In the binder \text{let } x = E_1 \text{ in } E_2, the
variable \( x \) is bound, with scope \( E_2 \). In the binder \( \lambda(x)E \), the variable \( x \) is bound, with scope \( E \). Let \( \text{fv}(E) \) be the set of variables occurring free in expression \( E \). Let a binder be\textit{ shadowed by an inner scope} in two cases: (1) it is \texttt{let x = E_1 in E_2} with a binder for \( x \) in \( E_2 \); (2) it is \( \lambda(x)E \) with a binder for \( x \) in \( E \). Let an expression be \textit{well-scoped} if and only if it has no subexpression that is a binder shadowed by an inner scope, and its bound variables are distinct from its free variables. For example, \texttt{let x = y in \lambda(x)x} is not well-scoped because the inner binder \( \lambda(x)x \) shadows the let-bound \( x \); and \texttt{let y = x in \lambda(x)y} is not well-scoped because the free variable \( x \) is bound by the lambda abstraction.

Intuitively, no binder within a well-scoped expression \( E \) re-defines either a variable free in or bound within \( E \).

Formally, we define \( \text{fv}(E) \) as follows.

\textbf{Definition 3.6 (Free Variables).}

\[
\begin{align*}
\text{fv}(x) & \triangleq \{x\} \\
\text{fv}(c) & \triangleq \{} \\
\text{fv}(\{\ell_i = E_i \mid i \in 1..n\}) & \triangleq \bigcup_{i \in 1..n} \text{fv}(E_i) \\
\text{fv}(E.\ell) & \triangleq \text{fv}(E) \\
\text{fv}(E_1 - E_2) & \triangleq \text{fv}(E_1) \cup \text{fv}(E_2) \\
\text{fv}(E_1 < E_2) & \triangleq \text{fv}(E_1) \cup \text{fv}(E_2) \\
\text{fv}(E_1 == E_2) & \triangleq \text{fv}(E_1) \cup \text{fv}(E_2) \\
\text{fv}(E_1 ? E_2 : E_3) & \triangleq \text{fv}(E_1) \cup \text{fv}(E_2) \cup \text{fv}(E_3) \\
\text{fv}(\text{let } x = E_1 \text{ in } E_2) & \triangleq \text{fv}(E_1) \cup (\text{fv}(E_2) \setminus \{x\}) \\
\text{fv}(\lambda(x)E) & \triangleq \text{fv}(E) \setminus \{x\} \\
\text{fv}(E_1 (E_2)) & \triangleq \text{fv}(E_1) \cup \text{fv}(E_2)
\end{align*}
\]

\textbf{Operational Semantics, Preservation, and Progress}

We define a standard small-step call-by-value \textit{reduction relation} on closed expressions, \( E \rightarrow E' \), meaning that expression \( E \) evolves in one step to expression \( E' \).

The \textit{reduction relation} \( E \rightarrow E' \) means that expression \( E \) evolves in one step to \( E' \). In the rules below, we write \( E[V/x] \) for the outcome of a capture-avoiding \textit{substitution} of the value \( V \) for each free occurrence of the variable \( x \) in the expression \( E \), with bound variables consistently renamed to result in a well-scoped expression. If \( E \) and \( V \) are well-scoped so is \( E[V/x] \).
Definition 3.7 (Reduction Rules).

\[
\begin{align*}
\text{(Red-1 Oplus)} & \quad E_1 \rightarrow E'_1 \quad \oplus \in \{-, <, >, ==\} \quad E_1 \oplus E_2 \rightarrow E'_1 \oplus E_2 \\
\text{(Red-2 Oplus)} & \quad E_2 \rightarrow E'_2 \quad V_1 \oplus E_2 \rightarrow V'_1 \oplus E'_2 \\
\text{(Red -)} & \quad c_3 = c_1 - c_2 \quad \text{if } c_1 < c_2 \text{ holds} \quad c_1 < c_2 \rightarrow \text{true} \quad c_1 < c_2 \rightarrow \text{false} \\
\text{(Red == True)} & \quad V_1 = V_2 \quad V_1 == V_2 \rightarrow \text{true} \quad V_1 \neq V_2 \rightarrow \text{false} \\
\text{(Red == False)} & \quad (V_1, V_2 \text{ first order}) \quad \text{if } c_1 < c_2 \text{ does not hold} \\
\text{(Red-1 Proj)} & \quad E \rightarrow E' \quad j \in 1..n \quad \ell = V_i \quad i \in 1..n \quad \ell_j \rightarrow V_j \\
\text{(Red Rcd)} & \quad E_j \rightarrow E'_j \quad j \in 1..n \quad \{\ell_i = V_i \quad i \in 1..j-1, \ell_j = E_j, \ell_k = E_k \quad k \in j+1..n\} \rightarrow \{\ell_i = V_i \quad i \in 1..j-1, \ell_j = E'_j, \ell_k = E'_k \quad k \in j+1..n\} \\
\text{(Red If)} & \quad E_1 \rightarrow E'_1 \quad \text{true?} \quad E_2 : E_3 \rightarrow E'_2 : E_3 \quad \text{false?} \quad E_2 : E_3 \rightarrow E_3 \\
\text{(Red-1 Let)} & \quad E_1 \rightarrow E'_1 \quad \text{let } x = E_1 \text{ in } E_2 \rightarrow \text{let } x = E'_1 \text{ in } E_2 \\
\text{(Red-2 Let)} & \quad E_1 \rightarrow E'_1 \quad \text{let } x = V_1 \text{ in } E_2 \rightarrow E_2[V_1/x] \\
\text{(Red-1 Lambda)} & \quad E_1 \rightarrow E'_1 \quad E_2 \rightarrow E'_2 \quad E_1(E_2) \rightarrow E'_1(E_2) \\
\text{(Red-2 Lambda)} & \quad V(E_2) \rightarrow V(E'_2) \quad (\lambda(x)E)(V) \rightarrow E[V/x] \\
\end{align*}
\]

Note that for the (Red == True) and (Red == False) we mean equality of values up to alpha-conversion. As, our type system requires that the type of the values be either Int or Bool, we only ever care about reductions of well-typed expressions, so we never end up testing equality between two function values.
3.5 Declarative Type System

An environment $\Gamma$ is a finite map from variables to types, written either as $x_1 : t_1, \ldots, x_n : t_n$ where $n > 0$ and the $x_i$ are pairwise distinct, or as $\emptyset$ for the empty environment. The purpose of an environment is to assign types to variables. The domain $\text{dom}(\Gamma)$ of an environment $\Gamma$ is the set of variables it assigns. Let $\text{dom}(x_1 : t_1, \ldots, x_n : t_n) = \{x_1, \ldots, x_n\}$ and $\text{dom}(\emptyset) = \emptyset$. In what follows, we use as a convention that if $\Gamma$ contains $x : t$ we write $\Gamma(x) = t$. The core judgment, $\Gamma \vdash E : t$, means that in environment $\Gamma$ the expression $E$ has the type $t$.

Declarative Typing Rules with Retyping

**Definition 3.8** (Declarative Typing Rules with Retyping).

\[
\begin{align*}
\text{(EXPR RETYPE)} & \quad \frac{}{\Gamma \vdash E : t \quad t <: : t'} \quad \Gamma \vdash E : t' \\
\text{(EXPR x)} & \quad \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : t} \\
\text{(EXPR b)} & \quad \frac{b \in \{\text{true}, \text{false}\}}{\Gamma \vdash b : \text{Bool}} \\
\text{(EXPR c)} & \quad \frac{\text{integer } c}{\Gamma \vdash c : \text{Int}} \\
\text{(EXPR \neg)} & \quad \frac{\Gamma \vdash E_1 : \text{Int}}{\Gamma \vdash E_1 - E_2 : \text{Int}} \\
\text{(EXPR <)} & \quad \frac{\Gamma \vdash E_1 : \text{Int} \quad \Gamma \vdash E_2 : \text{Int}}{\Gamma \vdash E_1 < E_2 : \text{Bool}} \\
\text{(EXPR ==)} & \quad \frac{\Gamma \vdash E_1 : t \quad \Gamma \vdash E_2 : t}{\Gamma \vdash E_1 == E_2 : \text{Bool}} \\
\text{(EXPR RCD)} & \quad \frac{\Gamma \vdash E_i : t_i \quad \forall i \in 1..n \quad \text{type } t = \{\ell_i : t_i \mid i \in 1..n\}}{\Gamma \vdash \{\ell_i = E_i \mid i \in 1..n\} : t} \\
\text{(EXPR PROJ)} & \quad \frac{\Gamma \vdash E : t \quad j \in 1..n \quad \text{type } t = \{\ell_i : t_i \mid i \in 1..n\}}{\Gamma \vdash E.\ell_j : t_j} \\
\text{(EXPR IF)} & \quad \frac{\Gamma \vdash E_1 : \text{Bool} \quad \Gamma \vdash E_2 : t \quad \Gamma \vdash E_3 : t}{\Gamma \vdash (E_1 ? E_2 : E_3) : t} \\
\text{(EXPR LET)} & \quad \frac{\Gamma \vdash E_1 : t_1 \quad \Gamma, x : t_1 \vdash E_2 : t_2}{\Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : t_2} \\
\text{(EXPR LAMBDA)} & \quad \frac{\Gamma, x : t_1 \vdash E : t_2 \quad \text{type } t = t_1 \rightarrow t_2}{\Gamma \vdash \lambda(x)E : t} \\
\text{(EXPR APPL)} & \quad \frac{\Gamma \vdash E_2 : t \quad \Gamma \vdash E_1 : t_1 \quad \text{type } t = t_1 \rightarrow t_2}{\Gamma \vdash E_2(E_1) : t_2}
\end{align*}
\]

The type system captures variable scoping:
Lemma 3.2. If $\Gamma \vdash E : t$ then $fv(E) \subseteq \text{dom}(\Gamma)$.

Proof. The proof is by induction on the depth of the derivation of $\Gamma \vdash E : t$. \hfill \square

Given a type environment, the type of a well-typed expression is unique up to type equality:

Lemma 3.3. If $\Gamma \vdash E : t_1$ and $\Gamma \vdash E : t_2$ then $t_1 <: t_2$.

Proof. The proof is by induction on the structure of $E$. \hfill \square

We state a standard weakening and a standard substitution lemma.

Lemma 3.4 (Weakening). If $\Gamma \vdash E' : t'$, then $\Gamma, x : t \vdash E' : t'$ for any $x \notin \text{dom}(\Gamma)$ and type $t$.

Proof. By induction on the height of the derivation of $\Gamma \vdash E' : t'$ using the syntax-directed rules. \hfill \square

Lemma 3.5 (Substitution). If $\Gamma, x : t, \Gamma' \vdash E' : t'$ and $\Gamma \vdash V : t$ then $\Gamma, \Gamma' \vdash E'[V/x] : t'$.

Proof. By induction on the height of the derivation of $\Gamma, x : t, \Gamma' \vdash E' : t'$ using the syntax-directed rules. \hfill \square

Type assumptions can be varied up to type equality:

Lemma 3.6 (Bound Equality). If $\Gamma, x : t \vdash E : \hat{t}$ and $t <: t'$ then $\Gamma, x : t' \vdash E : \hat{t}$.

Proof. By induction on the height of the derivation of $\Gamma, x : t \vdash E : \hat{t}$ using the declarative typing rules. \hfill \square

We now state progress and preservation theorem for our type system. Progress means that a closed well-typed program can always make positive progress, as opposed to failing in a stuck state. Preservation means that the outcome of a progression is well-typed, so long as the starting point is.

Theorem 3.1 (Progress). If $\emptyset \vdash E : t$ either (1) there is a value $V$ such that $E = V$, or (2) there is an expression $E'$ such that $E \rightarrow E'$.

Proof. By induction on the height of the derivation of $\emptyset \vdash E : t$. \hfill \square

Theorem 3.2 (Preservation). If $\Gamma \vdash E : t$ and $E \rightarrow E'$ then $\Gamma \vdash E' : t$. 
Proof. By induction on the height of the derivation of $E \rightarrow E'$.

In the next section we introduce syntax-directed typing rules as an alternative set of type rules for deriving judgments of the form $\Gamma \vdash E : t$. We do so as these rules because make some proofs easier, but we present the rules with (Expr Retype) as primary because they do not need the auxiliary relation.

Next we present the syntax-directed typing rules.

### Syntax-directed Presentation of Declarative Type System

This section presents an alternative syntax-directed set of rules for the judgment $\Gamma \vdash E : t$, and shows that the relations inductively defined by the rules in (Declarative Typing Rules with Retyping) and (Syntax-directed Declarative Typing Rules) are one and the same (Proposition 3.3).

This alternative set of rules is syntax-directed in the sense that there is one rule for each syntactic form of expression, unlike the rules above because of (Expr Retype); the work done by (Expr Retype) of closing the judgment up to type equality is moved into each of the syntax-directed rules, using an auxiliary relation $t <\!: S$ defined in Definition 3.3.

**Definition 3.9 (Syntax-directed Declarative Typing Rules).**

In these rules, the notation $t_1 <\!: t_2 <\!: \iota$ means $t_1 <\!: \iota$ and $t_2 <\!: \iota$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Decl Expr  x)</td>
<td>$x \in \text{dom}(\Gamma)$ $\Gamma(x) = t$ $t &lt;!: t'$</td>
<td>$\Gamma \vdash x : t'$</td>
</tr>
<tr>
<td>(Decl Expr  b)</td>
<td>$b \in {\text{true, false}}$ $t &lt;!: \text{Bool}$</td>
<td>$\Gamma \vdash b : t$</td>
</tr>
<tr>
<td>(Decl Expr  c)</td>
<td>integer $c$ $t &lt;!: \text{Int}$</td>
<td>$\Gamma \vdash c : t$</td>
</tr>
<tr>
<td>(Decl Expr  (-)) (t_1 &lt;!: t_2 &lt;!: \text{Int})</td>
<td>$\Gamma \vdash E_1 : t_1$ $\Gamma \vdash E_2 : t_2$ $t &lt;!: \text{Int}$</td>
<td>$\Gamma \vdash E_1 - E_2 : t$</td>
</tr>
<tr>
<td>(Decl Expr  &lt;) (t_1 &lt;!: t_2 &lt;!: \text{Int})</td>
<td>$\Gamma \vdash E_1 : t_1$ $\Gamma \vdash E_2 : t_2$ $t &lt;!: \text{Bool}$</td>
<td>$\Gamma \vdash E_1 &lt; E_2 : t$</td>
</tr>
<tr>
<td>(Decl Expr  ==) (t_1 &lt;!: t_2 &lt;!: \text{Bool or t_1 &lt;!: t_2 &lt;!: \text{Int}})</td>
<td>$\Gamma \vdash E_1 : t_1$ $\Gamma \vdash E_2 : t_2$ $t &lt;!: \text{Bool}$</td>
<td>$\Gamma \vdash E_1 == E_2 : t$</td>
</tr>
<tr>
<td>(Decl Expr  Rcd)</td>
<td>$\Gamma \vdash E_i : t_i$ $\forall i \in 1..n$ $t &lt;!: {\ell_i : t_i ; i \in 1..n}$</td>
<td>$\Gamma \vdash {\ell_i = E_i ; i \in 1..n} : t$</td>
</tr>
<tr>
<td>(Decl Expr  Proj) (t &lt;!: {t_i : i \in 1..n})</td>
<td>$\Gamma \vdash E : t$ $j \in 1..n$</td>
<td>$\Gamma \vdash E.\ell_j : t_j$</td>
</tr>
</tbody>
</table>
3.5. DECLARATIVE TYPE SYSTEM

We proceed by a case analysis of the derivation of $\Gamma \vdash E : t$ (of (Syntax-directed Declarative Typing Rules)).

**Lemma 3.7.** For all name types $t$, $t'$, and structures $S$, if $t <:: S$ and $t' <:: S$ then $t <:: t'$.

**Proof.** By a case analysis.

**Proposition 3.2.** If $\Gamma \vdash E : t$ using the rules of (Syntax-directed Declarative Typing Rules) and $t <:: t'$ then $\Gamma \vdash E : t'$.

**Proof.** We proceed by a case analysis of the derivation of $\Gamma \vdash E : t$ according to the rules of (Syntax-directed Declarative Typing Rules).

- **(Decl Expr x)** Suppose that $\Gamma \vdash x : t$ because $x \in \text{dom}(\Gamma)$ and $\Gamma(x) = t''$ and $t <:: t''$. Suppose also that $t <:: t'$. By transitivity we have that $t'' <:: t'$. By applying (Decl Expr x) for $x \in \text{dom}(\Gamma)$ and $\Gamma(x) = t$ and $t <:: t''$ we get that $\Gamma \vdash x : t''$.

- **(Decl Expr b)** Suppose that $\Gamma \vdash b : t$ because $b \in \{\text{true}, \text{false}\}$ and $t <:: \text{Bool}$. Suppose also that $t <:: t'$. By transitivity we have that $t' <:: \text{Bool}$. By applying (Decl Expr b) for $b \in \{\text{true}, \text{false}\}$ and $t' <:: \text{Bool}$ we get that $\Gamma \vdash b : t'$.

- **(Decl Expr c)** Suppose that $\Gamma \vdash c : t$ because integer $c$ and $t <:: \text{Int}$. Suppose also that $t <:: t'$. By transitivity we have that $t' <:: \text{Int}$. By applying (Decl Expr b) for integer $c$ and $t' <:: \text{Int}$ we get that $\Gamma \vdash c : \text{Int}$.

- **(Decl Expr −)** Suppose that $\Gamma \vdash E_1 - E_2 : t$ because $\Gamma \vdash E_1 : t_1$, $\Gamma \vdash E_2 : t_2$, $t_1 <:: t_2 <:: \text{Int}$, and $t <:: \text{Int}$. Suppose also that $t <:: t'$. By transitivity we have that $t' <:: \text{Int}$. By applying (Decl Expr −) for $\Gamma \vdash E_1 : t_1$, $\Gamma \vdash E_2 : t_2$, $t_1 <:: t_2 <:: \text{Int}$, and $t' <:: \text{Int}$ we get that $\Gamma \vdash E_1 - E_2 : t'$.

- **(Decl Expr <)** Suppose that $\Gamma \vdash E_1 < E_2 : t$ because $\Gamma \vdash E_1 : t_1$, $\Gamma \vdash E_2 : t_2$, $t_1 <:: t_2 <:: \text{Int}$ and $t <:: \text{Bool}$. Suppose also that $t <:: t'$. By transitivity we have that $t' <:: \text{Bool}$. By applying (Decl Expr <) for $\Gamma \vdash E_1 : t_1$, $\Gamma \vdash E_2 : t_2$, $t_1 <:: t_2 <:: \text{Int}$, and $t' <:: \text{Bool}$ we get that $\Gamma \vdash E_1 < E_2 : t'$. 

\[
\begin{align*}
(\text{Decl Expr IF}) & \quad (t' <:: \text{Bool}) & (\text{Decl Expr LET}) & \quad (x \notin \text{dom}(\Gamma)) \\
\Gamma \vdash E_1 : t' & \quad \Gamma \vdash E_2 : t & \quad \Gamma \vdash E_3 : t & \quad \Gamma \vdash E_1 : t_1, \quad \Gamma, x : t_1 \vdash E_2 : t_2 \\
\Gamma \vdash (E_1 ? E_2 : E_3) : t & \quad \Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : t_2
\end{align*}
\]
(Decl Expr \texttt{==}) Suppose that $\Gamma \vdash E_1 == E_2 : t$ because $\Gamma \vdash E_1 : t_1$, $\Gamma \vdash E_2 : t_2$, $t_1 ::= t_2 ::= \text{Bool}$ or $t_1 ::= t_2 ::= \text{Int}$, and $t ::= \text{Bool}$. Suppose also that $t ::= t'$. By transitivity we have that $t' ::= \text{Bool}$. By applying (Decl Expr \texttt{==}) for $\Gamma \vdash E_1 : t_1$, $\Gamma \vdash E_2 : t_2$, $t_1 ::= t_2 ::= \text{Bool}$ or $t_1 ::= t_2 ::= \text{Int}$, and $t ::= \text{Bool}$ we get that $\Gamma \vdash E_1 == E_2 : t'$.

(Decl Expr Rcd) Suppose that $\Gamma \vdash \{\ell_i = E_i^{i \in 1..n} \}: t$ because $\Gamma \vdash E_i : t_i$, $\forall i \in 1..n$ and $t ::= \{\ell_i : t_i^{i \in 1..n} \}$. Suppose also that $t ::= t'$.

Definition 3.3 it must be that there is $J = \{\ell_j^{j \in 1..n} = \{\ell_i^{i \in 1..n} \} \text{ and } t'_j \text{ such that type } t' = \{\ell_j : t_j^{j \in 1..n} \text{ and } t_j ::= t'_j \text{ for each } j \in 1..n. \text{ By induction for all } j \in 1..n, \text{ if } t_j ::= t'_j \text{ and } \Gamma \vdash E_j : t_j \text{ then } \Gamma \vdash E_j : t'_j. \text{ Now, by applying (Decl Expr Rcd) for } \Gamma \vdash E_j : t_j \forall j \in 1..n \text{ t ::= } \{\ell_j : t_j^{j \in 1..n} \} \text{ we get that } \Gamma \vdash \{\ell_j = E_j^{j \in 1..n} : t'. \text{ By transitivity we have that } t' ::= \{\ell_i : t_i^{i \in 1..n} \}. \text{ By applying (Decl Expr Rcd) for } \Gamma \vdash E_i : t_i \forall i \in 1..n \text{ and } t' ::= \{\ell_i : t_i^{i \in 1..n} \} \text{ we get that } \Gamma \vdash \{\ell_i = E_i^{i \in 1..n} : t'. \}

(Decl Expr Pro) Suppose that $\Gamma \vdash E.\ell_j : t_j$ because $\Gamma \vdash E : E : t \quad j \in 1..n$ and $t ::= \{\ell_i : t_i^{i \in 1..n} \}$. Suppose also that $t_j ::= t'_j$. By Definition 3.3 it must be that there are type $t' = \{\ell_i : t_i^{i \in 1..n} \}$. By applying (Decl Expr Pro) for $\Gamma \vdash E : t \quad j \in 1..n$ and $\Gamma \vdash E.\ell_j : t_j$ and $t' ::= t'_j$.

(Decl Expr If) Suppose that $\Gamma \vdash (E_1 ? E_2 : E_3) : t$ because $\Gamma \vdash E_1 : t_1$, $\Gamma \vdash E_2 : t_2$, $\Gamma \vdash E_3 : t t' ::= \text{Bool}$. Suppose also that $t ::= t'$.

By transitivity we have that $t' ::= \text{Bool}$. By applying (Decl Expr If) for $\Gamma \vdash E_1 : t'$ $\Gamma \vdash E_2 : t$ $\Gamma \vdash E_3 : t$ and for $t' ::= \text{Bool}$ we get that $\Gamma \vdash (E_1 ? E_2 : E_3) : t'$.

(Decl Expr Let) Suppose that $\Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : t_2$ because $\Gamma \vdash E_1 : t_1$, $\Gamma \vdash E_2 : t_2$. Suppose also that $t_2 ::= t'$. By transitivity we have that $t' ::= t_2$. By applying (Decl Expr Let) for $\Gamma \vdash E_1 : t_1$ $\Gamma \vdash x : t_1 \vdash E_2 : t_2$ and $t' ::= t_2$ we get that $\Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : t'$.

(Decl Expr Lambda) Suppose that $\Gamma \vdash \lambda(x)E : t$ because $t ::= \$. Suppose also that $t ::= t'$. By transitivity we have that $t' ::= t$. By applying (Decl Expr Lambda) for $\Gamma \vdash \lambda(x)E : t$ and $t' ::= t'$ we get that $\Gamma \vdash t'$. Suppose that $\Gamma \vdash \lambda(x)E : t$ because type $t = t_1 \rightarrow t_2$ and $\Gamma \vdash x : t_1 \vdash E : t_2$. We also assume that $t ::= t'$. By 3.3 it must also be that type $t' = t'_1 \rightarrow t'_2$ such that $t'_1 ::= t_1$ and $t_2 ::= t'_2$. Now, by applying (Decl Expr Lambda) for $\Gamma \vdash x : t'_1 \vdash E : t'_2$, we get $\Gamma \vdash \lambda x.E : t'$. 

(Decl Expr Appl) Suppose that $\Gamma \vdash E_2(E_1) : t$ because $t_2 ::= t_1 \rightarrow t$ and $E_2 : t_2$. Suppose also that $t \rightarrow t'$. By transitivity we have that $t' ::= t$. By 3.3 it
must also be that \( \text{type } t' = t'_1 \rightarrow t'_2 \) such that \( t'_1 <: t_1 \) and \( t_2 <: t'_2 \). By applying \((\text{DECL} \text{\ EXPR} \text{\ APPL})\) for \( \Gamma \vdash E_2 : t_2 \) \( \Gamma \vdash E_1 : t_1 \) and \( t_2 <: t'_1 \) \( t_1 <: t' \) we get that \( \Gamma \vdash E_2(E_1) : t' \).

(Declarative Typing Rules with Retyping) and (Syntax-directed Declarative Typing Rules) define one and the same relation.

**Proposition 3.3.** The judgment \( \Gamma \vdash E : t \) is derivable using the rules of (Declarative Typing Rules with Retyping) if and only if \( \Gamma \vdash E : t \) is derivable using the rules of (Syntax-directed Declarative Typing Rules).

**Proof.** The proof breaks down into proving the two halves separately.

1. If \( \Gamma \vdash E : t \) is derivable using (Declarative Typing Rules with Retyping) then \( \Gamma \vdash E : t \) is derivable using (Syntax-directed Declarative Typing Rules).

2. If \( \Gamma \vdash E : t \) is derivable using (Syntax-directed Declarative Typing Rules) then \( \Gamma \vdash E : t \) is derivable using the (Declarative Typing Rules with Retyping).

The proof of part (1) is by induction on the structure of \( E \). In each case, we consider that the derivation must follow from one of the syntax-directed rules of (Declarative Typing Rules with Retyping) together with zero or more usages of (EXPR RETYPE). We build a derivation using the corresponding rule from (Syntax-directed Declarative Typing Rules), and apply Proposition 3.2 to complete the derivation (doing the work of (EXPR RETYPE)).

The proof of part (2) is by induction on the structure of \( E \). In each case, we build a derivation using the corresponding rule from (Declarative Typing Rules with Retyping), and apply (EXPR RETYPE) to complete the derivation.

**Background: Structural vs. Nominal Type Systems**

Pierce (2002) writes: “Type systems like Java’s, in which names are significant and subtyping is explicitly declared, are called nominal.” In contrast, a structural type system is one where “names are inessential and subtyping is defined directly on the structures of types.” In our setting, names are inessential to type equality (or to the subtyping relation obtained as the union of all simulations). Therefore, our type system is structural and not nominal. Still, it is worth noting that our typing judgment \( \Gamma \vdash E : t \) is name-based and not structure-based, in the sense that it ascribes only a type name \( t \) to an expression, and
not a syntactic structure $S$. In contrast, Featherweight Java (Igarashi et al., 2001) has both a typing judgment that is name-based and a nominal type system.

## 3.6 The Type Inference Problem for Function Definitions

We have now assembled enough formal definitions to define the task introduced in Chapter 1.

**Problem 3.1 (Type Signature).** Consider a top-level untyped function definition with the following syntax:

```
function f(x_1, \ldots, x_n) return E
```

Let a type signature for $f$ be a tuple $\Sigma \triangleq (x_1 : t_1, \ldots, x_n : t_n, f : t)$ of type names, such that the following type assignment is derivable:

```
x_1 : t_1, \ldots, x_n : t_n \vdash E : t
```

If $(x_1 : t_1, \ldots, x_n : t_n, f : t)$ is a type signature, we can introduce a type so that the value $\lambda(x_1) \ldots \lambda(x_n)E$ has type $t$, and hence can be called as a function from typed code.

As shown in Section 3.1 our three running examples all exhibit ambiguities that we will resolve using natural information.
Type Inference by Accumulating Logical Constraints

The purpose of this chapter is to define an algorithmic typing system for the core language of Section 7.3. The goal of this type system is to encode typing constraints by accumulating logical constraints over type variables; this step will enable us to constructively combine the logical constraints with other sources of information in Chapter 5. We first describe related definitions and syntax for these typing constraints, and then present the corresponding algorithmic type system. We continue by proving some formal properties for this system and by showing what are the logical constraints that the algorithmic type system produces for our motivating examples. We then conclude this chapter by giving some references to related work on type inference by accumulating constraints.

Section 4.1 defines basic concepts that are needed for the description and usage of the logical constraints in the rest of the dissertation. We construct these constraints as logical formulae (Definition 4.2) over type variables (Definition 4.1). Thus we need to define a valuation that maps type variables to nominal types of Section 7.3 (Definition 4.3) and what does logical satisfaction mean in this case (Definition 4.4). Additionally, in Definition 4.6 we give a compact derived syntax to generalize the Definition 3.4 for type variables; we use this syntax to describe the algorithmic type system that follows.

Section 4.2 provides the (Algorithmic Typing Rules), where essentially for each rule of the (Declarative Typing Rules with Retyping) we define a corresponding rule to the new system. The core judgement of the algorithmic type system proposed in this chapter, emits a logical formula $C$ which encodes the desired constraints using Definition 4.6, along with a set of pairs $V$, which associates variable identifiers to their corresponding type
variables. The type system is designed in such a way that any type that can be ascribed to \( E \) by Definition 3.8 corresponds to a variable valuation \( \mu \) that satisfies the formula. Hence the formula captures logically the ways in which the expression may be type correct. Intuitively, we need these constraints to make sure that when we later combine them with the natural ones, our predictions will be sound.

In Section 4.3 we prove termination, soundness and completeness for (Algorithmic Typing Rules) and we state an important corollary (Corollary 4.1) that enables to show how this system corresponds to the claims we make in Section 3.1 about our three motivating examples of untyped function definitions. Finally, in Section 4.3 we apply Corollary 4.1 to our motivating examples and show the corresponding logical constraints.

4.1 Type Variables, Logical Constraints, Type Variable Valuation

To encode constraints \( C \) (Definition 4.2) on type variables \( \alpha \) (Definition 4.1) we use formulas in Equational logic. Equational logic relies on Boolean operators and equational predicates of the form \( \alpha = t \), which denote that a type variable \( \alpha \) is of type \( t \); the logic considers types simply as atomic names and does not depend on any other properties, such as type equality.

Hence, a constraint \( \alpha = \text{String} \) holds if \( \alpha \) denotes the type name \( \text{String} \), but does not hold if \( \alpha \) denotes \( \text{IntArray} \), even though \( \text{String} <\!: > \text{IntArray} \).

**Definition 4.1 (Type Variables).**

\[
\begin{align*}
\alpha & \quad \text{type variable} \\
\end{align*}
\]

Let \( A \) be a set to range over finite sets of type variables. Let \( \text{tyvar}(\alpha) = \{\alpha\} \).

**Definition 4.2 (Equational Logic).**

\[
\begin{align*}
C := & \quad \text{logical formula or constraint} \\
\text{true} & \quad \text{equation, between type variable } \alpha \text{ and type name } t \\
\alpha = t & \quad \text{negation} \\
\neg C & \\
C \lor C & \quad \text{disjunction}
\end{align*}
\]
Let \( \text{tyvar}(C) \) be the set of type variables occurring in \( C \).
Let \( C \) be the set of all constraints.

As usual in classical logic, we define false \( \triangleq \neg \text{true} \) and \( C_1 \land C_2 \triangleq (\neg C_1) \lor (\neg C_2) \).

**Definition 4.3 (Type Variable Valuation \( \mu \)).** A valuation \( \mu \) is a finite map \( \{ \alpha_i = t_i \mid i \in 1..n \} \) that maps a type variable \( \alpha_i \) to a nominal type \( t_i \). Let \( \text{dom}(\mu) = \{ \alpha_i \mid i \in 1..n \} \), the finite set of type variables in the domain of \( \mu \).

We also introduce a notation \( \mu(\Gamma^\circ) \) that lifts a variable valuation \( \mu \) to apply to an algorithmic environment \( \Gamma^\circ \). Let \( \mu(\Gamma^\circ) = x_1 : \mu(\alpha_1), \ldots, x_n : \mu(\alpha_n) \) if environment \( \Gamma^\circ = x_1 : \alpha_1, \ldots, x_n : \alpha_n \).

We define a logical satisfaction relation \( \mu \models C \), when \( \text{tyvar}(C) \subseteq \text{dom}(\mu) \), by induction on the size of \( C \), as follows:

**Definition 4.4 (Logical Satisfaction Relation).**

\[
\begin{align*}
\mu \models \text{true} & \quad \text{always} \\
\mu \models \alpha = t & \quad \text{if and only if} \quad \mu(\alpha) = t \\
\mu \models \neg C & \quad \text{if and only if} \quad \neg \mu \models C \\
\mu \models C_1 \lor C_2 & \quad \text{if and only if} \quad \mu \models C_1 \lor \mu \models C_2
\end{align*}
\]

Let constraint \( C \) be satisfiable if and only if there is a valuation \( \mu \) such that \( \mu \models C \).
We say that such an \( \mu \) is a model for \( C \).

Our standard notations behave as expected:

- \( \mu \models \text{false} \) never;
- \( \mu \models C_1 \land C_2 \) if and only if \( \mu \models C_1 \) and \( \mu \models C_2 \).

**Background: Type Inference by Accumulating Constraints**

While it is standard to accumulate sets of constraints on type variables, since Milner’s Algorithm W (Milner, 1978), our introduction of disjunctive constraints to handle ambiguity is less common. While Milner’s system finds the principal type scheme of a function definition, ours finds the maximally natural subject to being sound. Milner uses unification to simplify constraints as they are generated. For now, we omit constraint simplification from our theory so as to emphasise the generation of constraints. But as a next step, in Chapter 7 we convert these logical constraints to a formula in propositional logic and
eventually as an input to a MaxSAT solver, which simplifies the constraints. However, for
the sake of readability we define logical equivalence between two formulas as follows:

**Definition 4.5** (Logical Equivalence). Let $C \sim C'$ mean that for all $\mu$ with $\text{tyvar}(C, C') \subseteq \text{dom}(\mu)$, $\mu \models C$ if and only if $\mu \models C'$.

Logical equivalence is reflexive, transitive, and symmetric, and is a congruence. The
relation satisfies expected laws of logical equivalence, including associative, commutative,
identity, and domination laws for each of $\land$ and $\lor$, and distributive laws relating them.
(There is a rich literature on simplifying type inclusion constraints (Aiken and Wimmers,
1993; Aiken et al., 1998).)

Additionally, we adopt some type-specific notations for constraints, to do so we use the
relation defined on Definition 3.4 and define a derived syntax to notate the relation
$\alpha \triangle <::> S'$, which we use on Definition 4.7. Note that $S'$ is defined in the same way as $S$
where a concrete type $t$ has been replaced with a type variable $\alpha$. We adopt standard
notation for conjunction and disjunction of sets of constraints. If we have a set of constraints
$\{C_i \mid i \in I\}$ for a finite indexing set $I = \{i_1, \ldots, i_n\}$, the notation $\land_{i \in I} C_i$ means
the conjunction $C_{i_1} \land \cdots \land C_{i_n}$, and in particular means $\text{true}$ if $I = \emptyset$. Similarly, the notation
$\lor_{i \in I} C_i$ means the disjunction $C_{i_1} \lor \cdots \lor C_{i_n}$, and in particular means $\text{false}$ if $I = \emptyset$.

**Definition 4.6** (Derived Syntax for Constraints).

\[
\begin{align*}
\alpha <::> t & \triangleq \lor_{t \in I} \alpha = t \text{ where } I = \{t \mid t <::> t\} \\
\alpha <::> \alpha' & \triangleq \lor_{(t,t') \in I} (\alpha = t \land \alpha' = t') \text{ where } I = \{(t, t') \mid t <::> t'\} \\
\alpha <::> \{\ell_i : \alpha_i \in \{1 \ldots n\}\} & \triangleq \lor_{(t_1, \ldots, t_n) \in I} (\alpha = t \land \land_{i \in 1 \ldots n} \alpha_i = t_i) \\
& \text{ where } I = \{(t, t_1, \ldots, t_n) \mid t <::> \{\ell_i : \alpha_i \in 1 \ldots n\}\}
\end{align*}
\]

\[
\begin{align*}
\alpha <::> \alpha' & \triangleq \lor_{(t, t') \in I} (\alpha = t \land \alpha' = t') \\
& \text{ where } I = \{(t, t') \mid \exists n, j \in 1 \ldots n, \ell_1, t_1, \ldots, \ell_n, t_n : t <::> \{\ell_i : \alpha_i \in 1 \ldots n\}; \ell = \ell_j, t' = t_j\}
\end{align*}
\]

\[
\begin{align*}
\alpha <::> \alpha_1 \rightarrow \alpha_2 & \triangleq \lor_{(t, t_1, t_2) \in I} (\alpha = t \land \alpha_1 = t_1 \land \alpha_2 = t_2) \\
& \text{ where } I = \{(t, t_1, t_2) \mid t <::> t_1 \rightarrow t_2\}
\end{align*}
\]

These derived forms of constraints are satisfied as follows:

**Lemma 4.1** (Logical Satisfaction for Derived Syntax).

a. $\mu \models \land_{i \in I} C_i$ if and only if $\forall i \in I : \mu \models C_i$

b. $\mu \models \lor_{i \in I} C_i$ if and only if $\exists i \in I : \mu \models C_i$

c. $\mu \models \alpha <::> t$ if and only if $\mu(\alpha) <::> t$

d. $\mu \models \alpha <::> \alpha' \text{ if and only if } \mu(\alpha) <::> \mu(\alpha')$
4.1. TYPE VARIABLES, LOGICAL CONSTRAINTS, TYPE VARIABLE VALUATION

\[ e. \quad \mu \models \alpha \leftarrow \{ \ell_i : \alpha_i \} \text{ if and only if } \mu(\alpha) \leftarrow \{ \ell_i : \mu(\alpha_i) \} \]

\[ f. \quad \mu \models \alpha' \leftarrow \alpha.\ell \text{ if and only if } \exists n, \ell_1, t_1, \ldots, \ell_n, t_n \text{ and } j \in \{1, \ldots, n\}:
\mu(\alpha) \leftarrow \{ \ell_i : t_i \} \text{ and } \mu(\alpha') \leftarrow t_j \text{ and } \ell = \ell_j \]

\[ g. \quad \mu \models \alpha \leftarrow \alpha_1 \rightarrow \alpha_2 \text{ if and only if } \mu(\alpha) \leftarrow \mu(\alpha_1) \rightarrow \mu(\alpha_2). \]

Proof.

a. Suppose that we have an empty indexing set \( I = \emptyset \).

\[ \mu \models \bigwedge_{i \in I} C_i \]

if and only if \( \text{true} \) (Definition of the notation)

b. Suppose that we have a finite indexing set \( I = \{i_1, \ldots, i_n\} \).

\[ \mu \models \bigwedge_{i \in I} C_i \]

if and only if \( \mu \models C_{i_1} \land \cdots \land C_{i_n} \) (Definition of the notation)

if and only if \( \forall i \in I : \mu \models C_i \) (Definition 4.4)

c. Suppose that we have an empty indexing set \( I = \emptyset \).

\[ \mu \models \bigvee_{i \in I} C_i \]

if and only if \( \text{false} \) (Definition of the notation)

Suppose that we have a finite indexing set \( I = \{i_1, \ldots, i_n\} \).

\[ \mu \models \bigvee_{i \in I} C_i \]

if and only if \( \mu \models C_{i_1} \lor \cdots \lor C_{i_n} \) (Definition of the notation)

if and only if \( \exists i \in I : \mu \models C_i \) (Definition 4.4)
d. Consider any \( \mu, \alpha, \iota \). Let \( I = \{ t \mid t <\!: \iota \} \)

\[
\mu \models \alpha <\!: \iota
\]

if and only if \( \mu \models \bigvee_{t \in I} (\alpha = t) \)  
\hspace*{10cm} (Definition 4.6)

if and only if \( \exists t \in I : \mu(\alpha) = t \)  
\hspace*{10cm} (Part b)

if and only if \( \exists t \in I : t <\!: \iota \) and \( \mu(\alpha) = t \)

if and only if \( \mu(\alpha) <\!: \iota \)  
\hspace*{10cm} (Definition 4.3)

e. Consider any \( \mu, \alpha, \alpha' \). Let \( I = \{(t, t') \mid t <\!: t'\} \)

\[
\mu \models \alpha <\!: \alpha'
\]

if and only if \( \mu \models \bigvee_{(t, t') \in I} (\alpha = t \land \alpha' = t') \)  
\hspace*{10cm} (Definition 4.6)

if and only if \( \exists (t, t') \in I : \mu(\alpha) = t \) and \( \mu(\alpha') = t' \)  
\hspace*{10cm} (Part b)

if and only if \( \exists (t, t') : t <\!: t' \) and \( \mu(\alpha) = t \) and \( \mu(\alpha') = t' \)

if and only if \( \mu(\alpha) <\!: \mu(\alpha') \)  
\hspace*{10cm} (Definition 4.3)

f. Consider any \( \mu, \alpha, n, \ell_1, \ldots, \ell_n \). Let \( I = \{(t, t_1, \ldots, t_n) \mid t <\!: \{\ell_i : t_i \in \{1, \ldots, n\}\} \}

\[
\mu \models \alpha <\!: \{\ell_i : \alpha_i \in \{1, \ldots, n\}\}
\]

if and only if \( \mu \models \bigvee_{(t, t_1, \ldots, t_n) \in I} (\alpha = t \land \bigwedge_{i \in \{1, \ldots, n\}} \alpha_i = t_i) \)  
\hspace*{10cm} (Definition 4.6)

if and only if \( \exists (t, t_1, \ldots, t_n) \in I : \mu(\alpha) = t \) and \( \mu(\alpha_1) = t_1 \) and \( \ldots \) and \( \mu(\alpha_n) = t_n \)  
\hspace*{10cm} (Part a, b)

if and only if \( \exists (t, t_1, \ldots, t_n) \in I : t <\!: \{\ell_i : t_i \in \{1, \ldots, n\}\} \) and \( \mu(\alpha) = t \) and \( \mu(\alpha_1) = t_1 \) and \( \ldots \) and \( \mu(\alpha_n) = t_n \)

if and only if \( \mu(\alpha) <\!: \{\ell_i : \mu(\alpha_i) \in \{1, \ldots, n\}\} \)  
\hspace*{10cm} (Definition 4.3)

g. Consider any \( \mu, \alpha, \alpha', \ell \). Let \( I = \{(t, t_j) \mid t <\!: \{\ell_i : t_i \in \{1, \ldots, n\}\}, \ell = \ell_j, j \in \{1, \ldots, n\}\} \)

\[
\mu \models \alpha' <\!: \alpha, \ell
\]

if and only if \( \mu \models \bigvee_{(t, t') \in I} (\alpha = t \land \alpha' = t') \)

if and only if \( \exists (t, t') \in I : \mu(\alpha) = t \) and \( \mu(\alpha') = t' \)  
\hspace*{10cm} (Part b)

if and only if \( \exists (t, t') \in I : t <\!: \{\ell_i : t_i \in \{1, \ldots, n\}\}, \ell = \ell_j, j \in \{1, \ldots, n\}\} \)
4.2 Algorithmic Type System

Our core type system does not include subtyping, and so the constraints we need are mostly type equality constraints; a type system with subtyping would use type inclusion constraints (Aiken and Wimmers, 1993; Aiken et al., 1998).

4.2 Algorithmic Type System

An algorithmic type environment $\Gamma^\circ$ is a finite map from identifiers to type variables, written either as $x_1 : \alpha_1, \ldots, x_n : \alpha_n$ where $n > 0$ and the $x_i$ are pairwise distinct, or as $\emptyset$ for the empty environment. Let dom($x_1 : \alpha_1, \ldots, x_n : \alpha_n$) = {$x_1, \ldots, x_n$} and dom($\emptyset$) = $\emptyset$. Let tyvar($x_1 : \alpha_1, \ldots, x_n : \alpha_n$) = {$\alpha_1, \ldots, \alpha_n$} and tyvar($\emptyset$) = $\emptyset$.

Additionally, the type system outputs an overall record $V$ of when an identifier corresponds to a type variable. The record $V$ is a finite set $V = \{(x_1, \alpha_1), \ldots, (x_n, \alpha_n)\}$ of what we call natural facts, pairs of identifiers $x_i$ and type variables $\alpha_i$. Section 5.1 shows how this set serves as an input to construct natural constraints for $E$. Let tyvar($V$) = {$\alpha \mid (x, \alpha) \in V$}.

The judgment of our algorithmic type system takes the form $\Gamma^\circ \vdash E \Rightarrow \alpha (C, V)$ meaning that in $\Gamma^\circ$ the expression $E$ emits a type variable $\alpha$, a logical formula $C$, which accumulates all the logical constraints in $E$, and a set $V$ of natural facts that associates exactly all
bound variables in $E$ with type variables from $\Gamma^0$ or $C$.

The rules for deriving $\Gamma^0 \vdash E \Rightarrow \alpha (C, V)$ are as follows. We rely on the notation $\text{tyvar}(\psi_1, \ldots, \psi_n)$ that means the set of variables $\text{tyvar}(\psi_1) \cup \cdots \cup \text{tyvar}(\psi_n)$, where each $\psi_i$ is a syntactic phrase that is either an environment $\Gamma^0$, a type variable $\alpha$, a logical constraint $C$, or the set $V$.

The auxiliary function $\text{newtyvar}(\Gamma^0, \alpha, C, V)$ determines the type variables that are fresh in the derivation of $\Gamma^0 \vdash E \Rightarrow \alpha (C, V)$, that is, the variables $\text{tyvar}(\alpha, C, V)$ occurring in the output that are not in $\text{tyvar}(\Gamma^0)$, the variables of the input. By definition, it follows that $\text{tyvar}(\alpha, C, V) \subseteq \text{tyvar}(\Gamma^0) \cup \text{newtyvar}(\Gamma^0, \alpha, C, V)$.

**Definition 4.7** (Algorithmic Typing Rules).

Let $\text{newtyvar}(\Gamma^0, \alpha, C, V) = \text{tyvar}(\alpha, C, V) \setminus \text{tyvar}(\Gamma^0)$.

\[
\begin{align*}
\text{(ALGO x)} & \quad x \in \text{dom}(\Gamma^0) \quad \Gamma^0(x) = \alpha \quad \text{true, } \emptyset \\
\Gamma^0 \vdash x & \Rightarrow \alpha (\text{true, } \emptyset) \\
\text{(ALGO b)} & \quad b \in \{\text{true, false}\} \\
\Gamma^0 \vdash b & \Rightarrow \alpha (\alpha <:\text{ Bool, } \emptyset) \\
\text{(ALGO c)} & \quad \text{integer } c \\
\Gamma^0 \vdash c & \Rightarrow \alpha (\alpha <:\text{ Int, } \emptyset)
\end{align*}
\]

\[
\begin{align*}
\text{(ALGO \neg)} & \quad (\alpha \notin \text{tyvar}(\Gamma^0, C_1, C_2, V_1, V_2) \text{ and } \text{newtyvar}(\Gamma^0, \alpha, C_1, V_1) \cap \text{newtyvar}(\Gamma^0, \alpha, C_2, V_2) = \emptyset) \\
\Gamma^0 \vdash E_1 & \Rightarrow \alpha_1 (C_1, V_1) \quad \Gamma^0 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2) \\
\Gamma^0 \vdash E_1 - E_2 & \Rightarrow \alpha (\alpha <:\text{ Int } \land \alpha_1 <:\text{ Int } \land \alpha_2 <:\text{ Int } \land C_1 \land C_2, V_1 \cup V_2)
\end{align*}
\]

\[
\begin{align*}
\text{(ALGO <)} & \quad (\alpha \notin \text{tyvar}(\Gamma^0, C_1, C_2, V_1, V_2) \text{ and } \text{newtyvar}(\Gamma^0, \alpha, C_1, V_1) \cap \text{newtyvar}(\Gamma^0, \alpha, C_2, V_2) = \emptyset) \\
\Gamma^0 \vdash E_1 & \Rightarrow \alpha_1 (C_1, V_1) \quad \Gamma^0 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2) \\
\Gamma^0 \vdash E_1 < E_2 & \Rightarrow \alpha (\alpha <:\text{ Bool } \land \alpha_1 <:\text{ Int } \land \alpha_2 <:\text{ Int } \land C_1 \land C_2, V_1 \cup V_2)
\end{align*}
\]

\[
\begin{align*}
\text{(ALGO ==)} & \quad (\alpha \notin \text{tyvar}(\Gamma^0, C_1, C_2, V_1, V_2) \text{ and } \text{newtyvar}(\Gamma^0, \alpha, C_1, V_1) \cap \text{newtyvar}(\Gamma^0, \alpha, C_2, V_2) = \emptyset) \\
\Gamma^0 \vdash E_1 & \equiv \alpha_1 (C_1, V_1) \quad \Gamma^0 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2) \\
\Gamma^0 \vdash E_1 == E_2 & \Rightarrow \alpha (\alpha <:\text{ Bool } \land \alpha_1 <:\text{ Int } \land \alpha_2 <:\text{ Int } \land C_1 \land C_2, V_1 \cup V_2)
\end{align*}
\]

\[
\begin{align*}
\text{(ALGO RCD)} & \quad (\alpha \notin \bigcup_{i \in 1..n} \text{tyvar}(\Gamma^0, \alpha, C_i, V_i) \text{ and sets } \text{newtyvar}(\Gamma^0, \alpha, C_i, V_i) \text{ disjoint}) \\
\Gamma^0 \vdash E_i & \Rightarrow \alpha_i (C_i, V_i) \quad \forall i \in 1..n \\
\Gamma^0 \vdash \{\ell_i = E_i \mid i \in 1..n\} & \Rightarrow \alpha (\alpha <:\{\ell_i \mid i \in 1..n\} \land \bigwedge_{i \in 1..n} C_i \land \bigcup_{i \in 1..n} V_i)
\end{align*}
\]

\[
\begin{align*}
\text{(ALGO PROJ)} & \quad (\alpha' \notin \text{tyvar}(\Gamma^0, \alpha, C, V)) \\
\Gamma^0 \vdash E & \Rightarrow \alpha (C, V) \\
\Gamma^0 \vdash E.l & \Rightarrow \alpha' (\alpha' <:\alpha.l \land C, V)
\end{align*}
\]

\[
\begin{align*}
\text{(ALGO IF)} & \quad (\text{sets } \text{newtyvar}(\Gamma^0, \alpha, C_i, V_i) \text{ disjoint}) \\
\Gamma^0 \vdash E_1 & \Rightarrow \alpha_1 (C_1, V_1) \quad \Gamma^0 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2) \quad \Gamma^0 \vdash E_3 \Rightarrow \alpha_3 (C_3, V_3) \\
\Gamma^0 \vdash (E_1 ? E_2 : E_3) & \Rightarrow \alpha_2 (\alpha_1 <:\text{ Bool } \land \alpha_2 <:\text{ Int } \land \alpha_3 \land \bigwedge_{i \in 1..3} C_i \land \bigcup_{i \in 1..3} V_i)
\end{align*}
\]
We call this relation *algorithmic* because it is a nondeterministic specification for an algorithm that given inputs $\Gamma^o$ and $E$ computes outputs $\alpha$, $C$, and $V$ such that $\Gamma^o \vdash E \Rightarrow \alpha (C,V)$. Most of the rules pick a variable $\alpha$ to represent the type of the expression; these variables are freshly generated in the sense of being chosen arbitrarily so long as they are distinct from existing variables. The only nondeterminism in the rules arises from the choice of these fresh type variables.

Take the rule (Algo −) for example.

\[
\text{(Algo −) (} \alpha \notin \text{tyvar}(\Gamma^o, C_1, C_2, V_1, V_2) \text{ and } \text{newtyvar}(\Gamma^o, \alpha_1, C_1, V_1) \cap \text{newtyvar}(\Gamma^o, \alpha_2, C_2, V_2) = \emptyset) \\
\Gamma^o \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \quad \Gamma^o \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2)
\]

\[
\Gamma^o \vdash E_1 - E_2 \Rightarrow \alpha (\alpha <: \text{Int} \land \alpha_1 <: \text{Int} \land \alpha_2 <: \text{Int} \land C_1 \land C_2, V_1 \cup V_2)
\]

The rule illustrates the two kinds of side-condition needed for freshness.

1. The first kind, exemplified by $\alpha \notin \text{tyvar}(\Gamma^o, C_1, C_2, V_1, V_2)$, ensures that the fresh variable $\alpha$ is distinct from other variables in the current derivation: distinct both from the input $\Gamma^o$ or the output components $C_1, C_2$ and $V_1, V_2$.

2. The second kind, exemplified by $\text{newtyvar}(\Gamma^o, \alpha_1, C_1, V_1) \cap \text{newtyvar}(\Gamma^o, \alpha_2, C_2, V_2) = \emptyset$, ensures the disjointness of the sets of fresh variables picked by independent parallel derivations, such as $\Gamma^o \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$ and $\Gamma^o \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2)$.

A third kind of side-condition is $x \notin \text{dom}(\Gamma^o)$ in (Algo Let) and (Algo Lambda); these ensure that the input expression $E$ is well-scoped, and that environments only bind distinct variables.

Altogether, these three kinds of side-conditions ensure the following basic property:

**Lemma 4.2.** If $\Gamma^o \vdash E \Rightarrow \alpha (C, V)$ then $E$ is well-scoped.
Proof. By induction on the depth of derivation of $\Gamma^o \vdash E \Rightarrow \alpha (C,V)$.

Disjunctive logical constraints (written either explicitly, or implicitly using derived notations of Definition 4.6) represent alternative typings in the following rules:

- (ALGO ==): which overloading of the equality operator;
- (ALGO RCD): which named record type to return;
- (ALGO PROJ): which named record type from which to extract a field;
- (ALGO LAMBDA): which named function type to return;
- (ALGO APPL): which named named function type to apply.

Type Variables may have Zero, One, or More Identifiers

In (Algorithmic Typing Rules) we need the set $V$ of natural facts as each type variable can be associated with zero, one, or more identifiers. For instance the (ALGO PROJ) rule introduces a fresh type variable $\alpha'$ which does not correspond to any identifier. While in the program below, we show that a type variable can be associated with three distinct value variables.

\[
\begin{align*}
\lambda(y) & \quad // V = \{(y, \alpha_1)\} \\
\text{let } x1 = y \text{ in } & \quad // V = \{(y, \alpha_1), (x1, \alpha_1)\} \\
\text{let } x2 = y \text{ in } & \quad // V = \{(y, \alpha_1), (x1, \alpha_1), (x2, \alpha_1)\} \\
x1 - x2
\end{align*}
\]

4.3 Formal Properties: Termination, Soundness, and Completeness

Every well-scoped expression determines a logical constraint and a set of (id,type variable) pairs.

**Theorem 4.1** (Termination). Suppose $E$ is well-scoped and $fv(E) \subseteq \{x_1, \ldots, x_n\}$. For pairwise distinct $\alpha_1, \ldots, \alpha_n$, there are $\alpha, C, V$ such that $x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash E \Rightarrow \alpha (C,V)$.

**Proof.** Existence holds by induction on the structure of $E$. ⊢

**Lemma 4.3.** If $\Gamma^o \vdash E \Rightarrow \alpha (C,V)$ then $\alpha \in \text{dom}(\Gamma^o) \cup \text{newtyvar}(\Gamma^o, \alpha, C,V)$.
Proof. By induction on the derivation of $\Gamma \vdash E \Rightarrow \alpha (C, V)$, with appeal to the definition that $\text{newtyvar}(\Gamma^o, \alpha, C, V) = \text{tyvar}(\alpha, C, V) \setminus \text{tyvar}(\Gamma^o)$.

Lemma 4.4. If $\Gamma^o \vdash E \Rightarrow \alpha (C, V)$ then $\text{tyvar}(C) \subseteq \text{dom}(\Gamma^o) \cup \text{newtyvar}(\Gamma^o, \alpha, C, V)$.

Proof. By definition, $\text{newtyvar}(\Gamma^o, \alpha, C, V) = \text{tyvar}(\alpha, C, V) \setminus \text{tyvar}(\Gamma^o)$.

Lemma 4.5. If $\mu \cup \mu' \models C$ and $\text{dom}(\mu') \cap \text{tyvar}(C) = \emptyset$ then $\mu \models C$.

Proof. By induction on the derivation of $\Gamma^o \vdash E \Rightarrow \alpha (C, V)$.

Lemma 4.6 (Soundness). If $\Gamma^o \vdash E \Rightarrow \alpha (C, V)$ and $\text{dom}(\mu) = \text{tyvar}(\Gamma^o)$ and $\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, C, V)$ and $(\mu \cup \mu')(\alpha) <:> t$ and $\mu \cup \mu' \models C$ then $\mu(\Gamma^o) \vdash E : t$.

Proof. For the reader’s convenience we present here only a representative subset of the total number of cases. The rest cases can be found in the Appendix A. It suffices to prove that for all $\Gamma^o, E, \alpha, C, V, \mu, \mu', t$ that, if

(1) $\Gamma^o \vdash E \Rightarrow \alpha (C, V)$
(2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^o)$
(3) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, C, V)$
(4) $(\mu \cup \mu')(\alpha) <:> t$
(5) $\mu \cup \mu' \models C$

then $\mu(\Gamma^o) \vdash E : t$.

The proof is by induction on the derivation of (1) $\Gamma^o \vdash E \Rightarrow \alpha (C, V)$. We proceed by considering each rule that can derive judgment (1). Notice that in each case, there can only be one rule from Definition 3.9 that can derive the declarative judgement. In each case, we can assume (2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^o)$, (3) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, C, V)$, (4) $(\mu \cup \mu')(\alpha) <:> t$, and (5) $\mu \cup \mu' \models C$.

Case $E = x$. Our judgment (1) is derived as follows, with $E = x$ and $C = \text{true}$ and $V = \emptyset$.

\[
\text{(ALGO } x) \quad \frac{x \in \text{dom}(\Gamma^o) \quad \Gamma^o(x) = \alpha}{\Gamma^o \vdash x \Rightarrow \alpha (\text{true, } \emptyset)}
\]

Given this and (2), (3), (4), and (5), we are to show $\mu(\Gamma^o) \vdash x : t$.

From (3) we have:

$\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, C, V)$
\[\text{tyvar}(\alpha, C, V) \\setminus \text{tyvar}(\Gamma^\circ)\]
\[= \text{tyvar}(\alpha, \text{true}, \emptyset) \\setminus \text{tyvar}(\Gamma^\circ)\]
\[= \emptyset, \text{ because } \alpha \in \text{tyvar}(\Gamma^\circ)\]

and so it must be that \(\mu' = \emptyset\).

Since \(x \in \text{dom}(\Gamma^\circ)\) and \(\Gamma^\circ(x) = \alpha\), by Definition 4.3 it must be that \(x \in \text{dom}(\mu(\Gamma^\circ))\) and \(\mu(\Gamma^\circ)(x) = \mu(\alpha)\). Hence, for \(\Gamma = \mu(\Gamma^\circ)\), \(t = \mu(\alpha)\) and \(t' = t\). We have \(t <:\!\!> t'\) because \(<:\!\!>\) is reflexive (Lemma 3.1), and hence we can derive the following, as desired.

\[
(\text{DECL Expr } x) \quad x \in \text{dom}(\Gamma) \quad \Gamma(x) = t \quad t <:\!\!> t' \quad \Gamma \vdash x : t'
\]

Case \(E = b\). Our judgment (1) is derived as follows, with \(E = b\) and \(C = \alpha <:\!\!> \text{Bool}\) and \(V = \emptyset\).

\[
(\text{ALGO } b) \quad (\alpha \notin \text{tyvar}(\Gamma^\circ))
\quad b \in \{\text{true}, \text{false}\}
\quad \Gamma^\circ \vdash b \Rightarrow \alpha \quad (\alpha <:\!\!> \text{Bool}, \emptyset)
\]

Given this and (2), (3), (4), and (5), we are to show \(\mu(\Gamma^\circ) \vdash b : t\).

From (3) we have:

\[
\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V)
\]
\[
= \text{tyvar}(\alpha, C, V) \setminus \text{tyvar}(\Gamma^\circ)
\]
\[
= \text{tyvar}(\alpha, \alpha <:\!\!> \text{Bool}, \emptyset) \setminus \text{tyvar}(\Gamma^\circ)
\]
\[
= \{\alpha\}, \text{ because } \alpha \notin \text{tyvar}(\Gamma^\circ)
\]

and so it must be that \(\alpha \in \text{dom}(\mu')\).

By (4) and Lemma 4.1(2) for \(\iota = \text{Bool}\), \(\mu'(\alpha) <:\!\!> \text{Bool}\) because \(\mu' \models \alpha <:\!\!> \text{Bool}\), and therefore it must be that \(t <:\!\!> \text{Bool}\).

Since \(b \in \{\text{true}, \text{false}\}\) and \(t <:\!\!> \text{Bool}\), we can derive the following, as desired.

\[
(\text{DECL Expr } b) \quad b \in \{\text{true}, \text{false}\} \quad t <:\!\!> \text{Bool}
\quad \mu(\Gamma^\circ) \vdash b : t
\]
Case $E = c$. Our judgment (1) is derived as follows, with $E = c$ and $C = \alpha <:: Int$ and $V = \emptyset$.

\[
\frac{(\text{ALGO } c) \ (\alpha \notin \text{tyvar}(\Gamma^0))}{\text{integer } c}
\]

\[\Gamma^0 \vdash c \Rightarrow \alpha \ (\alpha <:: \text{Int, } \emptyset)\]

Given this and (2), (3), (4), and (5), we are to show $\mu(\Gamma^0) \vdash c : t$.

From (3) we have:

\[
\text{dom}(\mu') = \text{newtyvar}(\Gamma^0, \alpha, C, V)
\]

\[
= \text{tyvar}(\alpha, C, V) \setminus \text{tyvar}(\Gamma^0)
\]

\[
= \text{tyvar}(\alpha, \alpha <:: \text{Int, } \emptyset) \setminus \text{tyvar}(\Gamma^0)
\]

\[
= \{\alpha\}, \text{ because } \alpha \notin \text{tyvar}(\Gamma^0)
\]

and so it must be that $\alpha \in \text{dom}(\mu')$.

By (4) and Lemma 4.1(2) for $\iota = \text{Int}$, $\mu'(\alpha) <:: \text{Int}$ because $\mu' \models \alpha <:: \text{Int}$, and therefore it must be that $t <:: \text{Int}$.

Since we have integer $c$ and $t <:: \text{Int}$, we can derive the desired judgment:

\[
\frac{(\text{DECL EXPR } c)}{\text{integer } c \ t <:: \text{Int}} \frac{\mu(\Gamma^0) \vdash c : t}{\mu(\Gamma^0) \vdash c : t}
\]

Case $E = E_1 - E_2$. Our judgment (1) is derived as follows, with $E = E_1 - E_2$ and $C = \alpha <:: \text{Int} \land \alpha_1 <:: \text{Int} \land \alpha_2 <:: \text{Int} \land C_1 \land C_2$ and $V = V_1 \cup V_2$.

\[
\frac{(\text{ALGO } -) \ (\alpha \notin \text{tyvar}(\Gamma^0, C_1, C_2, V_1, V_2) \text{ and } \text{newtyvar}(\Gamma^0, \alpha_1, C_1, V_1) \cap \text{newtyvar}(\Gamma^0, \alpha_2, C_2, V_2) = \emptyset)}{\Gamma^0 \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \quad \Gamma^0 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2)}
\]

\[\Gamma^0 \vdash E_1 - E_2 \Rightarrow \alpha \ (\alpha <:: \text{Int} \land \alpha_1 <:: \text{Int} \land \alpha_2 <:: \text{Int} \land C_1 \land C_2, V_1 \cup V_2)\]

From (2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^0)$ and (3) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^0, \alpha, C, V)$, and (4) $(\mu \cup \mu')(\alpha) = t$, and (5) $\mu \cup \mu' \models \alpha <:: \text{Int} \land \alpha_1 <:: \text{Int} \land \alpha_2 <:: \text{Int} \land C_1 \land C_2$, we are to show $\mu(\Gamma^0) \vdash E_1 - E_2 : t$.

By (5) and Definition 4.4, we have $\mu \cup \mu' \models \alpha <:: \text{Int}$ and $\mu \cup \mu' \models \alpha_1 <:: \text{Int}$ and $\mu \cup \mu' \models \alpha_2 <:: \text{Int}$ and $\mu \cup \mu' \models C_1$ and $\mu \cup \mu' \models C_2$.

By applying Lemma 4.1(c) for $\iota = \text{Int}$, we get that:

i. $(\mu \cup \mu')(\alpha) <:: \text{Int}$ because $\mu \cup \mu' \models \alpha <:: \text{Int}$,
Case Let. Our judgment (1) is derived as follows, with 

\[ (\mu \cup \mu')(\alpha_1) \iff \text{Int} \text{ because } \mu \cup \mu' \models \alpha_1 \iff \text{Int}, \]

\[ (\mu \cup \mu')(\alpha_2) \iff \text{Int} \text{ because } \mu \cup \mu' \models \alpha_2 \iff \text{Int}. \]

From (i) and (4) it must be that \( t \iff \text{Int} \). Let \( t_1 = (\mu \cup \mu')(\alpha_1) \) and \( t_2 = (\mu \cup \mu')(\alpha_2) \).

Hence, from (ii) and (iii), it must be that \( t_1 \iff \text{Int} \) and \( t_2 \iff \text{Int} \).

Let \( \mu' = \mu' \upharpoonright \newtyvar(\Gamma^0, \alpha_1, C_1, V_1) \).

Hence, we have (3.1) \( \dom(\mu') = \newtyvar(\Gamma^0, \alpha_1, C_1, V_1) \).

From \( \Gamma^0 \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \) and Lemma 4.3 we get \( \alpha_1 \in \dom(\Gamma^0) \cup \newtyvar(\Gamma^0, \alpha_1, C_1, V_1) \).

Hence, from (3.1) and (ii) we get (4.1) \( (\mu \cup \mu')(\alpha_1) \iff \text{Int} \).

From \( \Gamma^0 \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \) and Lemma 4.4 we get that \( \tyvar(C_1) \subseteq \dom(\Gamma^0) \cup \newtyvar(\Gamma^0, \alpha_1, C_1, V_1) \).

Hence, from \( \mu \cup \mu' \models C_1 \) and Lemma 4.5 we get (5.1) \( \mu \cup \mu' \models C_1 \).

By induction hypothesis, \( \Gamma^0 \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \), (2), (3.1), (4.1), (5.1) imply \( \mu(\Gamma^0) \vdash E_1 : t_1 \).

By symmetric reasoning, we obtain \( \mu(\Gamma^0) \vdash E_2 : t_2 \).

Hence, we can derive the desired judgment

\[
\frac{(\text{DECL \ Expr} \ - ) \ (t_1 \iff t_2 \iff \text{Int})}{\mu(\Gamma^0) \vdash E_1 : t_1 \quad \mu(\Gamma^0) \vdash E_2 : t_2 \quad t \iff \text{Int}} \quad \mu(\Gamma^0) \vdash E_1 - E_2 : t
\]

Case Let. Our judgment (1) is derived as follows, with \( E = \text{let } x = E_1 \text{ in } E_2 \) and 

\( C = C_1 \land C_2 \) and \( V = \{ (x, \alpha_1) \} \cup V_1 \cup V_2 \).

\[
(\text{ALGO \ Let}) \ (x \notin \dom(\Gamma^0) \text{ and } \alpha_1 \notin \tyvar(\Gamma^0) \text{ and } \newtyvar(\Gamma^0, \alpha_1, C_1, V_1) \cap \newtyvar(\Gamma^0, x : \alpha_1, \alpha_2, C_2, V_2) = \emptyset)
\]

\[
\frac{\Gamma^0 \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \quad \Gamma^0, x : \alpha_1 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2)}{\Gamma^0 \vdash \text{let } x = E_1 \text{ in } E_2 \Rightarrow \alpha_2 (C_1 \land C_2, \{ (x, \alpha_1) \} \cup V_1 \cup V_2)}
\]

From (2) \( \dom(\mu) = \tyvar(\Gamma^0) \) and (3) \( \dom(\mu') = \newtyvar(\Gamma^0, \alpha, C, V) \), and (4) \( (\mu \cup \mu')(\alpha_2) \iff t_2 \), and (5) \( \mu \cup \mu' \models C_1 \land C_2 \), we are to show \( \mu(\Gamma^0) \vdash \text{let } x = E_1 \text{ in } E_2 : t_2 \).

By Definition 4.4, we have \( \mu \cup \mu' \models C_1 \) and \( \mu \cup \mu' \models C_2 \).

Let \( \mu' = \mu' \upharpoonright \newtyvar(\Gamma^0, \alpha_1, C_1, V_1) \).

Hence we have (3.1) \( \dom(\mu') = \newtyvar(\Gamma^0, \alpha_1, C_1, V_1) \).
From $\Gamma^o \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$ and Lemma 4.3 we get $\alpha_1 \in \text{dom}(\Gamma^o) \cup \text{newtyvar}(\Gamma^o, \alpha_1, C_1, V_1)$.

Let $t_1 = (\mu \cup \mu')(\alpha_1)$. Hence, from (3.1) we get (4.1) $(\mu \cup \mu')(\alpha_1) \iff t_1$.

From $\Gamma^o \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$ and Lemma 4.4 we get that $\text{tyvar}(C_1) \subseteq \text{dom}(\Gamma^o) \cup \text{newtyvar}(\Gamma^o, \alpha_1, C_1, V_1)$.

Hence, from $\mu \cup \mu' \models C_1$ and Lemma 4.5 we get (5.1) $\mu \cup \mu' \models C_1$.

By induction hypothesis, $\Gamma^o \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$, (2), (3.1), (4.1), (5.1) imply $\mu(\Gamma^o) \vdash E_1 : t_1$.

From (3), we have:

$$\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha_2, (C_1 \land C_2), \{(x, \alpha_1)\} \cup V_1) \cup V_2)$$
$$= \{(\alpha_2, \alpha_1) \cup \text{tyvar}(C_1) \cup \text{tyvar}(C_2) \cup \text{tyvar}(V_1) \cup \text{tyvar}(V_2)\} \setminus \text{tyvar}(\Gamma^o)$$

Hence, $\alpha_1 \in \text{dom}(\mu')$ since $\alpha_1 \notin \text{tyvar}(\Gamma^o) = \text{dom}(\mu)$.

Consider $\mu_2 = \mu \cup \{\alpha_1 = t_1\}$ and $\mu'_2 = \mu' \upharpoonright (\text{dom}(\mu') \setminus \{\alpha_1\} \setminus \text{dom}(\mu'))$.

The sets $\{\alpha_1\}$ and $\text{dom}(\mu)$ are disjoint, because of the condition $\alpha_1 \notin \text{tyvar}(\Gamma^o) = \text{dom}(\mu)$.

Hence, we have (2.2) $\text{dom}(\mu_2) = \text{tyvar}(\Gamma^o, \alpha_1)$.

We can verify that (3.2) $\text{dom}(\mu'_2) = \text{newtyvar}((\Gamma^o, x : \alpha_1), \alpha_2, C_2, V_2)$ by the following calculation:

$$LHS = \text{dom}(\mu'_2)$$
$$= \text{dom}(\mu') \setminus \{\alpha_1\}$$
$$= (\{\alpha_2, \alpha_1\} \cup \text{tyvar}(C_2) \cup \text{tyvar}(V_2)) \setminus \text{tyvar}(\Gamma^o) \setminus \{\alpha_1\}$$
$$RHS = \text{newtyvar}((\Gamma^o, x : \alpha_1), \alpha_2, C_2, V_2)$$
$$= \text{tyvar}(\alpha_2, C_2, V_2) \setminus \text{tyvar}(\Gamma^o, x : \alpha_1)$$
$$= \text{tyvar}(\alpha_2, C_2, V_2) \setminus \text{tyvar}(\Gamma^o) \setminus \{\alpha_1\}$$
$$= (\{\alpha_2\} \cup \text{tyvar}(C_2) \cup \text{tyvar}(V_2)) \setminus \text{tyvar}(\Gamma^o) \setminus \{\alpha_1\}$$

We have (4.2) $(\mu_2 \cup \mu'_2)(\alpha_2) \iff t_2$ because $(\mu \cup \mu')(\alpha_2) \iff t_2$ and $\mu_2 \cup \mu'_2 = \mu \cup \mu'$.

We have (5.2) $\mu_2 \cup \mu'_2 \models C_2$ because $\mu \cup \mu' \models C_2$ and $\mu_2 \cup \mu'_2 = \mu \cup \mu'$. 
By induction hypothesis, $\Gamma^o, x : \alpha_1 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2)$, (2.2), (3.2), (4.2), (5.2) imply $\mu_2(\Gamma^o, x : \alpha_1) \vdash E_2 : t_2$.

Hence, we have $\mu(\Gamma^o), x : t_1 \vdash E_2 : t_2$.

Hence, we can derive the desired judgment

\[(\text{DECL} \ \text{EXPR} \ \text{LET}) \ (x \notin \text{dom}(\Gamma)) \]
\[
\frac{\mu(\Gamma^o) \vdash E_1 : t_1 \quad \mu(\Gamma^o), x : t_1 \vdash E_2 : t_2}{\mu(\Gamma^o) \vdash \text{let} \ x = E_1 \ \text{in} \ E_2 \ : t_2}
\]

Case Lambda. Our judgment (1) is derived as follows, with $E = \lambda(x)E'$ and $C = \alpha <:\alpha_1 \to \alpha_2 \land C'$ and $V = \{(x, \alpha_1)\} \cup V'$.

\[(\text{ALGO} \ \text{LAMBDA}) \ (x \notin \text{dom}(\Gamma^o) \text{ and } \alpha \notin \text{tyvar}(\Gamma^o) \text{ and } \alpha \notin \text{tyvar}(\Gamma^o, \alpha_1, \alpha_2, C', V')) \]
\[
\frac{\Gamma^o, x : \alpha_1 \vdash E' \Rightarrow \alpha_2 (C', V')}{\Gamma^o \vdash \lambda(x)E' \Rightarrow \alpha (\alpha <:\alpha_1 \to \alpha_2 \land C', \{(x, \alpha_1)\} \cup V')}
\]

From (2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^o)$ and (3) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, C, V)$, and (4) $(\mu \cup \mu')(\alpha) <:\alpha$, and (5) $\mu \cup \mu' \models \alpha <:\alpha_1 \to \alpha_2 \land C'$, we are to show $\mu(\Gamma^o) \vdash \lambda(x)E' : t$.

By Definition 4.4, we have $\mu \cup \mu' \models \alpha <:\alpha_1 \to \alpha_2$ and $\mu \cup \mu' \models C'$.

By applying Lemma 4.1(g) we have that $(\mu \cup \mu')(\alpha) <:\alpha (\mu \cup \mu')(\alpha_1) \to (\mu \cup \mu')(\alpha_2)$, because $\mu \cup \mu' \models \alpha <:\alpha_1 \to \alpha_2$.

Let $t_1 = (\mu \cup \mu')(\alpha_1)$ and $t_2 = (\mu \cup \mu')(\alpha_2)$. Hence, from (4), it must be that $t <:\alpha_1 \to t_2$.

From (3), we have:

$$\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, (\alpha <:\alpha_1 \to \alpha_2 \land C'), (\{(x, \alpha_1)\} \cup V'))$$

$$= (\{\alpha, \alpha_1, \alpha_2\} \cup \text{tyvar}(C') \cup \text{tyvar}(V')) \setminus \text{tyvar}(\Gamma^o)$$

Hence, $\alpha_1 \in \text{dom}(\mu')$ since $\alpha \notin \text{tyvar}(\Gamma^o) = \text{dom}(\mu)$.

Consider $\mu_1 = \mu \cup \{\alpha_1 = t_1\}$ and $\mu'_1 = \mu' \upharpoonright (\text{dom}(\mu') \setminus \{\alpha, \alpha_1\})$.

Observe that $\mu \cup \mu' = \mu_1 \cup \mu'_1 \cup \{\alpha = t\}$

$$\mu_1 \cup \mu'_1 \cup \{\alpha = t\} = \mu \cup \{\alpha_1 = t_1\} \cup \mu' \upharpoonright (\text{dom}(\mu') \setminus \{\alpha, \alpha_1\}) \cup \{\alpha = t\}$$

$$= \mu \cup \mu'$$
4.3. FORMAL PROPERTIES: TERMINATION, SOUNDNESS, AND COMPLETENESS

The sets \( \{\alpha_1\} \) and \( \text{dom}(\mu) \) are disjoint, because of the condition \( \alpha_1 \notin \text{tyvar}(\Gamma^\circ) = \text{dom}(\mu) \).

Hence, we have \((2.1) \text{dom}(\mu_1) = \text{tyvar}(\Gamma^\circ, \alpha_1) \).

We can verify that \((3.1) \text{dom}(\mu'_1) = \text{newtyvar}(\Gamma^\circ, x : \alpha_1, \alpha_2, C', \mathcal{V}') \) by the following calculation:

\[
\begin{align*}
\text{LHS} &= \text{dom}(\mu'_1) \\
&= \text{dom}(\mu') \setminus \{\alpha, \alpha_1\} \\
&= (\{\alpha, \alpha_1, \alpha_2\} \cup \text{tyvar}(C') \cup \text{tyvar}(\mathcal{V}')) \setminus \text{tyvar}(\Gamma^\circ) \setminus \{\alpha, \alpha_1\} \\
&= (\{\alpha_2\} \cup \text{tyvar}(C') \cup \text{tyvar}(\mathcal{V}')) \setminus \text{tyvar}(\Gamma^\circ) \setminus \{\alpha_1\}
\end{align*}
\]

\[
\begin{align*}
\text{RHS} &= \text{newtyvar}(\Gamma^\circ, x : \alpha_1, \alpha_2, C', \mathcal{V}') \\
&= \text{tyvar}(\alpha_2, C', \mathcal{V}') \setminus \text{tyvar}(\Gamma^\circ, x : \alpha_1) \\
&= (\{\alpha_2\} \cup \text{tyvar}(C') \cup \text{tyvar}(\mathcal{V}')) \setminus \text{tyvar}(\Gamma^\circ) \setminus \{\alpha_1\}
\end{align*}
\]

We have \((4.1) (\mu_1 \cup \mu'_1)(\alpha_2) <\!:\! t_2 \) because \((\mu \cup \mu')(\alpha_2) <\!:\! t_2 \) and \(\mu_1 \cup \mu'_1 \cup \{\alpha = t\} = \mu \cup \mu'\).

We have \((5.1) \mu_1 \cup \mu'_1 \models C' \) because \(\mu \cup \mu' \models C' \) and \(\mu_1 \cup \mu'_1 \cup \{\alpha = t\} = \mu \cup \mu'\).

By induction hypothesis, \(\Gamma^\circ, x : \alpha_1 \models E' \Rightarrow \alpha_2 (C', \mathcal{V}'), (2.1), (3.1), (4.1), (5.1) \) imply \(\mu_1(\Gamma^\circ, x : \alpha_1) \models E' : t_2 \).

Hence, we have \(\mu(\Gamma^\circ), x : t_1 \models E' : t_2 \).

We can conclude as follows:

\[
(\text{DECL}
\text{ EXPR}
\text{ LAMBDA})
(x \notin \text{dom}(\mu(\Gamma^\circ)))
\]

\[
\begin{align*}
\mu(\Gamma^\circ), x : t_1 &\models E' : t_2 \\
&\Rightarrow t <\!:\! t_1 \rightarrow t_2 \\
\mu(\Gamma^\circ) &\models \lambda(x) E' : t
\end{align*}
\]

Case Appl. We have:

\[
\begin{align*}
(\text{ALGO APPL})
&\ (\alpha \notin \text{tyvar}(\Gamma^\circ, C_1, C_2, \mathcal{V}_1, \mathcal{V}_2) \text{ and } \text{newtyvar}(\Gamma^\circ, \alpha_2, C_2, \mathcal{V}_2) \cap \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, \mathcal{V}_1) = \emptyset) \\
&\Gamma^\circ \models E_2 \Rightarrow \alpha_2 (C_2, \mathcal{V}_2) \quad \Gamma^\circ \models E_1 \Rightarrow \alpha_1 (C_1, \mathcal{V}_1) \\
&\Gamma^\circ \models E_2(E_1) \Rightarrow \alpha_2 <\!:\! \alpha_1 \rightarrow \alpha \land C_1 \land C_2, \mathcal{V}_1 \cup \mathcal{V}_2)
\end{align*}
\]

From \((2) \text{dom}(\mu) = \text{tyvar}(\Gamma^\circ) \) and \((3) \text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, \mathcal{V}) \), and \((4) (\mu \cup \mu')(\alpha) <\!:\! t, \) and \((5) \mu \cup \mu' \models \alpha_2 <\!:\! \alpha_1 \rightarrow \alpha \land C_1 \land C_2, \) we are to show \(\mu(\Gamma^\circ) \models E_2(E_1) : t\).
By Definition 4.4, we have $\mu \models \alpha_2 <: \alpha_1 \rightarrow \alpha$ and $\mu \models C_1$ and $\mu \models C_2$.

By applying Lemma 4.1(g) we get $(\mu \cup \mu')(\alpha_2) <: (\mu \cup \mu')(\alpha_1 \rightarrow \alpha)$ because $\mu \cup \mu' \models \alpha_2 <: \alpha_1 \rightarrow \alpha$.

Let $t_1 = (\mu \cup \mu')(\alpha_1)$ and $t_2 = (\mu \cup \mu')(\alpha_2)$. Hence, from (4), it must be that $t_2 <: t_1 \rightarrow t$.

Let $\mu'_1 = \mu' \upharpoonright \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1)$.

Hence, we have (3.1) $\text{dom}(\mu'_1) = \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1)$.

From $\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$ and Lemma 4.3 we get $\alpha_1 \in \text{dom}(\Gamma^\circ) \cup \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1)$.

Hence, from (3.1) and because $t_1 = (\mu \cup \mu')(\alpha_1)$ we can get (4.1) $(\mu \cup \mu'')(\alpha_1) <: t_1$.

From $\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$ and Lemma 4.4 we get that $\text{tyvar}(C_1) \subseteq \text{dom}(\Gamma^\circ) \cup \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1)$.

Hence, from $\mu \cup \mu' \models C_1$ and Lemma 4.5 we get (5.1) $\mu \cup \mu'_1 \models C_1$.

By induction hypothesis, $\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$, (2), (3.1), (4.1), (5.1) imply $\mu(\Gamma^\circ) \vdash E_1 : t_1$.

By symmetric reasoning, we obtain $\mu(\Gamma^\circ) \vdash E_2 : t_2$.

Hence, for $\Gamma = \mu(\Gamma^\circ)$ we can derive the desired judgment

$$
\frac{\text{(DECL EXPR APPL) } (t_2 <: t_1 \rightarrow t)}{
\frac{\mu(\Gamma^\circ) \vdash E_2 : t_2 \quad \mu(\Gamma^\circ) \vdash E_1 : t_1}
{\mu(\Gamma^\circ) \vdash E_2(E_1) : t}
}
$$

\[ \square \]

**Lemma 4.7** (Completeness). Consider $\Gamma^\circ \vdash E \Rightarrow \alpha (C, V)$ and type $t$.

For all $\mu$ with $\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)$, if $\mu(\Gamma^\circ) \vdash E : t$ then there is $\mu'$ with $\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V)$ and $(\mu \cup \mu')(\alpha) <: t$ and $\mu \cup \mu' \models C$.

**Proof.** As before, for the reader’s convenience we present here only a representative subset of the total cases. The rest cases can be found in the Appendix A.

It suffices to prove that for all $\Gamma^\circ, E, \alpha, C, V, \mu, t$ that, if

1. $\Gamma^\circ \vdash E \Rightarrow \alpha (C, V)$
2. $\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)$
3. $\mu(\Gamma^\circ) \vdash E : t$


then there is $\mu'$ with (A) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V)$, (B) $(\mu \cup \mu')(\alpha) <: t$, and (C) $\mu \cup \mu' \models C$.

The proof is by induction on the height of the derivation of the algorithmic judgment (1). We proceed by a case analysis of $E$. For each rule from (Algorithmic Typing Rules), only one of the syntax-directed rules from (Syntax-directed Declarative Typing Rules) can have derived declarative judgment (3). Hence we can obtain the desired satisfaction relation by a detailed case analysis.

Recall that $\text{newtyvar}(\Gamma^\circ, \alpha, C, V) = \text{tyvar}(\alpha, C, V) \setminus \text{tyvar}(\Gamma^\circ)$.

Case $E = x$. In this case, and our assumptions (1), (2) and (3) take the forms:

(\text{ALGO } x)
\[
\begin{align*}
& x \in \text{dom}(\Gamma^\circ) \quad \Gamma^\circ(x) = \alpha \\
\Gamma^\circ \vdash x \Rightarrow \alpha (\text{true}, \varnothing)
\end{align*}
\]
\[
\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)
\]

(\text{DECL EXPR } x)
\[
\begin{align*}
& x \in \text{dom}(\mu(\Gamma^\circ)) \quad \mu(\Gamma^\circ)(x) = \alpha \quad \alpha < : > t' \\
& \mu(\Gamma^\circ) \vdash x : t'
\end{align*}
\]

We are to find $\mu'$ with (A) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V)$ (B) $(\mu \cup \mu')(\alpha) <: t$, and (C) $\mu \cup \mu' \models C$, where $C = \text{true}$ and $V = \varnothing$.

Let $\mu' = \varnothing$.

We have (A), since
\[
\text{dom}(\mu') = \varnothing
\]
\[
\text{newtyvar}(\Gamma^\circ, \alpha, C, V) = \text{tyvar}(\alpha, C, V) \setminus \text{tyvar}(\Gamma^\circ)
\]
\[
= \text{tyvar}(\alpha, \text{true}, \varnothing) \setminus \{\alpha\}
\]
\[
= \varnothing
\]

We have (B) $(\mu \cup \mu')(\alpha) <: t$, since $x \in \text{dom}(\Gamma^\circ)$ and $\Gamma^\circ(x) = \alpha$, by Definition 4.3 it must be that $x \in \text{dom}(\mu(\Gamma^\circ))$ and $\mu(\Gamma^\circ)(x) = \mu(\alpha)$, and by the premises of (DECL EXPR x) we get that $\Gamma = \mu(\Gamma^\circ)$ and $t <: > \mu(\alpha)$.

We have (C) $\mu \cup \mu' \models \text{true}$, by Definition 4.4.

Hence $\mu'$ has properties (A), (B), and (C) as desired.
Case $E = b$. In this case, and our assumptions (1), (2) and (3) take the forms:

\[
\begin{align*}
\text{(ALGO b) } (\alpha \notin \text{tyvar}(\Gamma^\circ)) \\
\quad b \in \{\text{true}, \text{false}\} \\
\Gamma^\circ \vdash b \Rightarrow \alpha (\alpha <\!: \text{Bool}, \emptyset) \\
\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)
\end{align*}
\]

\[
\begin{align*}
\text{(DECL Expr b)} \\
\quad b \in \{\text{true}, \text{false}\} \quad t <\!: \text{Bool} \\
\mu(\Gamma^\circ) \vdash b : t
\end{align*}
\]

We are to find $\mu'$ with (A) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V)$ (B) $(\mu \cup \mu')(\alpha) <\!: t$, and (C) $\mu \cup \mu' \models C$, where $C = \text{true}$ and $V = \emptyset$.

Let $\mu' = \{\alpha \mapsto t\}$.

We have (A), since

\[
\begin{align*}
\text{dom}(\mu') &= \{\alpha\} \\
\text{newtyvar}(\Gamma^\circ, \alpha, C, V) &= \text{tyvar}(\alpha, \alpha <\!: \text{Bool}, \emptyset) \setminus \text{tyvar}(\Gamma^\circ) \text{ and } \alpha \notin \text{tyvar}(\Gamma^\circ) \\
&= \{\alpha\}
\end{align*}
\]

We have (B) $(\mu \cup \mu')(\alpha) <\!: t$, because $\mu'(\alpha) <\!: t$ by definition of $\mu'$.

We have (C) $\mu \cup \mu' \models \alpha <\!: \text{Bool}$ since $(\mu \cup \mu')(\alpha) <\!: t$ and $t <\!: \text{Bool}$.

Hence $\mu'$ has properties (A), (B), and (C) as desired.

Case $E = c$. Similar to the case for $E = b$. In this case, and our assumptions (1), (2) and (3) take the forms:

\[
\begin{align*}
\text{(ALGO c) } (\alpha \notin \text{tyvar}(\Gamma^\circ)) \\
\quad \text{integer } c \\
\Gamma^\circ \vdash c \Rightarrow \alpha (\alpha <\!: \text{Int}, \emptyset) \\
\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)
\end{align*}
\]

\[
\begin{align*}
\text{(DECL Expr c)} \\
\quad \text{integer } c \quad t <\!: \text{Int} \\
\mu(\Gamma^\circ) \vdash c : t
\end{align*}
\]

We are to find $\mu'$ with (A) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V)$ (B) $(\mu \cup \mu')(\alpha) <\!: t$, and (C) $\mu \cup \mu' \models C$, where $C = \alpha <\!: \text{Int}$ and $V = \emptyset$. 
Let $\mu' = \{\alpha \mapsto t\}$.

We have (A), since

$$\text{dom}(\mu') = \{\alpha\}$$

$$\text{newtyvar}(\Gamma^0, \alpha, C, V) = \text{tyvar}(\alpha, \alpha <\!: \text{Int}, \emptyset) \setminus \text{tyvar}(\Gamma^0)$$

$$= \{\alpha\}$$

We have (B) $(\mu \cup \mu')(\alpha) <\!: t$, because $\mu'(\alpha) <\!: t$ by definition of $\mu'$.

We have (C) $(\mu \cup \mu')\models \alpha <\!: \text{Int}$ since $(\mu \cup \mu')(\alpha) <\!: t$ and $t <\!: \text{Int}$.

Hence $\mu'$ has properties (A), (B), and (C) as desired.

Case $E = E_1 - E_2$. In this case, our assumptions (1), (2) and (3) take the forms:

$$\text{newtyvar}(\Gamma^0, \alpha, C, V) \cap \text{newtyvar}(\Gamma^0, \alpha_2, C_2, V_2) = \emptyset$$

$$\Gamma^0 \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \quad \Gamma^0 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2)$$

$$\Gamma^0 \vdash E_1 - E_2 \Rightarrow \alpha (\alpha <\!: \text{Int} \wedge \alpha_1 <\!: \text{Int} \wedge \alpha_2 <\!: \text{Int} \wedge C_1 \wedge C_2, V_1 \cup V_2)$$

$$\text{dom}(\mu) = \text{tyvar}(\Gamma^0)$$

$$(\text{DECL EXPR } -) (t_1 <\!: \text{Int} \text{ and } t_2 <\!: \text{Int})$$

$$\mu(\Gamma^0) \vdash E_1 : t_1 \quad \mu(\Gamma^0) \vdash E_2 : t_2 \quad t <\!: \text{Int}$$

We are to find $\mu'$ with (A) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^0, \alpha, C, V)$, (B) $(\mu \cup \mu')(\alpha) <\!: t$, and (C) $\mu \cup \mu' \models C$, where $C = \alpha <\!: \text{Int} \wedge \alpha_1 <\!: \text{Int} \wedge \alpha_2 <\!: \text{Int} \wedge C_1 \wedge C_2$.

By induction hypothesis, (1) $\Gamma^0 \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$, (2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^0)$, and (3) $\mu(\Gamma^0) \vdash E_1 : t_1$ imply there is $\mu'_1$ with (A) $\text{dom}(\mu'_1) = \text{newtyvar}(\Gamma^0, \alpha_1, C_1, V_1)$, (B) $(\mu \cup \mu'_1)(\alpha_1) <\!: t_1$, and (C) $\mu \cup \mu'_1 \models C_1$.

By induction hypothesis, (1) $\Gamma^0 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2)$, (2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^0)$, and (3) $\mu(\Gamma^0) \vdash E_2 : t_2$ imply there is $\mu'_2$ with (A) $\text{dom}(\mu'_2) = \text{newtyvar}(\Gamma^0, \alpha_2, C_2, V_2)$, (B) $(\mu \cup \mu'_2)(\alpha_2) <\!: t_2$, and (C) $\mu \cup \mu'_2 \models C_2$.

The sets $\{\alpha\}$, $\text{dom}(\mu'_1)$, and $\text{dom}(\mu'_2)$ are disjoint, because of the conditions $\alpha \notin \text{tyvar}(\Gamma^0, C_1, C_2, V_1, V_2)$ and $\text{newtyvar}(\Gamma^0, \alpha_1, C_1, V_1) \cap \text{newtyvar}(\Gamma^0, \alpha_2, C_2, V_2) = \emptyset$.

Let $\mu' = \{\alpha \mapsto t\} \cup \mu'_1 \cup \mu'_2$, a well-formed finite map.
CHAPTER 4. TYPE INFERENCE BY ACCUMULATING LOGICAL CONSTRAINTS

56

Case Let. In this case, our assumptions (1), (2) and (3) take the forms

We have (A), since

\[ \text{dom}(\mu^\prime) = \{\alpha\} \cup \text{dom}(\mu_1^\prime) \cup \text{dom}(\mu_2^\prime) \]

\[ \text{newtyvar}(\Gamma^\circ, \alpha, C, V) = \{\alpha\} \cup \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1) \cup \text{newtyvar}(\Gamma^\circ, \alpha_2, C_2, V_2) \]

and \( \alpha \notin \text{tyvar}(\Gamma^\circ, C_1, C_2, V_1, V_2) \)

\[ = \{\alpha\} \cup \text{dom}(\mu_1^\prime) \cup \text{dom}(\mu_2^\prime) \]

We have (B) \((\mu \cup \mu^\prime)(\alpha) :<: t\) because \(\mu^\prime(\alpha) = t\) by definition of \(\mu^\prime\).

We have (C) \(\mu \cup \mu^\prime \models C\) because:

- \(\{\alpha \mapsto t\} \models \alpha :<: \text{Int}\) since \(t :<: \text{Int}\)
- \(\mu \cup \mu^\prime \models \alpha_1 :<: \text{Int}\) since \((\mu \cup \mu^\prime)(\alpha_1) :<: t_1\) and \(t_1 :<: \text{Int}\)
- \(\mu \cup \mu^\prime \models \alpha_2 :<: \text{Int}\) since \((\mu \cup \mu^\prime)(\alpha_2) :<: t_2\) and \(t_2 :<: \text{Int}\)
- \(\mu \cup \mu^\prime \models C_1\)
- \(\mu \cup \mu^\prime \models C_2\)

Hence \(\mu^\prime\) has properties (A), (B), and (C) as desired.

Case Let. In this case, our assumptions (1), (2) and (3) take the forms

\[ (\text{ALGO LET}) \ (x \notin \text{dom}(\Gamma^\circ) \text{ and } \alpha_1 \notin \text{tyvar}(\Gamma^\circ) \text{ and } \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1) \cap \text{newtyvar}(\Gamma^\circ, x : \alpha_1, \alpha_2, C_2, V_2) = \emptyset) \]

\[ \Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \quad \Gamma^\circ, \alpha_1 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2) \]

\[ \Gamma^\circ \vdash \text{let} \ x = E_1 \text{ in } E_2 \Rightarrow \alpha_2 (C_1 \land C_2, \{(x, \alpha_1)\} \cup V_1 \cup V_2) \]

\[ \text{dom}(\mu) = \text{tyvar}(\Gamma^\circ) \]

\[ (\text{DECL EXPR LET}) \ (x \notin \text{dom}(\Gamma)) \]

\[ \mu(\Gamma^\circ) \vdash E_1 : t_1 \quad \mu(\Gamma^\circ), x : t_1 \vdash E_2 : t_2 \]

\[ \mu(\Gamma^\circ) \vdash \text{let} \ x = E_1 \text{ in } E_2 : t_2 \]

We are to find \(\mu^\prime\) with

- (A) \(\text{dom}(\mu^\prime) = \text{newtyvar}(\Gamma^\circ, \alpha_2, C_1 \land C_2, \{(x, \alpha_1)\} \cup V_1 \cup V_2)\),
- (B) \((\mu \cup \mu^\prime)(\alpha_2) :<: t_2\), and
- (C) \(\mu \cup \mu^\prime \models C_1 \land C_2\).

By induction hypothesis, (1) \(\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)\), (2) \(\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)\), and (3) \(\mu(\Gamma^\circ) \vdash E_1 : t_1\) imply there is \(\mu^\prime\) with (A) \(\text{dom}(\mu^\prime_1) = \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1)\), (B) \((\mu \cup \mu^\prime_1)(\alpha_1) :<: t_1\), and (C) \(\mu \cup \mu^\prime_1 \models C_1\). Note, that because \(\alpha_1 \notin \text{tyvar}(\Gamma^\circ)\), it must be \(\alpha_1 \in \text{dom}(\mu^\prime_1)\).
Let $\mu_2 = \mu \cup \{\alpha_1 = \mu_1'(\alpha_1)\}$ and hence $\mu_2(\alpha_1) \iff t_1$.

We have $\text{dom}(\mu_2) = \text{tyvar}(\Gamma^o, x : \alpha_1)$ and $\mu_2(\Gamma^o, x : \alpha_1) = \mu(\Gamma^o), x : \mu_2(\alpha_1)$.

By Lemma 3.6, $\mu(\Gamma^o), x : \alpha_1 \vdash E_2 : t_2$ and $\mu_2(\alpha_1) \iff t_1$ imply $\mu(\Gamma^o), x : \mu_2(\alpha_1) \vdash E_2 : t_2$, and thus $\mu_2(\Gamma^o, x : \alpha_1) \vdash E_2 : t_2$.

By induction hypothesis, (1) $\Gamma^o, x : \alpha_1 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2)$, (2) dom($\mu_2$) = $\text{tyvar}(\Gamma^o, x : \alpha_1)$, and (3) $\mu_2(\Gamma^o, x : \alpha_1) \vdash E_2 : t_2$ imply there is $\mu'_2$ with

- (A) dom($\mu'_2$) = $\text{newtyvar}(\Gamma^o, x : \alpha_1), \alpha_2, C_2, V_2)$,
- (B) ($\mu_2 \cup \mu'_2)(\alpha_2) \iff t_2$, and
- (C) $\mu_2 \cup \mu'_2 \models C_2$.

Let $\mu' = \mu'_2 \cup \mu'_2$. The sets dom($\mu'_1$) and dom($\mu'_2$) are disjoint, because $\text{newtyvar}(\Gamma^o, \alpha_1, C_1, V_1) \cap \text{newtyvar}(\Gamma^o, x : \alpha_1), \alpha_2, C_2, V_2) = \emptyset$.

We have $\mu \cup \mu' = \mu_2 \cup \mu'_1 \cup \mu'_2$:

\[
\begin{align*}
\mu_2 \cup \mu'_1 \cup \mu'_2 &= \mu \cup \{\alpha_1 = \mu'_1(\alpha_1)\} \cup \mu'_1 \cup \mu'_2 \\
&= \mu \cup \mu'_1 \cup \mu'_2 \\
&= \mu \cup \mu'
\end{align*}
\]

We have (A) dom($\mu'$) = $\text{newtyvar}(\Gamma^o, \alpha_2, C_1 \land C_2, \{(x, \alpha_1)\} \cup V_1 \cup V_2)$ as shown by the following calculation.

\[
\begin{align*}
LHS &= \text{dom}(\mu') \\
&= \text{dom}(\mu'_1 \cup \mu'_2) \\
&= \text{dom}(\mu'_1) \cup \text{dom}(\mu'_2) \\
&= \text{newtyvar}(\Gamma^o, \alpha_1, C_1, V_1) \cup \text{newtyvar}(\Gamma^o, x : \alpha_1), \alpha_2, C_2, V_2) \\
&= (\text{tyvar}(\alpha_1, C_1, V_1) \setminus \text{tyvar}(\Gamma^o)) \cup (\text{tyvar}(\alpha_2, C_2, V_2) \setminus \text{tyvar}(\Gamma^o, x : \alpha_1)) \\
&= (\{\alpha_1\} \cup \text{tyvar}(C_1, V_1)) \setminus \text{tyvar}(\Gamma^o)) \cup (\{\alpha_2\} \cup \text{tyvar}(C_2, V_2)) \setminus \text{tyvar}(\Gamma^o)) \setminus \{\alpha_1\}) \\
&= (\{\alpha_1, \alpha_2\} \cup \text{tyvar}(C_1, C_2, V_1, V_2)) \setminus \text{tyvar}(\Gamma^o) \\
RHS &= \text{newtyvar}(\Gamma^o, \alpha_2, C_1 \land C_2, \{(x, \alpha_1)\} \cup V_1 \cup V_2) \\
&= (\{\alpha_1, \alpha_2\} \cup \text{tyvar}(C_1, C_2, V_1, V_2)) \setminus \text{tyvar}(\Gamma^o)
\end{align*}
\]

We have (B) $(\mu \cup \mu')(\alpha_2) \iff t_2$ because $\mu \cup \mu' = \mu_2 \cup \mu'_1 \cup \mu'_2$ and $(\mu_2 \cup \mu'_2)(\alpha_2) \iff t_2$.

We have (C) $\mu \cup \mu' \models C_1 \land C_2$ because $\mu \cup \mu'_1 \models C_1$ and $\mu_2 \cup \mu'_2 \models C_2$. 
Hence $\mu'$ has properties (A), (B), and (C) as desired.

Case Lambda. In this case, our assumptions (1), (2) and (3) take the forms:

\[
\text{(ALGO LAMBDA)} \quad (x \notin \text{dom}(\Gamma^o) \text{ and } \alpha_1 \notin \text{tyvar}(\Gamma^o) \text{ and } \alpha \notin \text{tyvar}((\Gamma^o, x : \alpha_1), \alpha_2, C, V))
\]

\[
\Gamma^o, x : \alpha_1 \vdash E \Rightarrow \alpha_2 \ (C, V)
\]

\[
\Gamma^o \vdash \lambda(x)E \Rightarrow \alpha \ (\alpha <:> \alpha_1 \rightarrow \alpha_2 \land C, \{(x, \alpha_1)\} \cup V)
\]

\[
\text{dom}(\mu) = \text{tyvar}(\Gamma^o)
\]

\[
\text{(DECL EXPR LAMBDA)} \quad (x \notin \text{dom}(\mu(\Gamma^o)))
\]

\[
\mu(\Gamma^o), x : t_1 \vdash E : t_2 \quad t <:> t_1 \rightarrow t_2
\]

\[
\mu(\Gamma^o) \vdash \lambda(x)E : t
\]

We are to find $\mu'$ with

- (A) \(\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, \alpha <:> \alpha_1 \rightarrow \alpha_2 \land C, \{(x, \alpha_1)\} \cup V)\),
- (B) \((\mu \cup \mu')(\alpha) <:> t\), and
- (C) \(\mu \cup \mu' \models \alpha <:> \alpha_1 \rightarrow \alpha_2 \land C\).

Let \(\mu_1 = \mu \cup \{\alpha_1 = t_1\}\).

We have \(\text{dom}(\mu_1) = \text{tyvar}(\Gamma^o, x : \alpha)\) and \(\mu_1(\Gamma^o, x : \alpha_1) = \mu(\Gamma^o), x : t_1\).

By induction hypothesis, (1) \(\Gamma^o, x : \alpha_1 \vdash E \Rightarrow \alpha_2 (C, V)\), (2) \(\text{dom}(\mu_1) = \text{tyvar}(\Gamma^o, x : \alpha_1)\), and (3) \(\mu_1(\Gamma^o, x : \alpha_1) \vdash E : t_2\) imply there is \(\mu'_1\) with

- (A) \(\text{dom}(\mu'_1) = \text{newtyvar}(\Gamma^o, x : \alpha_1, \alpha_2, C, V)\),
- (B) \((\mu_1 \cup \mu'_1)(\alpha_2) <:> t_2\), and
- (C) \(\mu_1 \cup \mu'_1 \models C\).

Let \(\mu' = \mu'_1 \cup \{\alpha_1 = t_1, \alpha = t\}\).

Observe that \(\mu \cup \mu' = \mu_1 \cup \mu'_1 \cup \{\alpha = t\}\)

\[
\mu \cup \mu' = \mu \cup \mu'_1 \cup \{\alpha_1 = t_1, \alpha = t\}
\]

\[
\mu_1 \cup \mu'_1 \cup \{\alpha = t\} = \mu \cup \{\alpha_1 = t_1\} \cup \mu'_1 \cup \{\alpha = t\}
\]

We have (A) \(\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, \alpha <:> \alpha_1 \rightarrow \alpha_2 \land C, \{(x, \alpha_1)\} \cup V)\) as shown by the following calculation.

\[
LHS = \text{dom}(\mu')
\]

\[
= \text{dom}(\mu'_1 \cup \{\alpha_1 = t_1, \alpha = t\})
\]
4.3. FORMAL PROPERTIES: TERMINATION, SOUNDNESS, AND COMPLETENESS

\[
\begin{align*}
= & \quad \text{dom}(\mu') \cup \{\alpha_1, \alpha\} \\
= & \quad \text{newtyvar}(\Gamma^\circ, x : \alpha_1, \alpha_2, C, V) \cup \{\alpha_1, \alpha\} \\
= & \quad (\text{tyvar}(\alpha_2, C, V) \setminus \text{tyvar}((\Gamma^\circ, x : \alpha_1))) \cup \{\alpha_1, \alpha\} \\
= & \quad (\{\alpha_2\} \cup \text{tyvar}(C, V)) \setminus \text{tyvar}(\Gamma^\circ) \cup \{\alpha_1\} \cup \{\alpha\} \\
= & \quad (\{\alpha, \alpha_2\} \cup \text{tyvar}(C, V)) \setminus \text{tyvar}(\Gamma^\circ) \cup \{\alpha_1\} \\
= & \quad (\{\alpha, \alpha_2\} \cup \text{tyvar}(C, V)) \setminus \text{tyvar}(\Gamma^\circ) \\
RHS = & \quad \text{newtyvar}(\Gamma^\circ, \alpha, \alpha \llt \alpha_1 \rightarrow \alpha_2 \land C, \{(x, \alpha_1)\} \cup V) \\
= & \quad (\{\alpha, \alpha_1, \alpha_2\} \cup \text{tyvar}(C, V)) \setminus \text{tyvar}(\Gamma^\circ)
\end{align*}
\]

We have (B) \((\mu \cup \mu')(\alpha) \llt t\) because \(\mu \cup \mu' = \mu_1 \cup \mu'_1 \cup \{\alpha = t\}\).

We have (C) \(\mu \cup \mu' \models \alpha \llt \alpha_1 \rightarrow \alpha_2 \land C\) because \(t \llt t_1 \rightarrow t_2\) and \(\mu' = \mu'_1 \cup \{\alpha_1 = t_1, \alpha = t\}\) and \(\mu \cup \mu' = \mu_1 \cup \mu'_1 \cup \{\alpha = t\}\) and \((\mu_1 \cup \mu'_1)(\alpha_2) = t_2\) and \(\mu_1 \cup \mu'_1 \models C\).

Hence \(\mu'\) has properties (A), (B), and (C) as desired.

Case Appl. In this case, our assumptions (1), (2) and (3) take the forms:

\[
\begin{align*}
\text{(ALGO APPL)} \quad & (\alpha \notin \text{tyvar}(\Gamma^\circ, C_2, C_1) \text{ and } \text{newtyvar}(\Gamma^\circ, \alpha_2, C_2, V_2) \cap \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1) = \emptyset) \\
& \Gamma^\circ \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2) \quad \Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \\
& \frac{}{\Gamma^\circ \vdash E_2(E_1) \Rightarrow \alpha (\alpha_2 \llt \alpha_1 \rightarrow \alpha \land C_1 \land C_2, V_1 \cup V_2)} \\
& \quad \text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)
\end{align*}
\]

\[
\begin{align*}
\text{(DECL EXPR APPL)} \quad & (t_2 \llt t_1 \rightarrow t) \\
& \frac{\mu(\Gamma^\circ) \vdash E_2 : t_2, \mu(\Gamma^\circ) \vdash E_1 : t_1}{\mu(\Gamma^\circ) \vdash E_2(E_1) : t}
\end{align*}
\]

We are to find \(\mu'\) with (A) \(\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V)\), (B) \((\mu \cup \mu')(\alpha) \llt t\), and (C) \(\mu \cup \mu' \models C\), where \(C = \alpha_2 \llt \alpha_1 \rightarrow \alpha \land C_1 \land C_2\) and \(V = V_1 \cup V_2\).

By induction hypothesis, (1) \(\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)\), (2) \(\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)\), and (3) \(\mu(\Gamma^\circ) \vdash E_1 : t_1\) imply there is \(\mu'_1\) with (A) \(\text{dom}(\mu'_1) = \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1)\), (B) \((\mu \cup \mu'_1)(\alpha_1) = t_1\), and (C) \(\mu \cup \mu'_1 \models C_1\).

By induction hypothesis, (1) \(\Gamma^\circ \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2)\), (2) \(\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)\), and (3) \(\mu(\Gamma^\circ) \vdash E_2 : t_2\) imply there is \(\mu'_2\) with (A) \(\text{dom}(\mu'_2) = \text{newtyvar}(\Gamma^\circ, \alpha_2, C_2, V_2)\), (B) \((\mu \cup \mu'_2)(\alpha_2) \llt t_2\), and (C) \(\mu \cup \mu'_2 \models C_2\).

The sets \(\{\alpha\}, \text{dom}(\mu'_1)\), and \(\text{dom}(\mu'_2)\) are disjoint, because of the conditions \(\alpha \notin \)}
tyvar(Γ°, C₁, C₂, V₁, V₂) and newtyvar(Γ°, α₁, C₁, V₁) ∩ newtyvar(Γ°, α₂, C₂, V₂) = ∅.

Let μ' = {α → t} ∪ μ'₁ ∪ μ'₂, a well-formed finite map.

We have (A) dom(μ') = newtyvar(Γ°, α, C, V) because newtyvar(Γ°, α, C, V) = {α} ∪ newtyvar(Γ°, α₁, C₁, V₁) ∪ newtyvar(Γ°, α₂, C₂, V₂).

We have (B) (μ ∪ μ')(α) <:: t because μ'(α) = t by definition of μ'.

We have (C) μ ∪ μ' |= C because:

• α₂ <:: α₁ → α since (μ ∪ μ')(α₂) <:: t₂, (μ ∪ μ')(α₁) <:: t₁, (μ ∪ μ')(α) <:: t, (μ ∪ μ')(α) <:: t, and t₂ <:: t₁ → t
• μ ∪ μ' |= C₁
• μ ∪ μ' |= C₂

Hence μ' has properties (A), (B), and (C) as desired.

\[ \square \]

**Theorem 4.2.** Suppose Γ° ⊢ E ⇒ α (C, V).

For all μ with dom(μ) = tyvar(Γ°) and type t:

(1) (Soundness) For all μ', if dom(μ') = newtyvar(Γ°, α, C, V) and (μ ∪ μ')(α) <:: t and μ ∪ μ' |= C then μ(Γ°) ⊢ E : t.

(2) (Completeness) If μ(Γ°) ⊢ E : t then there is μ' with newtyvar(Γ°, α, C, V) with (μ ∪ μ')(α) <:: t and μ ∪ μ' |= C.

**Proof.** (1) follows from Lemma 4.6 and (2) follows from Lemma 4.7. \[ \square \]

The following corollary can prove the claims made in Section 3.1 about our three motivating examples.

**Corollary 4.1.** Suppose that:

• function \( f(x₁, \ldots, xₙ) \) return \( E \) is an untyped function definition with \( \text{fv}(E) \subseteq \{x₁, \ldots, xₙ\} \)
• \( Γ° = x₁ : α₁, \ldots, xₙ : αₙ \) for fresh type variables \( αᵢ \).
• \( Γ° \vdash E ⇒ α \ (C, V) \)

Then, for all tuples \( (t₁, \ldots, tₙ, t) \), the following propositions are logically equivalent:

(1) \( (x₁ : t₁, \ldots, xₙ : tₙ, f : t) \) is a type signature for \( f \)
(2) \( x₁ : t₁, \ldots, xₙ : tₙ \vdash E : t \)
(3) for all \( \mu = \{ \alpha_1 = t_1, \ldots, \alpha_n = t_n \} \): \( \mu \cup \mu' \models C \) for some \( \mu' \) such that \( \text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, C, \forall) \) and \( (\mu \cup \mu')(\alpha) <: t \).

**Proof.** It is trivial that (1) and (2) are equivalent. By Theorem 4.2 we get that (2) is equivalent to (3). And thus, (1)-(3) are logically equivalent. \( \square \)

**Logical Constraints for our Three Motivating Examples**

In this subsection we show how Corollary 4.1 applies to our motivating examples (see Section 3.6). We show in full details how we extract these for the type signature for the \texttt{diffRange} function. For \texttt{uppercase} and \texttt{intEqual3} we present only the first and the final step of our reasoning.

We have checked these derivations by running our examples through a direct implementation of the algorithmic typing rules for our typed \( \lambda \)-calculus. In the examples that follow for readability we use the subscripts 1, 2 to shorten the definitions, for example \( E_{1,2} \triangleq \text{range}_{1,2}.\text{length} \) should expand to \( E_1 \triangleq \text{range}_1.\text{length} \) and \( E_2 \triangleq \text{range}_2.\text{length} \).

**Example: diffRange**

Recall that the definition for \texttt{diffRange} is:

\[
\begin{align*}
\text{function} \ \texttt{diffRange}(\text{range}_1, \text{range}_2) & \ \text{return} \ \text{range}_1.\text{length} - \text{range}_2.\text{length} \\
\end{align*}
\]

Starting with \( \Gamma^o = \text{range}_{1,2} : \beta_{1,2} \) let:

\[
\begin{align*}
E_{1,2} & \triangleq \text{range}_{1,2}.\text{length}, \quad E_{\text{diffRange}} \triangleq E_1 - E_2 \\
\end{align*}
\]

By Definition 4.7, the following is derivable:

\[
\begin{align*}
\Gamma^o \vdash \text{range}_{1,2} & \Rightarrow \beta_{1,2} \ (\text{true, } \emptyset) \quad \text{Algo } x \\
\alpha_{1,2} \notin \{ \beta_{1,2} \} & \quad \text{Algo Proj} \\
\Gamma^o \vdash A_{1,2} & \Rightarrow (\alpha_{1,2} <: \beta_{1,2}.\text{length} \land \text{true, } \emptyset) \quad \text{Algo Proj} \\
\beta \notin \{ \beta_{1,2}, \alpha_{1,2} \} \text{ and } \{ a_1 \} \cap \{ a_2 \} = \emptyset & \quad \text{Algo } \vdash \text{range}_{1,2} \Rightarrow \beta (\beta <: \text{Int} \land \alpha_{1,2} <: \text{Int} \land \alpha_{1,2} <: \beta_{1,2}.\text{length} \land \text{true, } \emptyset) \\
\Gamma^o \vdash E_{\text{diffRange}} & \Rightarrow \beta (\beta <: \text{Int} \land \alpha_{1,2} <: \text{Int} \land \alpha_{1,2} <: \beta_{1,2}.\text{length} \land \text{true, } \emptyset) \\
\end{align*}
\]

Given this result, for all tuples \( (t_1, t_2, t) \), Corollary 4.1 tells us that each of the following propositions are logically equivalent:

(1) \( (\text{range}_1 : t_1, \text{range}_2 : t_2, \text{diffRange} : t) \) is a type signature for \texttt{diffRange}

(2) \( \text{range}_1 : t_1, \text{range}_2 : t_2 \vdash E_{\text{diffRange}} : t \)
(3) there is $\mu'$ with $\text{dom}(\mu') = \{\alpha_1, \alpha_2\}$ and

$$\{\beta_1 = t_1, \beta_2 = t_2, \beta = t\} \cup \mu' \models (\beta <: Int \land \alpha_1 <: Int \land \alpha_2 <: Int \land \beta_1.\text{length} \land \alpha_1.\text{length} < \beta_2.\text{length} \land \text{true})$$

By further simplification, the following propositions are also equivalent:

(4) by Lemma 4.1(f), there is $\mu'$ with $\text{dom}(\mu') = \{\alpha_1, \alpha_2\}$ and

$$\{\beta_1 = t_1, \beta_2 = t_2, \beta = t\} \cup \mu' \models (\beta = \text{Int} \land \alpha_1 = \text{Int} \land \alpha_2 = \text{Int} \land ((\beta_1 = \text{String} \land \alpha_1 = \text{Int}) \lor (\beta_1 = \text{IntArray} \land \alpha_1 = \text{Int}) \lor (\beta_1 = \text{Range} \land \alpha_1 = \text{Int})) \land ((\beta_2 = \text{String} \land \alpha_2 = \text{Int}) \lor (\beta_2 = \text{IntArray} \land \alpha_2 = \text{Int}) \lor (\beta_2 = \text{Range} \land \alpha_2 = \text{Int})) \land \text{true})$$

(5) by Definition 4.5, there is $\mu'$ with $\text{dom}(\mu') = \{\alpha_1, \alpha_2\}$ and

$$\{\beta_1 = t_1, \beta_2 = t_2, \beta = t\} \cup \mu' \models (\beta = \text{Int} \land \alpha_1 = \text{Int} \land \alpha_2 = \text{Int} \land (\beta_1 = \text{String} \lor \beta_1 = \text{IntArray} \lor \beta_1 = \text{Range}) \land (\beta_2 = \text{String} \lor \beta_2 = \text{IntArray} \lor \beta_2 = \text{Range}))$$

(6) by Definition 4.4

$$(t_1 = \text{String} \lor t_1 = \text{IntArray} \lor t_1 = \text{Range}) \text{ and } (t_2 = \text{String} \lor t_2 = \text{IntArray} \lor t_2 = \text{Range}) \text{ and } t = \text{Int}$$

To summarize our chain of reasoning, we have that:

$$(\text{range1} : t_1, \text{range2} : t_2, \text{diffRange} : t) \text{ is a type signature for } \text{diffRange} \text{ if and only if }$$

$$(t_1 = \text{String} \lor t_1 = \text{IntArray} \lor t_1 = \text{Range}) \text{ and } (t_2 = \text{String} \lor t_2 = \text{IntArray} \lor t_2 = \text{Range}) \text{ and } t = \text{Int}$$
Example: uppercase

By Corollary 4.1 and similar reasoning to the previous example, we get:

\[(str : t_1, uppercase : t) \text{ is a type signature for } uppercase\]
\[\text{if and only if}\]
\[((t_1 = IntArray \text{ or } t_1 = String) \text{ and } (t = IntArray \text{ or } t = String))\]

Example: intEqual3

Again, by Corollary 4.1 and similar reasoning we get:

\[(int1 : t_1, int2 : t_2, int3 : t_3, intEqual3 : t) \text{ is a type signature for } intEqual3\]
\[\text{if and only if}\]
\[((t_1 = Bool \text{ and } t_2 = Bool \text{ and } t_3 = Bool) \text{ or}\]
\[(t_1 = Int \text{ and } t_2 = Int \text{ and } t_3 = Int)\] and \(t = Bool\)
Defining Natural Type Inference

In this chapter, we first formalize the notion of natural information in our setting, what we call natural constraints, in Section 5.1. We then define what is a maximally natural satisfying type valuation, as an environment that both satisfies the logical constraints defined in Chapter 4 and maximizes the information we take from natural constraints defined in this chapter. Equipped with that, we define a new natural type inference problem for function definitions, where we extend the type inference problem defined in Problem 3.1 with the requirement to get the maximally natural type signature Problem 5.2. Finally, we give an overall algorithm on how the natural type signature problem can be solved. On this outline of the algorithm we abstract the exact way of how we combine the logical and natural constraints, by just setting the requirements of the problem. In the next two chapters we will show two different ways of unifying the two channels of information and also prove the correctness of our overall algorithm.

5.1 Formalizing Textual Information as Natural Constraints

Source code is bimodal: it interlinks a formal, algorithmic channel aimed at devices and a natural language channel of identifiers and comments aimed at developers (Casalnuovo et al., 2020). Particularly, in our work we use the bimodality of code to enhance classical type inference by using statistical dependencies in the source code. As discussed in Section 2.3 there is a growing interest in various learning-based methods for type inference that exploits some form of Natural Language Processing (NLP) (Jurafsky and Martin, 2009) to extract information from source code. To mention some examples here, names
of variables provide information about their types (Xu et al., 2016), natural language in
method-level comments provide information about function types (Malik et al., 2019),
and lexically nearby tokens provide information about a variable’s type (Wei et al., 2020;
Hellendoorn et al., 2018).

This source of information is indirect, and thus difficult to formalize, but we can still hope
to exploit it by applying machine learning to large corpora of source code. Following the
idea that the naturalness in source code (Hindle et al., 2012) may be in part responsible for
the effectiveness of learning-based type inference, we refer generically to indirect, statistical
constraints about types as natural constraints. As this type of constraints are uncertain,
they are naturally formalized as probabilities.

The inference systems of the previous chapter assume unbounded sets of type variables $\alpha$
and type names $t$. The freshness conditions in the (Algorithmic Typing Rules) assume we
can always pick fresh type variables. However, in the setting of our top-level problem, of
inferring types for a function definition \texttt{function} $f(x_1, \ldots, x_n)$ \texttt{return} $E$, we can limit
our attention to the finite sets of type variables needed to generate constraints, and a
finite library of definitions for type names. More specifically, when inferring types for a
particular function definition, we consider a finite sequence of $V > 0$ type variables $\alpha_1,$
$\ldots, \alpha_V$, and a finite sequence of $T > 0$ type names $t_1, \ldots, t_T$.

We generate natural constraints for each type variable $\alpha_v$ based on the set $V$ of (identifiers,
type variables) pairs, which are obtained as the output of the (Algorithmic Typing Rules).
We now describe the formal details of formulating the natural constraints.

**Natural Constraints**

As discussed in Section 2.3 researchers have recently developed innovative learning-based
techniques to predict missing type annotations (Raychev et al., 2015; Hellendoorn et al.,
2018; Wei et al., 2020; Pradel et al., 2020; Allamanis et al., 2020; Mir et al., 2022). We
can abstract these techniques as a function $NC$ that provides a probability distribution
based on the identifiers associated with each type variable. We later exploit this function
$NC$ for each type variables to construct a natural matrix $N$ (Definition 5.1) that forms an
integral part of our type inference problem.

Formally, $NC$ is a function that maps a set of identifiers to a distribution over type names.
In practice (and as done in our implementation as well), the function $NC$ is computed by
training a machine learning model. Thus, the function can return a distribution over type
names for any set of identifiers.
However, we intend to predict the types only based on the identifiers associated with a type variable gathered in the set $V$ from our (Algorithmic Typing Rules). Thus, for each type variable $\alpha_v$, we define a set $\text{id}^v = \{id | (id, \alpha_v) \in V\}$, which is the finite set of identifiers associated with a type variable.

Now, for a set $\text{id}^v$, $\text{NC}(\text{id}^v)$ is a probability vector $n_v$ of length $T$ representing the distribution over named types for $\alpha_v$. A type variables maybe associated with zero, one, or more identifiers, as explained in Section 4.2. In the special case, when $V_v = \emptyset$ for a type variable $\alpha$, that is when there are no identifiers associated with $\alpha$, we set $\text{NC}(\emptyset)$ to be the uniform probability vector $[1/T, \ldots, 1/T]$. This is because, without any associated identifiers, one cannot get any natural language information for a type variable.

We now provide the definition of natural constraints.

**Definition 5.1 (Natural Constraints).** Let $V$ be a set of pairs (identifiers, type variable) and $\text{NC}$ be the function that maps sets of identifiers to distribution over type names. Further, let $\text{id}^v = \{id | (id, \alpha_v) \in V\}$ be the set of identifiers associated with type variable $\alpha_v$ for each $v \in 1..V$. We then define a natural constraint for a type variable $\alpha_v$ to be the probability vector $\text{NC}(\text{id}^v) \triangleq n_v = [n_{v,1}, \ldots, n_{v,T}]$. We aggregate the natural constraints $n_v$ for all $v \in 1..V$ into a natural matrix of size $V \times T$ using a function $\text{NatConstr}$ as follows:

$$\text{NatConstr}(V) \triangleq \left[ \text{NC}(\text{id}_1) \ldots \text{NC}(\text{id}_V) \right] = \left[ n_1^\top \ldots n_V^\top \right]^\top.$$

For the sake of brevity, when $V$ and $\text{NC}$ are clear from the context, we represent the natural matrix $\text{NatConstr}(V)$ simply as $N$.

The above definition can easily be generalized for any other properties, for instance comments or lexical scope, that can be associated with a type variable during training.

**Natural Value**

Our aim is to obtain a type valuation by utilizing information from the natural constraints. To this end, we first rewrite a type valuation $\mu$ as a $V \times T$ type valuation matrix. Such a rewrite enables us to be consistent with the notation of a subsequent chapter, Chapter 6, where we relax type valuations to provide a distribution over named types.

**Definition 5.2 (Type Valuation Matrix).** Given a type valuation $\mu$ over the type variables $\{\alpha_v | v \in 1..V\}$, we define a type valuation matrix as:

$$\text{Bin}(\mu) \triangleq \left[ m_1^\top \ldots m_V^\top \right]^\top.$$  

(5.1)
where \( m_v \triangleq \begin{bmatrix} m_{v,1} & \ldots & m_{v,T} \end{bmatrix} \) is a row one-hot vector for a type variable \( \alpha_v \) with \( m_{v,\tau} = 1 \) if \( \mu(\alpha_v) = t_\tau \), and 0 otherwise, where \( \tau \in 1..T \). The row \( m_v \) is an alternative way of providing a valuation of \( \alpha_v \) over named types. Further, we define \( \text{Bin}^{-1} \) to be the inverse transformation that allows us to go from a type valuation matrix \( \text{Bin}(\mu) \) to a valuation \( \mu \). Let \( \mathcal{M}^{V \times T} \) be the set of all type valuation matrices.

We proceed by defining a metric that enables us to determine the naturalness of a type valuation matrix \( M \) as the following.

**Definition 5.3 (Natural Value).** Given a type valuation matrix \( M \in \mathcal{M}^{V \times T} \) and a natural matrix \( N \in \mathcal{P}^{V \times T} \) we define natural value as

\[
\text{NatVal}_N(M) \triangleq \sum_{v=1}^{V} m_v \cdot n_v = \sum_{v=1}^{V} \sum_{\tau=1}^{T} n_{v,\tau} \cdot m_{v,\tau}
\] (5.2)

### 5.2 The Natural Type Inference Problem for Function Definitions

The goal of our problem is to choose a valuation \( \mu \) that both satisfies \( C \) and simultaneously, maximizes the information we obtain from \( N \). We can formally define this, as follows:

**Problem 5.1 (Maximally Natural Type Valuation).** Given a formula of logical constraints \( C \) in equational logic over \( \text{tyvar}(C) = \{ \alpha_v \mid v \in 1..V \} \) and a natural matrix \( N \) of size \( V \times T \), find a type assignment \( \mu^* \) with \( \text{dom}(\mu^*) = \text{tyvar}(C) \) that has the following properties:

1. \( \mu^* \models C \); and
2. for every type valuation \( \mu \) that satisfy the logical constraints \( C \) (that is, \( \mu \models C \)) the following holds:

\[
\text{NatVal}_N(\text{Bin}(\mu)) \leq \text{NatVal}_N(\text{Bin}(\mu^*)).
\]

We refer to a type assignment \( \mu^* \) that satisfies the properties (1) and (2) as a *maximally natural type valuation* with respect to logical constraints \( C \) and natural matrix \( N \). Note that \( \mu^* \) may not be unique, but we are interested in finding any one. Also, for the sake of readability, we represent such a type valuation \( \mu^* \) that satisfies the above mentioned properties using the notation \( \mu^* \models (N \mid C) \).

We now frame the problem of finding the maximally natural type signature. By combining Corollary 4.1 and Problem 5.1, the problem describes as follows:
5.3. High-level Overview of the Algorithm for Natural Type Inference

Problem 5.2 (maximally natural Type Signature). Suppose that:

- function \( f(x_1, \ldots, x_n) \) return \( E \) is a function definition with \( \text{fv}(E) \subseteq \{x_1, \ldots, x_n\} \)
- \( \Gamma^\circ = x_1 : \alpha_1, \ldots, x_n : \alpha_n \) for fresh type variables \( \alpha_i, i \in 1..n \)
- \( \Gamma^\circ \vdash E \Rightarrow \alpha (C, V) \)
- \( N = \text{NatConstr}(V \cup \{(x_1, \alpha_1), \ldots, (x_n : \alpha_n), (f, \alpha)\}) \)

Find a tuple for \( f \)

\[
\Sigma^* = (x_1 : \mu^*(\alpha_1), \ldots, x_n : \mu^*(\alpha_n), f : \mu^*(\alpha)),
\]

with \( \text{dom}(\mu^*) = \text{tyvar}(\Gamma^\circ) \) and \( \mu^* \upharpoonright (N \mid C) \).

We define the type signature \( \Sigma^* \) as the maximally natural type signature. We regard as natural type signature only the signatures that both satisfy the logical constraints and maximize the natural ones. In this chapter, we present our first approach to unify information from two sources, namely logical constraints derived from the algorithmic type system defined in Chapter 4 and natural constraints using machine learning described earlier in this chapter.

5.3 High-level Overview of the Algorithm for Natural Type Inference

We now present the high-level overview of the algorithm to solve Problem 5.2, that is to determine the natural type signature. Our algorithm has a couple of global parameters:

- We have an existing ambient library of type definitions:

  \[
  \text{type } t_1 = S_1 \quad \ldots \quad \text{type } t_T = S_T
  \]

- We assume that we are given a function \( \text{NatConstr} \) that maps identifiers to probability distribution over types.

We consider as input a function definition:

\[
\text{function } f(x_1, \ldots, x_n) \text{ return } E
\]

and our goal is to find a natural type signature \( \Sigma^* = (x_1 : t^*_1, \ldots, x_n : t^*_n, f : t^*) \) for \( f \).

(1) Check that \( E \) is well-scoped with \( \text{fv}(E) \subseteq \{x_1, \ldots, x_n\} \), and if not, terminate with an
error.

(2) Let \( \Gamma^\circ = x_1 : \alpha_1, \ldots, x_n : \alpha_n \) for fresh type variables \( \alpha_i \).

(3) Run the algorithmic typing rules to derive \( \Gamma \vdash E \Rightarrow \alpha (C, V_E) \).

(4) Let \( V = V_E \cup \{(x_1, \alpha_1), \ldots, (x_n, \alpha_n), (f, \alpha)\} \). Let \( N = \text{NatConstr}(V) \).

(5) Solve the natural type inference problem and find a valuation \( \mu^* \) with \( \text{dom}(\mu^*) = \text{tyvar}(\Gamma^\circ) \) such that:

(a) for some \( \mu' \) with \( \text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V) \) and \( \mu \cup \mu'(\alpha) = t \)

\[ \{\mu^*(\alpha_1) = t_1^*, \ldots, \mu^*(\alpha_n) = t_n^*\} \cup \mu' \models C; \]

and

(b) \( \mu^* \cup \mu' \upharpoonright (N \mid C) \).

(6) If Step (5) terminates successfully, then return \( \Sigma^* = (x_1 : \mu^*(\alpha_1), \ldots, x_n : \mu^*(\alpha_n), f : \mu^*(\alpha)) \) as the maximally natural type signature for \( f \).

The most important (and challenging) step of the above algorithm is Step (5) that solves Problem 5.1, that is to find the natural type valuation \( \mu^* \). In the next two chapters (Chapters 6 and 7), we will examine different approaches of solving this problem using tools from mathematical optimization. Later, in Chapter 8, we provide an empirical comparison of the different approaches.

The correctness and termination of the above algorithm also depends on the approach used for Step (5). This is because, we can already assert that the rest of the steps are correct and terminate based on the results discussed so far. Steps (1), (2) and (6) are trivially correct and terminate by construction. For Step (3), Theorem 4.1 and Theorem 4.2 prove the termination and correctness, respectively. Step (4) simply invokes a pre-trained machine learning model on the identifiers of the source code and thus, is correct and terminates. Hence, all depends on the guarantees of the approach used for Step (5). In the subsequent chapters, we expand upon what theoretical guarantees one can expect from the presented approach.
Natural Type Inference using Continuous Optimization

In this chapter, we present a continuous optimization approach to unify information from two sources, namely logical constraints derived from the algorithmic type system defined in Chapter 4 and natural constraints using machine learning described in Chapter 5. A logical constraint is a formula $C$ that describes necessary conditions for $E$ to be well-typed.

As described in Section 4.2, an instance of an algorithmic typing judgment takes the form $\Gamma^o \vdash E \Rightarrow \alpha (C, V)$. In principle, $C$ can be any formula such that if $\mu_1(\Gamma^o) \vdash E : (\mu_1 \cup \mu_2)(\alpha)$ with $\text{dom}(\mu_1) = \text{tyvar}(\Gamma^o)$ and a $\mu_2$ with $\text{dom}(\mu_2) = \text{newtyvar}(\Gamma^o, \alpha, C, V)$, then $\mu_1 \cup \mu_2 \models C$. From now on, let $\mu = \mu_1 \cup \mu_2$, this is for reader’s convenience as we are interested in finding a valuation for all type variables and not only the ones that are apparent in $\Gamma^o$. As discussed in Chapter 5 we can also exploit natural constraints $N$ to help us determine the maximally natural type valuation (Problem 7.1). Remember, that we have already relaxed $\mu$ to a $V \times T$ matrix in Definition 5.2. To combine these two constraints, we relax the boolean operations $C$ to continuous operators on $[0, 1]$ using fuzzy logic, which we briefly review in Section 6.1. We then define a continuous interpretation of the semantics of $C$ (Section 6.2) and describe its properties (Section 6.3). Most importantly, we provide conditions under which the continuous relaxation satisfies the boolean constraints Theorem 6.1.

Having done this, we first establish a constrained optimization problem (6.3) which directly solves natural type inference problem. We then propose a method of solving the constrained problem using a standard method from numerical optimization, called the penalty method, which re-formulates the problem as a series of unconstrained optimization problems (6.7). By a standard result Theorem 6.3 shows that the two formulations are equivalent. Finally,
we state Theorem 6.2 that shows how this formulation can give a solution to the natural type inference problem. In Chapter 8 we evaluate the algorithm in practice.

6.1 Background: Fuzzy Logic

The concept of fuzzy logic was introduced by Zadeh (1965) as an extension of classical logic that allows for the representation and processing of imprecise or vague information. In particular, t-norm based fuzzy logics take the real unit interval \([0,1]\) as a set of truth values, and interpret the various binary operations as functions on the unit interval \([0,1]\). A t-norm is a commutative and associative binary operation in \([0,1]\), non-decreasing in both variables and having 1 and 0 as neutral and absorbent elements respectively (Hájek, 1998). The are three widely know t-norm based logics: Gödel logic, Łukasiewicz logic, and product logic, with the latter attracting recently more interest from the machine learning community (Rocktäschel et al., 2015; Ryan et al., 2020; Yao et al., 2020). The main definition in product logic is that the classical boolean conjunction is interpreted as the product of the relaxed semantics of the conjuncts (Hájek et al., 1996). Product logic is well-suited for our optimization-based approach as its relaxations are smooth, and thus allows backpropagation (Evans and Grefenstette, 2018), which is essential for learning techniques.

6.2 Relaxation of the Logical Constraints

We proceed by defining a relaxation for the discrete logical constraints (see Definition 4.2). In Definition 6.1 we have defined a probability matrix \(P\) to hold relaxed probabilistic types. This is the object we optimize in our key formulation, (6.3), below. To do so we first define relaxed semantics for the logical constraints \(C\) over \(P\), and establish important properties of this semantics. Our semantics is based on product logic (Section 6.1); however, our atomic propositions \(\alpha_v = t_r\) and their interpretation via the matrix \(P\) are original (Definition 6.2).

**Definition 6.1 (Typing Probability Matrix \(P\)).** We define a probability matrix of size \(V \times T\) as

\[
P \triangleq \begin{bmatrix} p_{1}^{T} & \cdots & p_{V}^{T} \end{bmatrix}^{T}
\]

where each \(p_v \triangleq \begin{bmatrix} p_{v,1} & \cdots & p_{v,T} \end{bmatrix}\) is a row vector that defines a probability distribution over named types. Let \(\mathcal{P}^{V \times T}\) be the set of all typing probability matrices of size \(V \times T\).

**Definition 6.2 (Relaxed Semantics).** The relaxed semantics of \(C\) is a function \([C]_P : \)

\[
\]
\( P^{\mathcal{V} \times \mathcal{T}} \times \mathcal{C} \rightarrow [0, 1] \), where \( v \in 1 \ldots \mathcal{V} \), \( \tau \in 1 \ldots \mathcal{T} \) and \( \mathcal{C} \) is the set of logical constraints, defined as:

\[
\begin{align*}
[\mathbf{true}]_P &= 1 \\
[\alpha_v = t_{v, \tau}]_P &= p_{v, \tau} \\
[\neg C]_P &= 1 - [C]_P \\
[C_1 \land C_2]_P &= [C_1]_P \cdot [C_2]_P.
\end{align*}
\]

Let the derived \( t \)-conorm be \([C_1 \lor C_2]_P = [C_1]_P + [C_2]_P - [C_1]_P \cdot [C_2]_P\). Now, we show that our relaxed semantics \([C]_P\) is bound within \([0..1]\) and establish useful equivalences on it, which we use to prove Theorem 6.1. Note that the definition of negation in Definition 6.2 follows previous related work (Rocktäschel et al., 2015; Yao et al., 2020) and is not the standard one for product logic. This choice is more appropriate in our case as we want to keep the relaxation as a smooth function in \([0..1]\).

**Lemma 6.1.** For all logical constraints \( C \) and all \( P \in P^{\mathcal{V} \times \mathcal{T}} \), we have \( 0 \leq [C]_P \leq 1 \).

*Proof.* By structural induction on the expression \( C \). \( \square \)

**Lemma 6.2.** For all \( C, C_1, C_2, \) and \( P \in P^{\mathcal{V} \times \mathcal{T}} \):

i. \( ([C]_P = 0) \) if and only if \( ([\neg C]_P = 1) \)

ii. \( ([\neg C]_P = 0) \) if and only if \( ([C]_P = 1) \)

iii. \( ([C_1]_P = 1 \text{ and } [C_2]_P = 1) \) if and only if \( ([C_1]_P \cdot [C_2]_P = 1) \).

*Proof.* These follow by cases analyses based on Lemma 6.1. \( \square \)

### 6.3 Relaxation and Rounding of Type Valuations

In optimization, a relaxation approximates a difficult (usually discrete) problem with a nearby, easier (often continuous) problem. Our approach relaxes the natural type inference problem and then recovers a discrete type valuation from the relaxation, all the while preserving validity. So we first show how to relax a type valuation, then define a continuous probability distribution over all possible type valuations, conditioned on a type valuation probability matrix \( P \) (Definition 6.1). We close by showing that this relaxation preserves validity.
The following asserts essentially that the relaxed semantics is a continuous function that always agrees with the logical semantics.

**Lemma 6.3** (Binary Relaxation). For all $C$ and $\mu$: $[C]_{\text{Bin}(\mu)} = 1$ if and only if $\mu \models C$.

**Proof.** By structural induction on the constraint $C$. □

We have just shown how to relax a type valuation. We also need to go the other way and recover a type valuation from a continuous relaxation, a procedure known as *rounding*. To this end, we introduce a random variable that denotes type valuation, which is instrumental in recovering a discrete type valuation.

**Definition 6.3.** Let $\tilde{\mu}$ be a random variable that denotes type valuations. We define the probability mass function of $\tilde{\mu}$ as follows:

$$
\Pr[\tilde{\mu} = \mu \mid P] = \prod_{v=1}^{V} \prod_{\tau=1}^{T} (p_{v,\tau})^{\delta(\mu(\alpha_v), t_\tau)},
$$

where $p_{v,\tau}$ denotes the probability that the type variable $\alpha_v$ is of type $t_\tau$ (as defined in Definition 6.1), $\delta$ is the Kronecker delta, defined as $\delta(i, j) = 1$ if $i = j$, otherwise 0. Also, we conventionally use $0^0 = 1$.

Additionally, when we know we are referring to $\tilde{\mu}$, we write $\Pr[\mu \mid P]$.

**Theorem 6.1** (Continuous Relaxation of Logical Constraints). Consider any $\mu$ and any $P \in \mathcal{P}^{V \times T}$, such that $\Pr[\mu \mid P] > 0$. For all $C$, we have that:

$$
[C]_P = 1 \implies \mu \models C \quad (6.2a)
$$

$$
[C]_P = 0 \implies \mu \models \neg C. \quad (6.2b)
$$

**Proof.** The proof is by induction on the structure of $C$, proceeding by a case analysis of $C$.

Case $(a_v = t_\tau)$. The base case is $C = (a_v = t_\tau)$.

For Equation (6.2a), we have that because $[a_v = t_\tau]_P = 1$, and $[a_v = t_\tau]_P = p_{v,\tau}$ by definition, then we have $p_{v,\tau} = 1$. Let $\mu(a_v) = t_{\tau'}$. Then $\Pr[\mu \mid P] = \epsilon p_{v,\tau}$ for some $\epsilon \in \mathbb{R}$. Since $p_{v,\tau} = 1$ and $p_v$ is a probability vector, then for $\tau'' \neq \tau$, we have $p_{v,\tau''} = 0$. Therefore, $\Pr[\mu \mid P] > 0$ implies that $\tau' = \tau$, which implies that $\mu \models C$.

For Equation (6.2b), we are to show that $[a_v = t_\tau]_P = 0$ implies $\mu \models \neg C$. We have that $[a_v = t_\tau]_P = p_{v,\tau}$ by definition, and so $p_{v,\tau} = 0$. We are to show that $\mu \models \neg (a_v = t_\tau)$. For a contradiction, we assume the contrary, which by definition
means that $\mu \models (a_v = t_\tau)$. If $\mu \models (a_v = t_\tau)$ then by definition $\mu(a_v) = t_\tau$. Then $\Pr[\mu \mid P] = \epsilon p_{v,\tau}$ for some $\epsilon \in \mathbb{R}$. Since $p_{v,\tau} = 0$ we obtain that $\Pr[\mu \mid P] = 0$, which contradicts our assumption that $\Pr[\mu \mid P] > 0$.

Case $C = \neg C'$. Both for Equation (6.2a), Equation (6.2b) we have that $\Pr[\mu \mid P] > 0$.

For Equation (6.2a), we suppose that $\llbracket \neg C' \rrbracket_P = 1$, which by Lemma 6.2i implies that $\llbracket C' \rrbracket_P = 0$. By the induction hypothesis of Equation (6.2b) we have that $\mu \models \neg C'$.

For Equation (6.2b), we suppose that $\llbracket \neg C' \rrbracket_P = 0$, which by Lemma 6.2i implies that $\llbracket C' \rrbracket_P = 1$. By the induction hypothesis of Equation (6.2a) we have that $\mu \models C'$, which by definition implies that $\mu \models \neg \neg C'$.

Case $C = (C_1 \land C_2)$. We have that $\Pr[\mu \mid P] > 0$.

For Equation (6.2a), we suppose that

\[
\llbracket C_1 \land C_2 \rrbracket_P = 1
\]

implies $\llbracket C_1 \rrbracket_P \cdot \llbracket C_2 \rrbracket_P = 1$ (Definition 6.2)

implies $\llbracket C_1 \rrbracket_P = 1$ and $\llbracket C_2 \rrbracket_P = 1$ (Lemma 6.1)

implies $\mu \models C_1$ and $\mu \models C_2$ (Inductive Hypothesis)

implies $\mu \models C_1 \land C_2$ (Definition 4.4).

For Equation (6.2b), we suppose that

\[
\llbracket (C_1 \land C_2) \rrbracket_P = 0
\]

implies $\llbracket C_1 \rrbracket_P \cdot \llbracket C_2 \rrbracket_P = 0$ (Definition 6.2)

implies $\llbracket C_1 \rrbracket_P = 0$ or $\llbracket C_2 \rrbracket_P = 0$

implies $\mu \models \neg C_1$ or $\mu \models \neg C_2$ (Inductive Hypothesis, twice)

implies $\mu \models (\neg C_1 \lor \neg C_2)$ (Definition 4.4)

implies $\mu \models \neg (C_1 \land C_2)$ (De Morgan’s Law).
6.4 Natural Type Inference as a Continuous Optimization Problem

Given a program that admits multiple, correct, concrete type valuations given a type library (like our motivating examples in Section 3.1), the core intuition of this dissertation is that, the logical and natural constraints can interact to expedite the process of finding a type valuation that (1) type checks and (2) is maximally natural (see Problem 7.1).

In this section, we focus on finding a probability matrix $P^*$ that has these properties; and then we discuss how to obtain a valuation $\mu^*$ from $P^*$. Our insight is that we can combine these two requirements into a constrained continuous optimization problem.

Intuitively, we design the optimization problem to be over probability matrices $P \in \mathcal{P}^{V \times \tau}$. In the problem, we wish to find the matrix $P^*$ that is maximally natural, in the sense that it has the maximum $\text{NatVal}_N$ among all probability matrices that satisfy the logical constraints. As described in the previous section, this translates to $[C]_{P^*} = 1$ which we pose as a hard constraint to the optimization problem. Note that, while, in Definition 5.3, we define $\text{NatVal}_N$ for type valuation matrices, we can extend it naturally to any other matrix that has a dimension same as $N$.

Hence, we obtain the constrained optimization problem $O_{C,N}(P)$ which is defined by

$$\max_{P \in \mathcal{P}^{V \times \tau}} \text{NatVal}_N(P) \quad \text{s.t.} \quad [C]_P = 1 \quad (6.3)$$

where $P$ is a typing probability matrix as defined in Definition 6.1.

We like to exploit the solution of 6.3 to extract a maximally natural type valuation, which is our original goal. Towards this, we first relate type valuations to continuous typing matrices using the following definition.

**Definition 6.4.** We say that a type valuation $\mu$ is “maximally consistent” with a typing probability matrix $P$ and natural matrix $M$ if the following holds:

1. for all $v \in 1..V$, $\mu(\alpha_v) = t_\tau$ implies $p_{v,\tau} > 0$.
2. for all $v \in 1..V$, $\mu(\alpha_v) = t_\tau$ implies $n_{v,\tau} \geq n_{v,\tau'}$ where $\tau'$ is any index such that $p_{v,\tau'} > 0$.

From a given typing probability matrix $P$ and natural matrix $N$, one can easily construct a maximally consistent type valuation $\mu$ as follows: for each $v \in 1..V$, set $\mu(\alpha_v) = t_\tau$, where
where $\tau$ is an index where the typing probability $p_{v,\tau}$ is non-zero and also, the natural value $n_{v,\tau}$ achieves maximum (chosen at random if there are multiple such indices).

We now state the result that enables us to utilize the solution of 6.3 to derive a maximally natural type valuation.

**Theorem 6.2.** Let $C$ be the logical constraints, $N$ be the natural matrix, $\text{NatVal}_N$ be the natural value function with respect to $N$, and $[C]_P$ be the relaxed semantics of $C$. Now, consider $P^*$ to be a solution of the problem $O_{C,N}(P)$, described in (6.3), and $\mu^*$ to be a type assignment maximally consistent with $P^*$ and $N$. Then, $\mu^*$ has the following properties:

1. $\mu^* \models C$; and
2. for all $\mu \neq \mu^*$ such that $\mu \models C$, $\text{NatVal}_N(\text{Bin}(\mu)) \leq \text{NatVal}_N(\text{Bin}(\mu^*))$.

**Proof.** For (1), we rely on the first criterion of $\mu^*$ being maximally consistent, mentioned in Definition 6.4, to show the following:

$$\text{Pr}[\mu^* \models P^*] = \prod_{v=1}^{V} p_{v,\text{ind}(\mu^*(v))} > 0,$$

(6.4)

where $\text{ind}$ provides the index of the named type. Moreover, we have $[C]_{P^*} = 1$, since $P^*$ is a solution to 6.3. Thus, by Theorem 6.1, $\text{Pr}[\mu^* \models P^*] > 0$ and $[C]_{P^*} = 1$ imply that $\mu^* \models C$, which gives us (1).

For (2), we show that $\text{NatVal}_N(P^*) = \text{NatVal}_N(\text{Bin}(\mu^*))$. First, observe that $\text{Bin}(\mu^*)$ is a typing probability matrix for which $[C]_{\text{Bin}(\mu^*)} = 1$ (since $\mu^* \models C$). Thus, $\text{NatVal}_N(P^*) \geq \text{NatVal}_N(\text{Bin}(\mu^*))$, since $P^*$ is a solution for 6.3. Next, observe the following inequality that is a result of the second criteria of $\mu^*$ being maximally consistent as defined in Definition 6.4:

$$\text{NatVal}_N(\text{Bin}(\mu^*)) = \sum_{v=1}^{V} \sum_{\tau=1}^{T} n_{v,\text{ind}(\mu^*(v))} p_{v,\tau} n_{v,\text{ind}(\mu^*(v))} \geq \sum_{v=1}^{V} \sum_{\tau=1}^{T} p_{v,\tau} n_{v,\tau} = \text{NatVal}_N(P^*)$$

Finally, for any type valuation $\mu \models C$, we have $\text{NatVal}_N(\text{Bin}(\mu^*)) = \text{NatVal}_N(P^*) \geq \text{NatVal}_N(\text{Bin}(\mu))$, since $\text{Bin}(\mu)$ is a typing probability matrix that satisfies $[C]_{\text{Bin}(\mu)} = 1$. \qed
Techniques to solve Optimization Problem 6.3

To solve the constrained optimization problem (6.3), we need to make some remarks about its structure. The most important one is that the constraint \([C]_P = 1\) can be a non-linear constraint over the optimization variables, as it contains a multiplication of more than two variables, making the problem challenging. In fact, one can check that it is possible that \([C]_P\) is an arbitrarily high-order polynomial over the optimization variables. In principle, we could give the above problem directly to an off-the-shelf optimizers that handles globally constrained optimization problems. However, most optimizers are effective for (or even allow) optimization problems with constraints consisting of only up to second-order polynomials and will not work directly.

A common workaround is to convert the constrained problem to an equivalent unconstrained problem (6.8), in an approach inspired by the penalty method (Bertsekas, 1982; Boyd and Vandenberghe, 2004; Luenberger and Ye, 2015). Penalty methods are commonly used as they offer a straightforward way to handle constrained problems without the need of sophisticated tools and instead use general off-the-shelf optimization methods. In short, we approximate the constrained problem by adding to the objective function a term that prescribes a high score for the violation of the constraints. This additional term includes a multiplier \(c\) that determines the severity of the penalty. Ideally as \(c\) increases towards infinity the solution point of the penalty problem will converge to a solution of the constrained problem.

To do so first, we reparameterize the problem to remove the probability constraints, by using the softmax function

\[
\sigma(x) = \left[ \frac{\exp\{x_1\}}{\sum_i \exp\{x_i\}}, \frac{\exp\{x_2\}}{\sum_i \exp\{x_i\}}, \ldots \right]^T,
\]

which maps real-valued vectors to probability vectors. Next, we sometimes abuse the definition of softmax to use it over matrices where each row is a probability vector, meaning that the softmax function is applied over each row.

Using (6.5), our transformed problem takes the form

\[
\max_{Y \in \mathbb{R}^{V \times T}} \text{NatVal}_N(\sigma(Y))
\]

s.t. \([C]_{\sigma(Y)} = 1\)

(6.6)

where \(N\) is the natural matrix and \(Y = \left[ y_1^T \ldots y_V^T \right]^T\) is a matrix of size \(V \times T\) consisting
of real variables and \( \sigma(Y) = \left[ (\sigma(y_1))^T \ldots (\sigma(y_V))^T \right]^T \).

Our transformation of the optimization problem ensures that if some real-valued matrix \( Y^* \) maximizes (6.6), then \( P^* = \sigma(Y^*) \) maximizes (6.3). This is because, any probability vector can, in principle, be represented as the softmax of a real-valued vector.

We now remove the constraint in (6.6) by introducing a penalty \( c_\ell \) yielding the final form of our unconstrained optimization problem

\[
\max_{Y \in \mathbb{R}^{V \times T}} U_{C,N}(c_\ell, Y) = \text{NatVal}_N(\sigma(Y)) - c_\ell \, \text{Pen}(\sigma(Y)) \tag{6.7}
\]

where \( N \) is the natural matrix, \( c \) is a positive constant and \( \text{Pen} \) is a function over \( \mathbb{R}^{V \times T} \) satisfying: (i) \( \text{Pen} \) is continuous, (ii) \( \text{Pen}(\sigma(Y)) \geq 0 \) for all \( \mathbb{R}^{V \times T} \), and (iii) \( \text{Pen}(\sigma(Y)) = 0 \) if and only if \( 1 - [C]_{\sigma(Y)} = 0 \).

By choosing the Lagrangian penalty method (Luenberger and Ye, 2015)(Section 13.1) where \( \text{Pen}(\sigma(Y)) = 1 - [C]_{\sigma(Y)} \) and \( \ell \) being an integer index indicating the sequence of optimization problems, we need to solve a sequence of unconstrained problems for different \( c_\ell \) where

\[
\max_{Y \in \mathbb{R}^{V \times T}} U_{C,N}(c_\ell, Y) = \text{NatVal}_N(\sigma(Y)) - c_\ell \, (1 - [C]_{\sigma(Y)}) \tag{6.8}
\]

In the unconstrained problem, we maximize the natural value minus a penalty term that penalizes \( Y \) if it does not satisfy \( C \) under the relaxed semantics.

One method to actually solve (6.8) is the following: Let \( \{c_\ell\}, \ell = 1, 2, \ldots \), be a sequence of real values tending to infinity, such that, for each \( \ell \), \( c_\ell \geq 0 \), \( c_{\ell+1} > c_\ell \). For each \( c_\ell \) solve problem (6.8) obtaining a solution point \( Y_\ell \).

Observe that if we let \( c_\ell \to \infty \) we will get back a solution that both maximizes the \text{NatVal} and satisfies the logical constraints, formally:

**Theorem 6.3.** Let \( N \) be a natural probability matrix, \( C \) be logical constraints, \( [C]_P \) denote the relaxed semantics of \( C \) and \( \{c_\ell\} \) be a sequence of real values tending to infinity. Now, consider \( \{Y_\ell\} \) to be the sequence generated by the sequence of penalty method problems

\[
\max_{Y \in \mathbb{R}^{V \times T}} U_{C,N}(c_\ell, Y_\ell) = \text{NatVal}_N(\sigma(Y_\ell)) - c_\ell \, (1 - [C]_{\sigma(Y_\ell)})
\]

Then, any limit point of the sequence \( \{Y_\ell\} \) is a solution to (6.6).

The proof of the above theorem directly corresponds to a standard result in non-linear optimization bibliography, which can be found in Luenberger and Ye (2015) (p. 459). For
this result, they consider a general optimization problem, stated as follows:

$$\min_{x} \quad f(x)$$
\[ \text{s.t.} \quad x \in \Omega \]  

where \( f \) is a continuous function of \( E^n \) and \( \Omega \) is a constraint set in \( E^n \). They consider its corresponding unconstrained problem to be: \( \min q(c, x) \triangleq f(x) + cP(x) \), where \( c \) is a positive constant and \( P \) is a function on \( E^n \) satisfying: (i) \( P \) is continuous, (ii) \( P(x) \geq 0 \) for all \( x \in E^n \), and (iii) \( P(x) = 0 \) if and only if \( x \in \Omega \). They the state the following result:

**Theorem 6.4 (Luenberger and Ye (2015)).** Let \( \{x_k\} \) be a sequence generated by the penalty method. Then, any limit point of the sequence is a solution to the general optimization problem 6.9.

To apply the aforementioned theorem in our case we simply need to replace \( f(x) = -\text{NatVal}_N(\sigma(Y)) \) and \( P(x) = 1 - \lceil C \rceil_{\sigma(Y)} \).
Natural Type Inference using Discrete Optimization

In this chapter, we present novel algorithms for solving the maximally natural type inference problem (Problem 5.1) using popular deductive techniques to solve discrete optimization problems. Our first algorithm, presented in Section 7.1, reduces the type inference problem to a problem in maximum satisfiability (MaxSAT) and exploits an off-the-shelf MaxSAT solver to search for a suitable type assignment. Our second algorithm, presented in Section 7.2, relies on an Integer Linear Program (ILP) formulation of the type inference problem and thereafter, exploits an industrial ILP solver. Both the algorithms provide a sound and complete method of predicting types by exploiting information from the algorithmic typing rules and source code text, as established by Theorem 7.1 and Theorem 7.2. Finally, in Section 7.3, we prove that the natural type inference problem is in fact an NP-hard problem, in Theorem 7.3, indicating that there can be no polynomial time algorithm to solve it (unless $P=NP$).

7.1 Natural Type Inference using MaxSAT

In this section, we present an approach that is principally different from the one presented in Chapter 6. This approach relies on reducing the problem of maximally natural type inference to an instance of the maximum satisfiability (MaxSAT) problem (Krentel, 1986). The problem MaxSAT, roughly speaking, deals with finding suitable solutions for formulas in propositional logic. To understand this better, let us first introduce some notation related to propositional logic.
Propositional Logic

Let \( \text{Var} \) be a set of propositional variables, which take Boolean values \( \{0, 1\} \) (0 represents \textit{false}, 1 represents \textit{true}). Formulas in propositional logic—usually denoted by capital Greek letters—are defined recursively as follows:

\[
\Phi := z \in \text{Var} | \neg \Phi | \Phi \lor \Phi
\]

In addition, as syntax sugar, we allow the following standard formulas: \( \text{true} \), \( \text{false} \), \( \Phi \land \Psi \triangleq \neg(\neg \Phi \lor \neg \Psi) \) and \( \Phi \rightarrow \Psi \triangleq \neg \Phi \lor \Psi \).

An assignment \( a : \text{Var} \rightarrow \{0, 1\} \) is a function that maps propositional variables to Boolean values. Given an assignment \( a \), a valuation function \( V(a, \Phi) \) provides the semantics of a propositional formula. It is inductively defined as follows:

\[
V(a, z) = a(z)
\]

\[
V(a, \neg \Psi) = 1 - V(a, \Psi)
\]

\[
V(a, \Psi \lor \Phi) = \max\{V(a, \Psi), V(v, \Phi)\}
\]

We say that \( a \) satisfies \( \Phi \) if \( V(a, \Phi) = 1 \), and call \( a \) a satisfying assignment of \( \Phi \). A propositional formula \( \Phi \) is satisfiable if there exists a satisfying assignment \( a \) of \( \Phi \).

The most well-known problem in propositional logic—the satisfiability (SAT) problem—is the problem of determining whether a propositional formula is satisfiable or not. For this problem, propositional formulas are often assumed to be in Conjunctive Normal Form (CNF). A formula \( \Phi \), in CNF, is a conjunction of clauses \( c \in C_\Phi \); \( C_\Phi \) being the set of clauses. Each clause is a disjunction of literals; a literal being a propositional variable \( z \) or its complement \( \neg z \).

MaxSAT and its Variants

We can now introduce the problem of MaxSAT. MaxSAT is a variant of the SAT problem that deals with the search of assignments that maximize the number of satisfied clauses for a given propositional formula.

For our inference problem we, however, use a more general version of the MaxSAT problem, known as \textit{Partial Weighted MaxSAT}, or PW-MaxSAT in short. In this version, for a given propositional formula there are weights associated with its clauses and suitable assignments must be searched based on these weights.
Formally, a formula $\Phi$ is associated with a weight function $w: C_\Phi \mapsto \mathbb{R} \cup \{\infty\}$ that assigns a weight to each of its clauses $c \in C_\Phi$. Based on the weights assigned, clauses are categorized into hard constraints $\mathcal{H}$ and soft constraints $\mathcal{S}$. A clause is a hard constraint $c \in \mathcal{H}$ if $w(c) = \infty$, while it is a soft constraint $c \in \mathcal{S}$ if $w(c) \in \mathbb{R}$. Given a propositional formula $\Phi$ with hard constraints $\mathcal{H}$ and soft constraints $\mathcal{S}$, the goal in PW-MaxSAT is to find an assignment $a$ that

1. satisfies all hard constraints, that is, $V(a, c) = 1$ for all $c \in \mathcal{H}$, and
2. maximizes the total weight of satisfied soft constraints, that is, maximizes $\sum_{c \in \mathcal{S}} w(c) \cdot V(a, c)$.

With the rapid development of SAT (and related) solvers (Li and Manyà, 2021), finding assignments to formulas even with millions of variables is feasible. For the MaxSAT problem and its variants, in particular, there are several dedicated solvers that have excellent performance (Bacchus et al., 2019). However, many modern SAT solvers—like the one used for our prototype, Z3 (de Moura and Bjørner, 2008)—have been extended with the capability of handling MaxSAT problems.

**The Encoding Using MaxSAT**

For reducing our inference problem to PW-MaxSAT, we construct a propositional formula $\Phi_{NTI}(C, N)$ and assign appropriate weights to its clauses. Our constructed formula $\Phi_{NTI}(C, N)$ has the property that, for any assignment $a$ that satisfies its hard constraints and maximizes the weight of the satisfied soft constraints, we can obtain a type assignment $\mu_a$ that: 1) satisfies the logical constraints $C$; and 2) is as natural as possible with respect to the natural matrix $N$. In short, $\mu_a \upharpoonright (N \mid C)$ based on the notation introduced in Problem 5.1. We formalize this at the end of this section, as Theorem 7.1.

Internally, $\Phi_{NTI}(C, N)$ is a conjunction of three formulas, described as follows:

$$\Phi_{NTI}(C, N) = \Phi^{LC}(C) \land \Phi^{VC}(C) \land \Phi^{NC}(N).$$

The first conjunct $\Phi^{LC}(C)$ ensures that the prospective type assignment $\mu_a$ satisfies the logical constraints arising from the typing rules in Definition 4.7. The second conjunct $\Phi^{VC}(C)$ ensures that $\mu_a$ is a valid type assignment, assigning exactly one type to each variable. The third conjunct $\Phi^{NC}(N)$ ensures that $\mu_a$ is as natural as possible, that is, it maximizes the function $\text{NatVal}_N$, described in Problem 5.1. The conjuncts $\Phi^{LC}(C)$ and $\Phi^{VC}(C)$ are hard constraints (that is, $\Phi^{LC}(C), \Phi^{VC}(C) \in \mathcal{H}$), while $\Phi^{NC}(N)$ consists of soft constraints (that is, $\Phi^{NC}(N) \in \mathcal{S}$). In the remainder of the section, we describe each
conjunction (and their corresponding weights) in detail.

Each of the conjuncts of $\Phi_{NITI}(C, N)$ rely on the following propositional variables: $z_{v, \tau}$ for each $v \in \mathcal{V}$ and $\tau \in \mathcal{T}$. A variable $z_{v, \tau}$ tracks whether the variable $v$ is assigned a type $t_\tau$. Precisely, $a(z_{v, \tau}) = 1$ if and only if $\mu_a(\alpha_v) = t_\tau$. We now exploit the introduced variables to construct our formulas.

We construct our first conjunct $\Phi^{LC}(C)$ based on the algorithmic typing rules described in Chapter 4. Let $C$ be the logical constraints obtained using Definition 4.7. We now convert $C$, which is a formula in equational logic, to a formula in propositional logic using a translation function $tr$. The function is recursively on the structure of $C$ as follows:

\[
tr(\alpha_v = t) = z_{v, \tau}; \\
tr(\neg C) = \neg tr(C); \\
tr(C_1 \lor C_2) = tr(C_1) \lor tr(C_2).
\]

Intuitively, the function $tr$ translates equations $\alpha_v = t_\tau$ to propositional variables $z_{v, \tau}$ and leaves the rest unchanged. We now simply set $\Phi^{LC}(C) = tr(C)$.

We construct the second conjunct $\Phi^{VC}(C)$ as follows:

\[
\Phi^{VC}(C) = \bigwedge_{v \in \mathcal{V}} \left[ \bigvee_{\tau \in \mathcal{T}} z_{v, \tau} \land \bigwedge_{\tau \neq \tau' \in \mathcal{T}} \left[ \neg z_{v, \tau} \lor \neg z_{v, \tau'} \right] \right].
\]

Intuitively, the above formula says that each variable $v$ must be assigned at least one type $t_\tau$ and there must not be two types $t_\tau \neq t_{\tau'}$ assigned to it.

Due to the validity constraints introduced above, we can derive a type assignment $\mu_a$ from a satisfying assignment $a$ in a straightforward manner: we simply set $\mu_a(\alpha_v)$ to be the unique $\tau \in \mathcal{T}$ for which $a(z_{v, \tau}) = 1$.

We now construct the third conjunct $\Phi^{NC}(N)$ as follows:

\[
\Phi^{NC}(N) = \bigwedge_{v \in \mathcal{V}} \bigwedge_{\tau \in \mathcal{T}} z_{v, \tau}
\]

Unlike the other conjuncts, $\Phi^{NC}(N)$ consists of soft constraints. As a result, we assign real-valued weights to its clauses using the following function $w$:

\[
w(z_{v, \tau}) = n_{v, \tau}, \text{ for all } v \in \mathcal{V} \text{ and } \tau \in \mathcal{T}.
\]

where $n_{v, \tau}$ is the probability that $\alpha_v$ is set to type $t_\tau$ as specified in the natural constraint
We conclude this section by stating a theorem that asserts the correctness of the constructed propositional formula \( \Phi_{NTI}(C, N) \).

**Theorem 7.1.** Let be the logical constraints \( C \) and natural matrix \( N \).

Let \( \text{NatVal}_N \) be the natural value function obtained from the natural matrix \( N \).

Let \( H \) and \( S \) be the hard and soft constraints contained in the formula \( \Phi_{NTI}(C, N) \).

(1) (**Soundness**) If \( a \) is an assignment to \( \Phi_{NTI}(C, N) \) that satisfies the hard constraints \( H \) and maximizes the weight of the satisfied soft constraints \( S \), then the corresponding type assignment \( \mu_a \) has the properties that:

(a) \( \mu_a \models C \); and

(b) for all \( \mu \neq \mu_a \) such that \( \mu \models C \), \( \text{NatVal}_N(\text{Bin}(\mu)) \leq \text{NatVal}_N(\text{Bin}(\mu_a)) \).

(2) (**Completeness**) If \( \mu \models C \), then there exists an assignment \( a \) to \( \Phi_{NTI}(C, N) \) that satisfies the hard constraints \( H \).

**Proof.** Let us assume that \( a \) is a desired assignment to \( \Phi_{NTI}(C, N) \) and \( \mu_a \) be the type assignment derived from \( a \). Now, to prove (1a) we use the following claim: If \( V(a, tr(C)) = 1 \) then \( \mu_a \models C \). We prove the claim via an induction on the structure of \( C \).

Base Case \( C = (\alpha_v = t_\tau) \): Let \( V(a, tr(\alpha_v = t_\tau)) = 1 \). Then, \( V(a, z_{v,\tau}) = 1 \), which, based on the definition of the valuation function, means \( a(z_{v,\tau}) = 1 \). Further, using our construction of \( \mu_a \), \( \mu_a(\alpha_v) = t_\tau \) and thus, \( \mu_a \models (\alpha_v = t_\tau) \).

Case \( C = \neg C_1 \):

\[
V(a, tr(\neg C_1)) = 1
\]

implies

\[
1 - V(a, tr(C_1)) = 1 \quad \text{(Definition of } tr)\]

implies

\[
V(a, tr(C_1)) = 0 \quad \text{(Inductive Hypothesis)}
\]

implies

\[
\neg \mu \models C_1 \quad \text{(Definition 4.4)}
\]

implies

\[
\mu_a \models \neg C_1
\]

implies

\[
\mu_a \models C
\]

Case \( C = C_1 \lor C_2 \):

\[
V(a, tr(C)) = 1
\]
implies $V(a, tr(C_1 \lor C_2)) = 1$
implies $V(a, (tr(C_1) \lor tr(C_2))) = 1$ (Definition of $tr$)
implies $V(a, (tr(C_1)) = 1$ or $V(a, (tr(C_2)) = 1$
implies $\mu_a \models C_1$ or $\mu_a \models C_2$ (Inductive Hypothesis)
implies $\mu_a \models (C_1 \lor C_2)$ (Definition 4.4)
implies $\mu_a \models C$

For (1b), we show that the weight of the satisfied soft constraints by an assignment $a$ satisfying the hard constraints (that is, $\Phi^{LC}$ and $\Phi^{VC}$) is equal to $\text{NatVal}_N(\text{Bin}(\mu_a))$. Precisely,

$$\text{NatVal}_N(\mu_a) = \sum_{v=1}^{\nu} n_{v,\mu_a(\alpha_v)}$$
$$= \sum_{v=1}^{\nu} \sum_{\tau \in 1..T} n_{v,\tau}$$ (Definition of $\mu_a$)
$$= \sum_{v=1}^{\nu} \sum_{\tau=1}^{T} w(z_{v,\tau}) \cdot V(a, z_{v,\tau})$$ (Definition of $V$).

Thus, maximizing the weight of the satisfied soft constraints corresponds to maximizing the $\text{NatVal}_N$ function.

For (2), using type assignment $\mu$, we construct an assignment $a$ as follows: we set $a(z_{v,\tau}) = 1$ if and only if $\mu(\alpha_v) = t_{\tau}$. Now, to show that such an assignment satisfies $\Phi^{LC}$, we first prove the following claim: If $\mu \models C$ then $V(a, tr(C)) = 1$. Similar to the proof for the claim in (1a), we use a structural induction on $C$.

Base Case $C = (\alpha_v = t_{\tau})$: Let $\mu \models (\alpha_v = t_{\tau})$. Then, by Definition 4.4 we get that $\mu(\alpha_v) = t_{\tau}$. Now, we construct an assignment $a$ by setting $a(z_{v,\tau}) = 1$. Then, based on the definition of the valuation function $V(a, z_{v,\tau}) = 1$, This can be rewritten using the definition of the translation function as $V(a, tr(\alpha_v = t_{\tau})) = 1$.

Case $C = \neg C_1$:

$$\mu \models \neg C_1$$
implies not $\mu \models C_1$ (Definition 4.4)
implies $V(a, tr(C_1)) = 0$ (Induction Hypothesis)
implies $1 - V(a, tr(C_1)) = 1$
implies \( V(a, tr(-C_1)) = 1 \) \hspace{2cm} \text{(Definition of } tr) \\
implies \( V(a, tr(C)) = 1 \)

Case \( C = C_1 \lor C_2 \):

\[
\mu \models C_1 \lor C_2 \\
\implies \mu \models C_1 \text{ or } \mu \models C_2 \hspace{2cm} \text{(Definition 4.4)} \\
\implies V(a, (tr(C_1)) = 1 \text{ or } V(a, (tr(C_2)) = 1 \hspace{2cm} \text{(Inductive Hypothesis)} \\
\implies V(a, (tr(C_1) \lor tr(C_2)) = 1 \hspace{2cm} \text{(Definition of } tr) \\
\implies V(a, tr(C_1 \lor C_2)) = 1 \\
\implies V(a, tr(C)) = 1
\]

\[\square \]

7.2 Natural Type Inference using ILP

We now describe an alternate way of combining logical and natural constraints by reducing to a well-known problem—Integer Linear Programming (ILP) (Papadimitriou, 1982). Integer Linear Programming expresses the optimization of a linear function subject to a set of linear constraints over integer variables. We now formally introduce ILP.

Integer Linear Programming

Let \( c = [c_1, \ldots, c_n] \in \mathbb{R}^n \) and \( b = [b_1, \ldots, b_m] \in \mathbb{R}^m \) be vectors over real numbers, and \( A \in \mathbb{R}^{m \times n} \) be an \( m \times n \) dimensional matrix with entries \( a_{i,j} \in \mathbb{R} \) for each \( i \in 1..m \) and \( j \in 1..n \). Also, let \( x = [x_1, \ldots, x_n] \) be a vector of variables where each \( x_i \) can assume only integer values. The general problem can now be written as the following optimization:

\[
\max_x \sum_{i=1}^{n} c_i x_i \\
\text{s.t. } \sum_{i=1}^{n} a_{i,j} x_i = b_j \text{ for each } j \in 1..m \\
x_i \geq 0 \text{ for each } i \in 1..n \\
x_i \in \mathbb{Z} \text{ for each } i \in 1..n.
\]
Before we present our ILP formulation, we clarify the theoretical consequence of such a formulation. In terms of complexity, both MaxSAT and ILP share the same hardness: the decision version of both the problems are known to be \( \text{NP} \)-complete. As the result, the reductions shown in both Section 7.1 and this section demonstrate that combining logical and natural constraints can potentially be as hard as solving an instance of MaxSAT or ILP. However, it is often seen that reducing a problem to one of the two problems happens to be more effective in practice. Now, which of the two problem turns out to be more effective for our type inference problem requires a empirical study, which we present in the next chapter. Our focus in this section is on providing an efficient reduction to ILP and thereafter, proving its correctness.

### The ILP Formulation

To set up our ILP formulation, we introduce integer variables \( y_{v,\tau} \) for each \( v \in 1..V \) and each \( \tau \in 1..T \). To refer to the variables in a concise manner, we use the vector \( y = [y_{1,1}, \ldots, y_{V,T}] \) of dimension \( V \cdot T \).

Further, to impose the logical constraints \( C \) on the integer variables, we introduce a translation function \( \text{tr} \) that converts \( C \) into linear expressions over the integer variables. The translation function we design can be applied only on clauses and thus, we consider \( C \) to be specified in (or, if necessary, converted to) CNF. We now define \( \text{tr} \) over a clause \( c = l_1 \lor l_2 \lor \cdots \lor l_k \) where \( l_i \)'s represent literals that can be of the form \((\alpha_v = t_\tau)\) or \((\neg(\alpha_v = t_\tau))\).

\[
\begin{align*}
\text{tr}(\alpha_v = t_\tau) &= y_{v,\tau}; \\
\text{tr}(\neg(\alpha_v = t_\tau)) &= 1 - y_{v,\tau}; \\
\text{tr}(l_1 \lor l_2 \lor \cdots \lor l_k) &= \text{tr}(l_1) + \text{tr}(l_2) + \cdots + \text{tr}(l_k)
\end{align*}
\]

We now specify the ILP formulation as follows:

\[
\begin{align*}
\max_y & \quad \sum_{v=1}^V \sum_{\tau=1}^T y_{v,\tau} \cdot n_{v,t} \\
\text{s.t.} & \quad 0 \leq y_{v,\tau} \leq 1 \text{ for each } v \in 1..V \text{ and } \tau \in 1..T \\
& \quad \sum_{\tau \in 1..T} y_{v,\tau} = 1 \text{ for each } v \in 1..V \\
& \quad \text{tr}(c_i) \geq 1 \text{ for each clause } c_i \text{ of } C \\
& \quad y_{v,\tau} \in \mathbb{Z} \text{ for each } v \in 1..V \text{ and } t \in 1..T.
\end{align*}
\]

We refer to the above formulation as \( \mathcal{I}_{NTI}(C, N) \). In \( \mathcal{I}_{NTI}(C, N) \), the first set of constraints ensure that the variables \( y_{v,\tau} \) assume values either 0 or 1. The second set of constraints
ensure that for each \( v \in 1..V \), exactly one variable \( y_{v,\tau} \) assumes the value 1. The third constraint ensures that the variables respect the linear equation obtained from the logical constraints \( C \) using the translation function \( \overline{tr} \).

In a solution of \( \mathcal{I}_{NTI}(C, N) \), let the value of \( y_{v,\tau} \) be denoted using \( a_{v,\tau} \) for each \( v \in 1..V \) and for each \( \tau \in 1..T \). From a solution \( a_{v,\tau} \), we construct a type assignment \( \mu_a \) as follows: we set \( \mu_a(\alpha_v) \) to be unique \( \tau \in 1..T \) for which \( a_{v,\tau} = 1 \).

We now prove that our ILP formulation is correct, which formally stated is as follows:

**Theorem 7.2.** Let \( C \) be the logical constraints and \( N \) the natural matrix. Let \( \text{NatVal}_N \) be the function that computes the natural value of a type assignment with respect to a natural matrix \( N \). Let \( \mathcal{I}_{NTI}(C, N) \) be the ILP formulation as described above.

(1) *(Soundness)* Let the \( \mu_a \) be the type assignment constructed based on a solution of \( \mathcal{I}_{NTI}(C, N) \). Then \( \mu_a \) has the following properties:

(a) \( \mu_a \models C \); and

(b) for all \( \mu \neq \mu_a \) such that \( \mu \models C \), \( \text{NatVal}_N(\text{Bin}(\mu)) \leq \text{NatVal}_N(\text{Bin}(\mu_a)) \).

(2) *(Completeness)* If \( \mu \models C \), then there exists a feasible solution of \( \mathcal{I}_{NTI}(C, N) \), i.e., a solution that satisfies all of the constraints.

**Proof.** To prove (1a), we prove that \( \mu_a \models c_i \) for some arbitrary clause \( c_i = l_1 \lor \cdots \lor l_k \) of \( C \). To this end, observe that \( \overline{tr}(l_i) \) can only assume values 0 or 1, since \( l_i \)’s are either \( y_{v,\tau} \) or \( 1 - y_{v,\tau} \). Now, \( \overline{tr}(c_i) \geq 1 \) since \( \mathcal{I}_{NTI}(C, N) \) admits a feasible solution. This implies that at least one \( \overline{tr}(l_i) \) has value 1. Thus, without loss of generality, let \( \overline{tr}(l_m) = 1 \) and \( l_m = (\alpha_v = t_{\tau}) \). Consequently, based on our definition of \( \mu_a \), \( \mu_a(\alpha_v) = t_{\tau} \). As a result, \( \mu_a \models l_m \) and hence, \( \mu_a \models c_i \).

For (1b), we show that the value of the optimization objective in \( \mathcal{I}_{NTI}(C, N) \) is equal to \( \text{NatVal}_N(\mu_a) \). Let \( a_{v,\tau} \) be the value assumed by variable \( y_{v,\tau} \) for each \( v \in 1..V \) and each \( \tau \in 1..T \). We now have the following:

\[
\text{NatVal}_N(\text{Bin}(\mu_a)) = \sum_{v=1}^{V} \text{Bin}(\mu_a)_v \cdot n_v \quad \text{(Definition 5.3)}
\]

\[
= \sum_{v=1}^{V} \sum_{\tau=1}^{T} a_{v,\tau} \cdot n_{v,\tau}.
\]

Thus, maximizing the objective function in \( \mathcal{I}_{NTI}(C, N) \) maximizes the \( \text{NatVal}_N \) function.

Finally, to prove (2), we construct a solution of \( \mathcal{I}_{NTI}(C, N) \) based on a satisfying type
assignment $\mu$. In particular, we set $a_{v, \tau} = 1$ if $\mu(\alpha_v) = \tau$, otherwise $a_{v, \tau} = 0$ for each $v \in 1..V$ and $\tau \in 1..T$. We now prove that this a feasible solution for $I_{NTI}(C, N)$. Clearly, the first set of constraints $\sum_{\tau \in 1..T} a_{v, \tau} = 1$ is satisfied, since $\mu(\alpha_v) = t_\tau$ for exactly one $\tau$. The second set of constraints is satisfied by definition since $a_{v, \tau}$ is either 0 or 1. Finally, we show that the third set of constraints hold by showing that it holds for an arbitrary clause $c_i = l_1 \lor \cdots \lor l_k$ of $C$. Now, since $\mu | c_i$, $\mu | l_m$ for some literal $l_m$. Without loss of generality, let $l_m = (\alpha_v = t_\tau)$. Now, $a_{v, \tau} = 1$ based on the definition of the solution. Thus, $t_F(c_i) \geq a_{v, \tau} = 1$.

### 7.3 Natural Type Inference is NP-hard

We now prove that the problem of combining logical and natural constraints, as we state in Problem 5.1, is in fact an NP-hard problem. Since Problem 5.1 is an optimization problem, we prove its NP-hardness by relying on its corresponding decision problem, which we state below.

**Problem 7.1** (Decision: Natural Satisfying Type Valuation). Given a formula $C$ in equational logic over $A = tyvar(C) = \{\alpha_v \mid v \in 1..V\}$, a natural matrix $N$ of size $V \times T$, and $k \in \mathbb{R}$, does there exist a type assignment $\mu$ with $\text{dom}(\mu) = A$ that has the properties:

1. $\mu | C$; and
2. $\text{NatVal}_N(\text{Bin}(\mu)) \geq k$?

We refer to the above problem as $\text{NatSAT}$ for brevity. In contrast to the optimization problem (Problem 5.1), the $\text{NatSAT}$ problem requires a real number $k$ to search for a satisfying type assignment $\mu$ that has the property $\text{NatVal}_N(\text{Bin}(\mu)) \geq k$. Now, to determine for which values of $k$ there exists a type assignment with the mentioned properties, one can simply solve the optimization problem. As a result, the optimization problem is harder than the stated decision problem $\text{NatSAT}$. In what follows, we show that the $\text{NatSAT}$ problem is NP-complete, making Problem 5.1 NP-hard.

**Theorem 7.3.** The $\text{NatSAT}$ problem is NP-complete.

It is straightforward to show that the $\text{NatSAT}$ problem is in NP. In particular, given a type assignment $\mu$, verifying whether $\mu | C$ can be done in time $O(\text{poly}(|C|))$, where $|C|$ is the size of the syntactic tree of $C$. Also, verifying $\text{NatVal}_N(\text{Bin}(\mu)) \geq k$ requires performing a number of arithmetic operations based on Definition 5.3, which can be done in time $O(\text{poly}(V \cdot T))$.

For proving the NP-hardness, we rely on the decision version of the MaxSAT problem, which is the following:
Problem 7.2 (Decision: MaxSAT). Given a propositional formula $\Phi$ in CNF over $n$ variables with $m$ clauses, and a natural number $k \leq m$, does there exist an assignment $a$ that satisfies at least $k$ clauses?

Again, for brevity, we refer to the above problem as MaxSAT. The MaxSAT problem is a well-known NP-complete problem (Papadimitriou, 2007).

We now provide a polynomial reduction of the MaxSAT problem to the NatSAT problem, that is, $\text{MaxSAT} \leq_{\text{m}} \text{NatSAT}$. The crux of our reduction is to construct logical constraints $C_\Phi$ and a natural matrix $N_\Phi$ from a propositional formula $\Phi$ such that the following holds:

Claim 7.1. There exists an assignment for $\Phi$ that satisfies at least $k$ of its clauses if and only if there exists a type assignment $\mu$ such that $\mu \models C_\Phi$ and $\text{NatVal}_N(\text{Bin}(\mu), \Phi) \geq k + \frac{n}{2}$.

In the remainder of the section, we provide our reduction and subsequently, prove the above claim.

Let $\text{Var} = \{z_1, z_2, \ldots, z_n\}$ be the set of variables in $\Phi$ and $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$ be the clauses of $\Phi$. For the construction, let us introduce propositional variables $d_i$ for each $i \in 1..m$. These variables track the valuation of each clause $c_i$ of $\Phi$. Precisely, $a(d_i) = 1$ if and only if $V(a, c_i) = 1$ for each $i \in 1..m$. To ensure the desired meaning of the variables $d_i$, we construct the following formula:

$$
\Psi_\Phi \triangleq \bigwedge_{i \in 1..m} (d_i \leftrightarrow c_i) \\
\triangleq \bigwedge_{i \in 1..m} (d_i \rightarrow c_i) \land (c_i \rightarrow d_i) \\
\triangleq \bigwedge_{i \in 1..m} (\neg d_i \lor c_i) \land (\neg c_i \lor d_i)
$$

As a running example, consider $\Phi = (z_1 \lor \neg z_2) \land (z_3 \lor \neg z_1)$, where $c_1 = z_1 \lor \neg z_2$ and $c_2 = z_3 \lor \neg z_1$. We then have $\Psi_\Phi = (d_1 \leftrightarrow c_1) \land (d_2 \leftrightarrow c_2)$, which can be expressed in CNF as $\Psi_\Phi = (\neg d_1 \lor z_1 \lor \neg z_2) \land (\neg z_1 \lor d_1) \land (z_2 \lor d_1) \land (\neg d_2 \lor z_3 \lor \neg z_1) \land (\neg z_3 \lor d_2) \land (z_1 \lor d_2)$.

We now describe the construction of $C_\Phi$. We build $C_\Phi$ over type variables $A = \text{tyvar}(C_\Phi) = \{\alpha_v \mid v \in 1..n + m\}$ and two types $\{t_1, t_2\}$. For constructing $C_\Phi$, we apply a translation function $\tilde{tr}$ to convert $\Psi_\Phi$ to obtain formula in equational logic. The function $\tilde{tr}$ is defined inductively as follows:

$$
\tilde{tr}(z_i) = (\alpha_i = t_1); \quad \tilde{tr}(d_i) = (\alpha_{n+i} = t_1) \\
\tilde{tr}(\neg \Phi) = \neg \tilde{tr}(\Phi); \quad \tilde{tr}(\Phi \lor \Psi) = \tilde{tr}(\Phi) \lor \tilde{tr}(\Psi)
$$
Now, $C_{\Phi}$ is simply $\tilde{t}(\Psi_{\Phi})$. For the running example, $C_{\Phi}$ is over type variables $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$ and types $\{t_1, t_2\}$. Applying $\tilde{t}$ on $\Psi_{\Phi}$ yields the following $C_{\Phi}$:

$$C_{\Phi} = \left[ \neg(\alpha_4 = t_1) \lor (\alpha_1 = t_1) \lor \neg(\alpha_2 = t_1) \right] \land \left[ (\alpha_2 = t_1) \lor (\alpha_4 = t_1) \lor \neg(\alpha_3 = t_1) \lor (\alpha_5 = t_1) \left[ (\alpha_1 = t_1) \lor (\alpha_4 = t_1) \right] \right]$$

Next, we construct the natural constraints $n_v$ as follows:

$$n_v = \begin{cases} [\frac{1}{2}, \frac{1}{2}] & \text{for } v \in 1..n \\ [1, 0] & \text{for } v \in n + 1..n + m \end{cases}$$

Intuitively, the natural constraints are constructed such that, for each type variables $\alpha_i$ where $i \in 1..n$, equal preference is given to be set to $t_1$ or $t_2$, but, for each type variables $\alpha_i$ where $i \in n + 1..n + m$, more preference is given to be set to $t_1$. The natural matrix $N = [n_1^T \cdots n_{n+m}^T]$ is defined based on the introduced natural constraints. For the running example, $N$ will be the following:

$$\begin{array}{c|cc} & t_1 & t_2 \\ \hline \\ \alpha_1 & \frac{1}{2} & \frac{1}{2} \\ \alpha_2 & \frac{1}{2} & \frac{1}{2} \\ \alpha_3 & \frac{1}{2} & \frac{1}{2} \\ \alpha_4 & 1 & 0 \\ \alpha_5 & 1 & 0 \end{array}$$

Proof of Claim 7.1. To prove the forward direction, we consider that there exist an assignment for $\Phi$, say $a$, which satisfies at least $k$ of its clauses. We extend $a$ to assign values for the introduced variables $d_i$ by simply setting $a(d_i) = V(a, c_i)$. 

Such an assignment $a$ satisfies $\Psi_{\Phi}$, that is, $a \models \Psi_{\Phi}$. Since $\Psi_{\Phi} = \bigwedge_{i \in 1..n} d_i \leftrightarrow c_i$, to show $a \models \Psi_{\Phi}$, it suffices to show $a \models d_i \leftrightarrow c_i$. This can be shown by expanding the inductive definition of the valuation function $V$ as follows:

$$V(a, d_i \leftrightarrow c_i) = \min\{V(a, d_i \rightarrow c_i), V(a, c_i \rightarrow d_i)\}$$

$$= \min\{\max\{1 - a(d_i), V(a, c_i)\}, \max\{1 - V(a, c_i), a(d_i)\}\}$$

$$= \min\{\max\{1 - a(d_i), a(d_i)\}, \max\{1 - a(d_i), a(d_i)\}\}$$

$$= 1$$
Using the considered assignment $a$, we construct $\mu$ as follows:

- for $i \in 1 \cdots n$, $\mu(\alpha_i) \triangleq t_1$ if $a(z_i) = 1$, $t_2$ otherwise; and
- for $i \in n + 1 \cdots n + m$, $\mu(\alpha_i) \triangleq t_1$ if $a(d_i) = 1$, $t_2$ otherwise.

We now prove that $\mu \models C_\Phi$. To this end, we prove that $V(a, \Omega) = 1$ if and only if $\mu \models \tilde{t}r(\Omega)$ for any subformula $\Omega$ of $\Psi_\Phi$. We show this using an induction on the structure of the formula $\Omega$ as follows:

Case $\Omega = z_i$:

\[
V(a, z_i) = 1
\]
if and only if $a(z_i) = 1$ (Definition of $V$)
if and only if $\mu(\alpha_i) = t_1$ (Definition of $\mu$)
if and only if $\mu \models (\alpha_i = t_1)$ (Definition of $V$)
if and only if $\mu \models \tilde{t}r(z_i)$

Case $\Omega = \neg \Omega'$:

\[
V(a, \neg \Omega') = 1
\]
if and only if $V(a, \Omega') = 0$ (Definition of $V$)
if and only if $\mu \not\models \tilde{t}r(\Omega')$ (Inductive Hypothesis)
if and only if $\mu \models \tilde{t}r(\neg \Omega')$ (Definition 4.4)

Case $\Omega = \Omega_1 \lor \Omega_2$:

\[
V(a, \Omega_1 \lor \Omega_2) = 1
\]
if and only if $V(a, \Omega_1) = 1$ or $V(a, \Omega_2) = 1$ (Definition of $V$)
if and only if $\mu \models \tilde{t}r(\Omega_1)$ or $\mu \models \tilde{t}r(\Omega_2)$ (Inductive Hypothesis)
if and only if $\mu \models \tilde{t}r(\Omega_1 \lor \Omega_2)$ (Definition 4.4)

For the inductive step, we skip the cases when $\Omega = d_i$ and $\Omega = \Omega_1 \land \Omega_2$ since they are almost identical to the cases where $\Omega = z_i$ and $\Omega = \Omega_1 \lor \Omega_2$, respectively.

Finally, using the fact that $V(a, \Psi_\Phi) = 1$, we prove that $\mu \models \tilde{t}r(\Psi_\Phi)$, that is, $\mu \models C_\Phi$

We now prove that $\text{NatVal}_N(\text{Bin}(\mu), N_\Phi) \geq \frac{n}{2} + k$. In the proof, we use the notation
1\text{condition} to represent the characteristic function which has value 1 if the condition holds, 0 otherwise. The proof is now as follows:

\[
\text{NatVal}_N(\text{Bin}(\mu), N_\Phi) = \sum_{i=1}^{n} \left( \frac{1}{2} \cdot \mathbb{1}_{\mu(\alpha_i) = t_1} + \frac{1}{2} \cdot \mathbb{1}_{\mu(\alpha_i) = t_2} \right) + \sum_{i=n+1}^{n+m} \mathbb{1}_{\mu(\alpha_i) = t_1} \\
= \frac{n}{2} + \sum_{i=n+1}^{n+m} \mathbb{1}_{\mu(\alpha_i) = t_1} = \frac{n}{2} + \sum_{i=1}^{m} \mathbb{1}_{a(d_i) = t_1} = \frac{n}{2} + \sum_{i=1}^{m} \mathbb{1}_{V(a,c_i) = 1} \\
\geq \frac{n}{2} + k
\]

In the above equation, note that \(1_{V(a,c_i) = 1} \geq k\) since \(a\) satisfies at least \(k\) clauses of \(\Phi\).

To prove the backward direction, let us consider \(\mu\) is an type assignment such that \(\mu \models C_\Phi\) and \(\text{NatVal}_N(\text{Bin}(\mu), N_\Phi) \geq \frac{n}{2} + k\). Using \(\mu\), we construct assignment \(a\) for \(\Phi\) in the following manner:

- for \(i \in 1 \cdots n\), \(a(z_i) = 1\) if \(\mu(\alpha_i) = 1\)
- for \(i \in n + 1 \cdots n + m\), \(a(d_i) = 1\) if \(\mu(\alpha_i) = 1\)

As in the other direction, we prove that \(V(a, \Omega) = 1\) if and only if \(\mu \models \text{fr}(\Omega)\) for any subformula \(\Omega\) of \(\Psi_\Phi\) using the exact same inductive proof. This proves that \(V(a, \Psi_\Phi) = 1\) since \(\mu \models \text{fr}(\Psi_\Phi)\).

Next, we prove that \(a(d_i) = V(a, c_i)\) for each \(i \in 1..n\), using contradiction. Consider that \(a(d_i) \neq V(a, c_i)\) for some \(i \in 1..n\). This implies \(a(d_i) = 1 - V(a, c_i)\). Based on this, we compute \(V(a, d_i \leftrightarrow c_i)\) as follows:

\[
V(a, d_i \leftrightarrow c_i) = \min\{V(a, d_i \rightarrow c_i), V(a, c_i \rightarrow d_i)\} \\
= \min\{\max\{1 - a(d_i), V(a, c_i)\}, \max\{1 - V(a, c_i), a(d_i)\}\} \\
= \min\{\max\{V(a, c_i), V(a, c_i)\}, \max\{1 - V(a, c_i), 1 - V(a, c_i)\}\} \\
= \min\{V(a, c_i), 1 - V(a, c_i)\} \\
= 0
\]

Thus, if \(V(a, d_i \leftrightarrow c_i) = 0\), \(V(a, \Psi_\Phi) = 0\) since \(\Psi_\Phi = \bigwedge_{i \in 1..m} d_i \leftrightarrow c_i\), leading to a contradiction.

Finally, we prove that \(a\) satisfies at least \(k\) clauses of \(\Phi\) by exploiting \(\text{NatVal}_N\). In particular, we have the following:

\[
\text{NatVal}_N(\text{Bin}(\mu), N_\Phi) \geq \frac{n}{2} + k
\]
implies \[ \frac{n}{2} + \sum_{i=n+1}^{n+m} 1_{\mu(\alpha_i)=\ell_1} \geq \frac{n}{2} + k \]  (Definition 5.3)

implies \[ \sum_{i=1}^{m} \mathbb{1}_{a(d_i)=1} \geq k \]  (Definition of \(a\))

implies \[ \sum_{i=1}^{m} \mathbb{1}_{V(a,c_i)=1} \geq k \]

showing that \(V(a,c_i) = 1\) for at least \(k\) clauses.

The result in this section shows that the problem of combining logical constraints \(C\) and natural constraints \(N\) to obtain a type assignment \(\mu\) that satisfies the logical constraints \(C\) and, at the same time, maximizes the natural value \(\text{NatVal}_N\) is \(\text{NP}\)-hard. However, this does not directly show that finding a maximally natural type assignment for a given program is \(\text{NP}\)-hard. This is because, in Problem 7.1 (which we prove to be \(\text{NP}\)-hard), we make an implicit assumption that one can obtain arbitrary (in terms of syntactic structure) logical constraints \(C\) from a program via the typing rules in Chapter \(\ldots\). We do not know whether this is true. Nevertheless, this generates an interesting line of future work in which one characterizes programs based on the syntactic structure of logical constraints derived from the program to determine the computational hardness of obtaining maximally natural type assignments.
In this chapter, we discuss our realization of natural type inference in a prototype for TypeScript, which we name Optyper. Our tool Optyper principally differs from existing tools (at least for TypeScript) in two ways: 1) it exploits typing rules to generate logical constraints for sound typing and does not need to learn them; and 2) it employs optimization to pick the maximally natural choice that the logical constraints allow.

Before diving into the implementation details of Optyper, we must clarify a restriction that we employ in our tool. While, in Section 7.3, we prove that our approach is sound over a $\lambda$-calculus of named types, TypeScript’s type system is vastly more expressive. Hence, as is standard practice with research prototypes, Optyper does not handle TypeScript’s type system in its entirety, and employs restrictions such as restricting Optyper to a fixed type vocabulary (see Section 8.2). That said, we know of no reason, in principle, that Optyper could not be extended to TypeScript’s full type system.

To explain its implementation, in this chapter, we first describe the different building blocks of Optyper. We begin by describing how Optyper generates the logical constraints from the TypeScript compiler (Microsoft, 2020) `tsc`, in Section 8.1. We then describe how Optyper extracts the natural information using a popular deep learning model—Long Short Term Memory (LSTM)—in Section 8.2. Finally, we present the implementation of the various techniques for joint optimization of logical and natural constraints in Section 8.3.

Specifically, for the joint optimization, we describe the implementation of continuous optimization from Section 6.4, MaxSAT from Section 7.1 and ILP from Section 7.2.

We then evaluate Optyper on real world benchmarks obtained from open source GitHub.
repositories. Through our evaluation, we answer a number of research questions regarding the efficacy of our approach. In particular, we compare the different joint optimization techniques and the benefits of combining logical and natural constraints. Finally, we compare Optyper against two state-of-the-art type inference tools LambdaNet (Wei et al., 2020) and TypeWriter (Pradel et al., 2020). We show that, leveraging its solid formal foundation and optimization techniques, Optyper outperforms the other tools in predicting types.

8.1 Logical Constraints for Optyper

In this section, we first provide some background regarding TypeScript’s compiler and then we describe how we use it to extract logical constraints from it.

Background: TypeScript’s Type System

TypeScript (Microsoft, 2020) is a typed superset of JavaScript designed for developing large-scale, stable applications. TypeScript’s compiler (tsc) typechecks TypeScript programs then emits plain JavaScript, to leverage the fact that JavaScript is the only cross-platform language that runs in any browser, any host, and any OS. TypeScript’s type system considers record types (classes), whose fields or members have the same names and types, to be equal. This implementation decision permits TypeScript to handle many JavaScript idioms that depend on dynamic typing. One of the main goals of TypeScript’s designers is to support idiomatic JavaScript to provide a smooth transition from JavaScript to TypeScript. Therefore, TypeScript’s type system is deliberately unsound (Bierman et al., 2014). It is an optional type system, whose annotations can be omitted and have no effect on runtime, as TypeScript erases them when transpiling to JavaScript (Bierman et al., 2014). TypeScript’s type system defaults to assigning its any type to unannotated parameters, and methods or properties.

Extracting Logical Constraints

We cannot rely on the TypeScript compiler (Microsoft, 2020) tsc directly to generate the logical constraints of Section 4.1, because tsc does not construct logical formulas explicitly during typechecking. Still, we can rely on a mode of operation where the compiler infers types on TypeScript code with no type annotations. To ensure that no types are present to the input files, the first step before extracting the logical constraints is to parse the original files and remove all type annotations, then we continue with generating the constraints.
Specifically, to generate logical constraints on argument types, we build on a command-line tool, named TypeStat (Goldberg, 2020), that calls \texttt{tsc} to predict type hints for variables, usually to provide codefixes within a development environment. We associate each variable \( x_v \), to a fresh generic type variable \( \alpha_v \), in correspondence to Definition 4.1, and append the pair to \( V = V \cup \{(x_v, \alpha_v)\} \). So, when the predicted type for an identifier \( x_v \), is a literal \( t_\tau \) within our universe, we emit the constraint \( \alpha_v = t_\tau \), where \( v \in 1..V \), \( \tau \in 1..T \). When the predicted type for \( x_v \) is a union type \( (t_1 | \cdots | t_n) \) of literals, we emit the disjunction \((\alpha_v = t_1) \lor \cdots \lor (\alpha_v = t_n))\). Overall, for each function or group of functions in a file, we return a conjunction of the logical constraints generated for variables, as described above.

We note that our logical constraints include propositional logic, and therefore are able to express a wide range of interesting type constraints (Odersky et al., 1999; Pottier and Rémy, 2005).

8.2 Natural Constraints for Optyper

In this section, we describe how we generate the natural constraints using a deep learning model: character-level LSTM, a variant of Recurrent Neural Networks (RNNs). While there are numerous machine learning models for learning-based type inference as we see in the related works—Section 2.3; for our purposes of bridging the gap between logical and learning-based inference, the chosen RNN model suffices.

In what follows, we first provide some background knowledge for character-level LSTM and then we explain how we instantiate it for our problem to learn natural constraints from naming conventions over types.

**Background: Character-level LSTM**

In principle, natural constraints can be calculated based on any property of the source code, including names and comments. In this dissertation, we consider a simple but practically effective example of natural constraint, namely, a deep network that predicts the type of an identifier from the characters in its name. We consider each identifier \( id \) to be a character sequence \( x^{id} = (c_1^{id} \ldots c_n^{id}) \), where each \( c_i^{id} \) is a character. (This instantiation of the natural constraint is defined only on types for identifiers that occur in the source code, such as a function identifier or a parameter identifier.) This is a classification problem, where the input is \( x^{id} \), and the output classes are the set of our concrete types. Ideally, the classifier would learn that identifier names that are lexically similar tend to have similar types, and specifically which subsequences of the character names, like \( lst \), are highly
predictive of the type, and which subsequences are less predictive. One simple way to do so is to use an RNN.

For our purposes, an RNN is simply a function \((h_{i-1}, w_i) \mapsto h_i\) that maps a state vector \(h_{i-1} \in \mathbb{R}^H\) and an arbitrary input \(w_i\) to an updated state vector \(h_i \in \mathbb{R}^H\). (The dimension \(H\) is one of the hyperparameters of the model, which can be tuned to obtain the best performance.) The RNN has continuous parameters that are learned to fit a given data set, but we elide these parameters to lighten the notation, because they are trained in a standard way. We use a particular variant of an RNN called character-level long-short term memory (LSTM) network, which has proven to be particularly effective both for natural language and for source code (Sundermeyer et al., 2012; Melis et al., 2018; White et al., 2015; Khanh Dam et al., 2016). Such a network consists of LSTM units (Hochreiter and Schmidhuber, 1997) and can process a sequence of characters. We now proceed to describe the specific details of the character-level LSTM network that we use.

**Character-level LSTM Network Architecture and Training details**

To mathematically denote the output of an LSTM unit, we use the notation \(\text{LSTM}(h_{i-1}, w_i)\). We now describe how our LSTM network processes identifiers. For a given identifier \(x^{id} = (c_{id}^1 \ldots c_{id}^n)\), our network inputs each character \(c_{id}^i\) to an LSTM unit sequentially and finally, outputs a state vector \(h_n\). This vector \(h_n\) is then passed as input to a simple neural network that outputs the natural constraint \(n_{id}\). Formally, we have the following:

\[
\begin{align*}
    h_i &= \text{LSTM}(h_{i-1}, c_{id}^i) \quad i \in 1, \ldots, n \tag{8.1a}
    \\
    n_{id} &= F(h_N), \tag{8.1b}
\end{align*}
\]

where \(F : \mathbb{R}^H \rightarrow \mathbb{R}^T\) is a simple (feedforward) neural network. This network structure is, by now, a fairly standard architectural motif in deep learning. We leave the possibility of incorporating sophisticated networks as promising future work.

We now also specify our choice of the architecture for \(F\). We choose \(F\) to be a feedforward neural network with no additional hidden layers, described as follows

\[
F(h) = \log \left( \sigma (hA^T + b) \right),
\]

where the log function is applied componentwise, and \(A\) and \(b\) are learnable weights and bias. The softmax function (6.5) corresponds to the last layer of our neural network and essentially maps the values of the previous layer to \([0, 1]\), while the sum of all values is 1 as expected for a probability vector as already explained. We work in log space to help
8.2. NATURAL CONSTRAINTS FOR OPTYPER

Figure 8.1: Pipeline of learning naming conventions with a character-level LSTM, represented by a probability vector for each identifier.

numerical stability since computing (6.5) directly can be problematic. As a result, $F$ outputs values in $[-\infty, 0]$.

We train our character-level LSTM network on $(id, type)$ pairs consisting of variable identifiers together with their known types, and minimizing a loss function. Our chosen loss function is the negative log likelihood function—conveniently combined with our log output—defined as

$$L(N) = -\sum_{i} \log(n_{id})$$  \hspace{1cm} (8.3)

where $n_{id}$ is a natural constraint as defined in (8.1b) and $N$ is a matrix of size $V \times T$ that aggregates all the natural constraints (Definition 5.1).

We set the maximum number of iterations to 2,000, which suffices in practice for the loss to stabilise.

We use ADAM (Kingma and Ba, 2014), an extension of stochastic gradient descent (Robbins and Monro, 1951), as our optimization algorithm for the natural constraints. The main difference between ADAM and classical stochastic gradient descent is the use of adaptive instead of fixed learning rates. Although there exist other algorithms with adaptive learning rates like ADAGRAD (Duchi et al., 2011) and RMSPROP (Tieleman and Hinton, 2014), ADAM tends to have better convergence (Ruder, 2016).

The model is written in PyTorch (Paszke et al., 2017) and trained in an NVIDIA GeForce GTX 1080 Ti GPU.

Training Data Set

Following the work of Wei et al. (2020) and Hellendoorn et al. (2018), to train our model we use as dataset the 300 most starred Typescript projects from Github, containing between 500 to 10,000 lines of code. Our dataset was randomly split by project into 70% training
data, 10% validation data and 20% test data. Figure 8.1 shows a summary of the pipeline used to train our model.

**Extracting Natural Constraints**

To obtain the natural probability matrix $N$ as described in Definition 5.1, we rely on the probability distributions over named types for each identifier returned by our character-level LSTM network. In particular, we define the probability vector $n_v$ associated with each type variable $\alpha_v$ as

$$n_v = \left\{ \begin{array}{l}
[1/T, \ldots, 1/T], \\
\text{Average}(\{n_{id} \mid id \in \text{id}_v \})
\end{array} \right. \tag{8.4}$$

where $\text{id}_v$ denotes the identifiers associated with type variable $\alpha_v$ (as defined in Section 5.1), and $n_{id}$ is the probability distribution output by our LSTM network (as defined in Equation (8.1b)). The function $\text{Average}$ when applied to a set of probability vectors simply computes the point-wise average for each element of the vectors.

### 8.3 Combining Logical and Natural Constraints with Optyper

**Combining Constraints using Continuous Optimization**

In our framework both solving the relaxed logical constraints alone, and combining them with the natural constraints correspond to an optimization problem as described in (6.3). Our implementation is written in PyTorch (Paszke et al., 2017), and, as the PyTorch’s optimization package does not solve non-linear constraints, we relied on the unconstrained optimization problem, presented in (6.7), to solve our problem. Our experiments show that choosing $k = 1000$ as penalty multiplier suffices in most cases to satisfy the logical constraints. To find the minimum of the generated function, we use RMSprop (Tieleman and Hinton, 2014), an alternative to stochastic gradient descent (Robbins and Monro, 1951), with an adaptive learning rate. Finally, we note that, for numerical stability we need to use logits instead of probabilities as described below.
Logical Constraints in the Logit Space

In Definition 6.2, we present the continuous interpretation based on probabilities. As already mentioned, in the actual implementation we use logit instead for numerical stability. The logit of a probability is the logarithm of the odds ratio. Precisely, for an element \( p \in [0, 1] \) of a probability vector, the logit \( \pi \) corresponds to

\[
\pi = \log \frac{p}{1-p}.
\]

It allows us to map probability values from \([0, 1]\) to \([-\infty, \infty]\).

Given the matrix \( L \), which corresponds to the logit of the matrix \( P \) in Definition 6.2, we have that \( \log([C]_P) = [C]_L \). We interpret a constraint \( C \) as a number \([C]_P \in \mathbb{R}\) as follows:

\[
\begin{align*}
[a_v = t_{\tau}]_L &= \pi_{a_v, \tau} \\
[-C]_L &= \log(1 - \exp([C]_L)) \\
[C_1 \land C_2]_L &= [C_1]_L + [C_2]_L \\
[C_1 \lor C_2]_L &= \log(\exp([C_1]_L) + \exp([C_2]_L) - \exp([C_1]_L + [C_2]_L)).
\end{align*}
\]

The sigmoid function is defined as

\[
\text{sigmoid}(a) = \frac{\exp\{a\}}{1 + \exp\{a\}},
\]

while the LogSumExp function is defined as

\[
\text{LogSumExp}(x) = \log \left( \sum_i \exp\{x_i\} \right).
\]

Combining Constraints using MaxSAT

To tackle the MaxSAT problem (defined in Section 7.1), we rely on an industrial SAT/SMT solver Z3 (de Moura and Björner, 2008) developed at Microsoft Research. It is one of the most widely used deductive solvers, having the ability to handle a large variety of problems related to constraint satisfiability. Due to its versatility, numerous software verification and program analysis tools, such as Boogie (Goues et al., 2011), Dafny (Leino, 2010) etc., are built on top of it.
While it started off as a SAT/SMT solver, it was later extended with the feature of handling MaxSAT/MaxSMT (Bjørner and Phan, 2014). Although there are many algorithms provided by Z3 to handle MaxSAT, we stick to the default algorithm known as MaxRes (Narodytska and Bacchus, 2014) since it is proven to scale well on large benchmarks. Roughly speaking, the MaxRes algorithm iteratively removes soft constraints having low weights, until it finds a subset of soft constraints that are consistent with the hard constraints.

Combining Constraints using ILP

To tackle the ILP problem (defined in Section 7.2), we rely on Gurobi (https://www.gurobi.com), one of the fastest mathematical programming solvers available. For solving LP problems, Gurobi uses a highly-optimized implementation of the Simplex algorithm (Dantzig, 1960) that incorporates many heuristics to scale down the size of the problem, including removing redundant variables and linear inequalities.

8.4 Evaluation of Optyper

In this section, we evaluate the ability our tool Optyper to predict types. To evaluate our tool systematically, we formulate a number of research questions, which we answer through our experiments. The research questions are as follows:

RQ1: How do the different methods for combining logical and natural constraints compare against each other?

RQ2: How effective is it to combine logical constraints and natural constraints for type inference?

RQ3: How does our approach compare with prior work in predicting types?

To answer the research questions, we use the same benchmark set as was used by Wei et al. (2020); Hellendoorn et al. (2018). This is the exactly the same 20% test set that was used for the evaluation of our character-level LSTM model. This test set consists of 1304 files from 60 GitHub packages with ∼10000 annotation slots, locations in the source where type annotations are permitted.

As we alluded in the beginning of the chapter, for all the experiments, we restrict Optyper’s type vocabulary to be the top-100 most common library types occurring in our training set. As Wei et al. (2020) report, this space covers 98% of the non-any annotations in the
dataset. While we impose this restriction, for extensive evaluation, one can easily extend Optyper to handle a larger vocabulary.

For handling polymorphic types, Optyper conforms to prior work (Wei et al., 2020; Hellendoorn et al., 2018; Xu et al., 2016; Raychev et al., 2015) and mapping the arguments of polymorphic types to \textit{any}. For example, it maps Promise<boolean> to Promise<any>. Moreover, Optyper in its type vocabulary maps higher-order functions to the type \textit{Function}. Note that Optyper does not include the \textit{any} type outside of polymorphic types or Out-Of-Vocabulary \textit{OOV} token.

In what follows, we answer the three research questions based on the results obtained from our experiments.
RQ1: Comparison of Joint Optimization Techniques

To answer RQ1, we compare the performance of three techniques presented for the joint optimization of the logical and natural constraints. To distinctly identify the techniques, we call the technique using continuous optimization from Section 6.4 as Optyper-ContOpt, the one using MaxSAT from Section 7.1 as Optyper-MaxSAT and the one using ILP from Section 7.2 as Optyper-ILP. We compare the algorithms based on two parameters, the time required to predict types and the accuracy obtained.

Figure 8.2 demonstrates the pair-wise comparison of the running times on all test files and Table 8.1 presents the average running time. From the charts, one can clearly see that both Optyper-MaxSAT and Optyper-ILP outperform Optyper-ContOpt in running time. This fact is also reflected in the average running time of the algorithms. A possible explanation for the underwhelming performance of Optyper-ContOpt is that, in its optimization formulation, it relies on a non-linear objective function. Both Optyper-MaxSAT and
8.4. EVALUATION OF OPTYPER

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Avg-Time (in sec)</th>
<th>Avg-Accuracy (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optyper-ContOpt</td>
<td>29.20</td>
<td>81</td>
</tr>
<tr>
<td>Optyper-MaxSAT</td>
<td>10.12</td>
<td>83</td>
</tr>
<tr>
<td>Optyper-ILP</td>
<td>1.44</td>
<td>83</td>
</tr>
</tbody>
</table>

Table 8.1: Comparison of average runtime and average accuracy of the different joint optimization techniques. Num-SAT denotes the number of instances where type assignment satisfies the logical constraints, Avg-Time denotes the average running time of the algorithm in seconds, and Avg-Accuracy denotes the average accuracy of the algorithm.

Optyper-ILP rely on linear constraints and objective functions.

Also, from both Figure 8.2 and Table 8.1, we can observe that Optyper-ILP is faster than Optyper-MaxSAT. The main reason is the tool used for solving the ILP formulation, Gurobi, is well suited for solving ILP instances. The tool used for solving MaxSAT, Z3, on the other hand, is not a dedicated MaxSAT solver. Using dedicated MaxSAT solvers can lead to the efficient solving of the MaxSAT instances and thus, improve running time.

Next, we compare the accuracy achieved when the type predictions are compared to the annotations in the test files (see Section 8.2). These annotations include the ones that the programmer has written explicitly and the ones that the TypeScript compiler can infer.

Figure 8.3 compares the pair-wise accuracy of the algorithm on individual test files, while Table 8.1 provides the average accuracy on all files. We observe that Optyper-ILP and Optyper-MaxSAT obtain an identical accuracy performance. This is because the tools being used to solve ILP and MaxSAT are known to be sound and complete and thus return optimal solutions, which in almost all cases are unique. Optyper-ContOpt, however, achieves different accuracy than Optyper-ILP and Optyper-MaxSAT in many cases. This is because, for solving the optimization in Optyper-ContOpt, we rely on RMSProp (an extension of gradient descent), which may not always converge to an optimal solution (typically, when the number of epochs performed is low).

To answer RQ1, in terms of runtime, Optyper-ILP displays the best performance, almost 10 times faster, on average, than other algorithms. In terms of accuracy, all the algorithms achieve comparable accuracy. Since Optyper-ILP has the best overall performance, for the next parts, if we write Optyper we mean Optyper-ILP.
RQ2: Effectiveness of Combining Logical and Natural Constraints

To answer RQ2, we compare Optyper against two other algorithms: LC-only and NC-only. The algorithm LC-only only considers the logical constraints and searches for type assignments that only satisfy the logical constraints. The algorithm NC-only, on the other hand, only considers the natural constraints and returns a type assignment that assigns a type variable with the top-1 type prediction made by the LSTM model. We now compare the algorithms on the test set based on three parameters, which we describe next. Table 8.2 summarizes the comparison results based on the parameters.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Num-SAT</th>
<th>Avg-NatVal</th>
<th>Avg-Accuracy (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC-only</td>
<td>1304</td>
<td>0.212</td>
<td>67</td>
</tr>
<tr>
<td>NC-only</td>
<td>228</td>
<td>0.238</td>
<td>52</td>
</tr>
<tr>
<td>Optyper-ILP</td>
<td>1304</td>
<td>0.236</td>
<td>83</td>
</tr>
</tbody>
</table>

First, we compare the algorithms based on the number of test files in which it produces a satisfiable type assignment based on the logical constraints. We observe that, as expected, algorithms Optyper and LC-only satisfy the logical constraints in all files. NC-only satisfies the logical constraints in only 17.4% of all instances. This indicates that simply relying on machine learning based models might result in unsound type assignments.

Second, we compare the algorithms based on the natural value of the resulting type assignments using the function NatVal, introduced in Definition 5.3. Here we observe that LC-only, in general, produces type assignments with the lowest NatVal values among the three algorithms. This indicates that simply satisfying the logical constraints need not produce natural types.

Finally, we compare the algorithms based on their accuracy, which measures how close the predicted types are to the original ones. We observe that Optyper has the best accuracy, asserting the effectiveness of combining information from two types of constraints.

To answer RQ2, we can say that combining logical and natural constraints greatly benefits the inference of natural and sound types, as is evident from Table 8.2.
RQ3: Comparison to Existing Techniques

We now present a comparison of Optyper against two state-of-the-art techniques in learning-based type inference, Lambdanet (Wei et al., 2020), and TypeWriter (Pradel et al., 2020). We compare against these techniques since the underlying idea for both of the techniques is to combine program logic based information and natural information. To do so, Lambdanet employs a deep-learning based method, while TypeWriter employs a search-based method guided by the natural information.

![Comparison of Optyper against Lambdanet and TypeWriter](image_url)

**Figure 8.4:** Number of SAT instances for LambdaNet, TypeWriter algorithms and Optyper in which none of the algorithms timed out.

![Accumulated runtime of Optyper, LambdaNet and TypeWriter](image_url)

**Figure 8.5:** Accumulated runtime of Optyper, LambdaNet and TypeWriter algorithms on all the benchmarks. The timeout of the algorithms was set to be 600 secs.
Table 8.3: Comparison of the number of sat instances and the accuracy of LambdaNet, NC-only, LC-only and Optyper only on variables for which LambdaNet can predict types. Num-SAT denotes the number of instances where type assignment satisfies the logical constraints and Avg-Accuracy denotes the average accuracy of the algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Num-SAT</th>
<th>Avg-Accuracy (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC-only</td>
<td>228</td>
<td>51.4</td>
</tr>
<tr>
<td>LC-only</td>
<td>1304</td>
<td>71.1</td>
</tr>
<tr>
<td>LambdaNet</td>
<td>366</td>
<td>68.5</td>
</tr>
<tr>
<td>Optyper-ILP</td>
<td>1304</td>
<td>84.1</td>
</tr>
</tbody>
</table>

Comparison to LambdaNet

The comparison with LambdaNet is straightforward because they provide a pretrained model trained on the same dataset and for the same set of types as ours. We compare Optyper against LambdaNet a number of different parameters.

Figure 8.5 shows the runtime comparison of LambdaNet with Optyper-algorithms (note that the figure also contains a comparison with TypeWriter). As evident, LambdaNet performs significantly faster than any other type inference method. This is because LambdaNet simply relies on a deep-learning model that can be queried very fast. The other methods employ search techniques to ensure the soundness of the type prediction, which can be time-consuming.

In Figure 8.4, we present the number of test files in which the type prediction satisfies the logical constraints that we generate. Being a method that simply relies on machine learning model, the type prediction made by LambdaNet can definitely be unsound. Nevertheless, we provide this result as an indication of how efficiently a model can learn logical constraints.

In Table 8.3, we present the average accuracy obtained when compared with the original annotations from the test files (see Section 8.2). We present the number for NC-only and LC-only to understand the result better. We see that our accuracy is significantly better than that of LambdaNet. We think that a reason for the small difference between LambdaNet’s performance in all declaration slots and the reported performance in (Wei et al., 2020) is due to the fact that LambdaNet fails to learn types that are inferable for the compiler but yet not apparent in the training data.

Figure 8.6 demonstrate two specific examples where LambdaNet fails to predict types, for which the logical constraints are apparent directly in the code. The parameters on
lines 3 and 4 actually have type `Window` and `Event`, as you can see in Section 8.4, which contains the developer-annotated ground truth. Section 8.4, the figure on the right, shows that LambdaNet mispredicts their types as `String` and `Number`. We conjecture that the misprediction is because of data sparsity. LambdaNet correctly predicts the type of the first parameter because uses of `boolean` are relatively common in the training data, while uses of `Window` and `Event` are not, so the assignments on lines 7 and 8 provide too little signal for LambdaNet to pick up. Optyper, in contrast, correctly predicts all three parameter types. Optyper succeeds here because the assignments on lines 7 and 8 generate hard logical constraints that Optyper incorporates, at test time, into its optimization search for a satisfying type environment. These examples may explain the difference between LambdaNet’s and Optyper’s prediction accuracy. This difference in performance between the two approaches will crop up whenever the training data lacks sufficient number of examples of a particular logical relation.

Finally, we note that we see LambdaNet and indeed any learning approach as complementary to our work. In theory, it is straightforward to treat them as an instantiation of the natural phase, as a probability distribution over types described in our theoretical framework (Section 5.3).

**Comparison to TypeWriter**

We now compare Optyper against the search-based technique used by TypeWriter (Pradel et al., 2020), which is illustrated in Algorithm 1 in their paper. The implementation of this algorithm, however, is not publicly available. Moreover, the tool is developed for Python, while we develop for TypeScript. Due to these reasons, based on the algorithm provided
in the paper, we faithfully re-implemented their approach in Python and use it for our comparison. While they provide two approaches, a greedy and a non-greedy one, for our experiments, we do not use the non-greedy approach, since it takes a prohibitively large amount of time. Further, their algorithm relies on a parameter $w$ which only searches for the type assignment in the top $w$ choices proposed by the LSTM model. We ran their algorithm with three values of $w$, 1, 5, 10, which resulted in the three algorithms TypeWriter-1, TypeWriter-5 and TypeWriter-10.

**Table 8.4:** Comparison of the accuracy of TypeWriter algorithms and Optyper on all benchmarks. Avg-Accuracy denotes the average accuracy of the algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Avg-Accuracy (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeWriter-1</td>
<td>51.7</td>
</tr>
<tr>
<td>TypeWriter-5</td>
<td>63.4</td>
</tr>
<tr>
<td>TypeWriter-10</td>
<td>68.9</td>
</tr>
<tr>
<td>Optyper-ILP</td>
<td>82.0</td>
</tr>
</tbody>
</table>

Figure 8.5 compares the runtimes of the different algorithms. Here, we notice that Optyper-ILP has a better runtime than all of the TypeWriter algorithms, which performs worse when $w$ increases.

Figure 8.4 presents a comparison of the algorithms based on the number of instances in which they return a type assignment that satisfies the logical constraints. We observe that Optyper, in all of the instances, in which none of the algorithms timeout, returns a type assignment that satisfies the logical constraints. All of the TypeWriter algorithms, however, perform significantly worse in returning a sound type assignment. This can be attributed to two main reasons. First, as it is a greedy approach, during its search TypeWriter can overlook the type assignments that could satisfy the logical constraints. Second, the parameter $w$, which TypeWriter relies upon to reduce its space of types, results in ignoring the types that the logical constraints assert.

Finally, Table 8.4 presents the average accuracy of the algorithms. Here again, Optyper displays the best accuracy due to its complete search technique.

To answer RQ3, we can safely say that Optyper outperforms the state-of-the-art techniques in many aspects. Compared to LambdaNet, Optyper achieves at least 10% higher average accuracy and also, ensures all type predictions are sound. Compared to all TypeWriter algorithms, Optyper searches types at least two times faster and achieves an average accuracy of at least 10% higher.
Conclusion

We conclude this dissertation by providing a summary of the research conducted, highlighting some of the current limitations and shedding light on some possible future directions.

9.1 Summary

This dissertation addresses the lack of a rich type inference process for dynamically typed languages. To tackle this, we combine logical constraints, that is, deterministic information from a type system, with natural constraints, that is, uncertain information about types, learned by machine learning techniques, while focusing on the satisfaction of the typing rules dictated by the language. The key insight of our method is to constructively combine the natural and logical part using different optimization techniques with theoretical guarantees.

We have thus formally developed an inference system, from the ground up, that assigns type names to arbitrary type structures. This type system captures key aspects of type inference in optionally typed languages used in industry, like TypeScript and Python. Crucially, we have validated this system by theorem and proof. This work fully formalizes and proves termination, and correctness for a natural type inference algorithm.

We describe Optyper, a realization of our algorithm for natural type inference for TypeScript, and demonstrate its effectiveness (Section 8.4). The combination of logical and natural constraints yields a large improvement in performance over either natural or logical constraints individually, improves accuracy over the-state-of-the-art tool for predicting types for TypeScript by Wei et al. (2020), and demonstrates a better search technique than other type inference algorithms such as TypeWriter (Pradel et al., 2020). Our framework
is extensible: it can incorporate information from arbitrary models into its natural part and type constraints generated by traditional deterministic type inference systems.

As our literature review makes clear, all work in learning-based type inference to date focuses on formalizing their method, none states theorems or formally proves its approach to be sound by construction, and all are empirically validated. The present work rises to address this challenge. We have formally developed an inference system, from the ground up, that assigns type names to arbitrary type structures. This type system captures key aspects of type inference in optionally typed languages used in industry, like TypeScript and Python. Crucially, we have validated this system by theorem and proof. This work is the first to formalize and prove termination and soundness for a natural type inference algorithm.

**Limitations** At this point, it is important to underline limitations of the discussed topics. These essentially stimulate the discussion about possible future directions and improvements. Our algorithm only chooses types from the given library of type definitions. Hence, an input expression will be rejected if it needs a record or function type missing from the library. Another limitation is that we ignore field names when generating natural constraints. We expect it would be straightforward to extend the inference algorithm to augment the given library with type equations defining additional record or function types, as needed, and to take field names into account.

A bigger challenge is to extend natural type inference to features including subtyping, parametric polymorphism, and intersection and union types, important for TypeScript and other languages.

### 9.2 Future Work

Next, we discuss some possible future directions in more details.

**Extend to a Formal System with Subtyping**

As follow up work it would be interesting to look at how we can extend our core ideas to Featherweight Java (Jangda and Anand, 2019). Featherweight Java is a pure subset of Java, as a classic formalization of a nominal type system with subtyping. As in the current formalization we have not considered subtyping, we could use Featherweight Java as a standard off-the-shelf formalization of object oriented subtyping. Furthermore, it would be interesting to explore how our system would extend to inheritance.
Predicting from an Open Type Vocabulary

A harder problem is trying to predict complex types. This could include objects, higher order functions, unions and intersection types, or user defined types. In this case, we need to find a way to handle an actual type lattice, consider the hierarchy in-between types and the possible infinite combinations of them. To do so, it seems necessary to start by defining a particular type lattice—the official documentation of TypeScript does not explicitly defines one. Regarding the learning part, this extension will probably lead us to learn over trees instead of multiple labels. Furthermore, we will need to find a way to embed this hierarchy into the optimisation problem, presumably by using some kind of linearization over the tree structure. LambdadNet has already shown that is possible, by using using a pointer-network-like architecture (Vinyals et al., 2015). A different approach would be to use Source Code Embeddings from Language Models (SCELMo) (Karampatsis and Sutton, 2020) where the embeddings of each token depend on its context of the input sequence and thus even out-of-vocabulary (OOV) tokens have effective input representations, based on Peters et al. (2018)’s work.

Add Terms to the Optimization Problem

Another aspect is to be able to handle infinite or unseen combinations of types. We think it is compelling to include information from dynamic analysis in our approach. For the pure dynamic analysis part there is already some related work for testing TypeScript declaration files dynamically that could be viewed as a starting point (Kristensen and Møller, 2017; Graves et al., 2014). What seems more interesting is how we will integrate the new source of information to our model; for this we will probably have to add a new weighted term to the optimization problem, which will depend on the evidence we get through the dynamic analysis. This particular extension could be extremely useful, as it opens a way towards automatic transformation from JavaScript to TypeScript code.

Learn the Logical Constraints

A reasonable extension is to incorporate the continuous optimization function in the training process of our LSTM. Doing so we our model would possibly learn relations between logical constraints and the corresponding type. (Selsam et al., 2018; Wang et al., 2019) on their recent work have shown different ways to learn how to solve SAT problems using deep learning; that suggests that we could encode the logical constraints directly on the learning phase instead of enforce them at prediction time.
Application on Automatic Program Repair

A common pattern seen in Automatic Program Repair (APR) (Gazzola et al., 2019) approaches is that of a neural model generating a sequence of source code tokens that serves as a replacement for a bug. However, available information on which tokens are legal at a given position, such as in-scope variables, programming language syntax is not always used. We believe that it should be possible to formulate optimization problems similar to the one described in this dissertation, namely consisting of a neural model that outputs source code tokens for the natural part, and logical constraints based on which tokens are legal at a given position (for instance based on information from a compiler or a static analysis tool), which could probably achieve good performance due to narrowing down the space of possible results via the logical constraints.
Appendix

A.1 Proofs for Soundness and Completeness of the Algorithmic Typing System

Lemma 4.6 (Soundness). If $\Gamma^o \vdash E \Rightarrow \alpha (C, V)$ and $\text{dom}(\mu) = \text{tyvar}(\Gamma^o)$ and $\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, C, V)$ and $(\mu \cup \mu')(\alpha) <: t$ and $\mu \cup \mu' \models C$ then $\mu(\Gamma^o) \vdash E : t$.

Proof. Note that here, we only provide the cases that we omit in the main text after Lemma 4.6.

It suffices to prove that for all $\Gamma^o, E, \alpha, C, V, \mu, \mu', t$ that, if

1. $\Gamma^o \vdash E \Rightarrow \alpha (C, V)$
2. $\text{dom}(\mu) = \text{tyvar}(\Gamma^o)$
3. $\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, C, V)$
4. $(\mu \cup \mu')(\alpha) <: t$
5. $\mu \cup \mu' \models C$

then $\mu(\Gamma^o) \vdash E : t$.

The proof is by induction on the derivation of $(1) \Gamma^o \vdash E \Rightarrow \alpha (C, V)$. We proceed by considering each rule that can derive judgment $(1)$. Notice that in each case, there can only be one rule from Definition 3.9 that can derive the declarative judgement. In each case, we can assume $(2) \text{dom}(\mu) = \text{tyvar}(\Gamma^o)$, $(3) \text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, C, V)$, $(4) (\mu \cup \mu')(\alpha) <: t$, and $(5) \mu \cup \mu' \models C$.

Case $E = E_1 < E_2$ Our judgment $(1)$ is derived as follows, with $E = E_1 < E_2$ and
\( C = \alpha \iff \text{Bool} \land \alpha_1 \iff \text{Int} \land \alpha_2 \iff \text{Int} \land C_1 \land C_2 \) and \( V = V_1 \cup V_2 \).

(Algo \( \langle \) \( (\alpha \notin \text{tyvar}(\Gamma^\circ, C_1, C_2, V_1, V_2) \) and \( \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1) \cap \text{newtyvar}(\Gamma^\circ, \alpha_2, C_2, V_2) = \emptyset) \)

\[
\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \quad \Gamma^\circ \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2)
\]

\( \Gamma^\circ \vdash E_1 < E_2 \Rightarrow \alpha (\alpha \iff \text{Bool} \land \alpha_1 \iff \text{Int} \land \alpha_2 \iff \text{Int} \land C_1 \land C_2) \)

From (2) \( \text{dom}(\mu) = \text{tyvar}(\Gamma^\circ) \) and (3) \( \text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V) \), and (4) \( (\mu \cup \mu')(\alpha) \iff t \), and (5) \( \mu \cup \mu' \models \alpha \iff \text{Bool} \land \alpha_1 \iff \text{Int} \land \alpha_2 \iff \text{Int} \land C_1 \land C_2 \), we are to show \( \mu(\Gamma^\circ) \vdash E_1 < E_2 : t \).

By (5) and Definition 4.4, we have \( \mu \cup \mu' \models \alpha \iff \text{Bool} \) and \( \mu \cup \mu' \models \alpha_1 \iff \text{Int} \) and \( \mu \cup \mu' \models \alpha_2 \iff \text{Int} \) and \( \mu \cup \mu' \models C_1 \) and \( \mu \cup \mu' \models C_2 \).

By (4) and Lemma 4.1(c) three times, we get

i. for \( \iota = \text{Bool} \), \( (\mu \cup \mu')(\alpha) \iff \text{Bool} \) because \( \mu \cup \mu' \models \alpha \iff \text{Bool} \),

ii. for \( \iota = \text{Int} \), \( (\mu \cup \mu')(\alpha_1) \iff \text{Int} \) because \( \mu \cup \mu' \models \alpha_1 \iff \text{Int} \),

iii. for \( \iota = \text{Int} \), \( (\mu \cup \mu')(\alpha_2) \iff \text{Int} \) because \( \mu \cup \mu' \models \alpha_2 \iff \text{Int} \).

From (i) and (4) it must be that \( t \iff \text{Bool} \). Let \( t_1 = (\mu \cup \mu')(\alpha_1) \) and \( t_2 = (\mu \cup \mu')(\alpha_2) \).

Hence, from (ii) and (iii), it must be that \( t_1 \iff \text{Int} \) and \( t_2 \iff \text{Int} \).

Let \( \mu'_1 = \mu' \upharpoonright \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1) \).

Hence, we have (3.1) \( \text{dom}(\mu'_1) = \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1) \).

From \( \Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \) and Lemma 4.3 we get \( \alpha_1 \in \text{dom}(\Gamma^\circ) \cup \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1) \).

Hence, from (3.1) and (ii) we get (4.1) \( (\mu \cup \mu'_1)(\alpha_1) \iff \text{Int} \).

From \( \Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \) and Lemma 4.4 we get that \( \text{tyvar}(C_1) \subseteq \text{dom}(\Gamma^\circ) \cup \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1) \).

Hence, from \( \mu \cup \mu' \models C_1 \) and Lemma 4.5 we get (5.1) \( \mu \cup \mu'_1 \models C_1 \).

By induction hypothesis, \( \Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \), (2), (3.1), (4.1), (5.1) imply \( \mu(\Gamma^\circ) \vdash E_1 : \text{Int} \).

By symmetric reasoning, we obtain \( \mu(\Gamma^\circ) \vdash E_2 : \text{Int} \).

Hence, for \( \Gamma = \mu(\Gamma^\circ) \) we can derive the desired judgment

\[
(\text{DECL EXPR} \langle \rangle \langle t_1 \iff \text{Int} \) and \( t_2 \iff \text{Int} \rangle)
\]

\[
\Gamma \vdash E_1 : t_1 \quad \Gamma \vdash E_2 : t_2 \quad t \iff \text{Bool}
\]

\[
\Gamma \vdash E_1 < E_2 : t
\]
A.1. PROOFS FOR SOUNDNESS AND COMPLETENESS OF THE ALGORITHMIC TYPING SYSTEM

Case $E = E_1 == E_2$ Our judgment (1) is derived as follows, with $E = E_1 == E_2$ and

$C = \alpha \leftarrow Boolean \land \bigvee_{i \in \{Boolean, Integer\}} (\alpha_1 \leftarrow i \land \alpha_2 \leftarrow i) \land C_1 \land C_2$ and $V = V_1 \cup V_2$.

$$\text{(ALGO} == \text{)} (\alpha \notin \text{tyvar}(\Gamma^\circ, C_1, C_2, V_1, V_2) \text{ and } \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1) \cap \text{newtyvar}(\Gamma^\circ, \alpha_2, C_2, V_2) = \emptyset)$$

$$\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \quad \Gamma^\circ \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2)$$

From (2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)$ and (3) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V)$, and (4) $\mu \cup \mu' (\alpha) \leftarrow t$, and (5) $\mu \cup \mu' \models \alpha \leftarrow Boolean \land \bigvee_{i \in \{Boolean, Integer\}} (\alpha_1 \leftarrow i \land \alpha_2 \leftarrow i) \land C_1 \land C_2$, we are to show $\mu(\Gamma^\circ) \vdash E_1 == E_2 : t$.

By (5) and Definition 4.4, we have $\mu \cup \mu' \models \alpha \leftarrow Boolean$ and $\mu \cup \mu' \models \alpha_1 \leftarrow Boolean$ or $\mu \cup \mu' \models \alpha_1 \leftarrow Int$ and $\mu \cup \mu' \models \alpha_2 \leftarrow Int$ and $\mu \cup \mu' \models C_1$ and $\mu \cup \mu' \models C_2$.

By (4) and Lemma 4.1(c) five times we get

i. for $t = Boolean$, $(\mu \cup \mu')(\alpha) \leftarrow Boolean$ because $\mu \cup \mu' \models \alpha \leftarrow Boolean$,

ii. for $t = Boolean$, $(\mu \cup \mu')(\alpha_1) \leftarrow Boolean$ because $\mu \cup \mu' \models \alpha_1 \leftarrow Boolean$,

iii. for $t = Boolean$, $(\mu \cup \mu')(\alpha_2) \leftarrow Boolean$ because $\mu \cup \mu' \models \alpha_2 \leftarrow Boolean$,

iv. for $t = Int$, $(\mu \cup \mu')(\alpha_1) \leftarrow Int$ because $\mu \cup \mu' \models \alpha_1 \leftarrow Int$,

v. for $t = Int$, $(\mu \cup \mu')(\alpha_2) \leftarrow Int$ because $\mu \cup \mu' \models \alpha_2 \leftarrow Int$.

Thus we have that $(\mu \cup \mu')(\alpha) \leftarrow Boolean$ and $(\mu \cup \mu')(\alpha_1) \leftarrow Boolean$ and $(\mu \cup \mu')(\alpha_2) \leftarrow Boolean$ or $((\mu \cup \mu')(\alpha_1) \leftarrow Int$ and $(\mu \cup \mu')(\alpha_2) \leftarrow Int$.

From (i), it must be that $t \leftarrow Boolean$.

Let $\mu' = \mu' \upharpoonright \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1)$.

Hence, we have (3.1) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1)$.

From $\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$ and Lemma 4.3 we get $\alpha_1 \in \text{dom}(\Gamma^\circ) \cup \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1)$.

Hence, from (3.1) and (ii) we get (4.1) $t_1 \leftarrow Boolean$ or $t_1 \leftarrow Int$.

From $\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$ and Lemma 4.4 we get that $\text{tyvar}(C_1) \subseteq \text{dom}(\Gamma^\circ) \cup \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1)$.

Hence, from $\mu \cup \mu' \models C_1$ and Lemma 4.5 we get (5.1) $\mu \cup \mu' \models C_1$.

By induction hypothesis, $\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$, (2), (3.1), (4.1), (5.1) imply $\mu(\Gamma^\circ) \vdash E_1 : t_1$.

By symmetric reasoning, we obtain $\mu(\Gamma^\circ) \vdash E_2 : t_2$. 
Hence, for $\Gamma = \mu(\Gamma^o)$ we can derive the desired judgment

$$(\text{DECL EXPRT} \equiv=:) \ (t_1 \llt Bool \text{ and } t_2 \llt Bool) \text{ or } (t_1 \llt \text{Int} \text{ and } t_2 \llt \text{Int})$$

$$\Gamma \vdash E_1 : t_1 \quad \Gamma \vdash E_2 : t_2 \quad t \llt Bool$$

$$\Gamma \vdash E_1 == E_2 : t$$

Case Rcd. Our judgment (1) is derived as follows, with $E = \{\ell_i = E_i \ i \in 1..n\}$ and $C = \alpha \llt \{\ell_i : \alpha_i \ i \in 1..n\} \land \bigwedge_{i \in 1..n} C_i$ and $V = \bigcup_{i \in 1..n} V_i$.

$$(\text{ALGO RCD}) \ (\alpha \notin \bigcup_{i \in 1..n} tyvar(\Gamma^o, \alpha_i, C_i, V_i) \text{ and sets } newtyvar(\Gamma^o, \alpha_i, C_i, V_i) \text{ disjoint})$$

$$\Gamma^o \vdash E_i \Rightarrow \alpha_i (C_i, V_i) \quad \forall i \in 1..n$$

From (2) $\text{dom}(\mu) = tyvar(\Gamma^o)$ and (3) $\text{dom}(\mu') = newtyvar(\Gamma^o, \alpha, C, V)$, and (4) $(\mu \cup \mu')(\alpha) \llt t$, and (5) $\mu \cup \mu' \models \alpha \llt \{\ell_i : \alpha_i \ i \in 1..n\} \land \bigwedge_{i \in 1..n} C_i$, we are to show $\mu(\Gamma^o) \vdash \{\ell_i = E_i \ i \in 1..n\} : t$.

By Definition 4.4, we have $\mu \cup \mu' \models \alpha \llt \{\ell_i : \alpha_i \ i \in 1..n\}$ and $\mu \cup \mu' \models C_1$ and ... and $\mu \cup \mu' \models C_n$.

By applying Lemma 4.1(e) we get that $(\mu \cup \mu')(\alpha) \llt \{\ell_i : (\mu \cup \mu')(\alpha_i) \ i \in 1..n\}$ because $\mu \cup \mu' \models \alpha \llt \{\ell_i : \alpha_i \ i \in 1..n\}$.

Let $t_i = (\mu \cup \mu')(\alpha_i)$, for all $i \in 1..n$. Thus, from (4) it must be that $t \llt \{\ell_i : t_i \ i \in 1..n\}$.

Let $\mu'_i = \mu' \upharpoonright newtyvar(\Gamma^o, \alpha_1, C_1, V_1)$.

Hence, we have (3.1) $\text{dom}(\mu'_i) = newtyvar(\Gamma^o, \alpha_1, C_1, V_1)$.

From $\Gamma^o \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$ and Lemma 4.3 we get $\alpha_1 \in \text{dom}(\Gamma^o) \cup newtyvar(\Gamma^o, \alpha_1, C_1, V_1)$.

Hence, from (3.1) and $t_1 = (\mu \cup \mu')(\alpha_1)$ we get (4.1) $(\mu \cup \mu'_i)(\alpha_1) \llt t_1$.

From $\Gamma^o \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$ and Lemma 4.4 we get that $tyvar(C_1) \subseteq \text{dom}(\Gamma^o) \cup newtyvar(\Gamma^o, \alpha_1, C_1, V_1)$.

Hence, from $\mu \cup \mu' \models C_1$ and Lemma 4.5 we get (5.1) $\mu \cup \mu'_i \models C_1$.

By induction hypothesis, $\Gamma^o \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1)$, (2), (3.1), (4.1), (5.1) imply $\mu(\Gamma^o) \vdash E_1 : t_1$.

By symmetric reasoning, for all $i \in 2..n$ we obtain $\mu(\Gamma^o) \vdash E_i : t_i$.

Hence, for $\Gamma = \mu(\Gamma^o)$ we can derive the desired judgment
A.1. PROOFS FOR SOUNDNESS AND COMPLETENESS OF THE ALGORITHMIC TYPING SYSTEM

Case Proj. Our judgment (1) is derived as follows, with \( E = \{ \ell_i : t_i \}_{i \in 1..n} \) and \( C = \alpha' <:> \alpha.\ell \land C' \) and \( V = V' \).

\[
(\text{ALGO Proj}) \quad (\alpha' \notin \text{tyvar}(\Gamma', \alpha, C', V')) \\
\Gamma' \vdash E \Rightarrow \alpha (C', V') \\
\Gamma \vdash E.\ell \Rightarrow \alpha' (\alpha' <:> \alpha.\ell \land C', V')
\]

From (2) \( \text{dom}(\mu) = \text{tyvar}(\Gamma') \) and (3) \( \text{dom}(\mu') = \text{newtyvar}(\Gamma', \alpha, C', V') \), and (4) \( (\mu \cup \mu')'(\alpha') <:> t' \), and (5) \( \mu \cup \mu' \models \alpha' <:> \alpha.\ell \), we are to show \( \mu(\Gamma') \vdash E.\ell : t \).

By Definition 4.4, we have \( \mu \cup \mu' \models \alpha' <:> \alpha.\ell \) and \( \mu \cup \mu' \models C' \).

By applying Lemma 4.1(f) we get that \( (\mu \cup \mu')(\alpha') <:> \{ \ell_i : (\mu \cup \mu')(\alpha_i) \}_{i \in 1..n} \) and \( (\mu \cup \mu')(\alpha') <:> (\mu \cup \mu')(\alpha_j) \) and \( \ell = \ell_j \) and \( j \in 1..n \) because \( \mu \cup \mu' \models \alpha' <:> \alpha.\ell \).

Let \( t_i = (\mu \cup \mu')(\alpha_i) \), for every \( i \in 1..n \), and let \( t = (\mu \cup \mu')(\alpha) \). Thus, we get \( \ell <:: \{ \ell_i : t_i \}_{i \in 1..n} \) and \( t' <:: t_j \).

Let \( \mu'_i = \mu' \upharpoonright \text{newtyvar}(\Gamma', \alpha, C', V') \).

Hence, we have (3.1) \( \text{dom}(\mu'_i) = \text{newtyvar}(\Gamma', \alpha, C', V') \).

From \( \Gamma' \vdash E \Rightarrow \alpha (C', V') \) and Lemma 4.3 we get \( \alpha \in \text{dom}(\Gamma') \cup \text{newtyvar}(\Gamma', \alpha, C', V') \).

Hence, from (3.1) and \( t = (\mu \cup \mu')(\alpha) \) we get (4.1) \( (\mu \cup \mu')(\alpha) <:> t \).

From \( \Gamma' \vdash E \Rightarrow \alpha (C', V') \) and Lemma 4.4 we get that \( \text{tyvar}(C') \subseteq \text{dom}(\Gamma') \cup \text{newtyvar}(\Gamma', \alpha, C', V') \).

Hence, from \( \mu \cup \mu' \models C' \) and Lemma 4.5 we get (5.1) \( \mu_1 \cup \mu'_1 \models C' \).

By induction hypothesis, \( \Gamma' \vdash E \Rightarrow \alpha (C', V') \), (2), (3.1), (4.1), (5.1) imply \( \mu(\Gamma') \vdash E : t \).

Hence, for \( \Gamma = \mu(\Gamma') \) we can derive the desired judgment

\[
(\text{DECL Expr Rcd}) \\

\mu(\Gamma) \vdash E : t' \\
\mu(\Gamma) \vdash E.\ell_j : t_j
\]

Case If. We have:
\[\text{(ALGO IF)} \quad \begin{array}{c}
\Gamma^o \vdash E_1 \Rightarrow \alpha_1 \ (C_1, V_1) \\
\Gamma^o \vdash E_2 \Rightarrow \alpha_2 \ (C_2, V_2) \\
\Gamma^o \vdash E_3 \Rightarrow \alpha_3 \ (C_3, V_3)
\end{array}\]

\[\Gamma^o \vdash (E_1 \ ? \ E_2 : E_3) \Rightarrow \alpha_2 \ (\alpha_1 \Leftrightarrow \alpha_3 \wedge \bigwedge_{i \in 1..3} C_i, \bigcup_{i \in 1..3} V_i)\]

From (2) \(\text{dom}(\mu) = tyvar(\Gamma^o)\) and (3) \(\text{dom}(\mu') = newtyvar(\Gamma^o, \alpha, C, V)\), and (4) \(\mu \cup \mu')(\alpha_2) \Leftrightarrow t\), and (5) \(\mu \cup \mu' \models \alpha_1 \Leftrightarrow \alpha_3 \wedge \bigwedge_{i \in 1..3} C_i\), we are to show \(\mu(\Gamma^o) \vdash (E_1 \ ? \ E_2 : E_3) : t\).

By Definition 4.4, we have \(\mu \cup \mu' \models \alpha_1 \Leftrightarrow \alpha_3 \wedge \bigwedge_{i \in 1..3} C_i\) and \(\mu \cup \mu' \models C_1\) and \(\mu \cup \mu' \models C_3\).

By applying Lemma 4.1(c) for \(\iota = \text{Bool}\) we get (i) \((\mu \cup \mu')(\alpha_1) \Leftrightarrow \text{Bool}\) because \(\mu \cup \mu' \models \alpha_1 \Leftrightarrow \text{Bool}\).

By applying Lemma 4.1(d) we get (ii) \((\mu \cup \mu')(\alpha_2) \Leftrightarrow (\mu \cup \mu')(\alpha_3)\) because \(\mu \cup \mu' \models \alpha_2 \Leftrightarrow \alpha_3\).

Let \(t_1 = (\mu \cup \mu')(\alpha_1), t_3 = (\mu \cup \mu')(\alpha_3)\). Hence, from (i) it must be that \(t_1 \Leftrightarrow \text{Bool}\) and from (ii) and (4) \(t_2 \Leftrightarrow t_3\).

Let \(\mu' = \mu' \upharpoonright \text{newtyvar} (\Gamma^o, \alpha_1, C_1, V_1)\).

Hence, we have \((3.1) \text{dom}(\mu') = newtyvar(\Gamma^o, \alpha_1, C_1, V_1)\).

From \(\Gamma^o \vdash E_1 \Rightarrow \alpha_1 \ (C_1, V_1)\) and Lemma 4.3 we get \(\alpha_1 \in \text{dom}(\Gamma^o) \cup newtyvar(\Gamma^o, \alpha_1, C_1, V_1)\).

Hence, from (3.1) and (ii) we get (4.1) \((\mu \cup \mu')(\alpha_1) \Leftrightarrow \text{Bool}\).

From \(\Gamma^o \vdash E_1 \Rightarrow \alpha_1 \ (C_1, V_1)\) and Lemma 4.4 we get that \(tyvar(C_1) \subseteq \text{dom}(\Gamma^o) \cup newtyvar(\Gamma^o, \alpha_1, C_1, V_1)\).

Hence, from \(\mu \cup \mu' \models C_1\) and Lemma 4.5 we get (5.1) \(\mu \cup \mu' \models C_1\).

By induction hypothesis, \(\Gamma^o \vdash E_1 \Rightarrow \alpha_1 \ (C_1, V_1), (2), (3.1), (4.1), (5.1)\) imply \(\mu(\Gamma^o) \vdash E_1 : t_1\).

By symmetric reasoning, we obtain \(\mu(\Gamma^o) \vdash E_2 : t_2\), and \(\mu(\Gamma^o) \vdash E_3 : t_3\).

By applying the (DECL EXPR RETYPE) rule for \(t_2 \Leftrightarrow t_3\), we can derive the desired judgment

\[\text{(DECL EXPR IF)} \quad (t_1 \Leftrightarrow \text{Bool})\]

\[\mu(\Gamma^o) \vdash E_1 : t_1 \quad \mu(\Gamma^o) \vdash E_2 : t_2 \quad \mu(\Gamma^o) \vdash E_3 : t_2\]

\[\mu(\Gamma^o) \vdash (E_1 \ ? \ E_2 : E_3) : t_2\]

\[\square\]
Lemma 4.7 (Completeness). Consider $\Gamma^\circ \vdash E \Rightarrow \alpha \ (C, V)$ and type $t$.

For all $\mu$ with $\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)$, if $\mu(\Gamma^\circ) \vdash E : t$ then there is $\mu'$ with $\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V)$ and $(\mu \cup \mu')(\alpha) \ll t$ and $\mu \cup \mu' \models C$.

Note that here, we only provide the cases that we omit in the main text after Lemma 4.7.

Proof. It suffices to prove that for all $\Gamma^\circ, E, \alpha, C, V, \mu, t$ that, if

1. $\Gamma^\circ \vdash E \Rightarrow \alpha \ (C, V)$
2. $\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)$
3. $\mu(\Gamma^\circ) \vdash E : t$

then there is $\mu'$ with (A) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V)$, (B) $(\mu \cup \mu')(\alpha) \ll t$, and (C) $\mu \cup \mu' \models C$.

The proof is by induction on the height of the derivation of the algorithmic judgment (1). We proceed by a case analysis of $E$. For each rule from (Algorithmic Typing Rules), only one of the syntax-directed rules from (Syntax-directed Declarative Typing Rules) can have derived declarative judgment (3). Hence we can obtain the desired satisfaction relation by a detailed case analysis.

Recall that $\text{newtyvar}(\Gamma^\circ, \alpha, C, V) = \text{tyvar}(\alpha, C, V) \setminus \text{tyvar}(\Gamma^\circ)$.

Case $E = E_1 < E_2$. In this case, our assumptions (1), (2) and (3) take the forms:

\[
\begin{array}{l}
\text{(ALGO <)} \quad (\alpha \notin \text{tyvar}(\Gamma^\circ, C_1, C_2, V_1, V_2) \text{ and } \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1) \cap \text{newtyvar}(\Gamma^\circ, \alpha_2, C_2, V_2) = \emptyset) \\
\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 \ (C_1, V_1) \\
\Gamma^\circ \vdash E_2 \Rightarrow \alpha_2 \ (C_2, V_2) \\
\Gamma^\circ \vdash E_1 < E_2 \Rightarrow \alpha \ (\alpha \ll \text{Bool} \land \alpha_1 \ll \text{Int} \land \alpha_2 \ll \text{Int} \land C_1 \land C_2, V_1 \cup V_2) \\
\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)
\end{array}
\]

\[
\begin{array}{l}
\mu(\Gamma^\circ) \vdash E_1 : t_1 \\
\mu(\Gamma^\circ) \vdash E_2 : t_2 \\
\mu(\Gamma^\circ) \vdash t < t_1 \land t_2 \\
\mu(\Gamma^\circ) \vdash t < t_1 \\
\end{array}
\]

We are to find $\mu'$ with (A) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V)$, (B) $(\mu \cup \mu')(\alpha) \ll t$, and (C) $\mu \cup \mu' \models C$, where $C = \alpha \ll \text{Bool} \land \alpha_1 \ll \text{Int} \land \alpha_2 \ll \text{Int} \land C_1 \land C_2$ and $V = V_1 \cup V_2$.

By induction hypothesis, (1) $\Gamma^\circ \vdash E_1 \Rightarrow \alpha_1 \ (C_1, V_1)$, (2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)$, and (3) $\mu(\Gamma^\circ) \vdash E_1 : t_1$ imply there is $\mu'$ with (A) $\text{dom}(\mu'_1) = \text{newtyvar}(\Gamma^\circ, \alpha_1, C_1, V_1)$, (B) $(\mu \cup \mu'_1)(\alpha_1) \ll t_1$, and (C) $\mu \cup \mu'_1 \models C_1$.

By induction hypothesis, (1) $\Gamma^\circ \vdash E_2 \Rightarrow \alpha_2 \ (C_2, V_2)$, (2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)$, and
(3) $\mu(\Gamma^o) \vdash E_2 : t_2$ imply there is $\mu'_2$ with (A) $\text{dom}(\mu'_2) = \text{newtyvar}(\Gamma^o, \alpha_2, C_2, V_2)$, (B) $(\mu \cup \mu'_2)(\alpha_2) \iff t_2$, and (C) $\mu \cup \mu'_2 \models C_2$.

The sets $\{\alpha\}$, $\text{dom}(\mu'_1)$, and $\text{dom}(\mu'_2)$ are disjoint, because of the conditions $\alpha \notin \text{tyvar}(\Gamma^o, C_1, C_2, V_1, V_2)$ and $\text{newtyvar}(\Gamma^o, \alpha_1, C_1, V_1) \cap \text{newtyvar}(\Gamma^o, \alpha_2, C_2, V_2) = \emptyset$.

Let $\mu' = \{\alpha \mapsto t\} \cup \mu'_1 \cup \mu'_2$, a well-formed finite map.

We have (A) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, C, V)$ because $\text{newtyvar}(\Gamma^o, \alpha, C, V) = \{\alpha\} \cup \text{newtyvar}(\Gamma^o, \alpha_1, C_1, V_1) \cup \text{newtyvar}(\Gamma^o, \alpha_2, C_2, V_2)$.

We have (B) $(\mu \cup \mu')(\alpha) \iff t$ because $\mu'(\alpha) = t$ by definition of $\mu'$.

We have (C) $\mu \cup \mu' \models C$ because:

- $\{\alpha \mapsto t\} \models \alpha \iff \text{Bool}$ since $t \iff \text{Bool}$
- $\mu \cup \mu' \models \alpha_1 \iff \text{Int}$ since $(\mu \cup \mu')(\alpha_1) \iff t_1$ and $t_1 \iff \text{Int}$
- $\mu \cup \mu' \models \alpha_2 \iff \text{Int}$ since $(\mu \cup \mu')(\alpha_2) \iff t_2$ and $t_2 \iff \text{Int}$
- $\mu \cup \mu' \models C_1$
- $\mu \cup \mu' \models C_2$

Hence $\mu'$ has properties (A), (B), and (C) as desired.

Case $E = E_1 == E_2$. In this case, our assumptions (1), (2) and (3) take the forms:

\[
\begin{array}{c}
\text{(ALGO ==)} \quad \alpha \notin \text{tyvar}(\Gamma^o, C_1, C_2, V_1, V_2) \text{ and } \text{newtyvar}(\Gamma^o, \alpha_1, C_1, V_1) \cap \text{newtyvar}(\Gamma^o, \alpha_2, C_2, V_2) = \emptyset \\
\Gamma^o \vdash E_1 \Rightarrow \alpha_1 \ (C_1, V_1) \quad \Gamma^o \vdash E_2 \Rightarrow \alpha_2 \ (C_2, V_2) \\
\Gamma^o \vdash E_1 == E_2 \Rightarrow \alpha \ (\alpha \iff \text{Bool} \land \bigvee_{\iota \in \{\text{Bool}, \text{Int}\}} (\alpha_1 \iff \iota \land \alpha_2 \iff \iota) \land C_1 \land C_2, V_1 \cup V_2) \\
\text{dom}(\mu) = \text{tyvar}(\Gamma^o) \\
\end{array}
\]

\[
\begin{array}{c}
\text{(DECL EXPR ==)} \quad (t_1 \iff \text{Bool} \text{ and } t_2 \iff \text{Bool}) \text{ or } (t_1 \iff \text{Int} \text{ and } t_2 \iff \text{Int}) \\
\mu(\Gamma^o) \vdash E_1 : t_1 \quad \mu(\Gamma^o) \vdash E_2 : t_2 \quad t \iff \text{Bool} \\
\mu(\Gamma^o) \vdash E_1 == E_2 : t \\
\end{array}
\]

We are to find $\mu'$ with (A) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^o, \alpha, C, V)$, (B) $(\mu \cup \mu')(\alpha) \iff t$, and (C) $\mu \cup \mu' \models C$, where $C = \alpha \iff \text{Bool} \land \bigvee_{\iota \in \{\text{Bool}, \text{Int}\}} (\alpha_1 \iff \iota \land \alpha_2 \iff \iota) \land C_1 \land C_2$.

By induction hypothesis, (1) $\Gamma^o \vdash E_1 \Rightarrow \alpha_1 \ (C_1, V_1)$, (2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^o)$, and (3) $\mu(\Gamma^o) \vdash E_1 : t_1$ imply there is $\mu'_1$ with (A) $\text{dom}(\mu'_1) = \text{newtyvar}(\Gamma^o, \alpha_1, C_1, V_1)$, (B) $(\mu \cup \mu'_1)(\alpha_1) \iff t_1$, and (C) $\mu \cup \mu'_1 \models C_1$.

By induction hypothesis, (1) $\Gamma^o \vdash E_2 \Rightarrow \alpha_2 \ (C_2, V_2)$, (2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^o)$, and (3) $\mu(\Gamma^o) \vdash E_2 : t_2$ imply there is $\mu'_2$ with (A) $\text{dom}(\mu'_2) = \text{newtyvar}(\Gamma^o, \alpha_2, C_2, V_2)$,
A.1. PROOFS FOR SOUNDNESS AND COMPLETENESS OF THE ALGORITHMIC
TYPING SYSTEM

(B) \((\mu \cup \mu'_2)(\alpha_2) \iff t_2\), and (C) \(\mu \cup \mu'_2 \models C_2\).

The sets \(\{\alpha\}\), \(\text{dom}(\mu'_1)\), and \(\text{dom}(\mu'_2)\) are disjoint, because of the conditions \(\alpha \notin \text{tyvar}(\Gamma^0, C_i, V_i)\) and \(\text{newtyvar}(\Gamma^0, \alpha_1, C_i, V_i) \cap \text{newtyvar}(\Gamma^0, \alpha_2, C_i, V_i) = \emptyset\).

Let \(\mu' = \{\alpha \mapsto t\} \cup \mu'_1 \cup \mu'_2\), a well-formed finite map.

We have (A) \(\text{dom}(\mu') = \text{newtyvar}(\Gamma^0, \alpha, C, V)\) because \(\text{newtyvar}(\Gamma^0, \alpha, C, V) = \{\alpha\} \cup \text{newtyvar}(\Gamma^0, \alpha_1, C_i, V_i) \cup \text{newtyvar}(\Gamma^0, \alpha_2, C_i, V_i)\).

We have (B) \((\mu \cup \mu')(\alpha) \iff t\) because \(\mu'(\alpha) = t\) by definition of \(\mu'\).

We have (C) \(\mu \cup \mu' \models C\) because:

- \(\{\alpha \mapsto t\} \models \alpha \iff \text{Bool}\) since \(t \iff \text{Bool}\)
- \((\mu \cup \mu') \models \alpha_1 \iff \text{Bool}\) and \(\mu \cup \mu' \models \alpha_2 \iff \text{Bool}\) or \((\mu \cup \mu') \models \alpha_1 \iff \text{Int}\) and \(\mu \cup \mu' \models \alpha_2 \iff \text{Int}\), since \((\mu \cup \mu')(\alpha_1) \iff t_1\) and \((\mu \cup \mu')(\alpha_2) \iff t_2\), and \(t_1 \iff \text{Bool}\) and \(t_2 \iff \text{Bool}\) or \((t_1 \iff \text{Int}\) and \(t_2 \iff \text{Int}\))
- \(\mu \cup \mu' \models C_1\)
- \(\mu \cup \mu' \models C_2\)

Hence \(\mu'\) has properties (A), (B), and (C) as desired.

Case Rcd. In this case, our assumptions (1), (2) and (3) take the forms:

\[
\frac{}{\text{(ALGO RCD)}} (\alpha \notin \bigcup_{i \in 1..n} \text{tyvar}(\Gamma^0, \alpha_i, C_i, V_i) \text{ and sets } \text{newtyvar}(\Gamma^0, \alpha_i, C_i, V_i) \text{ disjoint})
\]

\[
\frac{\Gamma^0 \vdash E_i \Rightarrow \alpha_i (C_i, V_i) \ \forall i \in 1..n}{\Gamma^0 \vdash \{\ell_i = E_i : i \in 1..n\} \Rightarrow \alpha} (\alpha \iff \{\ell_i : \alpha_i : i \in 1..n\} \land \bigwedge_{i \in 1..n} C_i \cup \bigcup_{i \in 1..n} V_i)
\]

\[
\text{dom}(\mu) = \text{tyvar}(\Gamma^0)
\]

\[
\frac{\mu(\Gamma^0) \vdash E_i : t_i \ \forall i \in 1..n \ t \iff \{\ell_i : t_i : i \in 1..n\}}{\mu(\Gamma^0) \vdash \{\ell_i = E_i : i \in 1..n\} : t}
\]

We are to find \(\mu'\) with (A) \(\text{dom}(\mu') = \text{newtyvar}(\Gamma^0, \alpha, C, V)\), (B) \(\mu(\Gamma^0) \vdash E_i : t_i \Rightarrow \alpha_i (C_i, V_i)\), (2) \(\mu(\Gamma^0) \vdash E_i : t_i \Rightarrow \alpha_i (C_i, V_i)\), (3) \(\mu(\Gamma^0) \vdash E_i : t_i \Rightarrow \alpha_i (C_i, V_i)\) imply there is \(\mu'_i\) with (A) \(\text{dom}(\mu'_i) = \text{newtyvar}(\Gamma^0, \alpha_i, C_i, V_i)\), (B) \(\mu(\Gamma^0) \vdash E_i : t_i \Rightarrow \alpha_i (C_i, V_i)\), (3) \(\mu(\Gamma^0) \vdash E_i : t_i \Rightarrow \alpha_i (C_i, V_i)\), and (C) \(\mu \cup \mu'_i \models C_i\).

By applying induction hypothesis \(n\) times, for all \(i \in 1..n\), (1) \(\Gamma^0 \vdash E_i \Rightarrow \alpha_i (C_i, V_i)\), (2) \(\text{dom}(\mu) = \text{tyvar}(\Gamma^0)\), and (3) \(\mu(\Gamma^0) \vdash E_i : t_i \Rightarrow \alpha_i (C_i, V_i)\) imply there is \(\mu'_i\) with (A) \(\text{dom}(\mu'_i) = \text{newtyvar}(\Gamma^0, \alpha_i, C_i, V_i)\), (B) \(\mu(\Gamma^0) \vdash E_i : t_i \Rightarrow \alpha_i (C_i, V_i)\), (3) \(\mu(\Gamma^0) \vdash E_i : t_i \Rightarrow \alpha_i (C_i, V_i)\), and (C) \(\mu \cup \mu'_i \models C_i\).

The sets \(\{\alpha\}\), and \(\text{dom}(\mu'_1)\), \ldots, and \(\text{dom}(\mu'_n)\) are disjoint, because of the conditions \(\alpha \notin \bigcup_{i \in 1..n} \text{tyvar}(\Gamma^0, \alpha_i, C_i, V_i)\) and sets \(\text{newtyvar}(\Gamma^0, \alpha_i, C_i, V_i)\) are disjoint.

Let \(\mu' = \{\alpha \mapsto t\} \cup \bigcup_{i \in 1..n} \mu'_i\), a well-formed finite map.
We have (A) \( \text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V) \) because \( \text{newtyvar}(\Gamma^\circ, \alpha, C, V) = \{\alpha\} \cup \bigcup_{i \in 1..n} \text{newtyvar}(\Gamma^\circ, \alpha_i, C_i, V_i) \).

We have (B) \( (\mu \cup \mu')(\alpha) <\: t \) because \( \mu'(\alpha) <\: t \) by definition of \( \mu' \).

We have (C) \( \mu \cup \mu' \models C \) because:

- \( \mu \cup \mu' \models \alpha <\: \{\ell_i : \alpha_i \in 1..n\} \) since for all \( i \in 1..n \), \( (\mu \cup \mu')(\alpha_i) <\: t_i \) and \( (\mu \cup \mu')(\alpha) <\: t \) and \( t <\: \{\ell_i : t_i \in 1..n\} \)
- \( \mu \cup \mu' \models C_1 \) and ... and \( \mu \cup \mu' \models C_n \).

Hence \( \mu' \) has properties (A), (B), and (C) as desired.

Case Proj. In this case, our assumptions (1), (2) and (3) take the forms:

\[
\begin{align*}
(\text{ALGO PROJ}) \quad (\alpha' \notin \text{tyvar}(\Gamma^\circ, \alpha, C, V)) \\
\Gamma^\circ \vdash E \Rightarrow \alpha (C, V) \\
\Gamma^\circ \vdash E.\ell \Rightarrow \alpha' (\alpha' <\: \alpha.\ell \land C, V) \\
\text{dom}(\mu) = \text{tyvar}(\Gamma^\circ)
\end{align*}
\]

\[
(\text{DECL EXPR PROJ}) \quad (t <\: \{\ell_i : t_i \in 1..n\}) \\
\mu(\Gamma^\circ) \vdash E : t \quad j \in 1..n \quad \ell = \ell_j \\
\mu(\Gamma^\circ) \vdash E.\ell : t_j
\]

We are to find \( \mu' \) with (A) \( \text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha', C, V) \), (B) \( (\mu \cup \mu')(\alpha') <\: t \), and (C) \( \mu \cup \mu' \models C' \), where \( C' = \alpha' <\: \alpha.\ell \land C \).

By induction hypothesis, (1) \( \Gamma^\circ \vdash E \Rightarrow \alpha (C, V) \), (2) \( \text{dom}(\mu) = \text{tyvar}(\Gamma^\circ) \), and (3) \( \mu(\Gamma^\circ) \vdash E : t \) imply there is \( \mu'' \) with (A) \( \text{dom}(\mu'') = \text{newtyvar}(\Gamma^\circ, \alpha, C, V) \), (B) \( (\mu \cup \mu'')(\alpha') <\: t \), and (C) \( \mu \cup \mu'' \models C \).

The sets \( \{\alpha'\} \), \( \text{dom}(\mu'') \) are disjoint, because \( \alpha' \notin \text{tyvar}(\Gamma^\circ, \alpha, C, V) \).

Let \( \mu' = \{\alpha' \mapsto t_j\} \cup \mu'' \), a well-formed finite map.

We have (A) \( \text{dom}(\mu') = \text{newtyvar}(\Gamma^\circ, \alpha', C, V) \) because \( \text{newtyvar}(\Gamma^\circ, \alpha', C, V) = \{\alpha'\} \cup \text{newtyvar}(\Gamma^\circ, \alpha, C, V) \).

We have (B) \( (\mu \cup \mu')(\alpha') = t_j \) because \( \mu'(\alpha') <\: t_j \) by definition of \( \mu' \).

We have (C) \( \mu \cup \mu' \models C' \) because:

- \( \mu \cup \mu' \models \alpha' <\: \alpha.\ell \) because \( (\mu \cup \mu')(\alpha) <\: t \) and \( t <\: \{\ell_i : t_i \in 1..n\} \) and \( (\mu \cup \mu')(\alpha') <\: t_j \) by Lemma 4.1(f).
- \( \mu \cup \mu'' \models C \)
A.1. PROOFS FOR SOUNDNESS AND COMPLETENESS OF THE ALGORITHMIC TYPING SYSTEM

Hence $\mu'$ has properties (A), (B), and (C) as desired.

Case If. In this case, our assumptions (1), (2) and (3) take the forms:

\[(\text{ALGO If}) \text{ (sets } \text{newtyvar}(\Gamma^0, \alpha_i, C_i, V_i) \text{ disjoint)}\]

\[
\Gamma^0 \vdash E_1 \Rightarrow \alpha_1 (C_1, V_1) \quad \Gamma^0 \vdash E_2 \Rightarrow \alpha_2 (C_2, V_2) \quad \Gamma^0 \vdash E_3 \Rightarrow \alpha_3 (C_3, V_3)
\]

\[
\Gamma^0 \vdash (E_1 ? E_2 : E_3) \Rightarrow \alpha_2 (\alpha_1 <: \alpha_2) \wedge \alpha_1 \wedge \bigwedge_{i \in 1..3} C_i \cup \bigcup_{i \in 1..3} V_i
\]

\[
\text{dom}(\mu) = \text{tyvar}(\Gamma^0)
\]

\[(\text{DECL EXPR IF}) (t <: \alpha)\]

\[
\mu(\Gamma^0) \vdash E_1 : t' \quad \mu(\Gamma^0) \vdash E_2 : t \quad \mu(\Gamma^0) \vdash E_3 : t
\]

We are to find $\mu'$ with (A) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^0, \alpha_2, C, V)$, (B) $(\mu \cup \mu')(\alpha_2) <: t$, and (C) $\mu \cup \mu' = C$, where $C = \alpha_1 <: \alpha_2 <: \alpha_3 \wedge \bigwedge_{i \in 1..3} C_i$ and $V = \bigcup_{i \in 1..3} V_i$.

Let $t_1 = t'$, $t_2 = t$, and $t_3 = t$. By applying induction hypothesis 3 times, for all $i \in 1..3$, (1) $\Gamma^0 \vdash E_i \Rightarrow \alpha_i (C_i, V_i)$, (2) $\text{dom}(\mu) = \text{tyvar}(\Gamma^0)$, and (3) $\mu(\Gamma^0) \vdash E_i : t_i$ imply there is $\mu'_i$ with (A) $\text{dom}(\mu'_i) = \text{newtyvar}(\Gamma^0, \alpha_i, C_i, V_i)$, (B) $(\mu \cup \mu'_i)(\alpha_i) <: t_i$, and (C) $\mu \cup \mu'_i = C_i$.

The sets $\text{dom}(\mu'_1), \ldots, \text{dom}(\mu'_3)$ are disjoint, because the sets $\text{newtyvar}(\Gamma^0, \alpha_i, C_i, V_i)$ are disjoint.

Let $\mu' = \bigcup_{i \in 1..3} \mu'_i$, a well-formed finite map.

We have (A) $\text{dom}(\mu') = \text{newtyvar}(\Gamma^0, \alpha_2, C, V)$ because $\text{newtyvar}(\Gamma^0, \alpha_2, C, V) = \bigcup_{i \in 1..3} \text{newtyvar}(\Gamma^0, \alpha_i, C_i, V_i)$.

We have (B) $(\mu \cup \mu')(\alpha_2) <: t$ because $\mu'_2(\alpha_2) <: t_2 <: t$, and hence $\mu'(\alpha_2) <: t$ by definition of $\mu'$.

We have (C) $\mu \cup \mu' = C$ because

- $\mu \cup \mu' = \alpha_1 <: \alpha_1$, since $t' <: \alpha_2$ and $(\mu \cup \mu')(\alpha_1) <: t'$, the latter because $\mu'_1(\alpha_1) <: t_1 <: t'$, and hence $\mu'(\alpha_1) <: t'$ by definition of $\mu'$.
- $\mu \cup \mu' = \alpha_2 <: \alpha_3$, since $(\mu \cup \mu')(\alpha_2) <: t_2$ and $(\mu \cup \mu')(\alpha_3) = t_3$, and by definition $t_2 <: t_3 <: t$.
- $\mu \cup \mu' = C_1$ and $\mu \cup \mu' = C_2$ and $\mu \cup \mu' = C_3$.

Hence $\mu'$ has properties (A), (B), and (C) as desired.  

\[\square\]
Bibliography


http://www.jmlr.org/papers/v12/duchi11a.html

https://doi.org/10.24963/ijcai.2018/792


https://doi.org/10.1109/ICSE.2017.75

https://doi.org/10.1109/TSE.2017.2755013


https://github.com

https://github.com/JoshuaKGoldberg/TypeStat


https://doi.org/10.1145/3211346.3211355

https://doi.org/10.1007/978-3-642-24690-6_28


Milod Kazerounian, Jeffrey S. Foster, and Bonan Min. 2021. SimTyper: Sound Type Inference for Ruby using Type Equality Prediction. In ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). Chicago, IL, USA.


Microsoft. 2020. TypeScript. https://github.com/microsoft/TypeScript


Tim Rocktäschel, Sameer Singh, and Sebastian Riedel. 2015. Injecting Logical Background Knowledge into Embeddings for Relation Extraction. In Conference of the North American Chapter of the Association for


