Type Inference as Optimization

Anonymous Author(s)
Affiliation
Address
email

Abstract

Optionally typed dynamic languages can permit multiple valid type assignments. When this happens, developers can prefer one valid type assignment over another because it better reflects how they think about the program and the problem it solves. Natural type inference (NTI) uses natural language text within source code, such as identifiers, to help choose valid programming language types. A growing body of techniques has been proposed for NTI. These techniques predict types; they seek to return natural type assignments (assignments that reflect developer preferences) while striving for correctness. They are empirically effective, but they are not sound by construction: they do not leverage programming language theory to formalize their algorithms and show correctness and termination. Filling this foundational gap is the purpose of this paper. We are the first to present a detailed algorithm for NTI that is validated with theorems and proofs. Valid type assignments obey logical constraints arising from type rules; natural type assignments obey natural constraints arising from the natural language text associated with a variable and its uses. The core intuition of this work is that logical and natural constraints can interact to speed finding a type valuation that 1. type checks (satisfies the logical constraints) and 2. is most natural. We formulate NTI as a joint optimization problem. To do this, we define a numerical relaxation over boolean logical constraints that give us a condition that we treat as a hard constraint, while simultaneously we minimize distance from natural constraints, which we treat as soft constraints for our optimization problem. Our main result, the first formal proof of soundness for natural type inference, is that our algorithm always terminates, either with an error or with a tuple that is guaranteed to be a type signature for its input.

1 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, naming a variable $username$ or $password$, gives different connotation to the developer, although both should have all the attributes of a type `string`. 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 an untyped language (such as Javascript or Python). For a canonical example, 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:

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

Although E is untyped (like JavaScript), we assume 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

Submitted to 35th Conference on Neural Information Processing Systems (aiplans 2021). Do not distribute.

typing expressions. Given this input, *type inference* is the task of computing a type signature for f, that is, a tuple $(t_1, \ldots, t_n, t)$, 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 an inference algorithm 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.

By now, there are several systems of natural type inference that resolve the ambiguity by relying on the textual cues in the natural language already present in source code. From these textual cues, we can learn natural constraints, which capture an affinity (or aversion) between identifiers based on how they are used. They are manifestations of Firth's famous observation — "You shall know a word by the company it keeps." — in source code Firth [1957]. In the context of type inference, we formalize them as distributions over types. Natural type inference approaches learn to predict types from their training data, which is some representation of source code. Most approaches, like the pioneering work JSNice Raychev et al. [2015] or the first neural approach DeepTyper Hellendoorn et al. [2018], learn both logical and natural constraints. This uniform treatment puts logical and natural constraints on the same level and is inherently unsound. LambdaNet Wei et al. [2020] explicitly models some logical constraints in its graph neural network. Despite biasing its architecture toward logical constraints, it sometimes fails to learn them.

The exceptions are TypeWriter Pradel et al. [2019] and OptTyper Pandi et al. [2020], which obey logical constraints. As usual, TypeWriter's learns both logical and natural constraints, ranks its type predictions, then relies on an external, optional type checker to filter them. OptTyper explicitly separates logical and natural constraints. It infers logical constraints and learns natural constraints, then combines them in a joint optimization, but does not prove the result to be sound.

Despite their success in utilizing natural language cues extracted from code corpora to resolve typing ambiguities, an important problem with these previous works on natural type inference is the absence of any programming language theory to formalize the algorithms and to show correctness. Filling this foundational gap is the purpose of this paper.

1.1 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.

$$\begin{array}{l}
\textbf{function}\ \mathit{uppercase}(\mathit{str})\ \textbf{return}\ \text{// 1st} \\
\quad \{\mathit{length} = \mathit{str}.\mathit{length}, \\
\quad\ \mathit{items} = \lambda(i)\ \textbf{let}\ \mathit{char} = \mathit{str}.\mathit{items}(i)\ \textbf{in}\ \mathit{char} < 97\,?\,\mathit{char} : \mathit{char} < 123\,?\,\mathit{char} - 32 : \mathit{char}\} \\
\textbf{function}\ \mathit{diffRange}(\mathit{range1}, \mathit{range2})\ \textbf{return}\ \mathit{range1}.\mathit{length} - \mathit{range2}.\mathit{length}\ \ \text{// 2nd} \\
\textbf{function}\ \mathit{intEqual3}(\mathit{int1}, \mathit{int2}, \mathit{int3})\ \textbf{return}\ \mathit{int1} == \mathit{int2}\,?\,\mathit{int2} == \mathit{int3} : \textbf{false}\ \text{// 3rd}
\end{array}$$

Consider the following library of type definitions.

$$\begin{array}{l}
\textbf{type}\ \mathit{Char} = \mathit{Int} \\
\textbf{type}\ \mathit{String} = \{\mathit{length} : \mathit{Int}, \mathit{items} : \mathit{Int} \to \mathit{Char}\} \\
\textbf{type}\ \mathit{IntArray} = \{\mathit{length} : \mathit{Int}, \mathit{items} : \mathit{Int} \to \mathit{Int}\} \\
\textbf{type}\ \mathit{Range} = \{\mathit{length} : \mathit{Int}, \mathit{breadth} : \mathit{Int}\}
\end{array}$$

We work in a λ-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 $\mathit{IntArray}$ are syntactically different (because they are different type names), they are structurally equivalent (because they have the same structure, if we ignore the names). This is not a purely academic concern: it arises naturally in TypeScript. For example, when given 2nd example, TypeScript infers the uninformative intersection `string | any []`' for both `range1` and `range2` and, when given the 3rd example, TypeScript infers `any` for each parameter, which is utterly uninformative.

Given such a type library, 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, $(String, String)$ versus $(IntArray, IntArray)$.

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

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

1.2 Natural Type Inference by Solving Logical and Natural Constraints

This paper presents a new formal algorithm and theory for natural type inference. To do so we are using two types of constraints. The *logical constraints*, represent the constraints classically produced by type inference algorithms. The *natural constraints* which are the output of a learning procedure and refer generically to indirect, statistical constraints about types. In this work we consider a specific kind of natural constraints, which maps an identifier name to a probability vector. The key idea of the algorithm is to extract both logical and natural constraints from the input program, and to consider natural type inference as an optimization problem.

During optimization, we frame logical constraints as hard constraints that must be satisfied, while the natural constraints are soft constraints that ought to be satisfied. To enforce this, we introduce a *natural distance* that measures the degree to which the natural constraints are satisfied, and then we optimize this function, subject to the constraint that the logical constraints are satisfied. This optimization problem optimizes a continuous function of type assignments, subject to a logical constraint. This is a difficult optimization problem. In this paper we analyze an optimization approach based on a *numerical relaxation* that converts this combinatorial optimization problem to an unconstrained continuous optimization problem. This formulation allows us to use off the shelf algorithms to effectively solve our problem.

Finally, the output is a type signature that is provably sound because the logical constraints are satisfied, which implies that the input program is well typed.

For example, given $diffRange$ (see above) as input, what is the type signature (t_1, t_2, t) to output?

Our type inference algorithm generates the following logical and natural constraints.

- The logical constraints, after simplification, are that (t_1, t_2, t) is a type signature for $diffRange$ if and only if (1) $t_1 = String$ or $t_1 = IntArray$ or $t_1 = Range$, (2) $t_2 = String$ or $t_2 = IntArray$ or $t_2 = Range$, and (3) $t = Int$.
- A natural constraint is a mapping from each identifier $\{range1, range2, diffRange\}$ to a probability vector over possible types.

In the context of the library types $Char$, $String$, $IntArray$, and $Range$, and the builtin types Int and $Bool$, the natural constraints $(range1, t_1)$ and $(range2, t_2)$ bias the choice of both t_1 and t_2 to be $Range$, consistent with logical constraints (1) and (2). The natural constraint $(diffRange, t)$ biases the choice of t to be $Range$, but this violates the logical constraint (3), which takes priority.

In all, we get that:

- The algorithm assigns $diffRange$ the type signature $(Range, Range, Int)$.

Similarly, on our other examples we get:

- The algorithm assigns $uppercase$ the type signature $(String, String)$.
- The algorithm assigns $intEqual3$ the type signature $(Int, Int, Int, Bool)$.

1.3 Contributions: Formal Foundations for Natural Type Inference

As the setting for our study, we define 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.

Our contributions are as follows:

- We present a new algorithmic type system that given an expression yields logical and natural constraints. The algorithm is terminating and the logical constraints are sound and complete with respect to the declarative type system. Our overall task is finding a type signature for an untyped function definition, is equivalent to satisfying the logical constraint extracted from the function definition.
- We show how to combine a numerical relaxation of the logical constraints with probability distributions over the library of types to form a joint optimization problem. Firstly, we show how to relate the logical semantics and its relaxations. And then we present our key theorem where we shod that the optimizer is guaranteed to terminate with the optimal solution to the natural constraints that satisfies the logical constraints.
- We describe an overall algorithm for natural type inference, building on the algorithmic type system and the constraint satisfaction algorithm. By the correctness theorem, the algorithm always terminates, either with an error, or with a tuple that is guaranteed to be a type signature for its input.

This work is the first to formalize and prove termination and soundness for a natural type inference algorithm. Our specific algorithm deals with ambiguities arising from overloading, dot-notation, and structural equality of type names. It provides formal foundations for OptTyper Pandi et al. [2020] and shows that the resulting type signatures are sound. Some of our definitions, including logical and natural constraints, the continuous relaxations, and the core optimization problem are based on Pandi et al. [2020], but all the theorems of this paper are new, as is the formulation of an algorithm for type-checking function definitions in a typed λ-calculus.

As our literature review makes clear, all work in learning-based type inference to date focuses on formalising 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.

Still, there are limitations that can be addressed in future work. 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.

References

Miltiadis Allamanis, Hao Peng, and Charles Sutton. A Convolutional Attention Network for Extreme Summarization of Source Code. In *International Conference in Machine Learning (ICML)*, 2016.

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81:1–81:37, July 2018. doi: 10.1145/3212695.

Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. *ArXiv*, abs/2004.10657, 2020.

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4): 575–631, September 1993. ISSN 0164-0925. doi: 10.1145/155183.155231. URL https://doi.org/10.1145/155183.155231.

Charles Arthur. Tech giants may be huge, but nothing matches big data. *The Guardian*, 2013. URL https://www.theguardian.com/technology/2013/aug/23/tech-giants-data.

M. Baaz. Infinite-valued Gödel logics with 0-1-projections and relativizations. In P. Hájek, editor, *Proc. Gödel'96, Logic Foundations of Mathematics, Computer Science and Physics – Kurt Gödel's Legacy*, Lecture Notes in Logic 6, pages 23–33, Brno, Czech Republic, 1996. Springer.

Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. Hinge-loss markov random fields and probabilistic soft logic. *Journal of Machine Learning Research*, 18(109):1–67, 2017. URL http://jmlr.org/papers/v18/15-631.html.

Dimitri Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Athena Scientific, 1982. ISBN 978-0-12-093480-5. doi: 10.1016/B978-0-12-093480-5.50001-5.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. doi: 10.1017/CBO9780511804441.

Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Informaticae*, 33(4):309–338, 1998. doi: 10.3233/FI-1998-33401. URL https://doi.org/10.3233/FI-1998-33401.

Stephen Cook and David Mitchell. Finding hard instances of the satisfiability problem: A survey. 35, 01 2000.

Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data (extended abstract). In *Journal of Artificial Intelligence Research*, pages 5598–5602, 07 2018. doi: 10.24963/ijcai.2018/792.

J. Firth. A synopsis of linguistic theory 1930-1955. In *Studies in Linguistic Analysis*. Philological Society, Oxford, 1957. reprinted in Palmer, F. (ed. 1968) Selected Papers of J. R. Firth, Longman, Harlow.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning (ICML)*, April 2017.

Andrew D. Gordon. A tutorial on co-induction and functional programming. In *Proceedings of the 1994 Glasgow Workshop on Functional Programming, Ayr, Scotland*, pages 78–95. Springer London, September 1994. ISBN 978-3-540-19914-4 (Print) 978-1-4471-3573-9 (Online). URL https://www.microsoft.com/en-us/research/publication/a-tutorial-on-co-induction-and-functional-programming/.

Petr Hájek. *Metamathematics of fuzzy logic*. Trends in logic ; v. 4. Kluwer, Dordrecht ; London, 1998. ISBN 0792352386. doi: 10.1007/978-94-011-5300-3.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 152–162, New York, NY, USA, 2018. ACM. doi: 10.1145/3236024.3236051.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

Stanisław Jaśkowski. Three contributions to the two-valued propositional calculus. *Studia Logica*, 34(1):121–132, March 1975. ISSN 1572-8730. doi: 10.1007/BF02314428. URL https://doi.org/10.1007/BF02314428.

Angelika Kimmig, Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A short introduction to probabilistic soft logic. In *NIPS 2012*, 2012.

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: Inferring javascript function types from natural language information. In *International Conference on Software Engineering*, pages 304–315, Piscataway, NJ, USA, 2019. IEEE. doi: 10.1109/ICSE.2019.00045.

Microsoft. TypeScript, 2020. URL https://github.com/microsoft/TypeScript.

Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py: Deep similarity learning-based type inference for python, 2021.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract dpll and abstract dpll modulo theories. pages 36–50, 01 2004. ISBN 978-3-540-25236-8. doi: 10.1007/978-3-540-32275-7-3.

Dominic A. Orchard, Andrew C. Rice, and Oleg Oshmyan. Evolving fortran types with inferred units-of-measure. *J. Comput. Sci.*, 9:156–162, 2015.

Eirene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. Opttyper: Probabilistic type inference by optimising logical and natural constraints. *CoRR*, abs/2004.00348, 2020.

Marco Patrignani, Eric Mark Martin, and Dominique Devriese. On the semantic expressiveness of recursive types. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434302. URL https://doi.org/10.1145/3434302.

Benjamin Pierce. *Types and Programming Languages*. MIT Press, London, 2002.

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. Typewriter: Neural type prediction with search-based validation. *ArXiv*, abs/1912.03768, 2019. To appear at ESEC/FSE 2020.

Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 111–124, New York, NY, USA, 2015. ACM. doi: 10.1145/2676726.2677009.

Tim Rocktäschel, Sameer Singh, and Sebastian Riedel. Injecting logical background knowledge into embeddings for relation extraction. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1119–1129, Denver, Colorado, May 2015. Association for Computational Linguistics. doi: 10.3115/v1/N15-1118.

Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. Cln2inv: Learning loop invariants with continuous logic networks. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HJlfuTEtvB.

Charles Sutton, Andrew McCallum, et al. An introduction to conditional random fields. *Foundations and Trends® in Machine Learning*, 4(4):267–373, 2012.

The-Mypy-Project. Mypy lang, 2014. URL http://mypy-lang.org/. http://mypy-lang.org/.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015. URL http://papers.nips.cc/paper/5866-pointer-networks.pdf.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. LambdaNet: Probabilistic type inference using graph neural networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=Hkx6hANtwH.

Jack Williams, Carina Negreanu, Andrew D. Gordon, and Advait Sarkar. Understanding and inferring units in spreadsheets. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, July 2020. URL https://www.microsoft.com/en-us/research/publication/understanding-and-inferring-units-in-spreadsheets/.

Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 607–618, New York, NY, USA, 2016. ACM. doi: 10.1145/2950290.2950343.

Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. pages 106–120, 06 2020. doi: 10.1145/3385412.3385986.

This appendix details the sound, probabilistic type inference outlined in the submission. Appendix A presents the syntax and typing rules of a core calculus . In Appendix B we show how these typing rules correspond to an algorithmic type system that gathers logical constraints, which we need to satisfy. Appendix C we show how we transform a type inference problem to a joint optimization that allows us to combine classic type rules and learning in a sound way that guarantees type correctness. First we show present a relaxation of the logical constraints and then we present our main theorem Theorem 4 which guarantees that the optimizer will terminate with a solution that satisfies the logical constraints. Appendix D presents our overall algorithm based on the algorithmic type system. Finally Appendix E discuss the related work in this field.

A A Type System for a Core Language with Named Types

We assemble a typed λ-calculus to formalize and verify natural type inference, and in particular the essence of the OptTyper algorithm Pandi et al. [2020]. Given a function definition, OptTyper infers types for its arguments and result, where each type is the name of a type definition from a given TypeScript library.

In our formalism, a *type* t is simply a name drawn from a finite set of type names (such as Int or $IntArray$). Each type name t is defined by a *type equation* $\mathbf{type}\ t = S$ where S is a *type structure*, either a base, record, or function type. For instance, the type equation $\mathbf{type}\ Range = \{length : Int, breadth : Int\}$ defines the type $Range$ to equal the type structure $\{length : Int, breadth : Int\}$.

The core judgment of our type system determines when a expression E may be assigned a type name t. Unusually for a λ-calculus, our formalism assigns type names t but not type structures S to expressions because of our aim to formalize OptTyper, which only infers type names and not type structures. 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.

OptTyper resolves ambiguities arising from dot-notation, overloading of arithmetic operators, and type aliases for structurally equivalent types. Our formalism represents each of these ambiguities.

A.1 Types, Type Structures, and Type Equality

Here are the formal definitions of types t and type structures S.

Definition 1 (Types)**.**

t, u	*type (drawn from a finite set of names, including Bool and Int)*
$\iota ::= Bool \mid Int$	*base type*
ℓ	*label for field in a record*
$S, T ::=$	*type structure*
$\quad \iota$	*base type*
$\quad \{\ell_i : t_i\ ^{i \in 1..n}\}$	*record type*
$\quad t_1 \to t_2$	*function type*

Each type t has a unique type equation: $\mathbf{type}\ t = S$
In particular, each base type ι has the type equation: $\mathbf{type}\ \iota = \iota$

Our core syntax does not allow for a type alias $\mathbf{type}\ t_1 = t_2$, and nor does it allow nested type structures such as $S_1 \to S_2$ or $\{\ell_i : S_i\ ^{i \in 1..n}\}$. However, we can interpret these as shorthands for the core syntax. Given the definition $\mathbf{type}\ t_2 = S$ for t_2, we can interpret the alias $\mathbf{type}\ t_1 = t_2$ as being short for $\mathbf{type}\ t_1 = S$. We can interpret $\mathbf{type}\ t = S_1 \to S_2$ as meaning $\mathbf{type}\ t = t_1 \to t_2$ where t_1 and t_2 are fresh type names defined by $\mathbf{type}\ t_1 = S_1$ and $\mathbf{type}\ t_2 = S_2$. Hence, we consider the notation $\mathbf{type}\ t_f = t_1 \to \cdots \to t_n \to t$ for a curried type definition as implicitly introducing intermediate types with fresh names. Similarly, we can interpret $\mathbf{type}\ t = \{\ell_i : S_i\ ^{i \in 1..n}\}$ by introducing fresh names t_i each defined to be S_i.

For example, our type library from Section 1 uses these shorthands, and omits the definitions of the base types $Bool$ and Int. Here is the library in the core syntax, where we have introduced the

$Int2Char$ and $Int2Int$ as names for the function types used in the example.

$$\begin{array}{l}
\textbf{type}\ Bool\ =\ Bool \\
\textbf{type}\ Int\ =\ Int \\
\textbf{type}\ Char\ =\ Int \\
\textbf{type}\ Int2Char\ =\ Int \rightarrow Char \\
\textbf{type}\ Int2Int\ =\ Int \rightarrow Int \\
\textbf{type}\ String\ =\ \{length : Int,\ items : Int2Char\} \\
\textbf{type}\ IntArray\ =\ \{length : Int,\ items : Int2Int\} \\
\textbf{type}\ Range\ =\ \{length : Int,\ breadth : Int\}
\end{array}$$

Amadio *et al.* define *recursive types* as types defined by recursive type equations Amadio and Cardelli [1993]. Here is an example from their paper, the recursive type $Cell$ of an integer-containing memory cell:

$$\begin{array}{l}
\textbf{type}\ Unit\ =\ \{\} \\
\textbf{type}\ Read\ =\ Unit \rightarrow Int \\
\textbf{type}\ Write\ =\ Int \rightarrow Cell \\
\textbf{type}\ Add\ =\ Cell \rightarrow Cell \\
\textbf{type}\ Cell\ =\ \{read : Read, write : Write, add : Add\}
\end{array}$$

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 2 (Simulation, Bisimulation, and Type Equality)**.**

- *A binary relation on types $\mathcal{R}$ is a* simulation *if and only if:*
  (1) *whenever $t\ \mathcal{R}\ t'$ and $\textbf{type}\ t'\ =\ \iota$, then $\textbf{type}\ t\ =\ \iota$;*
  (2) *whenever $t\ \mathcal{R}\ t'$ and $\textbf{type}\ t'\ =\ \{\ell_i : t'_i{}^{\ i \in 1..n}\}$, there are t_i with $\textbf{type}\ t\ =\ \{\ell_i : t_i{}^{i \in 1..n}\}$ and $t_i\ \mathcal{R}\ t'_i$ for each $i \in 1..n$;*
  (3) *whenever $t\ \mathcal{R}\ t'$ and $\textbf{type}\ t'\ =\ t'_1 \rightarrow t'_2$ there are t_1, t_2 such that $\textbf{type}\ t\ =\ t_1 \rightarrow t_2$ and $t'_1\ \mathcal{R}\ t_1$ and $t_2\ \mathcal{R}\ t'_2$.*
- *A relation $\mathcal{R}$ is a* bisimulation *if and only if both $\mathcal{R}$ and its converse $\mathcal{R}^{-1}$ are simulations.*
- *The* type equality *relation $<:>$ is the union of all bisimulations.*

The following holds by standard constructions Milner [1989].

Lemma 1. *Type equality is reflexive, symmetric, and transitive, and is the largest bisimulation.*

Proposition 1. *Type equality $t <:> t'$ is decidable.*

Proof. As a corollary of Lemma 1, it follows that $t <:> t'$ if and only if there is a bisimulation $\mathcal{R}$ such that $t\ \mathcal{R}\ t'$. 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'). □

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 Section 1 partition into five equivalence classes:

- $\{Bool\}$
- $\{Int, Char\}$
- $\{Int2Char, Int2Int\}$

- $\{String, IntArray\}$
- $\{Range\}$

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

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

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

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 λ-calculi with recursive types use the notation $\mu t.S$ for the recursive type defined by the equation $\mathbf{type}\ t = S$. Patrignani *et al.* include a comprehensive survey Patrignani et al. [2021]. We do not use the $\mu t.S$ notation because our goal is an explicit formalism of named types defined by type equations; although recursive types are not needed for the motivating examples in Section 1, they arise naturally from our formalism and in practice.

A.2 Expressions and Values

We have a set of *variables* used both for values and for functions. It is a single set, but we conventionally use the metavariable x for variables used as values, and metavariable f for variables used as functions. Variables can occur either free or bound in an expression.

Separately, we have a set of *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 (Syntax of Expressions and Values)**.**

x, y, z	*(value) variable*
ℓ	*record label*
$E ::=$	*expression*
$\quad x$	*variable*
$\quad b$	*Boolean literal ($b \in \{\mathbf{true}, \mathbf{false}\}$)*
$\quad c$	*integer literal ($c \in \mathbb{Z}$)*
$\quad E_1 \oplus E_2 \qquad \oplus \in \{-, <, ==\}$	*selection of binary operators*
$\quad \{\ell_i = E_i\ ^{i \in 1..n}\}$	*record ($n \geq 0$)*
$\quad E.\ell$	*projection*
$\quad E_1\ ?\ E_2\ :\ E_3$	*conditional expression*
$\quad \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2$	*let-expression*
$\quad \lambda(x)E$	*lambda abstraction*
$\quad E_1\ (E_2)$	*application*
$V ::=$	*value*
$\quad b \mid c \mid \{\ell_i = V_i\ ^{i \in 1..n}\} \mid \lambda(x)E$	

Two of the expression forms are variable binders. In the binder $\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2$, the variable x is bound, with scope E_2. In the binder $\lambda(x)E$, the variable x is bound, with scope E. Let $fv(E)$ be the set of variables occurring free in expression E. Let a binder be *shadowed by an inner scope* in two cases: (1) it is $\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2$ 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 *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. Intuitively, no binder within a well-scoped expression E re-defines either a variable free in or bound within E. Our type system rejects inputs that are not well-scoped, a minor limitation because every expression has a well-scoped alpha-variant obtained by renaming bound variables.

A.3 Declarative Type System

The core judgment, $\Gamma \vdash E : t$, means that in environment Γ the expression E has the type t.

An *environment* Γ 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 $\varnothing$ for the empty environment. The purpose of

an environment is to assign types to variables. The *domain* $\mathrm{dom}(\Gamma)$ of an environment Γ is the set of variables it assigns. Let $\mathrm{dom}(x_1 : t_1, \ldots, x_n : t_n) = \{x_1, \ldots, x_n\}$ and $\mathrm{dom}(\varnothing) = \varnothing$. In what follows, we use as a convention that if Γ contains $x : t$ we write $\Gamma(x) = t$.

Definition 4 (Declarative Typing Rules with Retyping)**.**

(Expr Retype)
$$\frac{\Gamma \vdash E : t \quad t <:> t'}{\Gamma \vdash E : t'}$$

(Expr x)
$$\frac{x \in \mathrm{dom}(\Gamma) \quad \Gamma(x) = t}{\Gamma \vdash x : t}$$

(Expr b)
$$\frac{b \in \{\mathbf{true}, \mathbf{false}\}}{\Gamma \vdash b : \mathit{Bool}}$$

(Expr c)
$$\frac{\mathit{integer}\ c}{\Gamma \vdash c : \mathit{Int}}$$

(Expr −)
$$\frac{\Gamma \vdash E_1 : \mathit{Int} \quad \Gamma \vdash E_2 : \mathit{Int}}{\Gamma \vdash E_1 \ - \ E_2 : \mathit{Int}}$$

(Expr <)
$$\frac{\Gamma \vdash E_1 : \mathit{Int} \quad \Gamma \vdash E_2 : \mathit{Int}}{\Gamma \vdash E_1 \ < \ E_2 : \mathit{Bool}}$$

(Expr ==) $(t \in \{\mathit{Bool}, \mathit{Int}\})$
$$\frac{\Gamma \vdash E_1 : t \quad \Gamma \vdash E_2 : t}{\Gamma \vdash E_1 \ == \ E_2 : \mathit{Bool}}$$

(Expr Rcd)
$$\frac{\Gamma \vdash E_i : t_i \quad \forall i \in 1..n \quad \mathbf{type}\ t \ = \ \{\ell_i : t_i\ ^{i \in 1..n}\}}{\Gamma \vdash \{\ell_i = E_i\ ^{i \in 1..n}\} : t}$$

(Expr Proj)
$$\frac{\Gamma \vdash E : t \quad j \in 1..n \quad \mathbf{type}\ t \ = \ \{\ell_i : t_i\ ^{i \in 1..n}\}}{\Gamma \vdash E.\ell_j : t_j}$$

(Expr If)
$$\frac{\Gamma \vdash E_1 : \mathit{Bool} \quad \Gamma \vdash E_2 : t \quad \Gamma \vdash E_3 : t}{\Gamma \vdash (E_1 \,?\, E_2 : E_3) : t}$$

(Expr Let) $(x \notin \mathrm{dom}(\Gamma))$
$$\frac{\Gamma \vdash E_1 : t_1 \quad \Gamma, x : t_1 \vdash E_2 : t_2}{\Gamma \vdash \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2 : t_2}$$

(Expr Lambda) $(x \notin \mathrm{dom}(\Gamma))$
$$\frac{\Gamma, x : t_1 \vdash E : t_2 \quad \mathbf{type}\ t \ = \ t_1 \to t_2}{\Gamma \vdash \lambda(x)E : t}$$

(Expr Appl)
$$\frac{\Gamma \vdash E_2 : t \quad \Gamma \vdash E_1 : t_1 \quad \mathbf{type}\ t \ = \ t_1 \to t_2}{\Gamma \vdash E_2(E_1) : t_2}$$

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

Lemma 2. *If* $\Gamma \vdash E : t_1$ *and* $\Gamma \vdash E : t_2$ *then* $t_1 <:> t_2$.

Definition 20 in Appendix A consists of an alternative set of type rules for deriving judgments of the form $\Gamma \vdash E : t$. The 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 Appendix B.1. The relations inductively defined by the rules in Definition 4 and Definition 20 are one and the same (Proposition 2). We introduce the syntax-directed rules because they make some proofs easier, but we present the rules with (Expr Retype) as primary because they do not need the auxiliary relation.

A.4 Operational Semantics, Preservation, and Progress

We define a standard small-step call-by-value *reduction relation*, $E \to E'$, meaning that expression E evolves in one step to expression E'. The relation is the least closed under the rules in Definition 21 in Appendix B. That appendix also includes standard preservation and progress theorems.

- If $\Gamma \vdash E : t$ and $E \to E'$ then $\Gamma \vdash E' : t$.
- If $\varnothing \vdash E : t$ either (1) there is a value V such that $E = V$, or (2) there is E' such that $E \to E'$.

These results amount to the type safety properties guaranteed by our type system.

A.5 The Type Inference Problem for Function Definitions

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

Consider a top-level untyped function definition with the following syntax:

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

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

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

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

We want to show here that our three running examples all exhibit ambiguities that we will resolve using natural information. For example, we can derive the following:

$$\begin{array}{l} \mathit{str} : \mathit{String} \vdash E_{\mathit{uppercase}} : \mathit{String} \\ \mathit{range1} : \mathit{Range}, \mathit{range2} : \mathit{Range} \vdash E_{\mathit{diffRange}} : \mathit{Range} \\ \mathit{int1} : \mathit{Int}, \mathit{int2} : \mathit{Int}, \mathit{int3} : \mathit{Int} \vdash E_{\mathit{intEqual3}} : \mathit{Bool} \end{array}$$

where:

$$\begin{array}{lcl} E_{\mathit{uppercase}} & \triangleq & \{\mathit{length} = \mathit{str}.\mathit{length}, \\ & & \quad \mathit{items} = \lambda(i)\ \mathbf{let}\ \mathit{char} = \mathit{str}.\mathit{items}(i)\ \mathbf{in}\ \mathit{char} < 97\,?\ \\ & & \quad \mathit{char} : \mathit{char} < 123\,?\ \mathit{char} - 32 : \mathit{char}\} \\ E_{\mathit{diffRange}} & \triangleq & \mathit{range1}.\mathit{length} - \mathit{range2}.\mathit{length} \\ E_{\mathit{intEqual3}} & \triangleq & \mathit{int1} == \mathit{int2}\,?\ \mathit{int2} == \mathit{int3} : \mathbf{false}. \end{array}$$

Indeed, each function has multiple type signatures, some of which we list here:

- The *uppercase* function has signatures $(\mathit{String}, \mathit{String})$,$(\mathit{String}, \mathit{IntArray})$, $(\mathit{IntArray}, \mathit{String})$
- The *diffRange* function has signatures $(\mathit{String}, \mathit{String}, \mathit{String})$, $(\mathit{IntArray}, \mathit{IntArray}, \mathit{IntArray})$, and $(\mathit{Range}, \mathit{Range}, \mathit{Range})$.
- The *intEqual3* function has signatures $(\mathit{Bool}, \mathit{Bool}, \mathit{Bool}, \mathit{Bool})$, $(\mathit{Int}, \mathit{Int}, \mathit{Int}, \mathit{Bool})$, and also $(\mathit{Char}, \mathit{Char}, \mathit{Char}, \mathit{Bool})$.

B Constructing Logical Constraints and Natural Facts

The purpose of this section is to describe a new algorithm that given Γ and E returns a logical formula C such that any type that can be ascribed to E corresponds to a variable valuation Ω that satisfies the formula. Hence the formula captures logically the ways in which the expression may be type correct. As well as returning the logical constraint, the algorithm returns *natural facts*, a set of texts associated with the type variables.

The following section shows how to optimize simultaneously for logical constraints, corresponding to the logical formula, and for natural constraints, induced by the natural facts.

B.1 Logical Constraints, Variable Valuations, and Logical Satisfaction

We let α and β range over a given infinite set of type variables. We use A to range over finite sets of type variables.

Definition 5 (Type Variables)**.**

α, β	*type variable*
Let tyvar$(\alpha) = \{\alpha\}$.	

Our logical constraints are an equational logic, whose variables α denote items from a finite domain, the set of type names. The logic consists of propositional formulas, plus two equational predicates: $\alpha = t$, meaning that variable α denotes the type t, (An extension would be to allow equations between two variables, $\alpha = \alpha'$ meaning that both α and α' denote the same type, but these equations are not needed by our algorithm.) The finite domain is the set of types; the logic considers types simply as atomic names and does not depend on any other properties, such as type equality. Hence, a constraint $\alpha = \mathit{String}$ holds if α denotes the type name String, but does not hold if α denotes $\mathit{IntArray}$, even though $\mathit{String} <:> \mathit{IntArray}$.

Definition 6 (Logical Constraints)**.**

$C ::=$	*logical formula or constraint*
$\alpha = t$	*equation, between type variable* α *and type name* t

true	*truth*
$\neg\, C$	*negation*
$C \wedge C$	*conjunction*
$C \vee C$	*disjunction*

Let tyvar(C) be the set of type variables occurring in C.

Definition 7 (Valuation Ω)**.** *A* valuation Ω *is a finite map* $\{\alpha_i = t_i{}^{\,i\in 1..n}\}$ *such that for any type variable α_v, there is a type t_τ with $\Omega(\alpha_v) = t_\tau$. Let* $\mathrm{dom}(\Omega) = \{\alpha_i \mid i \in 1..n\}$*, the finite set of type variables in the domain of Ω.*

Definition 8 (Logical Satisfaction Relation)**.** *We define a logical satisfaction relation $\Omega \models C$, when tyvar$(C) \subseteq \mathrm{dom}(\Omega)$, by induction on the size of C, as follows:*

$$\begin{array}{rll}
\Omega \models \textit{true} & \textit{always} & \\
\Omega \models \alpha = t & \textit{if and only if} & \Omega(\alpha) = t \\
\Omega \models \neg\, C & \textit{if and only if} & \textit{not}\ \Omega \models C \\
\Omega \models C_1 \wedge C_2 & \textit{if and only if} & \Omega \models C_1\ \textit{and}\ \Omega \models C_2 \\
\Omega \models C_1 \vee C_2 & \textit{if and only if} & \Omega \models C_1\ \textit{or}\ \Omega \models C_2.
\end{array}$$

Let constraint C be satisfiable *if and only if there is a valuation Ω such that $\Omega \models C$.*

Definition 9 (Logical Equivalence)**.** *Let $C \sim C'$ mean that for all Ω with tyvar$(C, C') \subseteq \mathrm{dom}(\Omega)$, $\Omega \models C$ if and only if $\Omega \models C'$.*

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 $\wedge$ and $\vee$, and distributive laws relating them.

We adopt standard conventions. We define $\mathsf{false} \triangleq \neg\, \mathsf{true}$. 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 $\bigwedge_{i\in I} C_i$ means the conjunction $C_{i_1} \wedge \cdots \wedge C_{i_n}$, and in particular means true if $I = \varnothing$. Similarly, the notation $\bigvee_{i\in I} C_i$ means the disjunction $C_{i_1} \vee \cdots \vee C_{i_n}$, and in particular means false if $I = \varnothing$.

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 10. *Let the relation $t <:> S$ between type t and type structure S hold as follows:*

(1) $t <:> \iota$*, means* $\mathbf{type}\ t\ =\ \iota$*;*

(2) $t <:> \{\ell_i : t_i{}^{i\in 1..n}\}$ *means there are t'_i with* $\mathbf{type}\ t\ =\ \{\ell_i : t'_i{}^{i\in 1..n}\}$ *and $t_i <:> t'_i$ for all $i \in 1..n$;*

(3) $t <:> t_1 \to t_2$ *means there are t'_i with* $\mathbf{type}\ t\ =\ t'_1 \to t'_2$ *and $t_i <:> t'_i$ for all $i \in 1..2$.*

Definition 11 (Derived Syntax for Constraints)**.**

$\alpha <:> \iota \triangleq \bigvee_{t\in I} \alpha = t$ *where* $I = \{t \mid \exists \iota : t <:> \iota\}$
$\alpha <:> \alpha' \triangleq \bigvee_{(t,t')\in I} (\alpha = t \wedge \alpha' = t')$ *where* $I = \{(t, t') \mid t <:> t'\}$
$\alpha <:> \{\ell_i : \alpha_i{}^{\,i\in 1..n}\} \triangleq \bigvee_{(t,t_1,\ldots,t_n)\in I} (\alpha = t \wedge \bigwedge_{i\in 1..n} \alpha_i = t_i)$
where $I = \{(t, t_1, \ldots, t_n) \mid t <:> \{\ell_i : t_i{}^{\,i\in 1..n}\}\}$
$\alpha' <:> \alpha.\ell \triangleq \bigvee_{(t,t')\in I} (\alpha = t \wedge \alpha' = t')$
where $I = \{(t, t') \mid \exists n, j \in 1..n, \ell_1, t_1, \ldots, \ell_n, t_n : t <:> \{\ell_i : t_i{}^{\,i\in 1..n}\}, \ell = \ell_j, t' = t_j\}$
$\alpha <:> \alpha_1 \to \alpha_2 \triangleq \bigvee_{(t,t_1,t_2)\in I} (\alpha = t \wedge \alpha_1 = t_1 \wedge \alpha_2 = t_2)$ *where* $I = \{(t, t_1, t_2) \mid t <:> t_1 \to t_2\}$

These derived forms of constraints are satisfied as follows:

Lemma 3 (Logical Satisfaction for Derived Syntax)**.**

a. $\Omega \models \bigwedge_{i\in I} C_i$ *if and only if* $\forall i \in I : \Omega \models C_i$

b. $\Omega \models \bigvee_{i\in I} C_i$ *if and only if* $\exists i \in I : \Omega \models C_i$

c. $\Omega \models \alpha <:> \iota$ *if and only if* $\Omega(\alpha) <:> \iota$

d. $\Omega \models \alpha <:> \alpha'$ *if and only if* $\Omega(\alpha) <:> \Omega(\alpha')$

e. $\Omega \models \alpha <:> \{\ell_i : \alpha_i{}^{i \in 1..n}\}$ *if and only if* $\Omega(\alpha) <:> \{\ell_i : \Omega(\alpha_i)^{i \in 1..n}\}$

f. $\Omega \models \alpha' <:> \alpha.\ell$ *if and only if* $\exists\, n,\ \ell_1,\ t_1,\ \ldots,\ \ell_n,\ t_n$ *and* $j \in 1..n : \Omega(\alpha) <:> \{\ell_i : t_i{}^{i \in 1..n}\}$ *and* $\Omega(\alpha') <:> t_j$ *and* $\ell = \ell_j$

g. $\Omega \models \alpha <:> \alpha_1 \to \alpha_2$ *if and only if* $\Omega(\alpha) <:> \Omega(\alpha_1) \to \Omega(\alpha_2)$.

B.2 Natural Facts

A natural fact is a pair $(text, \alpha)$, where $text$ is a text, and α is a type variable, meaning that the text $text$, an identifier in this paper, is associated with the type variable α_i. Let N range over finite sets of natural facts. Appendix C.3 shows how we construct natural constraints from these facts.

Definition 12 (Natural Facts)**.**

$N ::= \{(text_1, \alpha_1), \ldots, (text_n, \alpha_n)\}$ *natural facts*

Let $tyvar(N) = \{\alpha_1, \ldots, \alpha_n\}$ *where* $N = \{(text_1, \alpha_1), \ldots, (text_n, \alpha_n)\}$.

B.3 Algorithmic Type System

An *algorithmic environment* Γ is a finite map from variables to types 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 $\varnothing$ for the empty environment. Let $\mathrm{dom}(x_1 : \alpha_1, \ldots, x_n : \alpha_n) = \{x_1, \ldots, x_n\}$ and $\mathrm{dom}(\varnothing) = \varnothing$. Let $tyvar(x_1 : \alpha_1, \ldots, x_n : \alpha_n) = \{\alpha_1, \ldots, \alpha_n\}$ and $tyvar(\varnothing) = \varnothing$. (We use the same metavariable Γ both for these algorithmic environments and also for the environments of Appendix A.3, which map variables to type names.)

The judgment of our algorithmic type system takes the form $\Gamma \vdash E \Rightarrow \alpha\ (C, N)$ meaning that in Γ the expression E has type α, a variable constrained by C, and that N is a set of natural facts that associate variables in E with type variables from Γ or C.

The rules are as follows. We rely on the notation $tyvar(\psi_1, \ldots, \psi_n)$ that means the set of variables $tyvar(\psi_1) \cup \cdots \cup tyvar(\psi_n)$, where each ψ_i is a syntactic phrase that is either an environment Γ, a type variable α, a logical constraint C, or a natural constraint N.

Definition 13 (Algorithmic Typing Rules)**.**

Let $fresh(\Gamma, \alpha, C, N) = tyvar(\alpha, C, N) \setminus tyvar(\Gamma)$.

(Algo x)
$$\frac{x \in \mathrm{dom}(\Gamma) \quad \Gamma(x) = \alpha}{\Gamma \vdash x \Rightarrow \alpha\ (\mathit{true}, \varnothing)}$$

(Algo b) $(\alpha \notin tyvar(\Gamma))$
$$\frac{b \in \{\mathbf{true}, \mathbf{false}\}}{\Gamma \vdash b \Rightarrow \alpha\ (\alpha <:> Bool, \varnothing)}$$

(Algo c) $(\alpha \notin tyvar(\Gamma))$
$$\frac{\textit{integer } c}{\Gamma \vdash c \Rightarrow \alpha\ (\alpha <:> Int, \varnothing)}$$

(Algo −) $(\alpha \notin tyvar(\Gamma, C_1, C_2)$ *and* $fresh(\Gamma, \alpha_1, C_1, N_1) \cap fresh(\Gamma, \alpha_2, C_2, N_2) = \varnothing)$
$$\frac{\Gamma \vdash E_1 \Rightarrow \alpha_1\ (C_1, N_1) \quad \Gamma \vdash E_2 \Rightarrow \alpha_2\ (C_2, N_2)}{\Gamma \vdash E_1\ -\ E_2 \Rightarrow \alpha\ (\alpha <:> Int \wedge \alpha_1 <:> Int \wedge \alpha_2 <:> Int \wedge C_1 \wedge C_2, N_1 \cup N_2)}$$

(Algo <) $(\alpha \notin tyvar(\Gamma, C_1, C_2)$ *and* $fresh(\Gamma, \alpha_1, C_1, N_1) \cap fresh(\Gamma, \alpha_2, C_2.N_2) = \varnothing)$
$$\frac{\Gamma \vdash E_1 \Rightarrow \alpha_1\ (C_1, N_1) \quad \Gamma \vdash E_2 \Rightarrow \alpha_2\ (C_2, N_2)}{\Gamma \vdash E_1\ <\ E_2 \Rightarrow \alpha\ (\alpha <:> Bool \wedge \alpha_1 <:> Int \wedge \alpha_2 <:> Int \wedge C_1 \wedge C_2, N_1 \cup N_2)}$$

(Algo ==) $(\alpha \notin tyvar(\Gamma, C_1, C_2)$ *and* $fresh(\Gamma, \alpha_1, C_1, N_1) \cap fresh(\Gamma, \alpha_2, C_2, N_2) = \varnothing)$
$$\frac{\Gamma \vdash E_1 \Rightarrow \alpha_1\ (C_1, N_1) \quad \Gamma \vdash E_2 \Rightarrow \alpha_2\ (C_2, N_2)}{\Gamma \vdash E_1\ ==\ E_2 \Rightarrow \alpha\ (\alpha <:> Bool \wedge \bigvee_{\iota \in \{Bool, Int\}} (\alpha_1 <:> \iota \wedge \alpha_2 <:> \iota) \wedge C_1 \wedge C_2, N_1 \cup N_2)}$$

(Algo Rcd) $(\alpha \notin \bigcup_{i \in 1..n} tyvar(\Gamma, \alpha_i, C_i, N_i)$ *and sets* $fresh(\Gamma, \alpha_i, C_i, N_i)$ *disjoint)*
$$\frac{\Gamma \vdash E_i \Rightarrow \alpha_i\ (C_i, N_i) \quad \forall i \in 1..n}{\Gamma \vdash \{\ell_i = E_i{}^{i \in 1..n}\} \Rightarrow \alpha\ (\alpha <:> \{\ell_i : \alpha_i{}^{i \in 1..n}\} \wedge \bigwedge_{i \in 1..n} C_i, \bigcup_{i \in 1..n} N_i)}$$

(Algo Proj) $(\alpha' \notin \mathit{tyvar}(\Gamma, \alpha, C, N))$

$$\frac{\Gamma \vdash E \Rightarrow \alpha\ (C, N)}{\Gamma \vdash E.\ell \Rightarrow \alpha'\ (\alpha' <:> \alpha.\ell \wedge C, N)}$$

(Algo If) *(sets* $\mathit{fresh}(\Gamma, \alpha_i, C_i, N_i)$ *disjoint)*

$$\frac{\Gamma \vdash E_1 \Rightarrow \alpha_1\ (C_1, N_1) \quad \Gamma \vdash E_2 \Rightarrow \alpha_2\ (C_2, N_2) \quad \Gamma \vdash E_3 \Rightarrow \alpha_3\ (C_3, N_3)}{\Gamma \vdash (E_1\ ?\ E_2\ :\ E_3) \Rightarrow \alpha_2\ (\alpha_1 <:> \mathit{Bool} \wedge \alpha_2 <:> \alpha_3 \wedge \bigwedge_{i \in 1..3} C_i, \bigcup_{i \in 1..3} N_i)}$$

(Algo Let) $(x \notin \mathrm{dom}(\Gamma)$ *and* $\mathit{fresh}(\Gamma, \alpha_1, C_1, N_1) \cap \mathit{fresh}(\Gamma, \alpha_2, C_2, N_2) = \varnothing)$

$$\frac{\Gamma \vdash E_1 \Rightarrow \alpha_1\ (C_1, N_1) \quad \Gamma, x : \alpha_1 \vdash E_2 \Rightarrow \alpha_2\ (C_2, N_2)}{\Gamma \vdash \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2 \Rightarrow\ \alpha_2\ (C_1 \wedge C_2, \{(x, \alpha_1)\} \cup N_1 \cup N_2)}$$

(Algo Lambda) $(x \notin \mathrm{dom}(\Gamma)$ *and* $\alpha \notin \mathit{tyvar}(\Gamma, \alpha_2, C, N))$

$$\frac{\Gamma, x : \alpha_1 \vdash E \Rightarrow \alpha_2\ (C, N)}{\Gamma \vdash \lambda(x)E \Rightarrow\ \alpha\ (\alpha <:> \alpha_1 \to \alpha_2 \wedge C, \{(x, \alpha_1)\} \cup N)}$$

(Algo Appl) $(\alpha \notin \mathit{tyvar}(\Gamma, C_2, C_1)$ *and* $\mathit{fresh}(\Gamma, \alpha_2, C_2, N_2) \cap \mathit{fresh}(\Gamma, \alpha_1, C_1, N_1) = \varnothing)$

$$\frac{\Gamma \vdash E_2 \Rightarrow \alpha_2\ (C_2, N_2) \quad \Gamma \vdash E_1 \Rightarrow \alpha_1\ (C_1, N_1)}{\Gamma \vdash E_2(E_1) \Rightarrow\ \alpha\ (\alpha_2 <:> \alpha_1 \to \alpha \wedge C_1 \wedge C_2, N_1 \cup N_2)}$$

We call this relation *algorithmic* because it is a nondeterministic specification for an algorithm that given *inputs* Γ and E computes *outputs* α, C, and N such that $\Gamma \vdash E \Rightarrow \alpha\ (C, N)$. Most of the rules pick a variable α 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.

(Algo $-$) $(\alpha \notin \mathit{tyvar}(\Gamma, C_1, C_2)$ and $\mathit{fresh}(\Gamma, \alpha_1, C_1, N_1) \cap \mathit{fresh}(\Gamma, \alpha_2, C_2, N_2) = \varnothing)$

$$\frac{\Gamma \vdash E_1 \Rightarrow \alpha_1\ (C_1, N_1) \quad \Gamma \vdash E_2 \Rightarrow \alpha_2\ (C_2, N_2)}{\Gamma \vdash E_1\ -\ E_2 \Rightarrow \alpha\ (\alpha <:> \mathit{Int} \wedge \alpha_1 <:> \mathit{Int} \wedge \alpha_2 <:> \mathit{Int} \wedge C_1 \wedge C_2, N_1 \cup N_2)}$$

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

(1) The first kind, exemplified by $\alpha \notin \mathit{tyvar}(\Gamma, C_1, C_2)$, ensures that the fresh variable α is distinct from other variables in the current derivation: distinct either from the input Γ or the output components C_1 and C_2.

(2) The second kind, exemplified by $\mathit{fresh}(\Gamma, \alpha_1, C_1, N_1) \cap \mathit{fresh}(\Gamma, \alpha_2, C_2, N_2) = \varnothing$, ensures the disjointness of the sets of fresh variables picked by independent parallel derivations, such as $\Gamma \vdash E_1 \Rightarrow \alpha_1\ (C_1, N_1)$ and $\Gamma \vdash E_2 \Rightarrow \alpha_2\ (C_2, N_2)$. The auxiliary function $\mathit{fresh}(\Gamma, \alpha, C, N)$ determines the type variables that are fresh in the derivation of $\Gamma \vdash E \Rightarrow \alpha\ (C, N)$, that is, the variables $\mathit{tyvar}(\alpha, C, N)$ occurring in the output that are not in $\mathit{tyvar}(\Gamma)$, the variables of the input.

A third kind of side-condition is $x \notin \mathrm{dom}(\Gamma)$ 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. *If* $\Gamma \vdash E \Rightarrow \alpha\ (C, N)$ *then* $\mathit{tyvar}(\alpha, N) \subseteq \mathit{tyvar}(\Gamma, C)$ *and* E *is well-scoped.*

Proof. By induction on the depth of derivation of $\Gamma \vdash E \Rightarrow \alpha\ (C, N)$. □

Disjunctive logical constraints 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.

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 most natural subject to being sound.

B.4 Formal Properties: Termination, Soundness, and Completeness

Every well-scoped expression determines a logical constraint and natural facts.

Theorem 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 α, C, N such that $x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash E \Rightarrow \alpha\ (C, N)$.*

Proof. Existence holds by induction on the structure of E. □

To state soundness and completeness of the algorithmic type system with respect to the syntax-directed declarative type system Definition 20, we introduce a notation $\Omega(\Gamma)$ that lifts a variable valuation Ω to apply to an algorithmic environment Γ. Let $\Omega(\Gamma) = x_1 : \Omega(\alpha_1), \ldots, x_n : \Omega(\alpha_n)$ if environment $\Gamma = x_1 : \alpha_1, \ldots, x_n : \alpha_n$,

Theorem 2. *Suppose $\Gamma \vdash E \Rightarrow \alpha\ (C, N)$ and $A =$ tyvar(Γ, α) and $A' =$ tyvar$(C) \setminus A$. For all Ω with $\mathrm{dom}(\Omega) = A$:*

(1) *(*SOUNDNESS*) For all Ω' with $\mathrm{dom}(\Omega') = A'$, if $\Omega \cup \Omega' \models C$ then $\Omega(\Gamma) \vdash E : \Omega(\alpha)$.*

(2) *(*COMPLETENESS*) If $\Omega(\Gamma) \vdash E : \Omega(\alpha)$ then $\Omega \cup \Omega' \models C$ for some Ω' with $\mathrm{dom}(\Omega') = A'$.*

The following corollary can prove the claims made in Section 1.1 about our three motivating examples of untyped function definitions.

Corollary 1. *Suppose that:*

- **function** $f(x_1, \ldots, x_n)$ **return** E *is an untyped function definition with fv$(E) \subseteq \{x_1, \ldots, x_n\}$*
- $\Gamma = x_1 : \beta_1, \ldots, x_n : \beta_n$ *for pairwise distinct type variables β_i*
- $\Gamma \vdash E \Rightarrow \beta\ (C, N)$
- $A = \{\beta_1, \ldots, \beta_n, \beta\}$ *and $A' =$ tyvar$(C) \setminus A$*

Then, for all tuples $(t_1, \ldots, t_n, t)$, the following propositions are logically equivalent:

(1) $(t_1, \ldots, t_n, t)$ *is a type signature for f*

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

(3) $\{\beta_1 = t_1, \ldots, \beta_n = t_n, \beta = t\} \cup \Omega' \models C$ *for some Ω' with* $\mathrm{dom}(\Omega') = A'$.

B.5 Revisiting our Three Motivating Examples

In this section we show how Corollary 1 applies to our motivating examples (see Appendix A.5). We show in full details how we extract these for the type signature for the *diffRange* function. For *uppercase* and *intEqual3* we present only the first and the final step of our reasoning.

B.5.1 Example: *diffRange*

Recall that the definition for *diffRange* is:

function *diffRange*(*range1*, *range2*) **return** *range1.length* − *range2.length* // 2nd

Starting with $\Gamma =$ *range*$_1 : \beta_1$, *range*$_2 : \beta_2$, let:

$$\begin{aligned} E_1 &\triangleq \textit{range1.length} \\ E_2 &\triangleq \textit{range2.length} \\ E_{\textit{diffRange}} &\triangleq E_1 - E_2 \end{aligned}$$

By Definition 13, the following is derivable:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathit{range_1} \Rightarrow \beta_1\ (\mathsf{true}, \varnothing)}\text{Algo } x \qquad \alpha_1 \notin \{\beta_1, \beta_2\}}{\Gamma \vdash E_1 \Rightarrow \alpha_1\ (\alpha_1 <:> \beta_1.\mathit{length} \wedge \mathsf{true}, \varnothing)}\text{Algo Proj} \quad \dfrac{\dfrac{}{\Gamma \vdash \mathit{range_2} \Rightarrow \beta_2\ (\mathsf{true}, \varnothing)}\text{Algo } x \qquad \alpha_2 \notin \{\beta_1, \beta_2\}}{\Gamma \vdash E_2 \Rightarrow \alpha_2\ (\alpha_2 <:> \beta_2.\mathit{length} \wedge \mathsf{true}, \varnothing)}\text{Algo Proj} \qquad \beta \notin \{\beta_1, \beta_2, \alpha_1, \alpha_2\} \text{ and } \{a_1\} \cap \{a_2\} = \varnothing}{\Gamma \vdash E_{\mathit{diffRange}} \Rightarrow \beta\ (\beta <:> \mathit{Int} \wedge \alpha_1 <:> \mathit{Int} \wedge \alpha_2 <:> \mathit{Int} \wedge \alpha_1 <:> \beta_1.\mathit{length} \wedge \alpha_2 <:> \beta_2.\mathit{length} \wedge \mathsf{true}, \varnothing)}\text{Algo} -$$

Given this result, for all tuples (t_1, t_2, t), Corollary 1 tells us that each of the following propositions are logically equivalent:

(1) (t_1, t_2, t) is a type signature for $\mathit{diffRange}$

(2) $\mathit{range}_1 : t_1, \mathit{range}_2 : t_2 \vdash E_{\mathit{diffRange}} : t$

(3) there is Ω' with $\mathrm{dom}(\Omega') = \{\alpha_1, \alpha_2\}$ and

$$\begin{aligned}&\{\beta_1 = t_1, \beta_2 = t_2, \beta = t\} \cup \Omega' \\ &\models (\beta <:> \mathit{Int} \wedge \alpha_1 <:> \mathit{Int} \wedge \alpha_2 <:> \mathit{Int} \wedge \alpha_1 <:> \beta_1.\mathit{length} \wedge \alpha_2 <:> \beta_2.\mathit{length} \wedge \mathsf{true})\end{aligned}$$

By further simplification, the following propositions are also equivalent:

(4) by Lemma 3(f), there is Ω' with $\mathrm{dom}(\Omega') = \{\alpha_1, \alpha_2\}$ and

$$\begin{aligned}&\{\beta_1 = t_1, \beta_2 = t_2, \beta = t\} \cup \Omega' \\ &\models \big(\beta = \mathit{Int} \wedge \alpha_1 = \mathit{Int} \wedge \alpha_2 = \mathit{Int} \wedge \\ &\quad ((\beta_1 = \mathit{String} \wedge \alpha_1 = \mathit{Int}) \vee (\beta_1 = \mathit{IntArray} \wedge \alpha_1 = \mathit{Int}) \vee (\beta_1 = \mathit{Range} \wedge \alpha_1 = \mathit{Int})) \wedge \\ &\quad ((\beta_2 = \mathit{String} \wedge \alpha_2 = \mathit{Int}) \vee (\beta_2 = \mathit{IntArray} \wedge \alpha_2 = \mathit{Int}) \vee (\beta_2 = \mathit{Range} \wedge \alpha_2 = \mathit{Int})) \wedge \mathsf{true}\big)\end{aligned}$$

(5) by Definition 9, there is Ω' with $\mathrm{dom}(\Omega') = \{\alpha_1, \alpha_2\}$ and

$$\begin{aligned}&\{\beta_1 = t_1, \beta_2 = t_2, \beta = t\} \cup \Omega' \\ &\models \big(\beta = \mathit{Int} \wedge \alpha_1 = \mathit{Int} \wedge \alpha_2 = \mathit{Int} \wedge \\ &(\beta_1 = \mathit{String} \vee \beta_1 = \mathit{IntArray} \vee \beta_1 = \mathit{Range}) \wedge \\ &(\beta_2 = \mathit{String} \vee \beta_2 = \mathit{IntArray} \vee \beta_2 = \mathit{Range})\big)\end{aligned}$$

(6) by Definition 8

$$\big((t_1 = \mathit{String} \text{ or } t_1 = \mathit{IntArray} \text{ or } t_1 = \mathit{Range}) \text{ and } (t_2 = \mathit{String} \text{ or } t_2 = \mathit{IntArray} \text{ or } t_2 = \mathit{Range})\big)$$

To summarize our chain of reasoning, we have that:

$$\begin{gathered}(t_1, t_2, t) \text{ is a type signature for } \mathit{diffRange} \\ \text{if and only if} \\ \big((t_1 = \mathit{String} \text{ or } t_1 = \mathit{IntArray} \text{ or } t_1 = \mathit{Range}) \text{ and } (t_2 = \mathit{String} \text{ or } t_2 = \mathit{IntArray} \text{ or } t_2 = \mathit{Range})\big)\end{gathered}$$

B.5.2 Example: *uppercase*

By Corollary 1 and similar reasoning to the previous example, we get:

$$\begin{gathered}(t_1, t) \text{ is a type signature for } \mathit{uppercase} \\ \text{if and only if} \\ \big((t_1 = \mathit{IntArray} \text{ or } t_1 = \mathit{String}) \text{ and } (t = \mathit{IntArray} \text{ or } t = \mathit{String})\big)\end{gathered}$$

B.5.3 Example: *intEqual3*

Again, by Corollary 1 and similar reasoning we get:

$$\begin{gathered}(t_1, t_2, t) \text{ is a type signature for } \mathit{intEqual3} \\ \text{if and only if} \\ \big((t_1 = \mathit{Bool} \text{ and } t_2 = \mathit{Bool} \text{ and } t_3 = \mathit{Bool}) \text{ or } (t_1 = \mathit{Int} \text{ and } t_2 = \mathit{Int} \text{ and } t_3 = \mathit{Int})\big) \text{ and } t = \mathit{Bool}\end{gathered}$$

C Solving Logical and Natural Constraints using Numerical Relaxation

In this section, our goal is to determine a way of choosing the valuation Ω to combine the logical constraints C and the natural facts N in a way that satisfies both requirements. To achieve this, we propose a method based on an optimization perspective, in which we numerically relax the problem of satisfying the logical constraints into a continuous optimization problem. First, we perform the numerical relaxation of the logical constraints, introducing a relaxed semantics of the constraints (Appendix C.1) and describing its properties (Appendix C.2). Afterwards, we define a continuous typing environment by introducing probability distributions over these relaxed semantics. This provides a numerical connection between the value of the relaxed semantics and the logical constraints satisfaction.

Next, we turn our attention towards the natural aspect of the problem and explain how to obtain natural typing constraints which are compatible with the logical constraints (Appendix C.3). Further, we describe how to combine the logical and natural constraints into a single formulation by introducing a joint optimization problem (Appendix C.4). Most importantly, we provide conditions under which the joint optimization problem converges to a solution that is guaranteed to satisfy both the logical and the natural constraints simultaneously.

Finally, we note that throughout this section we consider a finite sequence of $\mathcal{V} > 0$ type variables $\alpha_1, \ldots, \alpha_{\mathcal{V}}$, and a finite sequence of $\mathcal{T} > 0$ type names $t_1, \ldots, t_{\mathcal{T}}$.

C.1 Numerical Relaxation of the Logical Constraints

We proceed by defining a *numerical relaxation* for the discrete logical constraints (see Definition 6). For this purpose, we turn to soft logic, which relaxes the range of the truth function to $[0, 1]$ and is differentiable almost everywhere Hájek [1998]. First, we define a probability matrix P to hold relaxed concrete type valuations. This is key object we optimise in our key theorem, Theorem 4, below. We then define relaxed semantics for the logical constraints C over P, and establish important properties of this semantics.

Definition 14 (Type Valuation Probability Matrix)**.** *For each type variable $\{\alpha_v \mid v \in 1..\mathcal{V}\}$, we define a* probability matrix *as $P = \begin{bmatrix}\boldsymbol{p}_1^\intercal & \ldots & \boldsymbol{p}_{\mathcal{V}}^\intercal\end{bmatrix}^\intercal$, where each $\boldsymbol{p}_v = [p_{v,1} \quad \ldots \quad p_{v,\mathcal{T}}]$ is a row vector that defines a probability distribution over concrete types. Let $\mathcal{P}^{\mathcal{V}\times\mathcal{T}}$ be the set of all probability matrices.*

As we discuss in Appendix E.2, soft logic offers different choices of triangular norms to represent conjunction (t-norms). To be consistent with boolean first order logic, a t-norm is by definition commutative, associative, non-decreasing, and 1 is a neutral element. Given a t-norm, we can derive the corresponding t-conorm, which behaves like logical disjunction. Inspired by previous work on relation inference in machine learning Rocktäschel et al. [2015], we chose to use product logic. The main benefit of this choice is that its relaxations are smooth and well-suited for our optimization-based approach.

Definition 15 (Relaxed Semantics)**.** *We define a* relaxed semantics *of C as a function $[\![C]\!]_P : \mathcal{P}^{\mathcal{V}\times\mathcal{T}} \times \mathcal{C} \to [0, 1]$, where $v \in 1 \ldots \mathcal{V}$, $\tau \in 1 \ldots \mathcal{T}$ and $\mathcal{C}$ is the set of logical constraints, defined as:*

$$
\begin{aligned}
[\![\mathit{true}]\!]_P &= 1 \\
[\![\alpha_v = t_\tau]\!]_P &= p_{v,\tau} \\
[\![\neg\, C]\!]_P &= 1 - [\![C]\!]_P \\
[\![C_1 \wedge C_2]\!]_P &= [\![C_1]\!]_P \cdot [\![C_2]\!]_P \\
[\![C_1 \vee C_2]\!]_P &= [\![C_1]\!]_P + [\![C_2]\!]_P - [\![C_1]\!]_P \cdot [\![C_2]\!]_P.
\end{aligned}
$$

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 3.

Lemma 5. *For all C and all $P \in \mathcal{P}^{\mathcal{V}\times\mathcal{T}}$, we have $0 \leq [\![C]\!]_P \leq 1$.*

Proof. By structural induction on the expression C. □

Lemma 6. *For all C, C_1, C_2, and $P \in \mathcal{P}^{\mathcal{V}\times\mathcal{T}}$:*

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. $([\![C_1]\!]_P = 1$ *and* $[\![C_2]\!]_P = 1)$ *if and only if* $([\![C_1]\!]_P \cdot [\![C_2]\!]_P = 1)$

iv. $([\![C_1]\!]_P = 1$ *or* $[\![C_2]\!]_P = 1)$ *if and only if* $([\![C_1]\!]_P + [\![C_2]\!]_P - [\![C_1]\!]_P \cdot [\![C_2]\!]_P = 1)$.

Proof. These follow by cases analyses based on Lemma 5. □

C.2 Relaxation and Rounding of Type Valuations

In optimisation, *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 continuous probability distribution over all possible type valuations, conditioned on a type valuation probability matrix P (Definition 14). We close by showing that this relaxation preserves validity.

To relax a type valuation, we define a binary matrix that converts it into a probability matrix and establish that the conversion preserves validity.

Definition 16 (Binary Environment)**.** *We define the binary matrix* $B(\Omega) \in \mathcal{P}^{\mathcal{V}\times\mathcal{T}}$ *by setting each element* $b_{v,\tau} = 1$ *if* $\Omega(\alpha_v) = t_\tau$*, and* 0 *otherwise.*

The following is Theorem 1 of Pandi et al. [2020], and is the main theoretical result of that paper.

Lemma 7 (Binary Relaxation)**.** *For all* C *and* Ω*:*

$$[\![C]\!]_{B(\Omega)} = 1 \text{ if and only if } \Omega \models C.$$

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 marginalise the per variable type valuations into a set, over which we define a random variable. It is this random variable that we sample to recover a discrete type valuation.

Definition 17 (Continuous Type Valuation Space)**.** *Given a probability matrix* P*, for each type variable* α_v*, let* $\mathsf{A}_v \in \{t_1, \dots t_\mathcal{T}\}$ *be a discrete random variable with marginal probability mass function*

$$\Pr[\mathsf{A}_v = t \mid P] = \prod_{\tau=1}^{\mathcal{T}} p_{v,\tau}^{\delta(t=t_\tau)}, \tag{1}$$

where δ *is the Kronecker delta. In the above equation, we conventionally use* $0^0 = 1$*. We define* $\{\mathsf{A}_1 \dots \mathsf{A}_\mathcal{V}\}$ *to be independent.*

Finally, define the random variable $\widetilde{\Omega} = \{(\alpha_v, \mathsf{A}_v) \mid v \in 1 \dots \mathcal{V}\}$*; this is a random set whose outcome is a valuation. We write its mass function as* $\Pr[\widetilde{\Omega} = \Omega \mid P]$*. Additionally, when we know we are referring to* $\widetilde{\Omega}$*, we write* $\Pr[\Omega \mid P]$*.*

Lemma 8. *The probability mass function of* $\widetilde{\Omega}$ *is*

$$\Pr[\widetilde{\Omega} = \Omega \mid P] = \prod_{v=1}^{\mathcal{V}} \prod_{\tau=1}^{\mathcal{T}} p_{v,\tau}^{\delta(\Omega(\alpha_v)=t_\tau)}. \tag{2}$$

Proof. Observe that $\widetilde{\Omega}$ is in 1:1 correspondence with the set of random variables $\{\mathsf{A}_1, \dots \mathsf{A}_\mathcal{V}\}$. Because these variables are independent, their joint pmf is

$$\Pr[\mathsf{A}_1 \dots \mathsf{A}_v \mid P] = \prod_{v=1}^{\mathcal{V}} \prod_{\tau=1}^{\mathcal{T}} p_{v,\tau}^{\delta(a_v=t_\tau)}. \tag{3}$$

By change of variables, we obtain Equation (2). □

Theorem 3 (Continuous Relaxation)**.** *Consider any* Ω *and any* $P \in \mathcal{P}^{\mathcal{V}\times\mathcal{T}}$*, such that* $\Pr[\Omega \mid P] > 0$*. For all* C*, we have that:*

$$[\![C]\!]_P = 1 \textit{ implies } \Omega \models C \tag{4a}$$

$$[\![C]\!]_P = 0 \textit{ implies } \Omega \models \neg\, C. \tag{4b}$$

Proof. By structural induction on the expression C. □

C.3 Natural Constraints

To generate probabilistic natural constraints from the natural facts (Definition 12) our typing rules gather (Definition 13), we first introduce the function *NC* that returns a probability distribution from texts to types. We then use *NC* to define natural typing constraints for each variable, which we aggregate into a natural constraint matrix.

Let *NC* be a function from any set of texts to a distribution over type names. For any finite set of texts $X = \{text_1, \ldots, text_m\}$, *NC*$(X)$ is a probability vector $\boldsymbol{n}$ of length $\mathcal{T}$, the number of type names. In the present work, *NC* corresponds to a pretrained machine learning model.

Definition 18 (*Natural Constraints*)**.** *For each type variable* $\{\alpha_v\ (C, N) \mid v \in 1..\mathcal{V}\}$*, let* $N_v = \{text \mid (text, _) \in N\}$*, be the texts bound to* v *in* α_v*. A* natural constraint *is the probability vector* *NC*$(N_v) = \boldsymbol{n}_v = [\mathrm{n}_{v,1}, \ldots, \mathrm{n}_{v,\mathcal{T}}]$ *over* $\mathcal{T}$ *for* v*. We aggregate the natural constraints* $\boldsymbol{n}_v$ *into a* natural probability matrix *as* $\mathcal{N} = \begin{bmatrix}\boldsymbol{n}_1^\mathsf{T} & \ldots & \boldsymbol{n}_\mathcal{V}^\mathsf{T}\end{bmatrix}^\mathsf{T}$.

Each $\boldsymbol{n}_v \in \mathcal{N}$ is *NC*'s output on v's natural facts N_v.

Intuitively, if we wish to find a $P \in \mathcal{P}^{\mathcal{V}\times\mathcal{T}}$ that is as close as possible to the natural constraints $\mathcal{N} \in \mathcal{P}^{\mathcal{V}\times\mathcal{T}}$, we need to minimize the distance between these matrices. A simple and pragmatic choice is the sum of the squared Euclidean distances between their rows. Recall that the Euclidean distance between two vectors $\mathbf{u} = [u_1, \ldots, u_D]$ and $\mathbf{v} = [v_1, \ldots, v_D]$ is

$$\|\mathbf{u} - \mathbf{v}\|_2 = \sqrt{\sum_{d=1}^{D} (u_d - v_d)^2}.$$

This allows us to define the notion of natural distance.

Definition 19 (Natural Distance)**.** *Given a probability matrix* $P \in \mathcal{P}^{\mathcal{V}\times\mathcal{T}}$ *and a natural probability matrix* $\mathcal{N} \in \mathcal{P}^{\mathcal{V}\times\mathcal{T}}$*, we define Natural Distance as the sum of the squared Euclidean distances of each* $p_v \in P$ *from* $n_v \in \mathcal{N}$*.*

$$\textit{NatDist}(P, \mathcal{N}) \triangleq \sum_{v=1}^{\mathcal{V}} ||\boldsymbol{p}_v - \boldsymbol{n}_v||_2^2. \tag{5}$$

C.4 The Joint Optimization Theorem

Given a program that admits multiple, correct, concrete type assignments given a type library (like our motivating examples in Section 1.1), the core intuition of this work is that logical and natural constraints can interact to speed finding a type valuation that 1) type checks (satisfies the logical constraints $[\![C]\!]_P$) and 2) is *most natural*, that is, that minimizes *NatDist*$(P, \mathcal{N})$ over all valuations that type check.

In this section, we focus on finding a probability matrix P^* that has these properties; later we will discuss how to obtain a valuation from P^*. In particular, we will seek a P^* that is most natural in the sense that it minimizes the natural distance over all probability matrices that satisfy the logical constraints C in the relaxed semantics. More precisely, the probability matrix P^* is most natural if *NatDist*$(P^*, \mathcal{N}) \leq$ *NatDist*$(P', \mathcal{N})$, for all $P' \in \mathcal{P}^{\mathcal{V}\times\mathcal{T}}$. Essentially, P^* is the solution to a constrained continuous optimization problem.

To solve this optimization problem, we convert this problem to an unconstrained optimization problem (Equation (6)), in an approach inspired by the penalty method Bertsekas [1982], Boyd and Vandenberghe [2004]. In the unconstrained problem, we minimize the natural distance plus a penalty

term that penalizes P^* if it does not satisfy C under the relaxed semantics. In fact, the following theorem shows that this penalty can be chosen in such a way that the logical constraints must be satisfied.

Theorem 4 (Penalty Existence)**.** *Suppose C is satisfiable. Given a natural probability matrix $\mathcal{N}$ associated with C, there exists $M \in \mathbb{R}$ such that for all $m > M$, if*

$$P^* = \operatorname*{argmin}_{P \in \mathcal{P}^{\mathcal{V} \times \mathcal{T}}} \textit{NatDist}(P, \mathcal{N}) - m[\![C]\!]_P \tag{6}$$

then $[\![C]\!]_{P^*} = 1$. Also P^* is the most natural probability matrix, in the sense that $\textit{NatDist}(P^*, \mathcal{N}) \leq \textit{NatDist}(P', \mathcal{N})$, for all $P' \in \mathcal{P}^{\mathcal{V} \times \mathcal{T}}$ with $[\![C]\!]_{P'} = 1$.

The proof first observes that satisfying C naturally defines a bipartition over P, dividing it into nonsatisfying $P^{<1}$ and satisfying P^1 assignments. Then it finds optimal values over these parts for $\textit{NatDist}(P, \mathcal{N})$, the natural constraints, and $[\![C]\!]_P$, the logical constraints. It uses these values to construct an M such that, if $m > M$ and P^* is optimal, P^* must fall into P^1, so $[\![C]\!]_{P^*} = 1$ and it is a most natural, satisfying type assignment relative to $\mathcal{N}$.

Proof. First, we find a probabilistic type assignment P by solving the optimization problem

$$\min_{P \in \mathcal{P}^{\mathcal{V} \times \mathcal{T}}} \textit{NatDist}(P, \mathcal{N}) - m[\![C]\!]_P. \tag{7}$$

For clarity, we refer to this objective function below as:

$$\mathcal{O}_m(P) = \textit{NatDist}(P, \mathcal{N}) - m[\![C]\!]_P. \tag{8}$$

What we would like to show is that optimizing $\mathcal{O}_m$ yields a continuous assignment P such that $[\![C]\!]_P = 1$. Consider a fixed expression C. To analyse the optimum of the relaxed semantics, we partition $\mathcal{P}^{\mathcal{V} \times \mathcal{T}}$ into two sets

$$\begin{aligned} \mathcal{P}^{<1} &\triangleq \{P \in \mathcal{P}^{\mathcal{V} \times \mathcal{T}} \mid [\![C]\!]_P < 1\} \\ \mathcal{P}^{1} &\triangleq \{P \in \mathcal{P}^{\mathcal{V} \times \mathcal{T}} \mid [\![C]\!]_P = 1\}. \end{aligned} \tag{9}$$

First, the above is indeed a partition. $\mathcal{P}^{<1}$ and $\mathcal{P}^1$ are disjoint, by definition. Further, we have $\mathcal{P}^{<1} \cup \mathcal{P}^1 = \mathcal{P}$ since, for all $P \in \mathcal{P}^{\mathcal{V} \times \mathcal{T}}$, $[\![C]\!]_P \leq 1$, by Lemma 5. Second, observe that $\mathcal{P}^1$ is nonempty. Because C is satisfiable, there is an Ω such that $\Omega \models C$. Pick $P = B(\Omega)$. Then, by Lemma 7, we obtain $[\![C]\!]_P = 1$. Therefore, $B(\Omega)$ is a member of $\mathcal{P}^1$, so $\mathcal{P}^1$ is nonempty. If $\mathcal{P}^{<1}$ is empty, the theorem is trivially true, as every solution is in $\mathcal{P}^1$. Thus, without loss of generality, we assume that $\mathcal{P}^{<1}$ is nonempty.

First, suppose we optimize the natural constraints only for assignments where $[\![C]\!]_P < 1$. Then, we would get a potentially different optimum

$$\nu^{<1} = \min_{P \in \mathcal{P}^{<1}} \textit{NatDist}(P, \mathcal{N}) \tag{10}$$

Now, consider the constrained optimum of the natural constraints

$$\nu^{1} = \min_{P \in \mathcal{P}^{1}} \textit{NatDist}(P, \mathcal{N}) \tag{11}$$

Intuitively, ν^1 is the best possible value of the natural constraints on environments that $[\![C]\!]_P = 1$. We can define similar quantities for the logical constraints by maximizing $[\![C]\!]$ separately over $\mathcal{P}^1$ and $\mathcal{P}^{<1}$

$$\ell^{<1} = \max_{P \in \mathcal{P}^{<1}} [\![C]\!]_P \tag{12}$$

$$\ell^{1} = \max_{P \in \mathcal{P}^{1}} [\![C]\!]_P. \tag{13}$$

By the definition in Equation (9), we have that $l^{<1} < 1$ and $l^1 = 1$.

Choose

$$M = \max\left\{0, \frac{\nu^{<1} - \nu^1}{1 - l^{<1}}\right\}. \tag{14}$$

And assume that $m > M$, let $P \in \mathcal{P}^{<1}$. Finally, let $P^{1'}$ be an arbitrary element of $\mathcal{P}^1$ such that *NatDist*$(P^{1'}, \mathcal{N}) = \nu^1$. Then we have

$$\begin{aligned}
\mathcal{O}_m(P) &= \textit{NatDist}(P, \mathcal{N}) - m[\![C]\!]_P && (15)\\
&\geq \nu^{<1} - ml^{<1} && (16)\\
&= \nu^{<1} + \nu^1 - \nu^1 + m - m - ml^{<1} && (17)\\
&= \nu^1 - m - (\nu^1 - \nu^{<1}) + m(1 - l^{<1}) && (18)\\
&> \nu^1 - m - (\nu^1 - \nu^{<1}) + \max\left\{0, \frac{\nu^1 - \nu^{<1}}{1 - l^{<1}}\right\}(1 - l^{<1}) && (19)\\
&= \nu^1 - m + \max\left\{-(\nu^1 - \nu^{<1}), -(\nu^1 - \nu^{<1}) + (1 - l^{<1})\frac{\nu^1 - \nu^{<1}}{1 - l^{<1}}\right\} && (20)\\
&= \nu^1 - m + \max\left\{-(\nu^1 - \nu^{<1}), 0\right\} && (21)\\
&\geq \nu^1 - m = \nu^1 - ml^1 = \mathcal{O}_m(P^{1'}) \geq \mathcal{O}_m(P^*). && (22)
\end{aligned}$$

In the above, Equation (16) follows because *NatDist*$(P, \mathcal{N}) \geq \nu^{<1}$ and $[\![C]\!]_P \leq l^{<1}$. Equation (19) follows because $m > M$. Equation (20) relies on the fact that $1 - l^{<1} > 0$, which follows from Equation (9), and similarly Equation (22) uses the fact that $l^1 = 1$.

We have shown that, for every P, if $P \in \mathcal{P}^{<1}$, then $\mathcal{O}_m(P) > \mathcal{O}_m(P^*)$. This implies that $P^* \notin \mathcal{P}^{<1}$. It must be that $P^* \in \mathcal{P}^1$, so $[\![C]\!]_{P^*} = 1$.

Finally, to show that P^* is most natural, consider P' such that $[\![C]\!]_{P'} = 1$. By definition of P^*, we have $\mathcal{O}_m(P^*) \leq \mathcal{O}_m(P')$. But

$$\textit{NatDist}(P^*, \mathcal{N}) - m = \mathcal{O}_m(P^*) \leq \mathcal{O}_m(P') = \textit{NatDist}(P', \mathcal{N}) - m.$$

Therefore *NatDist*$(P^*, \mathcal{N}) \leq$ *NatDist*$(P', \mathcal{N})$. □

D Algorithm for Natural Type Inference

Our algorithm has a couple of global parameters:

- We have an existing ambient library of type definitions:

$$\textbf{type}\, t_1 \;=\; S_1 \quad \ldots \quad \textbf{type}\, t_{\mathcal{T}} \;=\; S_{\mathcal{T}}$$

- We assume a probability distribution *NC* from texts to type names as defined in Appendix C.3.

We consider as input the untyped function definition:

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

and our goal is to find the most natural type signature $(t_1, \ldots, t_n, t)$ for f.

(1) Check that E is well-scoped with $fv(E) \subseteq \{x_1, \ldots, x_n\}$, and if not, terminate with an error.
(2) Let $\Gamma = x_1 : \beta_1, \ldots, x_n : \beta_n$ for fresh type variables β_i.
(3) Run the algorithmic typing rules to derive $\Gamma \vdash E \Rightarrow \beta\ (C, N_E)$.
(4) Let $A \triangleq \{\alpha_1, \ldots, \alpha_{\mathcal{V}}\}$ be the union of $\{\beta_1, \ldots, \beta_n, \beta\}$ with the set of type variables in C or N.
(5) Let $N = N_E \cup \{(f, \beta_1), \ldots, (x_n, \beta_n)\}$.
(6) For each type variable $\alpha_v \in A$, let $\boldsymbol{n}_v = NC(X)$ where $X = \{text \mid (text, \alpha_v) \in N\}$, the set of texts associated with the type variable α_v by N. Let $\mathcal{N} = \begin{bmatrix}\boldsymbol{n}_1^\mathsf{T} & \ldots & \boldsymbol{n}_{\mathcal{V}}^\mathsf{T}\end{bmatrix}^\mathsf{T}$.

(7) Decide whether C is satisfiable or not. If not, terminate with an error.

(8) Pick $m \geq 0$ and solve the unconstrained problem:

$$P = \underset{P \in \mathcal{P}^{\mathcal{V} \times \mathcal{T}}}{\operatorname{argmin}} \; \textit{NatDist}(P, \mathcal{N}) - m[\![C]\!]_P.$$

If the computed P satisfies the constraints, that is $[\![C]\!]_P = 1$, then we set $P^* = P$, and the optimum solution is found. Otherwise, increase m and repeat.

(9) Define a valuation Ω as follows: for each $v \in 1..\mathcal{V}$, we set $\Omega(\alpha_v) = t_\tau$, where $\tau \in 1..\mathcal{T}$ is the index of α_v's most likely type in $\boldsymbol{p}_v$, that is, the maximum in the $1 \times \mathcal{T}$ row-vector $\boldsymbol{p}_v$ (a component of P^*). If there is a tie between maximum values we choose one at random.

(10) Terminate successfully, and return $(\Omega(\beta_1), \ldots, \Omega(\beta_n), \Omega(\beta))$.

Theorem 5 (Correctness)**.** *Given a function definition,*

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

the algorithm terminates, and if it terminates successfully with a tuple $(t_1, \ldots, t_n, t)$*, then that tuple is a type signature for* f*.*

Proof. We prove termination first. Steps 1-2 terminate by construction. Step 3 terminates by Theorem 1 because we have checked that E is well-scoped and $\mathit{fv}(E) \subseteq \{x_1, \ldots, x_n\} = \mathrm{dom}(\Gamma)$). Steps 4-6 terminate by construction. To decide satisfiability of C we need to convert C to a CNF sentence and use classic algorithms like DDPL Cook and Mitchell [2000] and DDPL(T)Nieuwenhuis et al. [2004] to determine satisfiability. The DDPL algorithm always terminates and thus Step 7. Step 8 terminates because of Theorem 4. Steps 9–10 terminate by construction.

For the second part, suppose the algorithm terminates with a signature $(t_1, \ldots, t_n, t)$. By the final step, the signature must take the form $(\Omega(\beta_1), \ldots, \Omega(\beta_n), \Omega(\beta))$. We are to show that $x_1 : \Omega(\beta_1), \ldots, x_n : \Omega(\beta_1) \vdash E : \Omega(\beta)$.

Once the algorithm terminates we have from Theorem 4 that: $[\![C]\!]_{P^*} = 1$, where P^* is the output of the optimization step. The valuation Ω of Step 8 gives

$$\Pr[\Omega \mid P^*] = \prod_{v=1}^{\mathcal{V}} \max(\boldsymbol{p}_v) > 0$$

as the max of each probability vector $\boldsymbol{p}$ is greater than zero. Thus, we have $\Pr[\Omega \mid P^*] > 0$. By Theorem 3, $\Pr[\Omega \mid P^*] > 0$ and $[\![C]\!]_{P^*} = 1$ imply that $\Omega \models C$. Recall that $x_1 : \beta_1, \ldots, x_n : \beta_n \vdash E \Rightarrow \beta\ (C, N_E)$ from Step 2 of the algorithm. We have both $x_1 : \beta_1, \ldots, x_n : \beta_n \vdash E \Rightarrow \beta\ (C, N_E)$ and $\Omega \models C$, so, by Theorem 2(1), we have that $x_1 : \Omega(\beta_1), \ldots, x_n : \Omega(\beta_n) \vdash E : \Omega(\beta)$, as desired. □

E Related Work

This section first positions our type system in the type system zoo, then succinctly presents soft logic, how we use it and why. The bulk of the section surveys learning-based type inference systems, focusing on what they formally detail and how they are evaluated.

E.1 Type Systems

Pierce writes: "Type systems like Java's, in which names are significant and subtyping is explicitly declared, are called *nominal*Pierce [2002]." 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 saying 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.

E.2 Soft Logic

Recently, there is a resurgence of interest in *soft logic* in machine learning. By soft logic, we mean a many-valued logic where the truth values lie on the unit interval $[0, 1]$ Hájek [1998]. The reason for this resurgence is twofold: First, soft logic allows the modelling of multiple notions of similarity. Second, and more relevant for our interests, the resulting compound formulas are amenable to continuous optimization approaches. Thus, they provide a framework to exploit the relational structure of different problems Kimmig et al. [2012]. Rocktäschel *et al.* use soft logic to improve the neural inference of relations from textual data Rocktäschel et al. [2015].

Three widely used t-norms are Gödel logic Baaz [1996], Łukasiewicz logic Jaśkowski [1975], and product t-norm Hájek [1998], with the latter two attracting more interest. For example, Bach *et al.* use Łukasiewicz logic due to its convenient relationship with their relaxed MAX-SAT problem formulation Bach et al. [2017]. Notably, the product t-norm admits backpropagation Evans and Grefenstette [2018]. Two recent works compare and contrast these different t-norms on the task of learning loop invariants Ryan et al. [2020], Yao et al. [2020]. In our natural type inference problem, the logical constraints are non-convex and we focus on smooth optimization formulations, so we choose product logic to represent them (Appendix C.1), because it is smooth while the other two are not and would require relaxation.

E.3 Empirically Validated Work on 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 3.5 was released with optional type annotations and the Mypy type checker The-Mypy-Project [2014], or equivalent TypeScript Microsoft [2020], which adds optional type definitions to JavaScript.

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 a hierarchical neural network, consisting of two recurrent neural networks. 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. [2021]. Type4Py tersely describes its model and validates its performance empirically. Like DeepTyper, NL2Type and Type4Py are both unsound.

[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].

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.* use Xu et al. [2016]. 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.* present an algorithm to infer unit types for numbers in spreadsheets cells Williams et al. [2020]. 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. As a result, all are unsound. Recognising this problem, researchers proposed OptTyper and TypeWriter. OptTyper Pandi et al. [2020] reformulates learning-based type inference as an optimization problem to predicts types. OptTyper statically extracts logical and natural type constraints from a program, then combines them into a single joint optimization problem to infer type signatures. Its presentation formally presents type inference as optimization; its evaluation is empirical, comparing its performance to baselines using accuracy. It is OptTyper's approach that we formalize here. TypeWriter Pradel et al. [2019] 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. TypeWriter is sound 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. As is standard, the authors formalize TypeWriter's model and present its guess-and-check procedure in pseudocode and report its performance using information retrieval measures.

A Syntax-Directed Presentation of Declarative Type System

As discussed in Appendix A.3, this appendix presents an alternative syntax-directed set of rules for the judgment $\Gamma \vdash E : t$, and shows that they define the same relation as the rules in that section.

Definition 20 (Syntax-Directed Declarative Typing Rules)**.**

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

(Decl Expr x)

$$\frac{x \in \mathrm{dom}(\Gamma) \quad \Gamma(x) = t \quad t <:> t'}{\Gamma \vdash x : t'}$$

(Decl Expr b)

$$\frac{b \in \{\mathbf{true}, \mathbf{false}\} \quad t <:> \mathit{Bool}}{\Gamma \vdash b : t}$$

(Decl Expr c)

$$\frac{\mathit{integer}\ c \quad t <:> \mathit{Int}}{\Gamma \vdash c : t}$$

(Decl Expr −) $(t_1 <:> t_2 <:> \mathit{Int})$

$$\frac{\Gamma \vdash E_1 : t_1 \quad \Gamma \vdash E_2 : t_2 \quad t <:> \mathit{Int}}{\Gamma \vdash E_1 \ - \ E_2 : t}$$

(Decl Expr <) $(t_1 <:> t_2 <:> \mathit{Int})$

$$\frac{\Gamma \vdash E_1 : t_1 \quad \Gamma \vdash E_2 : t_2 \quad t <:> \mathit{Bool}}{\Gamma \vdash E_1 \ < \ E_2 : t}$$

(Decl Expr ==) $(t_1 <:> t_2 <:> \mathit{Bool}$ *or* $t_1 <:> t_2 <:> \mathit{Int})$

$$\frac{\Gamma \vdash E_1 : t_1 \quad \Gamma \vdash E_2 : t_2 \quad t <:> \mathit{Bool}}{\Gamma \vdash E_1 \ == \ E_2 : t}$$

(Decl Expr Rcd)

$$\frac{\Gamma \vdash E_i : t_i \quad \forall i \in 1..n \quad t <:> \{\ell_i : t_i\ ^{i \in 1..n}\}}{\Gamma \vdash \{\ell_i = E_i\ ^{i \in 1..n}\} : t}$$

(Decl Expr Proj) $(t <:> \{\ell_i : t_i\ ^{i \in 1..n}\})$

$$\frac{\Gamma \vdash E : t \quad j \in 1..n}{\Gamma \vdash E.\ell_j : t_j}$$

(Decl Expr If) $(t' <:> \mathit{Bool})$

$$\frac{\Gamma \vdash E_1 : t' \quad \Gamma \vdash E_2 : t \quad \Gamma \vdash E_3 : t}{\Gamma \vdash (E_1 \,?\, E_2 : E_3) : t}$$

(Decl Expr Let) $(x \notin \mathrm{dom}(\Gamma))$

$$\frac{\Gamma \vdash E_1 : t_1 \quad \Gamma, x : t_1 \vdash E_2 : t_2}{\Gamma \vdash \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2 : t_2}$$

(Decl Expr Lambda) $(x \notin \mathrm{dom}(\Gamma))$

$$\frac{\Gamma, x : t_1 \vdash E : t_2 \quad t <:> t_1 \to t_2}{\Gamma \vdash \lambda(x)E : t}$$

(Decl Expr Appl) $(t_2 <:> t_1 \to t)$

$$\frac{\Gamma \vdash E_2 : t_2 \quad \Gamma \vdash E_1 : t_1}{\Gamma \vdash E_2(E_1) : t}$$

Proposition 2. *The judgment* $\Gamma \vdash E : t$ *is derivable using the rules of Definition 4 if and only if* $\Gamma \vdash E : t$ *is derivable using the rules of Definition 20.*

B Small-Step Operational Semantics

The *reduction relation* $E \to E'$ 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 *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 21 (Reduction Rules)**.**

(Red-1 Oplus)

$$\frac{E_1 \to E_1' \quad \oplus \in \{-, <, >, ==\}}{E_1 \ \oplus \ E_2 \to E_1' \ \oplus \ E_2}$$

(Red-2 Oplus)

$$\frac{E_2 \to E_2'}{V_1 \ \oplus \ E_2 \to V_1 \ \oplus \ E_2'}$$

(Red −)

$$\frac{c_3 = c_1 - c_2}{c_1 \ - \ c_2 \to c_3}$$

(Red < True)

$$\frac{\textit{if } c_1 < c_2 \textit{ holds}}{c_1 \ < \ c_2 \to \mathbf{true}}$$

(Red < False)

$$\frac{\textit{if } c_1 < c_2 \textit{ does not hold}}{c_1 \ < \ c_2 \to \mathbf{false}}$$

(Red == True)

$$\frac{V_1 = V_2}{V_1 \ == \ V_2 \to \mathbf{true}}$$

(Red == False)

$$\frac{V_1 \neq V_2}{V_1 \ == \ V_2 \to \mathbf{false}}$$

(Red-1 Proj)

$$\frac{E \to E'}{E.\ell \to E'.\ell}$$

(Red-2 Proj)

$$\frac{j \in 1..n}{\{\ell_i = V_i\ ^{i \in 1..n}\}.\ell_j \to V_j}$$

(Red Rcd)

$$\frac{E_j \to E_j' \quad j \in 1..n}{\{\ell_i = V_i\ ^{i \in 1..j-1}, \ell_j = E_j, \ell_k = E_k\ ^{k \in j+1..n}\} \to \{\ell_i = V_i\ ^{i \in 1..j-1}, \ell_j = E_j', \ell_k = E_k\ ^{k \in j+1..n}\}}$$

(Red If)

$$\frac{E_1 \to E_1'}{E_1 \,?\, E_2 : \ E_3 \to E_1' \,?\, E_2 : \ E_3}$$

(Red If True)

$$\frac{}{\mathbf{true} \,?\, E_2 : \ E_3 \to E_2}$$

(Red If False)

$$\frac{}{\mathbf{false} \,?\, E_2 : \ E_3 \to E_3}$$

(Red-1 Let)

$$\frac{E_1 \to E_1'}{\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2 \to \mathbf{let}\ x = E_1'\ \mathbf{in}\ E_2}$$

(Red-2 Let)

$$\frac{}{\mathbf{let}\ x = V_1\ \mathbf{in}\ E_2 \to E_2[V_1/x]}$$

(Red-1 Lambda)

$$\frac{E_1 \to E_1'}{E_1(E_2) \to E_1'(E_2)}$$

(Red-2 Lambda)

$$\frac{E_2 \to E_2'}{V(E_2) \to V(E_2')}$$

(Red Appl)

$$\frac{}{(\lambda(x)E)(V) \to E[V/x]}$$

Theorem 6 (Preservation)**.** *If* $\Gamma \vdash E : t$ *and* $E \to E'$ *then* $\Gamma \vdash E' : t$*.*

Theorem 7 (Progress)**.** *If* $\varnothing \vdash E : t$ *either (1) there is a value* V *such that* $E = V$*, or (2) there is an expression* E' *such that* $E \to E'$*.*