In Search of a Type Theory for Fuzzy Properties

Youyou Cong

Tokyo Institute of Technology

1 Introduction

Types are a powerful language for describing properties. Over the past decades, researchers have used this aspect of types for diverse purposes, ranging from verification of compilers [14, 1], to representation of discourses [21, 2], and to formalization of music theory [24, 7]. These applications are all built on top of the *Curry-Howard isomorphism*, which relates types to propositions and programs to proofs.

Typically, whether an object satisfies a property is considered as a binary question. For instance, in the case of compiler verification, we say that a compiler is correct only if it is fully consistent with its specification. Such binary questions can be easily translated to a type inhabitation problem $e : \tau$, where e is a term representing the object we are interested in, and τ is a type encoding the property we would like e to satisfy.

On the other hand, there are cases where satisfaction of a property is not a binary question. As an example, in music composition, a choice of a note may be regarded better or worse than other choices, rather than absolutely good or bad. Such fuzzy properties require a type system with some form of *grades*, telling us to what extent an object satisfies a certain property.

In this abstract, we describe our work in progress on developing a type theory for expressing fuzzy properties. We begin by comparing three existing type systems with the notion of grades, designed for programming [3], linguistics [8], and music [6]. We then compare these systems to effect systems and coeffect systems, which allow grades-like annotations to appear in restricted places. It is our belief that a general theory of fuzzy properties will open up new possibilities of transferring knowledge between different areas.

2 Existing Type Systems with Grades

2.1 Precision Types

Boston et al. [3] implement *precision types* in the context of approximate computing, a technique for improving performance by allowing imprecise results. In their type system, types are annotated with a probability, representing how precise a value is. As an example, consider the `square` function below.

```
@Approx(0.8) int square(@Approx(0.9) int x) {
  @Approx(0.8) int xSquared = x * x;
  return xSquared;
}
```

In the signature of `square`, we state that the input `x` is precise with probability 0.9, and that we would like the output to be precise with probability 0.8. Assuming multiplication always produces a precise

result, we can deduce that the product `xSquared` is precise with probability $0.9 \times 0.9 = 0.81$. This probability is greater than the expected probability 0.8, hence the function is judged well-typed. Thus, using the information of probabilities, we can soundly reason about the quality of a computation.

2.2 Probabilistic Type Theory

Cooper et al. [8] formulate a *probabilistic type theory* for natural language semantics, with a special focus on the modelling of human cognition and learning. In their framework, typing judgments may be associated with a probability, representing how likely a situation is true. For instance, in a situation where we have two strawberries, three red apples, and five oranges, we have the following judgments:

$$\begin{aligned} p(a : \text{Apple}) &= 0.3 \\ p(a : \text{Red}) &= 0.5 \\ p(a : \text{Apple} \mid a : \text{Red}) &= 0.6 \end{aligned}$$

The first judgment states that an object a is an apple with probability 0.3, and similarly, the second one states that a is red with probability 0.5. The last judgment states that a is an apple with probability 0.6 given that a is red. Using judgments like these, we can learn classifiers of situations and compute semantic values of sentences.

2.3 Weighted Refinement Types

Cong [6] proposes *weighted refinement types* as a means to formalize the rules of counterpoint, a style of composition where one composes a melody against another melody. In a weighted refinement type, every refinement predicate is paired with a weight, representing the reward one gets by choosing an interval satisfying that predicate. To illustrate this idea, we give a partial encoding of the rules for composing first-species counterpoint [10] as well as two example compositions.

```
CP : Type
CP = List (Pitch * Interval) < (isImperfect @ 30), (isDissonant @ -100) >

cp1 : CP @ 150
cp1 = [(c, per8), (d, maj6), (e, min6), (f, maj3),
       (e, min3), (d, maj6), (c, per8)]

cp2 : CP @ -10
cp2 = [(c, per8), (d, per5), (e, min3), (f, aug4),
       (e, min6), (d, maj6), (c, per8)]
```

The type `CP` defines counterpoint as a list of pitch-interval pairs refined by two predicates, each of which is coupled with a reward. The rewards encode the guidance that imperfect intervals (*i.e.*, thirds and sixths) are preferred and dissonant ones (*i.e.*, seconds, fourths, and sevenths) should be avoided. By summing up the rewards, we can discuss the theoretical correctness of counterpoint compositions. In the above example, `cp1` is considered more correct than `cp2` as it has more imperfect intervals and no dissonant ones.

3 Generalizing Existing Type Systems

We have seen three type systems that express fuzzy properties by means of grades. Now we turn to our research question: How can we design a general theory that subsumes these type systems? We do

not have a concrete answer yet, but we conjecture that the resulting theory would share similarities with *effect systems* and *coeffect systems*.

Effect systems [16, 18] are a framework for tracking what side effect a program causes to the environment. Representative applications of effect systems include checked exceptions in Java, which prevent runtime errors, and the Cats Effect library of Scala, which guarantees resource safety of asynchronous programs. In an effect system, computation types or typing judgments carry an annotation, such as a set of exceptions and reading/writing actions. These annotations are similar to the precision on the output type of the `square` function from Section 2.1, and the probability on the judgments of situations from Section 2.2.

Dual to effect systems, coeffect systems [5, 20] are a framework for tracking what resource a program demands of the environment. Notable applications of coeffect systems include bounded variable use, where each variable can only be used a certain number of times, and secure information flow, where high-security data are guaranteed not to leak. In a coeffect system, variable bindings carry an annotation, such as the usage bound and security level. These annotations are similar to the precisions on the input types of the `square` function from Section 2.1.

There also exists a concept, called *graded modal types* [19], that allows simultaneous handling of effects and coeffects. In a system with graded modal types, computations and variable bindings may both carry an annotation, representing either a side effect or a resource demand. These annotations might subsume the three domain-specific annotations we saw in the previous section, including the rewards on refinement predicates we saw in Section 2.3 if we emulate refinement types as dependent pair types [23].

Once we have a uniform theory for fuzzy properties, we can explore opportunities to transfer techniques developed in one area to another area. For instance, in a type system that counts variable use, we can exploit the usage information to guide synthesis of programs [4]. By replacing variable use with probabilities, we could potentially extend this idea to perform proof search for sentences. As a different example, in a type system with probabilistic judgments, we can model incremental interpretation of utterances [12]. By viewing probabilities as rewards, we could possibly apply this idea to model incremental composition of music.

4 Related Work

The idea of grading types first appeared in the effect system of Lucassen and Gifford [16], which is designed for finding scheduling constraints of parallel computations. Since then, researchers have developed various graded type systems that enable fine-grained reasoning of programs [15, 22, 26], as well as techniques that make those systems usable in practice [25, 27].

In the context of natural language, grades have been used to give a compositional account of "linguistic effects", including anaphora resolution [11] and scope ambiguity [13]. We are however not aware of similar work on "linguistic coeffects", although there exist linguistic applications [9, 17] of a special case of coeffects (namely linear logic).

The recent years have also seen implementations of type systems with grades, such as Granule [19], Idris 2 [4], and Lambda VL [26]. These languages cannot currently express the three type systems discussed in Section 2, although their future versions may be able to do so by allowing the programmer to define custom grades.

5 Conclusion

We described our ongoing work on developing a general type theory for expressing fuzzy properties. As a first step toward this goal, we observed three type systems featuring grades and identified their

similarities to effect and coeffect systems. We hope our work will stimulate interesting discussions across research communities and lead to better understanding of fuzzy phenomena.

Acknowledgments

The author is grateful to the anonymous reviewers for their encouraging comments. This work was supported by JST, ACT-X Grant Number JPMJAX210C, Japan.

References

[1] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Z. Weaver. CertiCoq : A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages*, CoqPL '17, 2017.

[2] Daisuke Bekki and Koji Mineshima. Context-passing and underspecification in dependent type semantics. In *Modern Perspectives in Type-Theoretical Semantics*, pages 11–41. Springer, 2017.

[3] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Tuning approximate computations with constraint-based type inference. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[4] Edwin Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP '21)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[5] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, page 351–370, Berlin, Heidelberg, 2014. Springer-Verlag.

[6] Youyou Cong. Towards type-based music composition. In *The 11th International Workshop on Trends in Functional Programming in Education*, TFPIE 2022, 2022.

[7] Youyou Cong and John Leo. Demo: Counterpoint by construction. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design*, FARM '19, page 22–24, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Robin Cooper, Simon Dobnik, Shalom Lappin, and Staffan Larsson. Probabilistic type theory and natural language semantics. In *Linguistic Issues in Language Technology, Volume 10*, 2015.

[9] Mary Dalrymple, John Lamping, Fernando Pereira, and Vijay Saraswat. Linear logic for meaning assembly. In *Proceedings of the Workshop on Computational Logic for Natural Language Processing*, 1995.

[10] Johann Joseph Fux. *The Study of Counterpoint*. W. W. Norton & Company, 1965.

[11] Julian Grove and Jean-Philippe Bernardy. Algebraic effects for extensible dynamic semantics. *Journal of Logic, Language and Information*, pages 1–27, 2022.

[12] Julian Hough and Matthew Purver. Probabilistic record type lattices for incremental reference processing. In *Modern perspectives in type-theoretical semantics*, pages 189–222. Springer, 2017.

[13] Gregory M Kobele. The cooper storage idiom. *Journal of Logic, Language and Information*, 27(2):95–131, 2018.

[14] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 42–54, New York, NY, USA, 2006. Association for Computing Machinery.

[15] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, page 108–118, New York, NY, USA, 2000. Association for Computing Machinery.

[16] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[17] Bruno Mery. Lexical semantics with linear types. In *Third Workshop on Natural Language and Computer Science*, volume 32 of *LNCS '15*, page 12. EasyChair, 2015.

[18] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science (LNCS)*, pages 114–136, Berlin, Heidelberg, 1999. Springer-Verlag.

[19] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.

[20] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: A calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 123–135, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Aarne Ranta. *Type-Theoretical Grammar*. Oxford University Press, Inc., 1995.

[22] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 157–168, New York, NY, USA, 2010. Association for Computing Machinery.

[23] Gan Shen and Lindsey Kuper. Toward SMT-based refinement types in Agda. *arXiv preprint arXiv:2110.05771*, 2021.

[24] Dmitrij Szamozvancev and Michael B. Gale. Well-typed music does not sound wrong (experience report). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, page 99–104, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, jun 1994.

[26] Yudai Tanabe, Luthfan Anshar Lubis, Tomoyuki Aotani, and Hidehiko Masuhara. A functional programming language with versions. *The Art, Science, and Engineering of Programming*, 6, 2022.

[27] Yan Mei Tang and Pierre Jouvelot. Effect systems with subtyping. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '95, page 45–53, New York, NY, USA, 1995. Association for Computing Machinery.