Recursive Types for Modular Programming: Foundations, Algorithms, and Applications

LITAO ZHOU, The University of Hong Kong, China

Recursive types describe self-referential data structures and self-returning object interfaces, both commonly used in programming languages, but composing independently developed recursive components remains challenging. This talk revisits that challenge through a modular programming problem known as the Expression Problem. Iso-recursive subtyping provides a partial remedy, and we review a line of research towards a reusable metatheory of iso-recursive subtyping, covering declarative specifications, efficient algorithmic formulations, connections to equi-recursive types, and extensions to various type systems. Finally, we show that, with intersection types and a merge operator, the composition of recursive components reduces to a single missing subtyping law: distributivity of intersections over recursive types. We present our ongoing work on the algorithmic treatment of this law. Our formulation restricts recursive distributivity to weakly positive comparisons and generalizes the existing unary positivity check to set-based polarity tracking. We believe this metatheory can provide a foundation for extending and composing recursive components in practical type systems and programming languages.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Object oriented languages**.

Additional Key Words and Phrases: Recursive types, Iso-recursive subtyping, Modular programming, Expression Problem, Intersection types

1 Introduction

Recursive types describe self-referential data structures, such as algebraic data types, and objects whose methods return the object's own type, which are commonly used in programming languages. In type systems, they are formalized in two disciplines: equi-recursive and iso-recursive. *Equi-recursive* types equate a recursive type $\mu\alpha.A$ with its unfolding $A[\alpha \mapsto \mu\alpha.A]$, which provides a flexible and convenient addition to type systems without recursive types, but they also come with heavyweight metatheory challenges, such as coinductive reasoning [3], algorithmic complexity [5], and subtle interactions with other features [18]. *Iso-recursive* types relate the two only through explicit fold/unfold operators, and typically enjoy a lighter core metatheory.

Their classical subtyping discipline, Cardelli's Amber rules [1, 6], has been widely used, while a complete declarative specification and algorithmic formulations for the *iso*-recursive setting were developed more recently [31, 32]. That line of work makes it practical to revisit Amber-style iso-recursive subtyping in richer type systems: with bounded quantification [29, 30], with efficient algorithms [27], and—our focus in this talk—with distributivity of intersections over recursive types, applied to the modular composition of recursive components in the *Expression Problem* [23].

The Expression Problem. Consider the Haskell program below, which illustrates the Expression Problem extended with a recursive method that `doubles` the literal nodes of an expression AST.

```
1 data Exp = Lit Int | Add Exp Exp
2
3 eval :: Exp → Int
4 eval (Lit n)   = n
5 eval (Add l r) = eval l + eval r
6 double :: Exp → Exp
7 double (Lit n)   = Lit (n + n)
8 double (Add l r) =
9   Add (double l) (double r)
```

The Expression Problem describes a classic challenge in reconciling modularity and extensibility. For functional programming languages, adding a new method is easy, just by defining a new

Author's Contact Information: Litao Zhou, The University of Hong Kong, Hong Kong SAR, China, ltzhou@cs.hku.hk.

function over the existing datatype. However, adding a new constructor is less modular, as it requires duplicating edits on all the method definitions.

```
-- modular extension: add a new method
print :: Exp → String
print (Lit n)   = show n
print (Add l r) =
 "(" ++ print l ++ "+" ++ print r ++ ")"
```

```
-- non-modular: add a new constructor
data Exp = ... | Neg Exp   -- to replace line 1
eval (Neg e)   = - (eval e)     -- after line 5
double (Neg e) = Neg (double e) -- after line 9
```

Object-oriented languages dualize the problem—consider the same expressions as objects:

```
interface ExpI { eval : Int; double : ExpI }  -- double returns ExpI itself
class Lit(n: Int) implements ExpI {
  eval = n;               double = new Lit(n+n) }
class Add(l: ExpI, r: ExpI) implements ExpI {
  eval = l.eval + r.eval; double = new Add(l.double, r.double) }
```

Now the modularity of the two extensions flips: adding `Neg` is just defining one new class modularly, but `print` must edit `interface ExpI` and every class implementing it.

Analyzing the problem. We can read the two programs with iso-recursive types. The Haskell datatype reads as a recursive sum, $Exp \triangleq \mu\alpha.\, \mathsf{Int}+\alpha\times\alpha$, while in the OOP reading, the interface reads as a recursive record, $ExpI \triangleq \mu\alpha.\, \{\mathsf{eval} : \mathsf{Int}, \mathsf{double} : \alpha\}$, its knot tied by the self-returning `double`. In practical programming languages, fold and unfold are inserted automatically at constructor and pattern matching positions, so we omit them for brevity. In both readings the non-modular dimension is the one that changes the recursive type: in FP languages, adding the constructor `Neg` re-ties the knot at $Exp \triangleq \mu\alpha.\, \mathsf{Int} + \alpha \times \alpha + \alpha$, while in OOP languages, adding the method `print` re-ties the knot at $ExpI' \triangleq \mu\alpha.\, \{\mathsf{eval} : \mathsf{Int}, \mathsf{double} : \alpha, \mathsf{print} : \mathsf{String}\}$, a program typed at the old knot must be retyped at the new one.

Recursive subtyping as a partial remedy. It is natural to consider subtyping between iso-recursive types as a remedy. We focus on the OOP program. Amber-style iso-recursive subtyping relates one recursive type to another, i.e. $ExpI' \leqslant ExpI$ as follows: [1]

$$\mu\alpha.\{\mathsf{eval} : \mathsf{Int}, \mathsf{print} : \mathsf{String}, \mathsf{double} : \alpha\} \leqslant \mu\alpha.\{\mathsf{eval} : \mathsf{Int}, \mathsf{double} : \alpha\}.$$

Therefore, the `Add` class implemented on lines 4–5, which has type $ExpI \to ExpI \to ExpI$, can indeed be generalized by contravariant subtyping to have type $ExpI' \to ExpI' \to ExpI$. The programming intuition matches the typing: the generalization on the *arguments* comes for free—`Add` only depends on `eval` and `double`, and does not reject extra methods on its inputs—so objects of type $ExpI'$ may safely flow in. The *result* type, however, cannot be extended to $ExpI'$: `print` is simply not defined in the existing class, so the constructor still returns only $ExpI$. Amber recovers this natural one-sided flexibility, but the `print` extension itself is still built by editing every class, failing to solve the Expression Problem.

Internalizing modular composition with the merge operator. Compositional Programming (CP) [25] offers a way out: a method extension is written as one new self-contained component, and a *merge* operator $(e_1\, ,, e_2)$ [12], backed by distributive subtyping [2], composes components without touching existing code (we refer to [25] for the full design). In the style of our running example, the `print` extension becomes:

[1] We can understand record types as intersection types of single-field records, e.g. $\{\mathsf{eval} : \mathsf{Int}, \mathsf{double} : \alpha\}$ is defined as $\{\mathsf{eval} : \mathsf{Int}\}$ & $\{\mathsf{double} : \alpha\}$, so that the subtyping between the recursive body is just standard intersection subtyping.

```
interface PrintI { print : String }
class LitP(n: Int) implements PrintI { print = show n }
class AddP(l: PrintI, r: PrintI) implements PrintI { ... }  -- analogous
-- hope: new Lit(n) ,, new LitP(n)  implements  ExpI' = ExpI & PrintI
```

The missing law. Without recursive methods like `double`, CP would already type the hope above. Rule S-DistArr, the standard BCD-style distributivity rule for functions, pushes merges under constructors and composes the resulting interface field-wise. What CP cannot type is *recursive* composition: to merge $\mathit{PrintI} \triangleq \mu\alpha.\{\text{print} : \text{String}\}$ with *ExpI* to obtain the extended interface $\mathit{ExpI}' \triangleq \mu\alpha.\{\text{eval} : \text{Int}, \text{print} : \text{String}, \text{double} : \alpha\}$ (i.e. $\mathit{ExpI} \,\&\, \mathit{PrintI} \leqslant \mathit{ExpI}'$), we would need rule S-DistMu, distributivity of intersections over recursive types. To our knowledge, no existing iso-recursive subtyping system derives this law: Amber-style rules relate recursive types one-to-one and cannot synthesize one recursive type from two.

S-AndL
$$\frac{A_i \leqslant C}{A_1 \,\&\, A_2 \leqslant C}\ i \in \{1,2\}$$

S-DistArr
$$\frac{}{(A \to B) \,\&\, (A \to C) \leqslant A \to (B \,\&\, C)}$$

S-DistMu
$$\frac{}{(\mu\alpha.A) \,\&\, (\mu\alpha.B) \leqslant \mu\alpha.(A \,\&\, B)}$$

Related approaches and roadmap. Many other solutions to the Expression Problem work around the gap with source-level encodings [11, 16, 22], which we discuss further in §4. CP [25] approaches the Expression Problem from a different angle: by internalizing composition into intersection types, the merge operator, and subtyping, it aims to keep a simpler source-level design and provides a more straightforward solution to the Expression Problem. Our work aims to extend this design with support for recursive methods by developing the subtyping principles needed to compose components that each define their own recursive interface. The remainder of this abstract places this goal within a broader roadmap for iso-recursive types. We review the reusable metatheory on which it rests (§2), briefly discuss our ongoing work on formalizing distributivity over recursive types (§3), and conclude with a vision for extending these foundations to practical type systems and programming languages (§4).

2 A Reusable Metatheory of Iso-Recursive Subtyping

Foundations. The starting point is Cardelli's Amber rules, introduced informally in 1985 [6]. Amadio and Cardelli later developed the semantic theory of equi-recursive subtyping [1]; the iso-recursive Amber metatheory was revisited by Zhou et al.. They gave a declarative *finite-unfolding specification* and an algorithmic formulation, *nominal unfolding*, and proved them equivalent to the Amber rules, in which they introduced another intermediate presentation, *weakly positive subtyping*, to characterize the positivity of the recursion variable in a recursive comparison $\alpha \in_+ A \leqslant B$. Various extensions to the metatheory followed, including intersection types and bounded quantification, by extending the *nominal unfolding* formulation. However, the nominal unfolding variant does not extend naturally to distributivity over recursive types. As we will show next, our work adapts and extends the *weakly positive subtyping* formulation.

We present the research roadmap of iso-recursive subtyping in Figure 1. In addition to the foundation-focused work discussed above, iso-recursive subtyping has also been studied algorithmically in QuickSub [27]. Moreover, full iso-recursive types [28] form a separate branch, in which fold and unfold are generalized to erasable deep casts and recover the expressive power of equi-recursive types. A source-level encoding to solve the same recursive Expression Problem as in §1 has been proposed as an alternative encoding to CP based on full iso-recursive types [26], but requires a more explicit encoding with deep casts and polymorphic interfaces. This work instead pursues a direct subtyping abstraction, aiming to keep CP's clean source-level design and internalize the composition as foundation-level distributive subtyping.

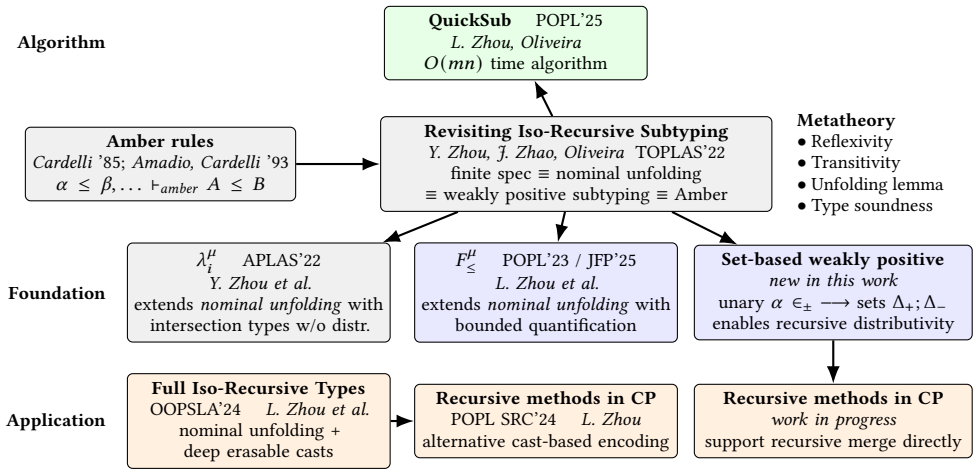

Fig. 1. Research roadmap organized by foundations, algorithms, and applications. Gray boxes denote prior work; colored boxes are the author's work.

3 Ongoing Work: Distributivity over Recursive Types, Algorithmically

We now return to the missing law S-DistMu and highlight two challenges in formulating it algorithmically.

Challenge 1: recursive distributivity is not an unrestricted equivalence. Rule S-DistArr is bidirectional. Its converse follows from S-AndL and the function subtyping rule. This enables algorithmic subtyping for BCD-style distributivity by splitting the type into two subtypes that can be subtyped separately [15, 19].

The analogous argument fails for S-DistMu. When the recursion variable occurs negatively, separately tying the recursive knots can create strict subtyping, so its two sides need not be equivalent. An unrestricted μ-splitter may replace a type by a strict subtype and fail to reconstruct even reflexivity. A natural path is to restrict S-DistMu to be applied only to recursive types that have positive polarity. This is still sufficient for the composition in §1, where recursive methods such as `double` return, rather than consume, the recursive interface.

Challenge 2: weak positivity must respect intersection backtracking. The characterization of whether a recursive variable acts positively during subtyping is subtle, as already discussed in [32], which defines weakly positive subtyping, with a *weakly positive restriction* that checks one recursion variable at a time: $\alpha \in_+ A \leqslant B$ means that for any $C \leqslant D$, substituting C and D for α on the two sides preserves the judgment: $A[\alpha \mapsto C] \leqslant B[\alpha \mapsto D]$ holds. This ensures that unfolding recursive types preserve the subtyping relation.

This unary check is insufficient in the presence of intersection subtyping. Rule S-AndL supports backtracking: to prove $(A_1 \,\&\, A_2) \leqslant B$, a derivation may drop either conjunct. Separate unary checks may therefore succeed by making incompatible choices: one variable through A_1 and another through A_2, even though no single subtyping derivation validates both variables.

Our solution is *set-based polarity tracking*. The judgment $\Delta_+; \Delta_- \vdash A \leqslant B$ tracks sets of variables that must occur weakly positively (resp. negatively) throughout a *single* derivation. Arrow cases flip the tracked sets. More importantly, an intersection-left rule makes one backtracking choice for the entire set, ensuring that all tracked variables follow the same conjunct.

With this set-based approach, we can formulate the restriction above precisely: a recursive type $\mu\alpha.T$ may be split into $(\mu\alpha.T_1)$ & $(\mu\alpha.T_2)$ precisely when the body T splits *and* the recursion variable is weakly positive in T up to the set-based tracking as described above. The proof and mechanization of this formulation are ongoing work.

We believe that weak positivity is a sufficient and useful restriction for the missing recursive distributivity law. A more permissive approach would be to admit negative occurrences, as the full power of S-DistMu allows. This would allow for binary methods [4], where the recursive variable occurs contravariantly but the corresponding components may still be mergeable. We leave this relaxation to future work.

4 Related Work and Outlook

The Expression Problem has accumulated many solutions within existing languages. Data Types à la Carte [22] delays tying the knot by taking the fixpoint of a coproduct of functor summands; object algebras [11] and finally tagless encodings [7] abstract over the carrier behind parameterized interfaces. In these library-based solutions, modular composition is *encoded at the source level*: programs are written against the encoding—injections, smart constructors, heavily parameterized carriers—rather than against the plain datatype. Polymorphic variants [16] and family polymorphism [14] instead make extensibility a language feature, and both can express recursive methods such as `double`—through explicit open recursion and manual knot-tying in the former, and through virtual classes with a more complex type system in the latter. Across all these solutions, recursive methods like `double` and binary methods [4] are the cases that demand the most machinery [24]. We believe our extension of iso-recursive subtyping to recursive distributivity and its integration into CP provides a principled and practical solution to the Expression Problem in the presence of recursive composition.

Outlook. A workable theory of distributivity over recursive types could also benefit settings beyond the Expression Problem, such as modular compiler passes over extensible ASTs and extensible object interfaces with self-returning methods. More broadly, the foundations of (iso-)recursive types connect to several practical language designs: recursive modules in module languages like Standard ML and OCaml [10, 13, 20], session types to support concurrent programming [8, 9, 17], and modern language eco-systems like the WebAssembly type system [21]. These systems face similar questions of how recursive types should be compared, unfolded, and modularly composed. We believe the foundations, algorithms, and applications reviewed in this talk lay the groundwork for answering these questions.

Acknowledgments

This research is supervised by Bruno C. d. S. Oliveira.

References

[1] Roberto M. Amadio and Luca Cardelli. 1993. Subtyping Recursive Types. *ACM Transactions on Programming Languages and Systems* 15, 4 (1993), 575–631. doi:10.1145/155183.155231

[2] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A filter lambda model and the completeness of type assignment1. *The journal of symbolic logic* 48, 4 (1983), 931–940.

[3] Michael Brandt and Fritz Henglein. 1998. Coinductive Axiomatization of Recursive Type Equality and Subtyping. *Fundamenta Informaticae* 33, 4 (1998), 309–338. doi:10.3233/FI-1998-33401

[4] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. 1995. On Binary Methods. *Theory Pract. Object Syst.* 1, 3 (1995), 221–242.

[5] Yufei Cai, Paolo G Giarrusso, and Klaus Ostermann. 2016. System F-omega with equirecursive types for datatype-generic programming. In *Principles of Programming Languages (POPL)*. 30–43.

[6] Luca Cardelli. 1985. Amber. *Combinators and Functional Programming Languages: Thirteenth Spring School of the LITP* (1985). doi:10.1007/3-540-17184-3

[7] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. doi:10.1017/S0956796809007205

[8] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. 2009. Foundations of session types. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*. 219–230.

[9] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2014. On the preciseness of subtyping in session types. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. 135–146.

[10] Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *Programming Language Design and Implementation (PLDI)*. 50–63. doi:10.1145/301618.301641

[11] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *European Conference on Object-Oriented Programming (ECOOP)*. 2–27. doi:10.1007/978-3-642-31057-7_2

[12] Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*. 364–377. doi:10.1145/2951913.2951945

[13] Derek R Dreyer, Robert Harper, and Karl Crary. 2001. *Toward a practical type theory for recursive modules*. Technical Report.

[14] Erik Ernst. 2004. The expression problem, Scandinavian style. *On Mechanisms For Specialization* 27 (2004).

[15] Andong Fan, Xuejing Huang, Han Xu, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2022. Direct Foundations for Compositional Programming. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:28. doi:10.4230/LIPICS.ECOOP.2022.18

[16] Jacques Garrigue. 2000. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Vol. 13.

[17] Simon J. Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2-3 (2005), 191–225. doi:10.1007/s00236-005-0177-z

[18] Giorgio Ghelli. 1993. Recursive Types Are Not Conservative over $F_{\leq}$. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 146–162. doi:10.1007/BFb0037104

[19] Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2021. Taming the Merge Operator. *J. Funct. Program.* 31 (2021), e28. doi:10.1017/S0956796821000186

[20] Keiko Nakata and Jacques Garrigue. 2006. Recursive modules for programming. *ACM SIGPLAN Notices* 41, 9 (2006), 74–86.

[21] Andreas Rossberg. 2023. Mutually Iso-Recursive Subtyping. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2, Article 234 (2023). doi:10.1145/3622809

[22] Wouter Swierstra. 2008. Data Types à la Carte. *Journal of Functional Programming* 18, 4 (2008), 423–436. doi:10.1017/S0956796808006758

[23] Philip Wadler. 1998. The expression problem. *Java-genericity mailing list* (1998).

[24] Matthias Zenger and Martin Odersky. 2005. Independently Extensible Solutions to the Expression Problem. In *FOOL*. http://zenger.org/papers/fool05.pdf

[25] Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021. Compositional Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 3 (2021), 1–61.

[26] Litao Zhou. 2024. Compositional Programming with Full Iso-recursive Types. (2024). https://ltzhou.com/static/POPL24SRC.pdf

[27] Litao Zhou and Bruno C. d. S. Oliveira. 2025. QuickSub: Efficient Iso-Recursive Subtyping. *Proceedings of the ACM on Programming Languages* 9, POPL, Article 20 (2025). doi:10.1145/3704869

[28] Litao Zhou, Qianyong Wan, and Bruno C. d. S. Oliveira. 2024. Full Iso-Recursive Types. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2, Article 278 (2024). doi:10.1145/3689718

[29] Litao Zhou, Yaoda Zhou, and Bruno C. d. S. Oliveira. 2023. Recursive Subtyping for All. *Proc. ACM Program. Lang.* 7, POPL (2023), 1396–1425. doi:10.1145/3571241

[30] Litao Zhou, Yaoda Zhou, Qianyong Wan, and Bruno C. d. S. Oliveira. 2025. Recursive Subtyping for All. *Journal of Functional Programming* (2025). doi:10.1017/S0956796825000036 Extended version of POPL 2023.

[31] Yaoda Zhou, Bruno C. d. S. Oliveira, and Andong Fan. 2022. A Calculus with Recursive Types, Record Concatenation and Subtyping. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 175–195. doi:10.1007/978-3-031-21037-2_9

[32] Yaoda Zhou, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2022. Revisiting Iso-Recursive Subtyping. *ACM Trans. Program. Lang. Syst.* 44, 4 (2022), 24:1–24:54. doi:10.1145/3549537