Compositional Programming with Full Iso-recursive Types

LITAO ZHOU, The University of Hong Kong, China

We propose a new formulation for typing iso-recursive types, by extending the fold and unfold operators in standard iso-recursive type systems to a *casting* operator that can transform terms between deep isomorphic types. We show that by integrating this design into the CP programming language, we can achieve a solution to the Expression Problem, with support for recursive methods.

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

Additional Key Words and Phrases: Iso-recursive types, Formalization, Expression Problem

1 EXPRESSION PROBLEM IN CP

We consider implementing an arithmetic expression datatype in Fig. 1. Our code is written in the CP programming language [18], that features compositional interfaces and nested trait composition, and provides a solution to the Expression Problem (EP) [16].

In CP, datatype constructors are modeled as functions that take datatype arguments as input and return all the methods on the datatype, as shown in Fig. 1b. One can naturally add new methods or new constructors by extending the compositional interface and providing a trait that implements the extension. For example, in Fig. 1c, `evalImpl` is defined for the method `eval` and constructors `Lit` and `Add`. The trait `evalImpl` can be combined with other components modularly and address EP, as we shall see.

```
type Exp = {
  eval   : Int,
  double : Exp,
};
```

(a) The interface we want to build

```
type NumSig = {
  Lit : Int → Exp;
  Add : Exp → Exp → Exp;
};
```

(b) The definition of datatype `Num`

```
type IEval = { eval : Int };
evalImpl = {
  Lit (val : Int) = { eval = val };
  Add (left right : IEval) =
    { eval = left.eval + right.eval }
};
-- evalImpl : {
--   Lit : Int → IEval
--   Add : IEval → IEval → IEval
-- }
```

(c) The implementation of the `eval` method and its type

Fig. 1. Motivating programming examples

CP has a structural type system that models objects as records, object types as record types. However, such formulation prevents us from expressing recursive methods. For example, in the interface of `Exp` in Fig. 1a, there is a `double` method that returns another arithmetic expression datatype with all the values in `Lit` nodes doubled. Such interface cannot be written in CP at present. One may naturally consider encoding the interface `Exp` as a recursive type: $\texttt{Exp} \triangleq \mu\,\alpha.\{\texttt{eval:Int, double:}\alpha\}$, and implement the method `double` as follows:

```
type IDouble = { double : Exp };
doubleImpl = fix self. {
  Lit (value : Int) = { double = self.Lit (value + value) };
  Add (left : Exp) (right : Exp) = { double = self.Add left.double right.double }
}; -- doubleImpl: {Lit: Int → IDouble, Add: Exp → Exp → IDouble}
```

However, if we naïvely extend CP with iso-recursive types, the composition below is not feasible!

```
implNum : NumSig = evalImpl ,, doubleImpl -- Type error!
```

The nested trait composition in CP is enabled by the merge operator (,,), which creates an intersection type of the two components `evalImpl` and `doubleImpl`. CP supports the BCD-style distributive subtyping on arrow types [1], and models record concatenation by intersections of single field record types, so if we look into the type checking process of `implNum`, we are left to transform `{eval: Int, double: Exp}` to `Exp`.

In this work we take an iso-recursive treatment of the recursive types, since it has an easier metatheory [22] and has demonstrated to be conservative to other programming features [20]. Iso-recursive types and their unfoldings should be explicitly converted via term-level `fold` and `unfold` constructs. For example, given an expression `e` of type `{eval: Int, double: Exp}`, `(fold [Exp] e)` gives the type `Exp`.

Nevertheless, in the example of `implNum` above, there are no ways to insert the `fold` operator, since we are implementing the components of this folded expression separately. The current iso-recursive type system [21] is not effective enough to modularly compose the recursive methods.

2 FULL ISO-RECURSIVE TYPES

The main idea of our solution is straightforward. Instead of allowing foldings/unfoldings on expressions of recursive types, we allow them to take place anywhere within an expression. Specifically, we introduce a new casting operator that represents the isomorphic folding/unfolding transformations between types, as described by the type casting rules in Fig. 2.

$$e ::= \ldots \mid \mathbf{cast}[c]e \qquad \textit{(Expressions)}$$
$$c ::= \mathsf{unfold}_A \mid \mathsf{fold}_A \mid \mathsf{id} \mid c_1 \to c_2 \mid \ldots \qquad \textit{(Casting operators)}$$

$\boxed{\vdash e : A}$ *(Full iso-recursive typing)*

$$\textsc{Typ-cast}\ \frac{\vdash e : A \qquad \vdash A \hookrightarrow B : c}{\vdash \mathbf{cast}\,[c]e : B}$$

$\boxed{\vdash A \hookrightarrow B : c}$ *(Type casting)*

$$\textsc{TCast-arrow}\ \frac{\vdash A_2 \hookrightarrow A_1 : c_1 \qquad \vdash B_1 \hookrightarrow B_2 : c_2}{\vdash A_1 \to B_1 \hookrightarrow A_2 \to B_2 : c_1 \to c_2}$$

$$\textsc{TCast-unfold}\ \frac{}{\vdash \mu\alpha.A \hookrightarrow A[\mu\alpha.A/\alpha] : \mathsf{unfold}_{\mu\alpha.A}}$$

$$\textsc{TCast-fold}\ \frac{}{\vdash A[\mu\alpha.A/\alpha] \hookrightarrow \mu\alpha.A : \mathsf{fold}_{\mu\alpha.A}}$$

$$\textsc{TCast-id}\ \frac{}{\vdash A \hookrightarrow A : \mathsf{id}}$$

Fig. 2. Selected typing and casting rules of full iso-recursive types

Compared to the standard iso-recursive typing rules, our type system is now able to express more terms. For example, with the casting operator

$$c_{\mathsf{Num}} \triangleq \{\mathtt{Lit} : \mathsf{id} \to \mathsf{fold}_{\mathsf{Exp}}\},,\{\mathtt{Add} : \mathsf{id} \to \mathsf{id} \to \mathsf{fold}_{\mathsf{Exp}}\}$$

we can achieve the desired typing $\mathbf{cast}\ [c_{\mathsf{Num}}]$ `implNum : NumSig`.

We have developed a type system for a calculus with record types, disjoint intersection types [8], BCD subtyping, iso-recursive types, and the casting operators. The calculus employs a call-by-value type-directed operational semantics [13, 14] and can encode all of the programming examples above. We formally prove the type safety of the type system in Coq.

3 NEXT STEP: MODULAR COMPOSITION VIA POLYMORPHISM

In our above implementation of `doubleImpl`, the input argument of the `Add` constructor is a fixed object interface `Exp`, which indicates that users need to decide what exact methods will be included in the interface in advance, and `doubleImpl` cannot be further composed with other constructors or methods. Therefore, we have not reached a complete solution to the EP. To fully support modular composition, we need polymorphism. Specifically, we modify the code as follows:

```
type NumSig[X] = { Lit: Int → X; Add: X → X → X };
type IDouble[X] = { double : X };
evalImpl (self: Top) = { ... }; -- evalImpl : Top → NumSig IEval
doubleImpl X (self: NumSig X) = {
  Lit (value : Int) = { double = self.Lit (value + value) };
  Add (left : IDouble X) (right : IDouble X) =
    { double = self.Add left.double right.double }
}; -- doubleImpl: forall X. NumSig X → NumSig (IDouble X)
```

Now the interfaces and implementations are parameterized by a type variable `X`. We can also add a new datatype constructor `Neg`:

```
type NegSig[X] = { Neg : X → X };
negImpl X (self: NegSig X) = { Neg (node : IEval & IDouble X ) = {
    eval = - node.eval;    double = self.Neg node.double
}}; -- negImpl: forall X. NegSig X → NegSig (IEval & IDouble X)
```

Finally, we can combine all the extensions by instantiating `X` to be the final recursive type of the object interface `Exp`, and insert a top-level casting operator to fold the recursive type at the correct places. Note that to support `self` reference in the implementations, we also need to close the expression by a general term-level fixpoint.

```
type Lang = { Lit: Int → Exp; Add: Exp → Exp → Exp; Neg: Exp → Exp };
langImpl : Lang = fix. (cast [id → (cNum ,, {Neg: id → foldExp})]
    (evalImpl ,,  doubleImpl Exp ,, negImpl Exp))
```

With polymorphism, our type casting relation can achieve the full power of modular composition of recursive methods. We expect polymorphism in our calculus to be feasible, based on prior works in disjoint polymorphism [2, 10]. The example demonstrates that we can address the Expression Problem in that both new constructors and new methods can be easily extended in our encoding.

4 RELATED WORK

There have been many solutions to EP in the literature and existing programming languages, including Object Algebras [7], ML module systems [15], family polymorphism [9, 19], polymorphic variants [12], the finally tagless encodings [5, 6], `super`-Charging OOP [11] and other techniques. Among those approaches, CP provides a natural solution which avoids the need of writing boilerplate or highly parameterized codes. This work addresses one of the most pressing extensions in CP. With full iso-recursive types, we have modeled the `double` operation described by [17] in this abstract. Moreover, the binary methods [3] can also be modeled.[1] It is worth noticing that recursive types are not the only approach to encoding object methods [4], but we believe that compared to others, full iso-recursive types are more natural to be embedded into the CP ecosystem.

ACKNOWLEDGMENTS

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

[1]The example in this abstract can be found at the online CP implementation https://plground.org/tony/binary.

REFERENCES

[1] 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.

[2] Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 381–409. https://doi.org/10.1007/978-3-030-17184-1_14

[3] 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.

[4] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. 1999. Comparing Object Encodings. *Inf. Comput.* 155, 1-2 (1999), 108–133. https://doi.org/10.1006/INCO.1999.2829

[5] Ralf Hinze Bruno C. d. S. Oliveira and Andres Loeh. 2007. Extensible and Modular Generics for the Masses. In *Trends in Functional Programming*, Henrik Nilsson (Ed.). Best student paper award.

[6] 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. https://doi.org/10.1017/S0956796809007205

[7] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7313)*, James Noble (Ed.). Springer, 2–27. https://doi.org/10.1007/978-3-642-31057-7_2

[8] Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 364–377. https://doi.org/10.1145/2951913.2951945

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

[10] 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. https://doi.org/10.4230/LIPICS.ECOOP.2022.18

[11] Andong Fan and Lionel Parreaux. 2023. super-Charging Object-Oriented Programming Through Precise Typing of Open Recursion. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:28. https://doi.org/10.4230/LIPICS.ECOOP.2023.11

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

[13] Xuejing Huang and Bruno C. d. S. Oliveira. 2020. A Type-Directed Operational Semantics For a Calculus with a Merge Operator. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26:1–26:32. https://doi.org/10.4230/LIPICS.ECOOP.2020.26

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

[15] Keiko Nakata and Jacques Garrigue. 2006. Recursive modules for programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, John H. Reppy and Julia Lawall (Eds.). ACM, 74–86. https://doi.org/10.1145/1159803.1159813

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

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

[18] 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.

[19] Yizhou Zhang and Andrew C. Myers. 2017. Familia: unifying interfaces, type classes, and family polymorphism. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 70:1–70:31. https://doi.org/10.1145/3133894

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

[21] Yaoda Zhou, Bruno C. d. S. Oliveira, and Andong Fan. 2022. A Calculus with Recursive Types, Record Concatenation and Subtyping. In *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13658)*, Ilya Sergey (Ed.). Springer, 175–195. https://doi.org/10.1007/978-3-031-21037-2_9

[22] 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. https://doi.org/10.1145/3549537