Simple Extensible Programming through Precisely-Typed Open Recursion

Andong Fan
The Hong Kong University of Science and Technology
Hong Kong, China

Abstract

In this abstract, we show that a small extension to the MLscript programming language gives a simple solution to the Expression Problem through precisely typed open recursion.

CCS Concepts: • **Software and its engineering → Object oriented languages**.

Keywords: modularity, union types, open recursion

ACM Reference Format:
Andong Fan. 2022. Simple Extensible Programming through Precisely-Typed Open Recursion. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '22), December 5–10, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3563768.3563951

1 The Expression Problem in MLscript

Consider the following motivating MLscript[1] [7] example, a minimal expression language that we will extend in several directions. We start by declaring expression constructors:

```
class Lit(value: Int)
class Add(lhs, rhs)
```

Note that the parameter types of `Add` are unspecified. Their ability to accept subexpressions of arbitrary types will be crucial to the extensibility of our approach, as we shall see.

```
trait EvalBase {
  eval: (Lit | Add('a, 'a) as 'a) → Int
  fun eval(e) = if e is
    Lit(n)        then n
    Add(lhs, rhs) then
      this.eval(lhs) + this.eval(rhs) }
```

[1]MLscript is a new programming language currently being developed at HKUST, available online at https://github.com/hkust-taco/mlscript/.

SPLASH Companion '22, December 5–10, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9901-2/22/12…$15.00
https://doi.org/10.1145/3563768.3563951

Here we define an `EvalBase` trait with an instance-matching method `eval` implementing expression evaluation. The type form `T as 'a` (which has least precedence, so here `T` is `Lit | Add('a, 'a)`) is a concise way of expressing recursive types; in negative positions, it is equivalent to `'a where 'a <: T`.

Now consider extending the code for pretty-printing:

```
trait PrettyBase {
 print: (Lit | Add('a, 'a) as 'a) → Str
 fun print(e) = if e is
   Lit(n)        then toString(n)
   Add(lhs, rhs) then
     this.print(lhs) ++ "+" ++ this.print(rhs) }
```

Next, consider another direction of code extension — defining a new expression constructor `Neg`:

```
class Neg(expr)
```

We then extend `EvalBase` and `PrettyBase` correspondingly:

```
trait EvalDerived extends EvalBase {
  override eval(e) =
    if e is Neg(d) then 0 - this.eval(d)
    else super.eval(e) }

trait PrettyDerived extends PrettyBase {
  override print(e) =
    if e is Neg(d) then
      "-(" ++ this.print(d) ++ ")"
    else super.print(e) }
```

The overriding methods `eval` and `print` implement evaluation and pretty-printing for negations, calling the base implementation for other expression types. Finally, we try to compose everything together:

```
val Lang = new EvalDerived with PrettyDerived
```

This may seem legitimate, but unfortunately it does not really work! For example, while `Lang.print(Neg(Lit(1)))` type checks, the following does *not*:

```
Lang.print(Add(Neg(Lit(1)), Lit(2)))
```

We get an error saying that `print` as declared in `PrettyBase` expects an argument of type `Lit | Add('a, 'a)` but receives `Neg(Lit(1))` instead. This is because MLscript infers the following insufficiently refined type for the overridden `print` definition of `PrettyDerived`:

```
print: (Neg('b) | (Lit | Add('a,'a) as 'a) as 'b) → Str
```

Notice that the recursive type variable `'a` only includes constructors `Add` and `Lit`. The refinement is only done on the outermost level of expressions: it is a *shallow* refinement, while we need a *deep* one. Indeed, MLscript has no way of

knowing that the `print` implementation of `PrettyBase` uses open recursion and *actually* handles `Neg` cases once its recursive calls are overridden — the type annotation of `eval` in `EvalBase` does not tell the full story!

2 Precise Typing of Open Recursion

We copy the *syntax* of method types with receivers from Kotlin: type $A.(B) \to C$ denotes a function with receiver A expecting B arguments and returning C. In Kotlin, this is used for extension methods, but it means something different in MLscript, where it reflects that in JavaScript, one can extract methods from classes and pass the receiver explicitly:

```
class C { val x = 2; fun foo(y) = this.x + y }
mth = C.foo // we have mth: C.(Int) → Int
mth.call(new C, 3) // statically type checked
```

As we shall see, we can also use receiver types to refine the type signatures of methods in the presence of open recursion.

2.1 Fixing The Motivating Example

We can now fix the motivating example of Section 1, which only requires small changes in `EvalBase` and `PrettyBase`:

```
// In trait EvalBase:
eval: {eval: 'A → Int}.(Lit | Add('A, 'A)) → Int

// In trait PrettyBase:
print: {print: 'A → Str}.(Lit | Add('A, 'A)) → Str
```

These signatures refine the implicit `this` parameter to contain an `eval` (or `print`) method accepting *any* argument of type variable `'A`, which corresponds to the type of subexpressions found in the `Add` constructor. Crucially, the individual `eval` method is *not* polymorphic in `'A` — since `'A` occurs in the *receiver part* of the method signature, it is considered quantified *at the class level*, and not at the method level. Class-level type variables can be understood as hidden type parameters which get constrained implicitly, and they are instantiated upon creation of objects.

By providing these signatures, we effectively *open* the types of these recursive functions so that they can later be refined in subclasses. Notice that these types are no longer recursive! Indeed, the recursive types only reappear once we *tie the knots* upon computing the final method types associated with each class, as we shall explain next.

Our motivating example now works perfectly, even though we did not change the definitions of the derived traits.

2.2 Type Checking Open Recursion

After type checking the contents of a class, the MLscript compiler makes sure that its class-level type variables are consistent with the receiver refinements used in the class — otherwise it means the class cannot be instantiated and an error should be reported. To do this, we simply make sure that `this` can be made a subtype of each receiver refinements. In the case of `EvalBase`, this means constraining `'A` such that `(Lit | Add('A, 'A)) → Int` (the `eval` function type) be a subtype of `'A → Int` (its receiver refinement), which results in the constraint `'A <: Lit | Add('A, 'A)`, recovering the original recursive type we had in Section 1.

To type check the `print` override in `PrettyDerived`, since no explicit signature is provided, we first assign an inference type variable `?p` to `this.print`. Moreover, we assign to `super.print` the method type inherited from `PrettyBase`, i.e., `(Lit | Add('A, 'A)) → Str`, together with the constraint `this <: {print: 'A → Str}` from the corresponding receiver refinement, reducing to `?p <: 'A → Str`. Taking all these together, the MLscript compiler infers the following principal type for `print` (we omit the details):

```
print: 'A → Str  where 'A <: Neg('A) | Lit | Add('A, 'A)
```

This is precisely the *deeply* refined method type we needed to type check the problematic example in the previous section.

Theoretically, MLscript could always infer receiver type refinements, making the motivating example work without any type annotations. We refrain from doing this and even require that all overridden methods be given explicit type signatures to mitigate the "fragile base class" problem — it is better to ask programmers to explicitly specify at which type a method is expected to be overridden, if any. Moreover, inferring receiver refinements can be counter-productive, as it may mask errors in methods definitions and report them at call sites instead.

2.3 Going Further

Our technique generalizes well to other forms of precisely-typed extensible programming. Consider an openly-recursive normalization method that deeply *reconstructs* expression:

```
type Exp<E> = Lit | Add(E, E) | Sub(E, E) | Neg(E)
n: {n: 'A → 'B}.(Exp<'A>) → Exp<'B>
```

We can override this definition in a subclass to *narrow down* the shape of the returned AST:

```
n: {n: 'A → 'B}.(Exp<'A>) → (Lit | Add('B, 'B) | Neg('B))
```

an implementation of which would look like:

```
override norm(e) = if super.norm(e) as ne is
  Sub(l, r) then Add(l, Neg(r)) else ne
```

3 Related Work

There is a sea of work in extensible programming, based on techniques such as polymorphic variants [3] in OCaml, recursive modules [5] in ML, and new programming paradigms [1, 6] like Compositional Programming [8]. The *polymorphic variants* solution [4] probably comes closest to our MLscript technique. But in the former, open recursion is implemented by passing an explicit parameter for the recursive call and manually tying recursive knots at the end, which is verbose and less natural. Moreover, polymorphic variants have many limitations [2], that can be fixed by embracing "proper" implicit subtyping as in MLscript [7]. In particular, we argue that union types are simpler than row polymorphism, which tries to imperfectly emulate subtyping through unification.

References

[1] 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 (sep 2009), 509–543. https://doi.org/10.1017/S0956796809007205

[2] Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyundefinedn. 2016. Set-Theoretic Types for Polymorphic Variants. *SIGPLAN Not.* 51, 9 (sep 2016), 378–391. https://doi.org/10.1145/3022670.2951928

[3] Jacques Garrigue. 1998. Programming with polymorphic variants. In *In ACM Workshop on ML*.

[4] Jacques Garrigue. 2000. Code reuse through polymorphic variants. In *In Workshop on Foundations of Software Engineering*.

[5] Keiko Nakata and Jacques Garrigue. 2006. Recursive Modules for Programming. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. Association for Computing Machinery, New York, NY, USA, 74–86. https://doi.org/10.1145/1159803.1159813

[6] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Beijing, China) *(ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/978-3-642-31057-7_2

[7] Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 141 (2022). https://doi.org/10.1145/3563304

[8] Weixin Zhang, Yaozhu Sun, and Bruno C. D. S. Oliveira. 2021. Compositional Programming. *ACM Trans. Program. Lang. Syst.* 43, 3, Article 9 (sep 2021), 61 pages. https://doi.org/10.1145/3460228