Language Composition through Product Extension and Its Use Cases for DSL Development

Marten Voorberg (voorberg@kth.se)

December 4, 2024 — KTH Royal Institute of Technology

Background

What are Sum and Product Types?

A **Sum Type** represents a value that takes on *one* of several types.
Sum Types are also called **Variants**, **Disjoint Unions**, and more

```
syn MaybeInt = None () | Some Int
syn IntList = Nil () | (Int, IntList)
syn Expr = TmInt Int | TmAdd (Expr, Expr)
```

A **Product Type** represents a value that consists of *multiple* values with different types.
Tuples are unlabelled product types
Records are labelled product types

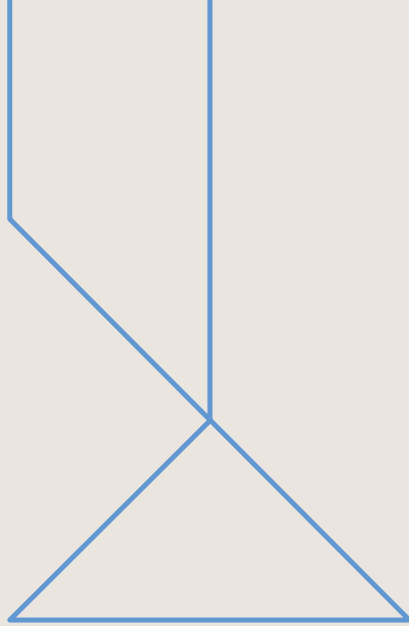
```
type IntPair = (Int, Int)
type Point = {x : Int, y : Int}
type TypeCheckEnv = {
    tyVars : Set String,
    varMap : Map String Type
}
```

Sum Extension in MLang

```
lang Arith
    syn Expr =
    | TmInt {val : Int}
    | TmAdd {lhs : Expr, rhs : Expr}

    sem eval =
    | TmInt t -> t.val
    | TmAdd t -> addi (eval t.lhs) (eval t.rhs)
end
```

```
lang ExtendedArith = Arith
    syn Expr +=
    | TmNeg {e : Expr}



    sem eval +=
    | TmNeg t -> negi (eval t.e)
end
```

The **extensibility of sum types** facilitates the re-use of languages!



Extensible Product Types

Use Case 1: Extensible Environments and Contexts

Suppose we would like to add **type checking** to our DSL.

`3 + 1`	`3 + "bar"`	`f 1`	`Foo 1 "bar"`
`-`	`-`	`Map Str Type`	`Map Str [Type]`

- But what is the type of the context?
- Since we combine multiple values, it should be a **product type**!
- When we extend the language, we need to be able to **extend this product type**!

Use Case 1: Extensible Environments and Contexts Cont'd

```
lang TypeCheck = Base
    erec Ctx = {}

    sem typeCheck : Ctx -> Expr -> Type
end
```

```
lang ArithTypeCheck = TypeCheck + Arith
    sem TypeCheck ctx +=
    | TmAdd t -> ...
end
```

```
lang FunTypeCheck = TypeCheck + FunApp
    erec Ctx *= {funType : Map String Type}

    sem TypeCheck ctx +=
    | TmFunApp t -> ...
end
```

```
lang ConTypeCheck = TypeCheck + ConApp
    erec Ctx *= {conType : Map String [Type]}

    sem TypeCheck ctx +=
    | TmConApp t -> ...
end
```

Use Case 2: Extending Constructor Payloads

$\lambda f.\, \lambda x.\, f(f\ x)$

```
lang LambdaCalculus
    syn Expr =
    | TmVar {id : String}
    | TmApp {lhs : Expr, rhs : Expr}
    | TmAbs {id : String, body : Expr}
    | TmInt {val : Int}
    | TmAdd {lhs : Expr, rhs : Expr}
end
```

$\lambda f : (\mathrm{Int} \to \mathrm{Int}).\, \lambda x : \mathrm{Int}.\, f(f\ x)$

```
lang STLC = LambdaCalculus
    syn Type =
    | TyArrow {lhs : Type, rhs : Type}
    | TyInt {}

    syn Expr *=
    | TmAbs {tyAnnot : Type}
end
```

Product extension of syntax types facilitates the extension of **existing constructs** with **additional information**

Conclusion

Conclusions

- Two use cases of product extension in DSL development:
  - Extensible environment or context types.
  - Extension of existing language constructs with additional information.
- Extensible records are a part of a larger upgrade to MLang including a stronger type system, co-semantic functions, and more.
- Currently, extensible records are an experimental feature.
- The presented work was done as a master thesis in computer science. Get in touch!


KTH
VETENSKAP
OCH KONST