An exploration of dependent types for data-centric programming

Introduction

Type systems define rules that assign types to expressions of a language, and can be backed by formalizations called **type theories**. The first (Russell, 1908) was devised to be a logical foundation of mathematics, though it had issues. **Dependently-typed** programming languages restore this goal with the "propositions as types" motto (Wadler, 2015), in which types correspond to mathematical propositions and programs to their proofs. As such, they are used to both write programs and formally verify them.

Objectives

- Study dependent types from the ground up;
- Apply a dependently-typed language to develop a verified data frame implementation.

Untyped λ-calculus

Most type theories have been developed within the λ-calculus, a formal system for computation. Its untyped variant has the following syntax and evaluation:

$$\begin{aligned} \lambda\text{-term } E &::= x && \text{(variable)} \\ &\mid E\,E && \text{(application)} \\ &\mid \lambda x.E && \text{(abstraction)} \end{aligned}$$

Definition (β-reduction). $((\lambda x.M)N) \to_\beta M[x := N]$

The untyped λ-calculus is Turing-complete (Turing, 1936), and as such, can be used to write any program. This also means evaluation of terms may never halt, so a term may not have a *normal form*. This makes it unfit as a proof system, since it can lead to inconsistency.

Simply-typed λ-calculus (STLC)

Adding types to λ-terms forces their evaluation to halt.

$$\begin{aligned} \text{type } T &::= \alpha && \text{(constant)} \\ &\mid T \to T && \text{(function)} \end{aligned} \qquad \begin{aligned} E &::= \ldots \\ &\mid \lambda x : T.E \end{aligned}$$

For each syntactic variant of a λ-term, we define a type assignment rule through Gentzen's sequent calculus:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash M : S \to T \quad \Gamma \vdash N : S}{\Gamma \vdash M\,N : T} \qquad \frac{\Gamma, x : S \vdash M : T}{\Gamma \vdash \lambda x.M : S \to T}$$

This calculus is equivalent to the implicational intuitionistic propositional calculus. Variations of STLC add polymorphism through type quantification, allowing types to depend on other types.

Dependently-typed λ-calculus

Dependent type theories allow types to also depend on terms. This way, functions can be from terms to terms, types to terms and vice-versa, and types to types. One is the Calculus of Constructions (Coquand and Huet, 1988).

Below is code in Lean 4 with a proof for $A \vee B \to B \vee A$:

```
theorem or_comm : A ∨ B → B ∨ A
  | Or.inl a => Or.inr a
  | Or.inr b => Or.inl b
```

Data-centric programming

In data-centric programming, the primary objective is to represent, query and transform data. There are many systems built with this purpose, such as SQL.

For this project, we decided to explore the Lean 4 programming language by implementing a framework for data frames akin to the pandas library (McKinney, 2010). We started by adapting a similar library for Idris 2 by Tejiščák (2020) and building upon it.

- Data frame representation

```
def exampleDF : DF
  ([ "name"    :- String
   , "age"     :- Int
   , "country" :- Option String
   ]) :=
  ⟨‖  "Artur" ¦    "Bob" ¦ "Claire" ¦ "Blorg" ¦ "Zorg" ¦
   ‖       32 ¦       54 ¦       41 ¦     101 ¦     99 ¦
   ‖ some "BR" ¦ some "US" ¦ some "FR" ¦   none ¦   none ¦⟩
```

- Queries and transformations

```
def alienAges := exampleDF
  |>.where (Option.isNone <$> col "country" (by repeat constructor))
  |>.select (
    • ("name"    :- col "name" (by repeat constructor))
    • ("age"     :- col "age"  (by repeat constructor))
    • ("century" :- over (col "age"  (by repeat constructor))
                         (FromInt.fromInt 100)))
  |>.orderBy [.asc (col "age" (by repeat constructor))]
  |>.get "century" (by repeat constructor)
```

Conclusion

Writing verified code is tricky, as the compiler might not always infer proofs. However, Lean's extensive macro system allows creating simpler interfaces as domain-specific languages, which improves the process. Our framework could be used to define critically important transformations, to later be translated to other languages.

References

- Coquand, Thierry and Gérard Huet (1988). "The calculus of constructions".
- McKinney, Wes (2010). "Data Structures for Statistical Computing in Python".
- Russell, Bertrand (1908). "Mathematical Logic as Based on the Theory of Types".
- Tejiščák, Matúš (2020). *idris-data-frame*.
- Turing, Alan M. (1936). "On Computable Numbers, with an Application to the Entscheidungsproblem".
- Wadler, Philip (Nov. 2015). "Propositions as types".

Eduardo Sandalo Porto (eduardo.sandalo@usp.br)

Supervisor: Prof. Ana Cristina Vieira de Melo

Department of Computer Science — University of São Paulo

IME