MiniDλ: A Simplified Lambda Calculus Interpreter, Dependently.

Hiroki Chen
Computer Science, Indiana University Bloomington, IN, USA
haobchen@iu.edu

Abstract

MiniDλ aims to be a simple interpreter for a variant of the lambda calculus extended with dependent types. This project involves parsing lambda expressions, type-checking them using the rules of dependent type theory and evaluating those expressions.

1 My Idea

Dependent types are a powerful feature in type systems that allow types to depend on values, which is specifically useful to enable the system to catch more bugs and prove more interesting properties. With dependent types, it's possible to encode more invariants and properties directly into the types of data structures and functions. For example, a type-checked program does not only mean that it contains no basic type error (like adding strings to numbers) but compiles with more sophisticated domain-specific rules (e.g., a list is sorted after it is obtained from a sorting algorithm). Due to these charming perks, many proof assistants (Coq, Agda, etc.) utilize dependent types for formal verification.

This necessitates a type checker that works for dependent types by extending, for instance, the *Simply Typed Lambda Calculus* (STLC). In this project, I want to implement a dependently typed lambda calculus containing the following features:

- **Support for basic DTs:** Vectors are one of the examples.
- **Extension to the inference rules:** Since STLC lacks inference rules for DTs, we must close this gap.
- **Proof for Type Safety (tentative):** We aim to use the proof assistant (Coq) for type safety, but this could be tricky and subtle, so I mark this as optional.

2 The Rationale

During the lectures we have been taught the notion of STLC and its type inference rules, the basic CEK machines, and the notion of DTs. One thing that seems missing, however, would be how to integrate DT into STLC, which is important as the modern FP family (like Haskell) heavily uses DTs in their core type systems. While STLC and DT are not something new, making them work in tandem is no trivial task that involves all the subtleties in type theory. Furthermore, we believe one only understands how type systems work by implementing a type checker hands-on.

3 Roadmap

- **Language Design and Scope:** This includes the basic syntax, type inference rules, and evaluation rules of STLC and those supporting DTs.

$$
\begin{array}{rll}
e ::= & e : \tau & \text{annotated terms} \\
| & \mathcal{U} & \text{type of types} \\
| & \forall x : \tau . \tau' & \text{dependent function space} \\
| & x & \text{variable} \\
| & e_1 e_2 & \text{application} \\
| & \lambda x . e & \text{abstraction}
\end{array}
\tag{1}
$$

However, the above definition cannot work well with DTs since values must appear inside types τ. There is no longer a syntactic distinction between these things [1]. Consider for example an inhabitant of the vector type:

$$
v : \forall a : \mathcal{U}, n : \mathbb{N}.\ \mathrm{Vec}\ a\ n \tag{2}
$$

Types are thus appearing directly inside each term.

$$
\begin{array}{rll}
e, \rho ::= & e : \rho & \text{annotated terms} \\
| & \mathcal{U} & \text{type of types} \\
| & \forall x : \rho . \rho' & \text{dependent function space} \\
| & x & \text{variable} \\
| & e_1 e_2 & \text{application} \\
| & \lambda x . e & \text{abstraction}
\end{array}
\tag{3}
$$

Now the types are something that can be evaluated, and the "normal forms" should also incorporate types.

$$
\begin{array}{rll}
v, \tau ::= & n & \text{Numbers, strings, etc.} \\
| & \mathcal{U} & \text{Type universe} \\
| & \forall x :: \tau . \tau' & \text{dependent function space} \\
| & \lambda x . e & \text{abstraction}
\end{array}
\tag{4}
$$

- **Implementation of the interpreter and type checker:** We use Rust as the main language because it allows very flexible and powerful pattern-matching on types. For example,

```rust
pub enum LambdaType {
    Basic(BasicType),
    Arrow{
        arg: Box<LambdaType>,
        ret: Box<LambdaType>,
    },
}

let ty = get_type();
match ty {
    Basic(bt) => println!("{bt:?}"),
    Arrow{arg, ret} => println!("{arg:?} -> {ret:?}"),
};
```

Such a shorthand makes development smoother. Rust's other fantastic features like built-in `Option` and `Result` types provide us with monad-like operations. While pure FP languages like Haskell may also be good candidates, the packages hosted at https://crates.io reduce our burden for pure engineering work.

References

[1] A. Löh, C. McBride, and W. Swierstra, "A tutorial implementation of a dependently typed lambda calculus," *Fundamenta informaticae*, vol. 102, no. 2, pp. 177–207, 2010.