Extensions on the MiniML Language

Camilo Brown-Pinilla

Computer Science 51

Harvard University

Introduction

MiniML is a Turing-complete subset of OCaml implemented in OCaml itself. Without extensions, the base version of MiniML only operates on expressions composed of integer and boolean values. There are only a handful of binary and unary operators defined on these values. The evaluation of MiniML expressions manifest either substitution or dynamically-scoped environment semantics. The extensions I present here augment this language by:

1. Adding support for new atomic types and operations on these new types, as well adding new operations on existing types
2. Introducing lexically scoped environment semantics, as seen manifested in the evalution of OCaml.

New Types and Operations

Types

As mentioned before, my extended implementation of MiniML now includes support for new atomic value types. Specifically, MiniML can now operate on floats, strings and chars in addition to bools and ints; all of the same atomic types found in OCaml save for units and bytes.

Operations

I have vastly expanded the operations supported in MiniML. I list their syntax and definition alongside the types they operate on, below:

- `^` Concatenation. Binary operator defined on strings
- `not` (Boolean) Negation. Unary operator defined on bools
- `&&` Boolean and. Binary operator defined on bools
- || Boolean or. Binary operator defined on bools
- `sin` Sine. Unary operator defined on floats
- `cos` Cosine. Unary operator defined on floats
- `**` Exponentiation. Binary operator defined on floats
- `/` Division. Binary operator on ints and floats
- `<>` Not equals. Binary operator on all types
- `>` Greater than. Binary operator on all types
- `>=` Greater than or equal to. Binary operator on all types
- `<=` Less than or equal to. Binary operator on all types

These operations are added on top of the operations defined in the base version of MiniML. Notably, MiniML syntax does not require users to use different operators for the same operation on different types (i.e. `*` vs `*.` for int and float multiplication, respectively. This can be seen as an improvement in usability, especially to beginner OCaml enthusiasts. Just like OCaml, all of these operations can still only operate on expressions of the same type.

Implementation

Implementation of these new types and operations followed exactly as the implementation of the base MiniML types and operators did. In addition to this, I augemented `miniml_lex.mll` and `miniml_parse.mly` to allow the parser to recognize my new definitions.

Demonstration

```
<== not (true || false) = false ;;
==> true


<== 42. ** cos 0. ;;
==> 42.


<== let c = "camilo" in let s51 = "lovesocaml" in c ^ s51 ;;
==> camilolovesocaml
```

Lexically scoped semantics

As mentioned in the introduction, MiniML now supports lexically scoped semantics and employs it by default, just as is done in OCaml.

Implementation

These semantics were implemented in much the same way as the substitution and dynamic semantics were. Indeed, lexically scoped and dynamically scoped semantics are almost the same, save for their handling of recursion, function evaluation, and function application.

Demonstration

There are many different subtle differences in evaluation between dynamically and lexically scoped semantics. Most easily seen, however, is the inclusion of closures in lexically scoped semantics. We see this behavior here:

```
<== let x = 1 in fun z -> z + 1 ;;
==> ((fun z -> z + 1), [(x, 1)])
```

Here, the closure of our function in the environment where x is mapped to 1 is printed.

Conclusion

By adding support for new types and operators and lexically scoped semantics, my extensions to the MiniML language make it much more similar to OCaml and, in doing so, make it much more usuable than the base MiniML. One nice improvement over standard OCaml is the lack of a need to differentiate between the same operators on different types (`*.` vs `*` for example). While my implementation of MiniML is still rather bare-bones, the software lends itself to further extension nicely. Implementing this language was difficult, but doing so greatly increased my appreciation for all the work done under the hood in parsing and evaluating code.