CSE 341 AB: Section 7

Josh Pollock

Office Hours: Tuesdays 3:00pm - 4:00pm

Questions?

Agenda

1. Interpreters and HW 5
    a. What is an interpreter?
    b. Well-formed MUPL programs
    c. Structs and type tags
    d. MUPL macros
2. Code as Data
    a. Quoting
    b. Quasiquoting
    c. Partial Evaluation

Interpreters and HW 5

Remark: Implementing recursive functions in MUPL is tricky.

What Is an Interpreter?

Recursively evaluates a source program (and runtime data) to a value.

Two flavors

- Small Step
  - One call to the function does one step of execution.
  - How we trace programs.
  - Recall abstract machine example from a few weeks ago.

- **Big Step (HW 5)**
  - One call to the function does the entire execution.
  - Really just chaining a bunch of small steps together implicitly with recursion.

Well-Formed Programs

Imagine that the AST is typed SML ADT.

- **`var`** always holds a string.
- **`int`** always holds a number.
- **`add`** always holds two expressions. (They don't necessarily evaluate to **`int`**s)
- …

Instead of a type system enforcing these, our nonexistent parser might ensure it.

It's fine for ill-formed programs to crash your interpreter.

Structs and Type Tags

A struct introduces a new type tag.

The type tag of a struct **`struct`** can be checked with **`struct?`**

`eval-exp` checks the type tags of its fields, **`eval-exp-wrong`** does not.

Explicitly checking tags lets you throw custom errors instead of contract violations.

Use custom errors rather than contract violations for <u>well-formed</u> programs!

Writing Macros in MUPL

Because we are manipulating MUPL ASTs, we can write functions that take in MUPL ASTs and return new MUPL ASTs.

These are macros for our language MUPL, but they aren’t hygienic.

Code as Data

In Lisps, It’s Easy to Manipulate Code

Code and lists have (almost) the same syntax.

This makes it easy to write code that manipulates other code.

But we need a way to turn code into data and back again.

Quoting

Racket provides functions for converting code to a list of tokens and vice versa.

`quote`: parses its argument as data

`eval`: takes data and evaluates it

```
'e := (quote e)

(eval (quote e)) = e
```

This is why you can use quotes for lists.

Quasiquoting

Sometimes we want to evaluate subexpressions before we quote them.

`quasiquote`: like quote, but allows unquoting inside

`unquote`: code inside unquote is evaluated before it's tokenized.
Can only be used inside **`quasiquote`***.*

```
`e := (quasiquote e)

(eval (quasiquote e)) = e

,e := (unquote e)
```

Partial Evaluation

Quoting allows us to *stage* computation by delaying evaluation.

We can run different parts of our program at different times.

Partial Evaluation

- We can *specialize* a program if we know some of the inputs.
- Sometimes makes code faster and/or smaller.
- This is called **partial evaluation**.
- Different than partial application, because we can evaluate under the lambda.

There are some interesting results regarding partial evaluation including connections between interpreters and compilers.
See the Futamura projections.