CSE 341: Programming Languages

Dan Grossman
Spring 2008
Lecture 19—DrScheme modules; abstraction with dynamic types; function equivalences
Modularity

Recall from our ML module lecture some good things about modules:

- Namespace management (help keep names short and separate)
- Make some bindings inaccessible (private functions, data)
- Enforce invariants by using abstract types
  - Data is reachable, but outside the module only limited things can be done with it
- In our example:
  - Rationals are always printed in reduced form.
  - Clients can’t tell if rationals are kept in reduced form.
Scheme vs. DrScheme

“Pure” Scheme (R5RS) has no module system or define-struct

- We’ll investigate how much of modules’ advantages we can get via other means

DrScheme has a module system

- But in a dynamically typed language, there won’t be signatures with abstract types

- We can get abstract types using define-struct instead
  - Because it makes a new type not equal to any other type
  - Quite different than ML approach but both work
Life without modules

- Can hide private things using `let`
  - Workable but awkward
  - Making the `define-struct` “private” is a huge help
The key to define-struct

It is essential to hiding parts of a define-struct that it is a fresh, different type than any other type.

• In our example, hid the accessors, mutators, and constructor.

• Sometimes exposing some accessors makes sense.

Otherwise, someone could use other features (e.g., cons or set-car!) to violate invariants.

It is still the case that any Scheme function can be called with any argument, but we can control invariants on rationals.
DrScheme modules

• provide for explicit list of what is available outside
  – Can be “part” of define-struct
  – Kind of like “part” of an ML datatype (kind of)

• require for using another module
  – With optional prefixing of names for namespace management
Function equivalences

There are 3 very general things you can do with functions that produce equivalent code. Recognizing them (and their subtle caveats) can make you a better programmer.

1. Systematic renaming of variables
2. “Inlining” by replacing a function call with a body + substitutions
3. Unnecessary function wrapping

Before considering each, it will help to define carefully the notion of free variables...
Free variables

An expression \( e \) has a set of free variables. The definition is:

- For each use of a variable, find the binding that defines that variable. (This uses the language’s scope rules.)
- If there is a use of \( x \) that is in \( e \) whose corresponding binding is outside \( e \), then \( x \) is in the free variables of \( e \).

Example:

```ml
fun f x =
  let val w = x + y
  val y = fn x => z + y + x
  val q = w + x
  in if g w then x+4 else f (x-1) end
```
Systematic Renaming

Is $\text{fn } x => e1$ is equivalent to $\text{fn } y => e2$ where $e2$ is $e1$ with every $x$ replaced by $y$?

(Generally a good property of languages; callers unaffected by code maintenance in callee.)
Scope matters

Is \( fn \ x \Rightarrow e_1 \) is equivalent to \( fn \ y \Rightarrow e_2 \) where \( e_2 \) is \( e_1 \) with every \( x \) replaced by \( y \)?

What if \( e_1 \) is \( y \)?

What if \( e_1 \) is \( fn \ x \Rightarrow x \)?

Need caveats:

\( fn \ x \Rightarrow e_1 \) is equivalent to \( fn \ y \Rightarrow e_2 \) where \( e_2 \) is \( e_1 \) with every free \( x \) replaced by \( y \).

But only if \( y \) is not already free in \( e_1 \)!
Inlining

Is \((\text{fn } x \Rightarrow e_1) \ e_2\) equivalent to \(e_3\) where \(e_3\) is \(e_1\) with every \(x\) replaced by \(e_2\)?

Example: Replace \((\text{fn } x \Rightarrow x+x) \ (2+3)\) with \((2+3) + (2+3)\)

Useful for simplifying or specializing code

Also a different (non-environment) way to think about what a function call is.
More scope mattering

Is \((\text{fn } x \Rightarrow e1) \ e2\) equivalent to \(e3\) where \(e3\) is \(e1\) with every \(x\) replaced by \(e2\)?

- Every free \(x\) (of course).
  - Example: \((\text{fn } x \Rightarrow (\text{fn } x \Rightarrow x)) \ 17\)

- A free variable in \(e2\) must not be bound at an occurrence of \(x\). (Called “capture”.)
  - Example: \((\text{fn } x \Rightarrow (\text{fn } y \Rightarrow x)) \ y\)

- Evaluating \(e2\) must terminate, not do assignments, not raise exceptions, not print, etc.
  - Because in ML and Scheme (but not all functional languages), \(e2\) is evaluated before the call
    - Example: \((\text{fn } x \Rightarrow x+x) \ ((\text{print } "hi" ; 5))\)

- Efficiency? Could be faster or slower. (Why?)
**Unnecessary Function Wrapping**

A common source of bad style for beginners

Is `e1` equivalent to `fn x => e1 x`? Sure, provided:

- `e1` effect-free (terminates, no mutation, printing, exceptions, etc.)
- `x` does not occur free in `e1`

Example:

```
List.map (fn x => SOME x) lst
List.map SOME lst
```

Notice variables, constructors, etc. are bound to values, so they are always effect-free (the value is already computed)

Another example:

```
(lambda () (f))
f
```