CSE 341: Programming Languages

Dan Grossman

Winter 2008

Lecture 16— define-struct; static vs. dynamic typing

Data in Scheme

Recall ML's approach to each-of, one-of, and self-referential types.

Pure Scheme's approach:

- There is One Big Datatype with built-in predicates.
- Primitives implicitly raise errors for "wrong variant"
- Use pairs (lists) for each-of types
- Can also use for one-of types with explicit "tags"
  - Like our force/delay with a boolean field
- Use helper functions like `caddr` (and/or define your own).

`define-struct`

DrScheme extends Scheme with `define-struct`, e.g.:

```
(define-struct card (suit value))
```

Semantics: Introduce several new bindings...

- *constructor* (`make-card`) that takes arguments and make values.
- *predicate* (`card?`) that takes 1 argument, return `#t` only for values made from the right constructor.
- *accessors* (`card-suit`, `card-value`) that take 1 argument, return a field, or call `error` for values not made from the right constructor.
- *mutators* (`set-card-suit!`, `set-card-value!`) that are like accessors except they mutate field contents.

Idiom for ML datatypes

Instead of a `datatype` with n constructors, you just use `define-struct` n times.

That "these n go together" is just convention.

Instead of `case`, you have a `cond` with n predicates and one "catch-all" error case.

`define-struct` is special

`define-struct` creates a new variant for The One Big Datatype.

Claim: `define-struct` is not a function.

Claim: `define-struct` is not a macro.

It could be a macro except for one key bit of its semantics: Values built from the constructor cause every *other* predicate (including all built-in ones) to return `#f`.

Advantage: abstraction

Disadvantage: Can't write "generic" code that has a case for every possible variant in every Scheme program.

Good and Bad Things About Types

Strong vs. *Weak* typing

In languages with weak typing, there exist programs that implementations *must* accept at compile-time, but at run-time the program can do *anything*, including blow-up your computer.

Examples: C, C++

Old "wisdom": "Strong types for weak minds"

New "wisdom": "Weak typing endangers society and costs billions a year"

Why weak typing? For efficiency and low-level implementation (important for 1% of low-level systems)

My view: Programming is hard enough without implementation-defined behavior. This has little to do with types.

Static vs. Dynamic Typing

In ML and Scheme `"hi" - "mom"` or `(- "hi" "mom")` are errors, but in ML it's at "compile-time" (static) and Scheme it's at "run-time" (dynamic).

- `(define (f) (- "hi" "mom"))` fine *until you call it*, but never type-checks in ML.

Indisputable facts:

- A language with static checks catches certain bugs without testing (earlier in the software-development cycle)
- It's impossible to catch exactly the buggy programs at compile-time
  - *Impossible* (undecidable) to know what code will execute in what environments, so may give *false positives*
  - Algorithm bugs remain (e.g., using + where you meant −)

Static Checking

Key questions for a compile-time check (e.g., ML type-checking):

1. What is it checking? Examples (and not):
   - Yes: Primitives (e.g., +) aren't applied to inappropriate values
   - Yes: Module interfaces are respected (e.g., don't use private functions)
   - No: `hd` is never applied to the empty list
2. Is it *sound*? (Does it ever accept a program that at run-time does what we claimed it could not? "false negative")
3. Is it *complete*? (Does it ever reject a program that could not do the "bad thing" at run-time? "false positive")

All non-trivial static analyses are either unsound or incomplete.

Good design leads to "useful subsets" of all programs, typically (but not always) ensuring soundness and sacrificing completeness.

A Question of Eagerness

Again, every static type system provides certain guarantees. Here are some things for which useful static checks have been developed, but are not commonly in type systems (yet?): null dereferences, division-by-zero, data races, ...

There is also more than "compile-time" or "run-time".

Consider `3 / 0`.

- Compile-time: reject if code is "reachable" (maybe dead branch)
- Link-time: reject if code is "reachable" (maybe unused function)
- Run-time: reject if code executes (maybe branch never taken)
- Even later: maybe delay error until "bad number" is used to index into an array or something.
  - Crazy? Floating-point allows division-by-zero; gives you `+inf.0`.

Exploring Some Arguments

1a. Dynamic typing is more convenient

```
(define (f x) (if (> x 0) (* 2 x) #f))
(let ([ans (f y)]) (if ans e1 e2))
```

```
datatype intOrBool = Int of int | Bool of bool
fun f x = if x > 0 then Int (2*x) else Bool false
case f y of
  Int i => e1
| Bool b => e2
```

Exploring Some Arguments

1b. Static typing is more convenient

```
(define (cube x) (if (not (number? x))
                     (error "bad arguments")
                     (* x x x)))
(cube 7)


fun cube x = x * x *x
cube 7
```

Exploring Some Arguments

2. Static typing prevents / doesn't prevent useful programs

- Overly restrictive type systems certainly can (e.g., without polymorphism a new list library for each list-element type)
- `datatype` gives you as much or as little flexibility as you want – can embed Scheme in ML:

```
datatype SchemeVal = Int of int | String of string
                   | Fun of SchemeVal -> SchemeVal
                   | Cons of SchemeVal * SchemeVal
if e1
then Fun (fn x => case x of Int i => i * i * i)
else Cons (Int 7, String ``hi'')
```

Viewed this way, Scheme is "unityped" with "implicit tag-checking" which is "just" a matter of convenience.

Exploring Some Arguments

3. Static/dynamic typing better for code evolution

- Dynamic: If you need to change the type of something, the program will still compile; easier to incrementally upgrade other code to support the change?
- Static: If you change the type of something, the type-checker guides you to all the places you need to change?

In practice, ML's pattern exhaustiveness is great for the latter.

Exploring Some Arguments

4. Types make code reuse harder/easier.

- Dynamic:
  - Sound types means you'll always restrict how code is used in some way that you need not
  - By using cons cells for everything, you can reuse lots of libraries
- Static:
  - Using separate types catches bugs and enforces abstractions (don't accidentally confuse two different uses of cons cells)
  - We can provide enough flexibility in practice (e.g., with polymorphism)

Design issue: Whether to build a new data structure or encode with existing ones (for libraries) is an important consideration.

Exploring Some Arguments

5. Types make programs faster/slower.

- Dynamic: Don't have to code around the type system or duplicate code; optimizer can remove provably unnecessary tag-tests
- Static: Programmer controls where tag-tests occur (in patterns) and knows that compiler need not have unnecessary tests (is argument to + a number).

Summary

There are real trade-offs here; you must know them.

It is possible to have rational discussions about them, informed by facts.

Almost every language checks some things statically and other things dynamically.

- It is really a question of *what* you check statically, but we have an informal understanding of what type-checking "normally checks for"