# CSE 341: Programming Languages

Dan Grossman

Fall 2004

Lecture 18— Tree-Iterator Example and Static vs. Dynamic Typing

# An Extended Example

You've seen several advanced and abstract notions:

- iterators for separating traversal from processing
- thunks for delaying evaluations
- passing continuations for "what to do next"
  - `let/cc` "forgets what you're doing" like exceptions
  - but passing a "continuation function" is a similar idiom
- tail recursion

An elegant example that puts it all together: a tree iterator

# Cool things about the tree iterator

Fundamentally, tree iteration requires a conceptual *stack*.

But using the call-stack is inconvenient because of delayed evaluation.

Instead the stack is implicit in our "continuation functions".

And everything is a tail call!

Food for thought: There is an automatic transformation that makes every function call in every program a tail call, eliminating a call-stack and using "continuation functions" instead. Called *continuation-passing style*.

# 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 efficient 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 adding strings (`‘‘hi’’ + ‘‘mom’’` or `(+ ‘‘hi’’ ‘‘mom’’)`) is an error, but in ML it's at "compile-time" (static) and Scheme it's at "run-time" (dynamic).

Indisputable facts:

- A language with static checks catches certain bugs without testing (earlier in the software-development cycle)
- It is impossible to catch exactly the buggy programs at compile-time
  - "Will a program add a string" trivially harder than "Will a program terminate"
  - Application-logic bugs remain (e.g., using factorial where you meant to use fibonacci)

# Static Checking

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

1. What is it checking? Examples (and not):
   - Primitives (+, apply, ...) are never applied to "inappropriate" values
   - `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 `x / 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 `nan`.

# Exploring Some Arguments

1. Dynamic/static 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
```

---

```
(define (cube x) (if (not (number? x))
                     (error 'cube ``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 (Pascal array sizes, lack of polymorphism)
- 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 should/shouldn't be extensible (new cases throughout program, at run-time, etc.)

- Dynamic: necessary for abstraction, necessary for an evolving world (ubicomp, service discovery, etc.), even ML does it for exceptions
- Static: can never establish exhaustiveness, must always have "default" clauses

My view: You probably want both options in your language and to think carefully in design phase.

# Exploring Some Arguments

5. Types make code reuse harder/easier.

- Dynamic: Soundness means you'll never be as flexible as somebody wants; if you use cons cells for everything, you can have a rich library for them
- Static: Using separate types catches bugs and enforces abstractions; 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

6. 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)

# 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.