<image/> <image/> <image/> <section-header><section-header><section-header><section-header><text><text><text></text></text></text></section-header></section-header></section-header></section-header>	<section-header> because the provided of the provided</section-header>
<section-header><section-header><section-header><section-header><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></section-header></section-header></section-header></section-header>	<section-header><section-header><section-header><code-block><code-block><code-block><code-block><code-block></code-block></code-block></code-block></code-block></code-block></section-header></section-header></section-header>

Racket from an ML Perspective

One way to describe Racket is that it has "one big datatype"

- All values have this type

datatype theType = Int of int | String of string | Pair of theType * theType | Fun of theType -> theType | ...
Constructors are applied implicitly (values are tagged) - 42 is really like Int 42 inttag 42
Primitives implicitly check tags and extract data, raising errors for wrong constructors

```
fun car v = case v of Pair(a,b) => a | _ => raise ...
fun pair? v = case v of Pair _ => true | _ => false
```

Spring 2013

Spring 2013

5

More on The One Type

pairs, symbols, procedures, etc.

adding to "theType"

• Built-in constructors for "theType": numbers, strings, booleans,

• Each struct-definition creates a new constructor, dynamically

Static checking Example: ML, what types prevent Static checking is anything done to reject a program after it In ML, type-checking ensures a program (when run) will never have: (successfully) parses but before it runs · A primitive operation used on a value of the wrong type Part of a PL's definition: what static checking is performed - Arithmetic on a non-number - A "helpful tool" could do more checking - e1 e2 where e1 does not evaluate to a function - A non-boolean between if and then Common way to define a PL's static checking is via a type system A variable not defined in the environment - Approach is to give each variable, expression, etc. a type A pattern-match with a redundant pattern - Purposes include preventing misuse of primitives (e.g., 4/"hi"), Code outside a module call a function not in the module's signature enforcing abstraction, and avoiding dynamic checking · Dynamic means at run-time (First two are "standard" for type systems, but different languages' Dynamically-typed languages do (almost) no static checking type systems ensure different things) Line is not absolute Spring 2013 Spring 2013 CSE341: Programming Languages 7 CSE341: Programming Languages Example: ML, what types allow Purpose is to prevent something In ML, type-checking does not prevent any of these errors Have discussed facts about what the ML type system does and does not prevent - Instead, detected at run-time - Separate from how (e.g., one type for each variable) though previously studied many of ML's typing rules Calling functions such that exceptions occur, e.g., hd [] An array-bounds error Language design includes deciding what is checked and how Division-by-zero Hard part is making sure the type system "achieves its purpose" In general, no type system prevents logic / algorithmic errors: That "the how" accomplishes "the what" Reversing the branches of a conditional - More precise definition next Calling f instead of g (Without a program specification, type-checker can't "read minds") 9 Spring 2013 CSE341: Programming Languages Spring 2013 CSE341: Programming Languages 10 Correctness A question of eagerness Suppose a type system is supposed to prevent X for some X "Catching a bug before it matters" is in inherent tension with A type system is sound if it never accepts a program that, when "Don't report a bug that might not matter" run with some input, does X No false negatives Static checking / dynamic checking are two points on a continuum A type system is complete if it never rejects a program that, no Silly example: Suppose we just want to prevent evaluating 3 / 0

- Keystroke time: disallow it in the editor
- Compile time: disallow it if seen in code
- Link time: disallow it if seen in code that may be called to evaluate main
- Run time: disallow it right when we get to the division
- Later: Instead of doing the division, return +inf.0 instead
- Just like 3.0 / 0.0 does in every (?) PL (it's useful!)

Notice soundness/completeness is with respect to X

matter what input it is run with, will not do X

- No false positives

rely on it) but not complete

Spring 2013

- "Fancy features" like generics aimed at "fewer false positives"

The goal is usually for a PL type system to be sound (so you can

Incompleteness

A few functions ML rejects even though they do not divide by a string

```
fun fl x = 4 div "hi" (* but fl never called *)
fun f2 x = if true then 0 else 4 div "hi"
fun f3 x = if x then 0 else 4 div "hi"
val x = f3 true
fun f4 x = if x <= abs x then 0 else 4 div "hi"
fun f5 x = 4 div x
val y = f5 (if true then 1 else "hi")
```

CSE341: Programming Languages

What about unsoundness?

Suppose a type system were unsound. What could the PL do?

- · Fix it with an updated language definition?
- Insert dynamic checks as needed to prevent X from happening?
- Just allow X to happen even if "tried to stop it"?
- Worse: Allow not just X, but anything to happen if "programmer gets something wrong"
 - Will discuss C and C++ next...

Spring 2013

Spring 2013

CSE341: Programming Languages

What weak typing has caused

 Racket is *not* weakly typed Old now-much-rarer saying: "strong types for weak minds" It just checks most things dynamically* - Idea was humans will always be smarter than a type system (cf. undecidability), so need to let them say "trust me" then it can optimize them away Reality: humans are really bad at avoiding bugs - We need all the help we can get! · Not having ML or Java's rules can be convenient And type systems have gotten much more expressive (fewer Cons cells can build anything false positives) - Anything except #f is true 1 bug in a 30-million line operating system written in C can - ... make an entire computer vulnerable - An important bug like this was probably announced this week (because there is one almost every week) 17 Spring 2013 CSE341: Programming Languages

15

Why incompleteness · Almost anything you might like to check statically is undecidable: - Any static checker cannot do all of: (1) always terminate, (2) be sound, (3) be complete – This is a mathematical theorem! Examples: - Will this function terminate on some input? - Will this function ever use a variable not in the environment? - Will this function treat a string as a function? - Will this function divide by zero? · Undecidability is an essential concept at the core of computing - The inherent approximation of static checking is probably its most important ramification 13 CSE341: Programming Languages 14 Spring 2013 Why weak typing (C/C++) Weak typing: There exist programs that, by definition, *must* pass static checking but then when run can "set the computer on fire"? Dynamic checking is optional and in practice not done - Why might anything happen? · Ease of language implementation: Checks left to the programmer · Performance: Dynamic checks take time · Lower level: Compiler does not insert information like array sizes,

so it cannot do the checks

Weak typing is a poor name: Really about doing neither static nor dynamic checks

 A big problem is array bounds, which most PLs check dynamically

Spring 2013

CSE341: Programming Languages

16

Example: Racket

- Dynamic checking is the *definition* - if the *implementation* can analyze the code to ensure some checks are not needed,

This is nothing like the "catch-fire semantics" of weak typing

*Checks macro usage and undefined-variables in modules statically



(define (f y) (if (> y 0) (+ y y) "hi")) (let ([ans (f x)]) (if (number? ans) (number->string ans) ans)) datatype t = Int of int | String of string fun f y = if y > 0 then Int(y+y) else String case f x of Int i => Int.toString i | String s => s Spring 2013 CSE341: Programming Languages

Can assume data has the expected type without cluttering code with dynamic checks or having errors far from the logical mistake

hi"		<pre>(define (cube x) (if (not (number? x)) (error "bad arguments") (* x x x))) (cube 7) fun cube x = x * x * x cube 7</pre>	
21	Spring 2013	CSE341: Programming Languages	22

Claim 2a: Static prevents useful programs

Any sound static type system forbids programs that do nothing wrong, forcing programmers to code around limitations

(define (f g) (cons (g 7) (g #t))) (define pair_of_pairs (f (lambda (x) (cons x x)))) fun f g = (g 7, g true) (* does not type-check *) val pair of pairs = f (fn $x \Rightarrow (x, x)$) Spring 2013 CSE341: Programming Languages 23

Claim 2b: Static lets you tag as needed

Rather than suffer time, space, and late-errors costs of tagging everything, statically typed languages let programmers "tag as needed" (e.g., with datatypes)

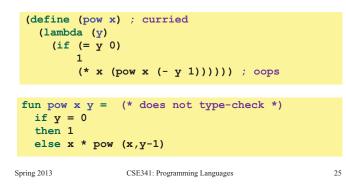
In the extreme, can use "TheOneRacketType" in ML Extreme rarely needed in practice

```
datatype tort = Int of int
                | String of string
                 Cons of tort * tort
                  Fun of tort -> tort
if el
then Fun (fn x => case x of Int i => Int (i*i*i))
else Cons (Int 7, String "hi")
Spring 2013
                  CSE341: Programming Languages
                                                      24
```

Claim 3a: Static catches bugs earlier

Static typing catches many simple bugs as soon as "compiled"

- Since such bugs are always caught, no need to test for them
- In fact, can code less carefully and "lean on" type-checker



Claim 4a: Static typing is faster

Language implementation:

- Does not need to store tags (space, time)
- Does not need to check tags (time)

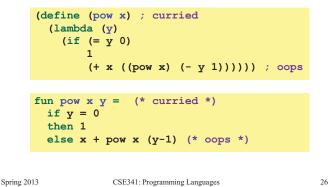
Your code:

Spring 2013

- Does not need to check arguments and results

Claim 3b: Static catches only easy bugs

But static often catches only "easy" bugs, so you still have to test your functions, which should find the "easy" bugs too



Claim 4b: Dynamic typing is faster

Language implementation:

- Can use optimization to remove some unnecessary tags and tests
 - Example: (let ([x (+ y y)]) (* x 4))
- While that is hard (impossible) in general, it is often easier for the performance-critical parts of a program

Your code:

 Do not need to "code around" type-system limitations with extra tags, functions etc.

Spring 2013

- CSE341: Programming Languages
- 28

Claim 5a: Code reuse easier with dynamic

CSE341: Programming Languages

Without a restrictive type system, more code can just be reused with data of different types

- If you use cons cells for everything, libraries that work on cons cells are useful
- Collections libraries are amazingly useful but often have very complicated static types
- Etc.

Claim 5b: Code reuse easier with static

- Modern type systems should support reasonable code reuse with features like generics and subtyping
- If you use cons cells for everything, you will confuse what represents what and get hard-to-debug errors
 - Use separate static types to keep ideas separate
 - Static types help avoid library misuse

27

So far	Claim 6a: Dynamic better for prototyping	
Considered 5 things important when writing code: 1. Convenience	Early on, you may not know what cases you need in datatypes and functions	
 Not preventing useful programs Catching bugs early 	 But static typing disallows code without having all cases; dynamic lets incomplete programs run 	
 Performance Code reuse 	 So you make premature commitments to data structures And end up writing code to appease the type-checker that you later throw away 	
But took the naive view that software is developed by taking an existing spec, coding it up, testing it, and declaring victory.	Particularly frustrating while prototyping	
Reality: – Often a lot of prototyping <i>before</i> a spec is stable – Often a lot of maintenance / evolution <i>after</i> version 1.0		
Spring 2013 CSE341: Programming Languages 31	Spring 2013 CSE341: Programming Languages 32	
Claim 6b: Static better for prototyping	Claim 7a: Dynamic better for evolution	
 What better way to document your evolving decisions on data structures and code-cases than with the type system? New, evolving code most likely to make inconsistent assumptions 	 Can change code to be more permissive without affecting old callers Example: Take an int or a string instead of an int All ML callers must now use a constructor on arguments and pattern-match on results Existing Racket callers can be oblivious 	
Easy to put in temporary stubs as necessary, such as _ => raise Unimplemented	(define (f x) (* 2 x)) (define (f x) (if (number? x)	
	(* 2 x) (string-append x x)))	
	<pre>fun f x = 2 * x fun f x = case f x of Int i => Int (2 * i) String s => String(s ^ s)</pre>	
Spring 2013 CSE341: Programming Languages 33	Spring 2013 CSE341: Programming Languages 34	
Claim 7b: Static better for evolution	Coda	
When we change type of data or code, the type-checker gives us a "to do" list of everything that must change — Avoids introducing bugs	 Static vs. dynamic typing is too coarse a question Better question: <i>What</i> should we enforce statically? 	
 The more of your spec that is in your types, the more the type-checker lists what to change when your spec changes 	 Legitimate trade-offs you should know Rational discussion informed by facts! 	
Example: Changing the return type of a function	 Ideal (?): Flexible languages allowing best-of-both-worlds? — Would programmers use such flexibility well? Who decides? 	
Example: Adding a new constructor to a datatype – Good reason not to use wildcard patterns		
Counter-argument: The to-do list is mandatory, which makes		

Counter-argument: The to-do list is mandatory, which makes evolution in pieces a pain: cannot test part-way through

Spring 2013

Spring 2013