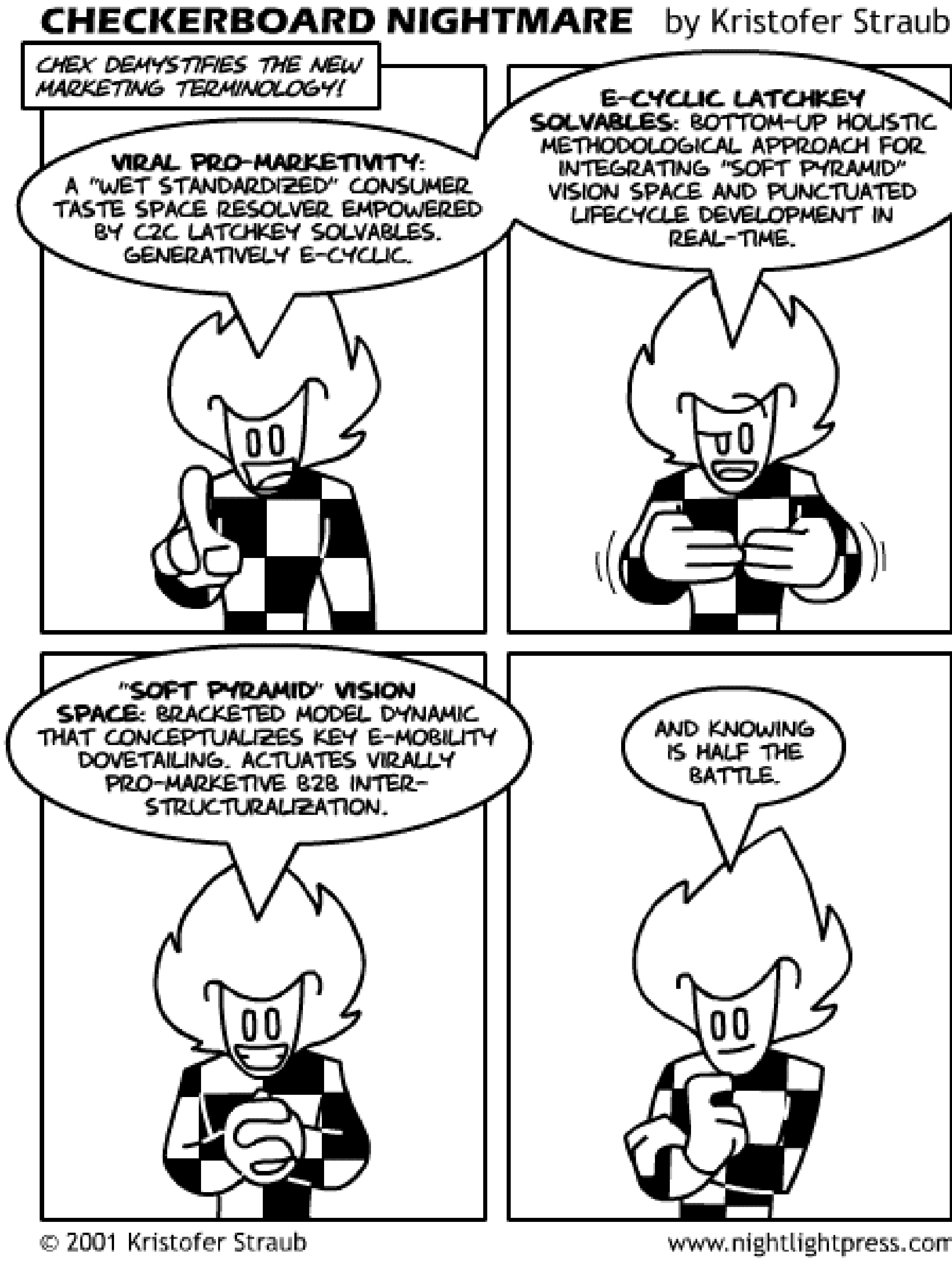


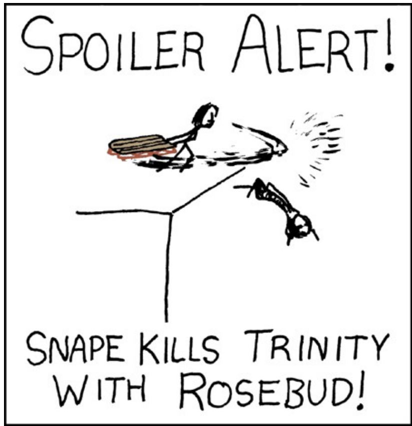
Second-Order Type Systems

One-Slide Summary

- A **polymorphic** type system is flexible: it allows one functions to be applied to many types of arguments.
- Parametric **impredicative** polymorphism allows any type to be used polymorphically. This has simple syntax but complicated expressive semantics and type reconstruction is undecidable.
- Parametric **predicative** polymorphism allows only monomorphic types as type variables.
- **Prenex** predicative polymorphism and the **value restriction** are two constrained, weaker versions of predicative polymorphism that are used in practice.

Upcoming Lectures

- We're now reaching the point where you have all of the tools and background to understand advanced topics.
- Upcoming Topics:
  - Dependent Types + Data Abstraction
  - Communication and Concurrency
  - Fault Localization
  - Automated Program Repair



Review: Modeling References

- A heap is a mapping from addresses to values

$$h ::= \cdot \mid h, a \leftarrow v : \tau$$

  - $a \in$ Addresses (Addresses $\neq \mathbb{Z}$?)
  - We tag the heap cells with their types
  - Types are useful only for static semantics. They are not needed for the evaluation $\Rightarrow$ are not a part of the implementation
- We call a program an expression with a heap

$$p ::= \text{heap } h \text{ in } e$$

  - The initial program is "heap $\cdot$ in e"
  - Heap addresses act as bound variables in the expression
  - This is a trick that allows easy *reuse of properties of local variables for heap addresses*
    - e.g., we can rename the address and its occurrences at will

Static Semantics of References

- Typing rules for expressions:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\texttt{ref}\ e : \tau) : \tau\ \texttt{ref}} \qquad \frac{\Gamma \vdash e : \tau\ \texttt{ref}}{\Gamma \vdash !e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \texttt{unit}}$$

- and for programs

$$\frac{\Gamma \vdash v_i : \tau_i\ (i = 1 \ldots n) \qquad \Gamma \vdash e : \tau}{\vdash \texttt{heap}\ h\ \texttt{in}\ e : \tau}$$

where $\Gamma = a_1 : \tau_1\ \texttt{ref}, \ldots, a_n : \tau_n\ \texttt{ref}$

and $h = a_1 \leftarrow v_1 : \tau_1, \ldots, a_n \leftarrow v_n : \tau_n$

Contextual Semantics for References

- Addresses are values: $v ::= \ldots \mid a$
- New contexts: $H ::= \text{ref } H \mid H_1 := e_2 \mid a_1 := H_2 \mid !\ H$
- No new *local* reduction rules
- But some new *global* reduction rules
  - heap h in H[ref v : τ] → heap h, a ← v : τ in H[a]
    - where a is fresh (this models allocation – the heap is extended)
  - heap h in H[! a] → heap h in H[v]
    - where a ← v : τ ∈ h (heap lookup – can we get stuck?)
  - heap h in H[a := v] → heap h[a ← v] in H[*]
    - where h[a ← v] means a heap like h except that the part "a ← v_1 : τ" in h is replaced by "a ← v : τ" (memory update)
- Global rules are used to propagate the effects of a write to the entire program (eval order matters!)

Example with References

- Consider these (the redex is underlined)
  - heap · in $(\lambda f{:}int \to int\ ref.\ !(f\ 5))\ \ (\lambda x{:}int.\ ref\ x : int)$
  - heap · in $!((\lambda x{:}int.\ ref\ x : int)\ \ 5)$
  - heap · in !(ref 5 : int)
  - heap a = 5 : int in !a
  - heap a = 5 : int in 5
- The resulting program has a useless memory cell
- An equivalent result would be

heap · in 5

- This is a simple way to model garbage collection

The Limitations of F_1

- In F_1 a function works exactly for one type
- Example: the identity function
  - id = $\lambda x{:}\tau.\ x : \tau \to \tau$
  - We need to write *one version for each type*
  - Worse: sort : $(\tau \to \tau \to \text{bool}) \to \tau$ array $\to \tau$ array
- The various sorting functions differ only in typing
  - At runtime they *perform exactly the same operations*
  - We need different versions only to keep the type checker happy
- Two alternatives:
  - Circumvent the type system (see C, Java, ...), or
  - Use a *more flexible type system* that lets us write only one sorting function (but use it on many types of objs)

Cunning Plan

- Introduce Polymorphism (much vocab)
- It's Strong: Encode Stuff
- It's Too Strong: Restrict
  - Still too strong … restrict more
- Final Answer:
  - Polymorphism works "as expect"
  - All the good stuff is handled
  - No tricky decideability problems
- Done early?

Polymorphism

- Informal definition

  A function is polymorphic if it can be applied to *"many"* types of arguments

- Various kinds of polymorphism depending on the definition of *"many"*
  - subtype polymorphism (aka bounded polymorphism)
    - "many" = all subtypes of a given type
  - ad-hoc polymorphism
    - "many" = depends on the function
    - choose behavior at runtime (depending on types, e.g. sizeof)
  - parametric *predicative* polymorphism
    - "many" = all monomorphic types
  - parametric *impredicative* polymorphism
    - "many" = all types

Parametric Polymorphism: Types as Parameters

- We introduce type variables and allow expressions to have variable types
- We introduce polymorphic types

Λ = Lambda

$$\tau ::= b \mid \tau_1 \to \tau_2 \mid t \mid \forall t.\ \tau$$

$$e ::= x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid \Lambda t.\ e \mid e[\tau]$$

  - $\Lambda t.\ e$ is type abstraction (or generalization, "for all t")
  - $e[\tau]$ is type application (or instantiation)
- Examples:
  - id = $\Lambda t.\lambda x{:}t.\ x$: $\forall t.t \to t$
  - id[int] = $\lambda x{:}int.\ x$: $int \to int$
  - id[bool] = $\lambda x{:}bool.\ x$: $bool \to bool$
  - "id 5" is invalid. Use "id[int] 5" instead

Impredicative Typing Rules

- The typing rules:

$$\frac{x : \tau \text{ in } \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda t.e : \forall t.\tau} \qquad t \text{ does not occur in } \Gamma$$

$$\frac{\Gamma \vdash e : \forall t.\tau'}{\Gamma \vdash e[\tau] : [\tau/t]\tau'}$$

Impredicative Polymorphism

- Verify that "id[int] 5" has type int
- Note the side-condition in the rule for type abstraction
  - Prevents ill-formed terms like: $\lambda x{:}t.\Lambda t.x$
- The evaluation rules are just like those of F_1
  - This means that type abstraction and application are all performed at compile time (*no run-time cost*)
  - We do not evaluate under Λ ($\Lambda t.\ e$ is a value)
  - We do not have to operate on types at run-time
  - This is called phase separation: type checking is separate from execution

(Aside:) Parametricity or "Theorems for Free" (P. Wadler)

- Can prove properties of a term *just from its type*
- There is only one value of type $\forall t.t \rightarrow t$
  - The identity function
- There is no value of type $\forall t.t$
- Take the function reverse : $\forall t.$ t List $\rightarrow$ t List
  - This function cannot inspect the elements of the list
  - It can only return a list of "original list elements"
  - If L_1 and L_2 have the same length and let "match" be a function that compares two lists element-wise according to an arbitrary predicate
  - then "match L_1 L_2" $\Rightarrow$ "match (reverse L_1) (reverse L_2)" !

Expressiveness of Impredicative Polymorphism

- This calculus is called
  - F_2
  - system F
  - second-order λ-calculus
  - polymorphic λ-calculus
- Polymorphism is *extremely expressive*
- We can encode many base and structured types in F_2

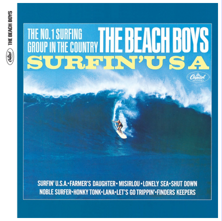
Encoding Base Types in F_2

- Booleans
  - bool = $\forall t.t \to t \to t$ (*given any two things, select one*)
  - There are exactly two values of this type!
  - true = $\Lambda t.\ \lambda x{:}t.\lambda y{:}t.\ x$
  - false = $\Lambda t.\ \lambda x{:}t.\lambda y{:}t.\ y$
  - not = $\lambda b{:}bool.\ \Lambda t.\lambda x{:}t.\lambda y{:}t.\ b\ [t]\ y\ x$
- Naturals
  - nat = $\forall t.\ (t \to t) \to t \to t$ (*given a successor and a zero element, compute a natural number*)
  - $0 = \Lambda t.\ \lambda s{:}t\to t.\lambda z{:}t.\ z$
  - $n = \Lambda t.\ \lambda s{:}t\to t.\lambda z{:}t.\ s\ (s\ (s\ldots s(n)))$
  - add = $\lambda n{:}nat.\ \lambda m{:}nat.\ \Lambda t.\ \lambda s{:}t\to t.\lambda z{:}t.\ n\ [t]\ s\ (m\ [t]\ s\ z)$
  - mul = $\lambda n{:}nat.\ \lambda m{:}nat.\ \Lambda t.\ \lambda s{:}t\to t.\lambda z{:}t.\ n\ [t]\ (m\ [t]\ s)\ z$

Expressiveness of F_2

- We can encode similarly:
  - $\tau_1 + \tau_2$ as $\forall t. (\tau_1 \to t) \to (\tau_2 \to t) \to t$
  - $\tau_1 \times \tau_2$ as $\forall t. (\tau_1 \to \tau_2 \to t) \to t$
  - unit as $\forall t. t \to t$
- We *cannot encode full recursion* (next lecture: $\mu t.\tau$)
  - We can encode primitive recursion but *not full recursion*
  - All terms in F_2 have a termination proof in second-order Peano arithmetic (Girard, 1971)
    - This is the set of naturals defined using zero, successor, induction along with quantification both over naturals and over sets of naturals

Computer Science



- This American Turing-award winner is known as the DARPA program manager in charge of funding groups developing TCP/IP. He funded and founded ICANN and the Internet Society. He helped develop the first commercial email system connected to the internet.

Computer Science, Mathematics

- This American mathematician did not win the Turing award, but developed in 1936, independently of Alan Turing, a model of computation that was equivalent to Turing Machines. The unsolvability of the *Entscheidung-sproblem* was exactly what was needed to obtain unsolvability results in the theory of formal languages.

Logic in Prose

148. Except living with others our whole life, we are both alone, solitary.

211. It was an uncomfortable silence. It was as if they were both as ease with each other.

222. He is just as powerful as myself, but not equally so.

270. He probably does know me but he where's a mask, so illogically he could be a number of people that I know.

426. Though her grades proved otherwise, Maeby wasn't an idiot.

Q: Books (702 / 842)

- This 1953 dystopian novel by Ray Bradbury has censorship as a major theme. The main character, Guy Montag, is a fireman.

What's Wrong with F_2

- Simple syntax but very complicated semantics
  - id can be applied to itself: "id [$\forall$t. t $\rightarrow$ t] id"
  - This can lead to paradoxical situations in a pure set-theoretic interpretation of types
  - e.g., the meaning of id is a function whose domain contains a set (the meaning of $\forall$t.t$\rightarrow$ t) that contains id!
  - This suggests that giving an interpretation to impredicative type abstraction is tricky
- Complicated termination proof (Girard)
- Type reconstruction (typeability) is *undecidable*
  - If the type application and abstraction are missing
- How to fix it?
  - Restrict the use of polymorphism

Predicative Polymorphism

- Restriction: type variables can be instantiated *only with monomorphic types*
- This restriction can be expressed syntactically

  $\tau ::= b \mid \tau_1 \to \tau_2 \mid t$ // monomorphic types

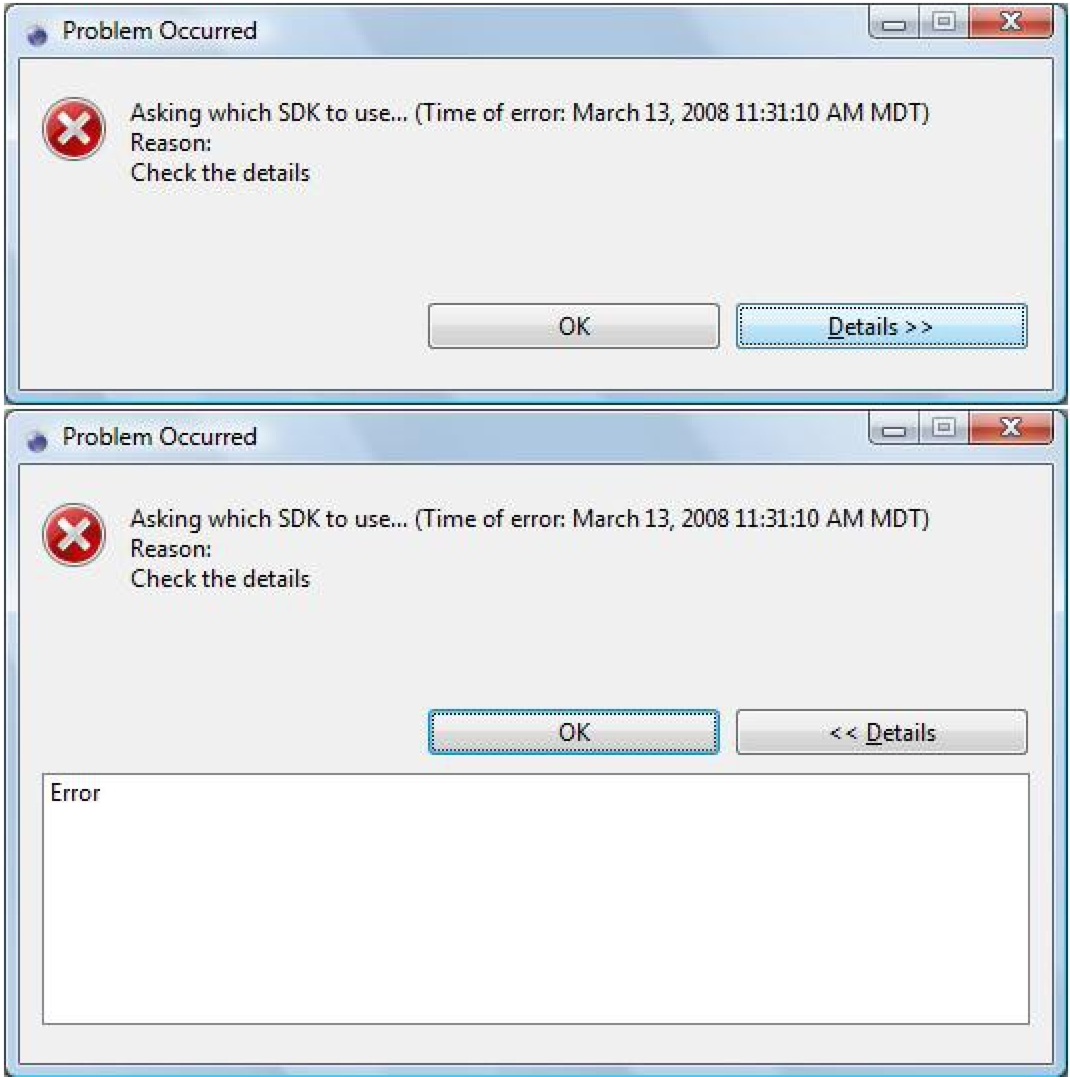
  $\sigma ::= \tau \mid \forall t.\ \sigma \mid \sigma_1 \to \sigma_2$ // polymorphic types

  $e ::= x \mid e_1\ e_2 \mid \lambda x{:}\sigma.\ e \mid \Lambda t.e \mid \mathbf{e\ [\tau]}$

  - Type application is restricted to mono types
  - Cannot apply "id" to itself anymore
- Same great typing rules
- Simple semantics and termination proof

Was that good enough?

- Type reconstruction still undecidable
- Must. Restrict. Further!



Prenex Predicative Polymorphism

- Restriction: polymorphic type constructor at *top level only*
- This restriction can also be expressed syntactically

  $\tau ::= b \mid \tau_1 \to \tau_2 \mid t$

  $\sigma ::= \tau \mid \forall t.\ \sigma$

  $e ::= x \mid e_1\ e_2 \mid \lambda x{:}\tau.\ e \mid \Lambda t.e \mid \mathbf{e\ [\tau]}$

  - Type application is predicative
  - Abstraction only on mono types
  - The only occurrences of ∀ are at the top level of a type

    $(\forall t.\ t \to t) \to (\forall t.\ t \to t)$ is not a valid type

- Same typing rules (less filling!)
- Simple semantics and termination proof
- Decidable type inference!

Expressiveness of Prenex Predicative F_2

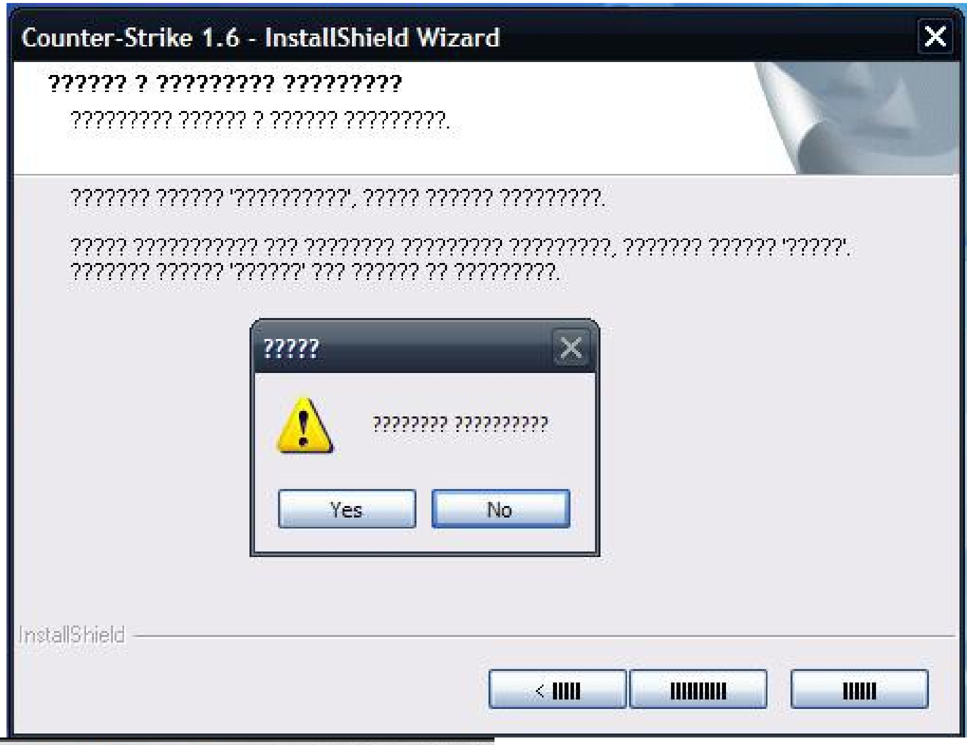
- We have simplified too much!
- Not expressive enough to encode nat, bool
  - But such encodings are only of theoretical interest anyway (cf. time wasting)
- Is it expressive enough in practice? Almost!
  - Cannot write something like

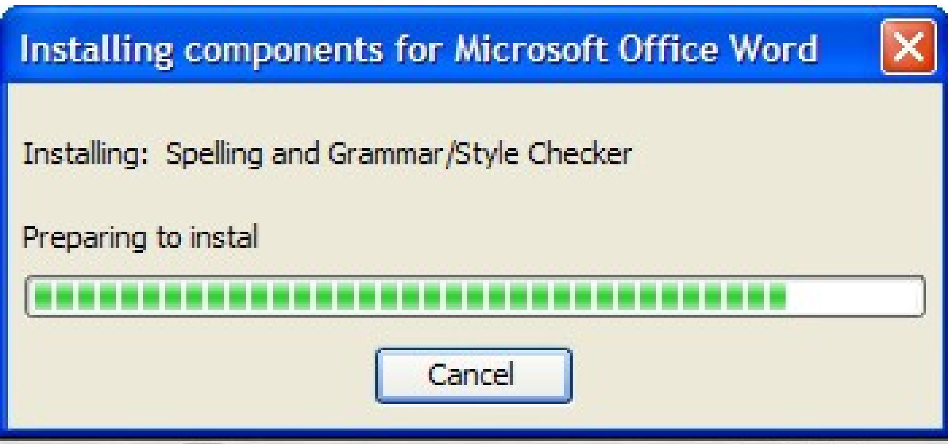
$(\lambda s{:}\forall t.\tau.\ \ldots\ s\ [nat]\ x\ \ldots\ \ s\ [bool]\ y)$

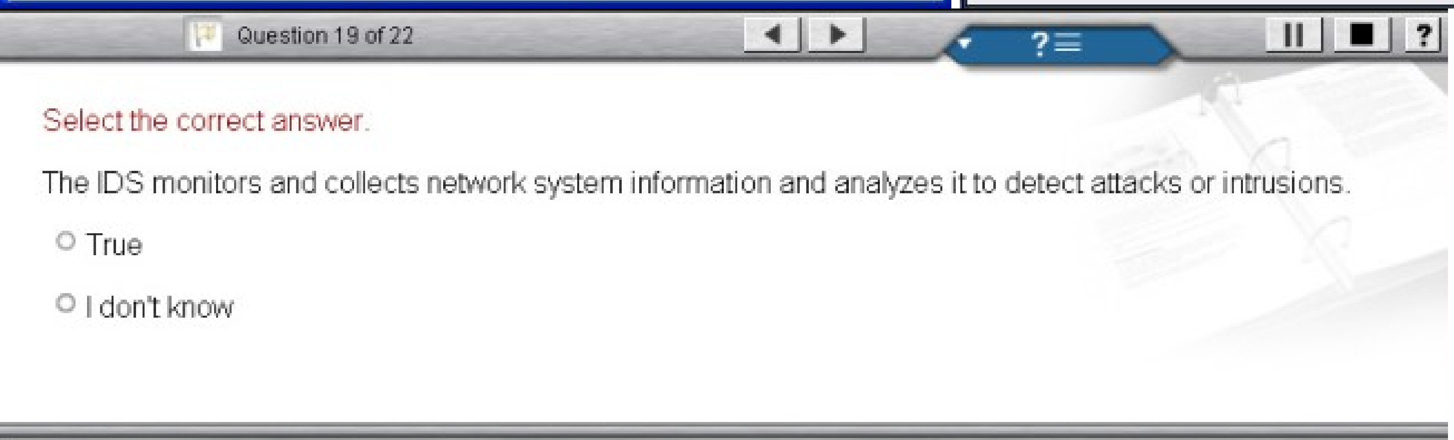
$(\Lambda t.\ \ldots$ code for sort$)$

  - Formal argument s cannot be polymorphic

What are we trying to do again?







ML and the Amazing Polymorphic Let-Coat

- ML solution: slight extension of the predicative F_2
  - Introduce "let x : σ = e_1 in e_2"
  - With the semantics of "(λx : σ.e_2) e_1"
  - And typed as "[e_1/x] e_2" (result: "fresh each time")

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x : \sigma\ = e_1\ \texttt{in}\ e_2 : \tau}$$

- This lets us write the polymorphic sort as

  let

  s : ∀t.τ = Λt. … code for polymorphic sort …

  in

  … s [nat] x …. s [bool] y

- We have found the sweet spot!

ML and the Amazing Polymorphic Let-Coat

- ML solution: slight extension of the predicative F_2
  - Introduce "let $x : \sigma = e_1$ in e_2"
  - With the semantics of "$(\lambda x : \sigma . e_2)\ e_1$"
  - And typed as "$[e_1/x]\ e_2$" (result: "fresh each time")

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x : \sigma\ = e_1\ \texttt{in}\ e_2 : \tau}$$

- This lets us write the polymorphic sort as

  let

      $s : \forall t.\tau = \Lambda t.$ … code for polymorphic sort …

  in

      … s [nat] x …. s [bool] y

- Surprise: this was a major ML design flaw!

ML Polymorphism and References

- let is evaluated using call-by-value but is typed using call-by-name
  - What if there are side effects?
- Example:

  let   x : $\forall$t. (t $\to$ t) ref = Λt.ref (λx : t. x)

  in

  x [bool] := λx: bool. not x ;

  (! x [int]) 5

  - Will apply "not" to 5
  - Recall previous lectures: invariant typing of references
  - Similar examples can be constructed with exceptions
- It took 10 years to find and agree on a clean solution

The Value Restriction in ML

- A type in a let is generalized *only for syntactic values*

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x : \sigma\ = e_1\ \texttt{in}\ e_2 : \tau}$$

e_1 is a syntactic value or σ is monomorphic

- Since e_1 is a value, its evaluation *cannot have side-effects*
- In this case call-by-name and call-by-value are the same
- In the previous example ref (λx:t. x) is not a value
- This is not too restrictive in practice!

Subtype Bounded Polymorphism

- We can bound the instances of a given type variable $\forall t < \tau.\ \sigma$
- Consider a function $f : \forall t < \tau.\ t \to \sigma$
- How is f different than $g : \tau \to \sigma$?
- One Answer: can invoke f on any subtype of τ
- Another: They are different if t appears in σ
  - e.g, let $f : \forall t < \tau. t \to t$ and $g : \tau \to \tau$ both be the identity function
  - Take $x : \tau'$ where $\tau' < \tau$
  - $f\ [\tau']\ x$ has static type τ'
  - $g\ x$ (using subsumption) has static type τ
  - Since both have dynamic type τ', we have lost information with g

Homework

- Homework 5
- Partners for HW6?