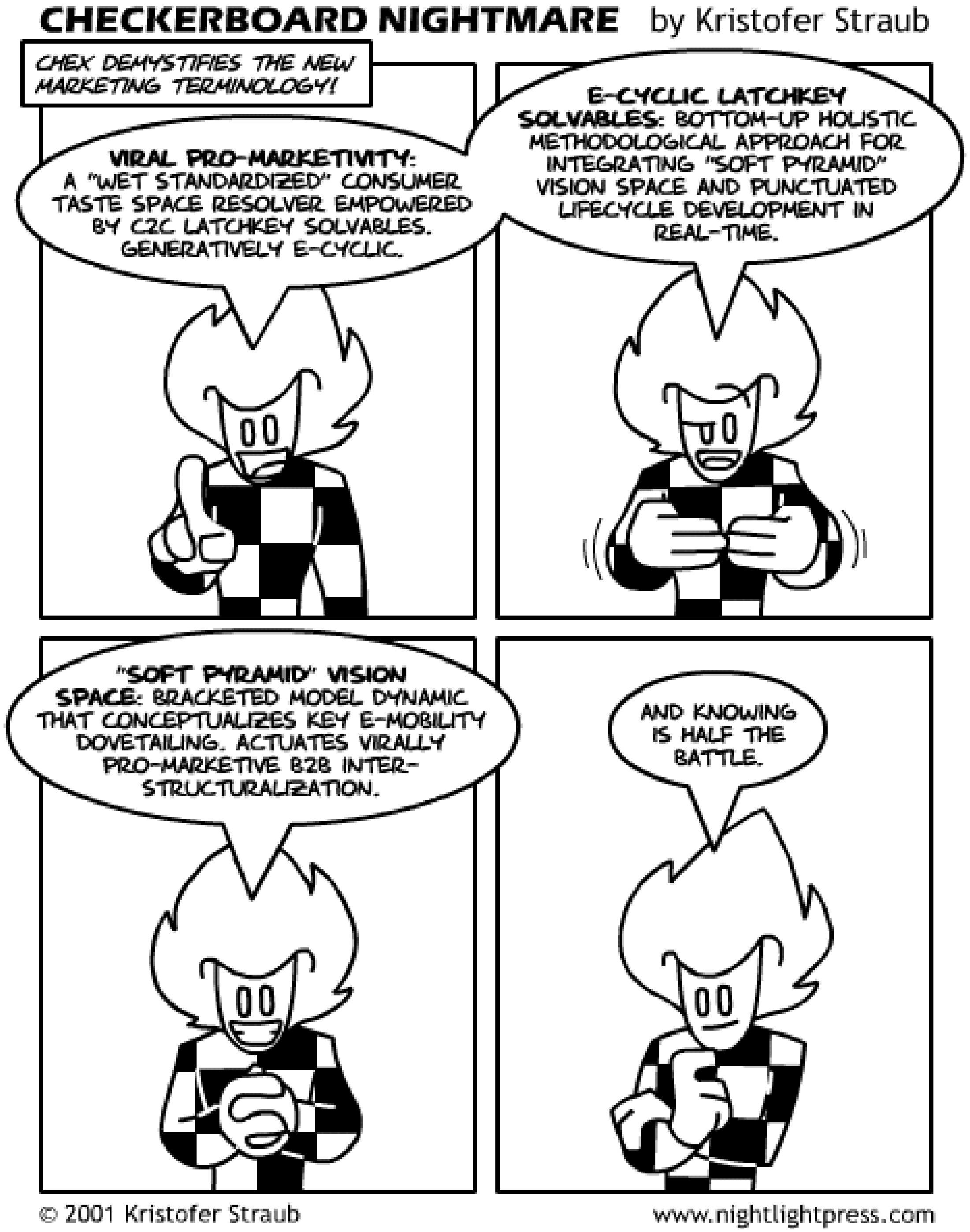

CHECKERBOARD NIGHTMARE by Kristofer Straub
CHEX DEMYSTIFIES THE NEW MARKETING TERMINOLOGY!
VIRAL PRO-MARKETIVITY: A "WET STANDARDIZED" CONSUMER TASTE SPACE RESOLVER EMPOWERED BY C2C LATCHKEY SOLVABLES. GENERATIVELY E-CYCLIC.
E-CYCLIC LATCHKEY SOLVABLES: BOTTOM-UP HOLISTIC METHODOLOGICAL APPROACH FOR INTEGRATING "SOFT PYRAMID" VISION SPACE AND PUNCTUATED LIFECYCLE DEVELOPMENT IN REAL-TIME.
"SOFT PYRAMID" VISION SPACE: BRACKETED MODEL DYNAMIC THAT CONCEPTUALIZES KEY E-MOBILITY DOVETAILING. ACTUATES VIRALLY PRO-MARKETIVE B2B INTER-STRUCTURALIZATION.
AND KNOWING IS HALF THE BATTLE.
© 2001 Kristofer Straub
www.nightlightpress.com

Second-Order Type Systems

Homework 5 Summary

Student : 37.9704
Student : 44.4466
ORIGINAL : 50.2442
Student : 50.8275
Student : 50.8633
Student : 50.9181
Student : 52.1347
Student : 52.1633
Student : 52.3775
Student : 53.5894
Student : 55.8466
Student : 56.4942
Student : 57.1276
Student : 57.1609
Student : 57.2013
Student : 57.8926
Student : 58.2259
Matthew Kin-Mo Yu : 69.5804

The number reported is the average coverage Percentage over all 50 subject programs.

What are the threats to validity?

What can we learn?

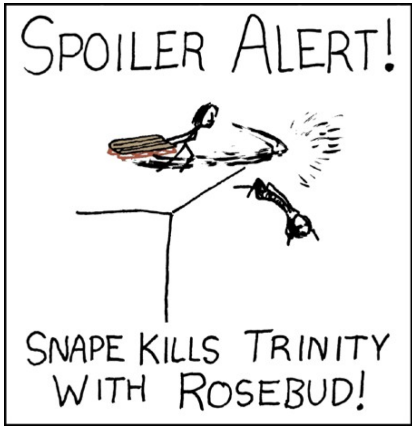
Support for:
Switch statements,
Floating point (recursive p/q determination),
Casts between floats and ints,
Structs (creation)
Structs with struct fields

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: simple syntax, complicated expressive semantics, 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.

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
  - Automated Theorem Proving + Proof Checking
  - Communication and Concurrency
  - Cooperative Bug Isolation
  - Automated Program Repair

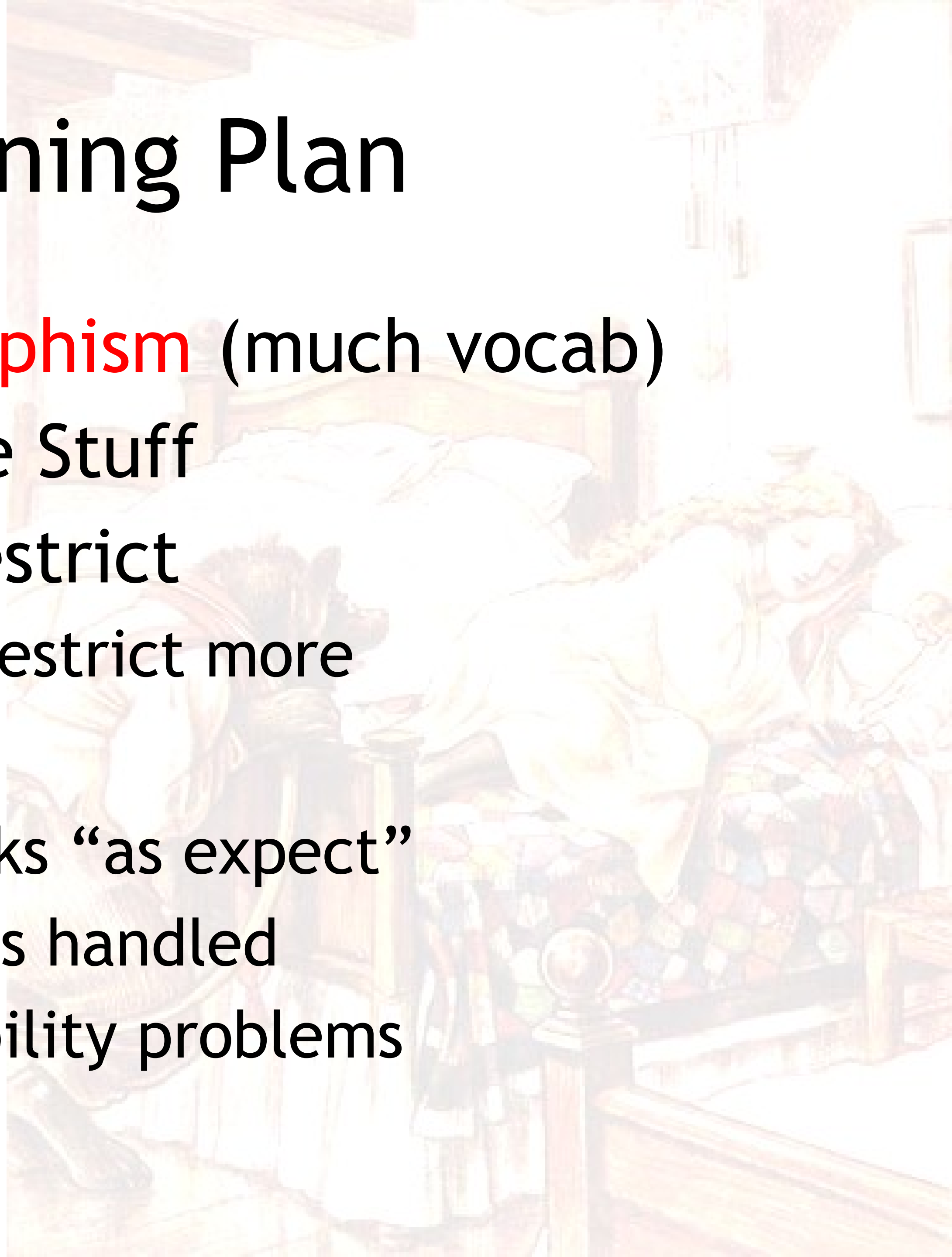


The Limitations of F_1

- 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 : $(\tau \to \tau \to \text{bool}) \to \tau\ \text{array} \to \tau\ \text{array}$
- 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

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

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

  $\Lambda t.\ e$ 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] = $\lambda x{:}\text{int}.\ x$: int → int
  - id[bool] = $\lambda x{:}\text{bool}.\ x$: bool → 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 $\forall t.t \to t$
  - The identity function
- There is no value of type $\forall t.t$
- Take the function reverse : $\forall t.$ t List $\to$ t List
  - This function cannot inspect the elements of the list
  - It can only produce a permutation of the original list
  - 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
  - $\text{bool} = \forall t.t \to t \to t$ (*given any two things, select one*)
  - There are exactly two values of this type!
  - $\text{true} \quad = \Lambda t.\ \lambda x{:}t.\lambda y{:}t.\ x$
  - $\text{false} \quad = \Lambda t.\ \lambda x{:}t.\lambda y{:}t.\ y$
  - $\text{not} \quad = \lambda b{:}\text{bool}.\ \Lambda t.\lambda x{:}t.\lambda y{:}t.\ b\ [t]\ y\ x$
- Naturals
  - $\text{nat} = \forall t.\ (t \to t) \to t \to t$ (*given a successor and a zero element, compute a natural number*)
  - $0 = \Lambda t.\ \lambda s{:}t\to t.\lambda z{:}t.\ z$
  - $n = \Lambda t.\ \lambda s{:}t\to t.\lambda z{:}t.\ s\ (s\ (s\ldots s(n)))$
  - $\text{add} = \lambda n{:}\text{nat}.\ \lambda m{:}\text{nat}.\ \Lambda t.\ \lambda s{:}t\to t.\lambda z{:}t.\ n\ [t]\ s\ (m\ [t]\ s\ z)$
  - $\text{mul} = \lambda n{:}\text{nat}.\ \lambda m{:}\text{nat}.\ \Lambda t.\ \lambda s{:}t\to t.\lambda z{:}t.\ n\ [t]\ (m\ [t]\ s)\ z$

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* $\mu t.\tau$
  - 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, Mathematics

- This Polish-Jewish 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 self-titled *Problem* was exactly what was needed to obtain unsolvability results in the theory of formal languages.

More Prose Logic

281. She couldn't exactly say that he looked different, although there was the possibility that she could pass him on the street without recognizing him. But, upon closer inspection, he was exactly the same.

94. She found her mother where she would always be, behind the bar, cooking food only the sober would eat.

126. It all starts six years after the end of it all.

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

378. "Are you alright?" His voice sounded scared but his voice looked calm.

What's Wrong with F_2

- Simple syntax but very complicated semantics
  - id can be applied to itself: "id [∀t. t → 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 ∀t.t→ 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

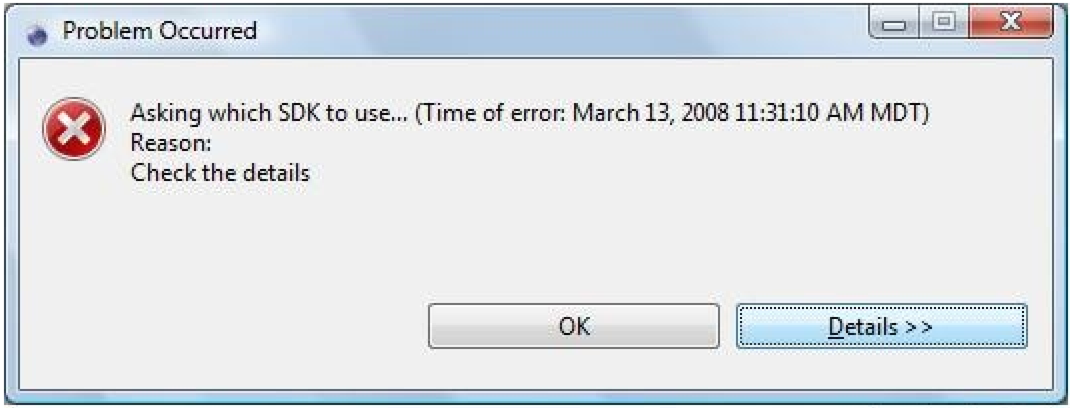
  $\sigma ::= \tau \mid \forall t.\ \sigma \mid \sigma_1 \to \sigma_2$ // polymorphic types

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

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

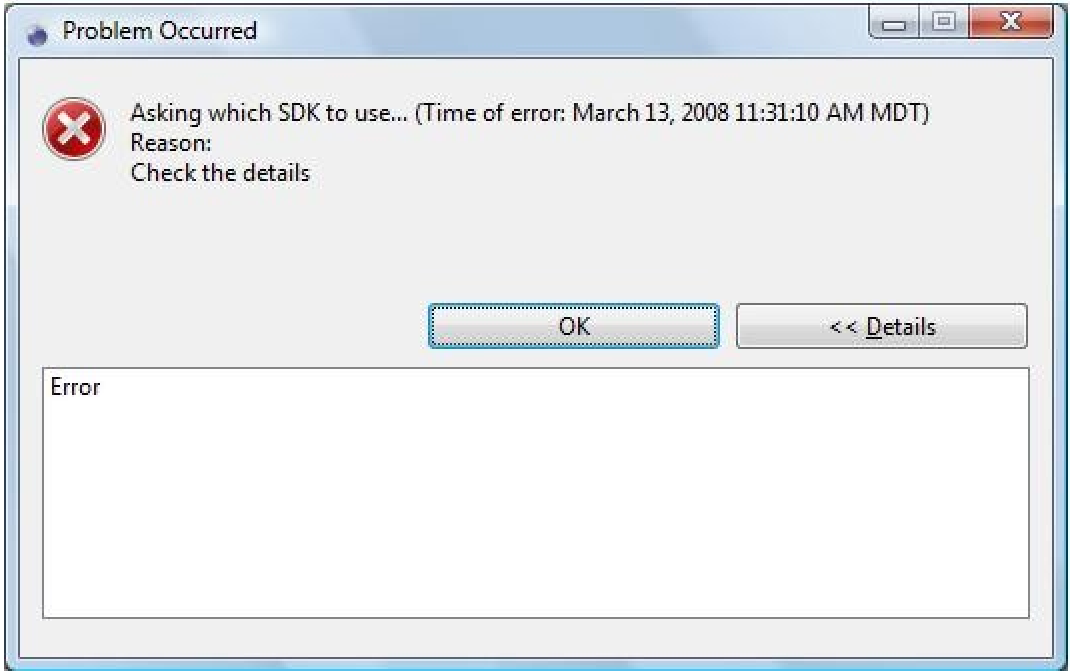
Was that good enough?

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



Problem Occurred

Asking which SDK to use... (Time of error: March 13, 2008 11:31:10 AM MDT)
Reason:
Check the details



Problem Occurred

Asking which SDK to use... (Time of error: March 13, 2008 11:31:10 AM MDT)
Reason:
Check the details

Error

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

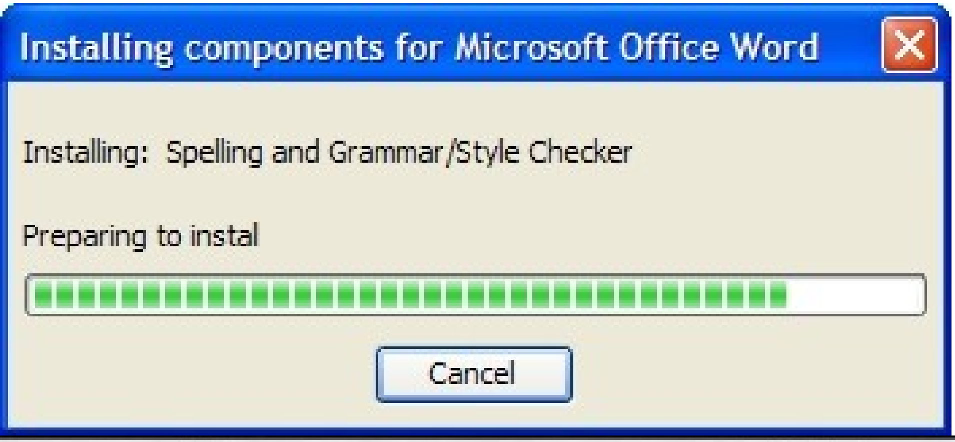
- We have simplified too much!
- Not expressive enough to encode nat, bool
  - But such encodings are only of theoretical interest anyway (cf. time wasting)
- Is it expressive enough in practice? Almost!
  - Cannot write something like

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

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

  - Formal argument s cannot be polymorphic

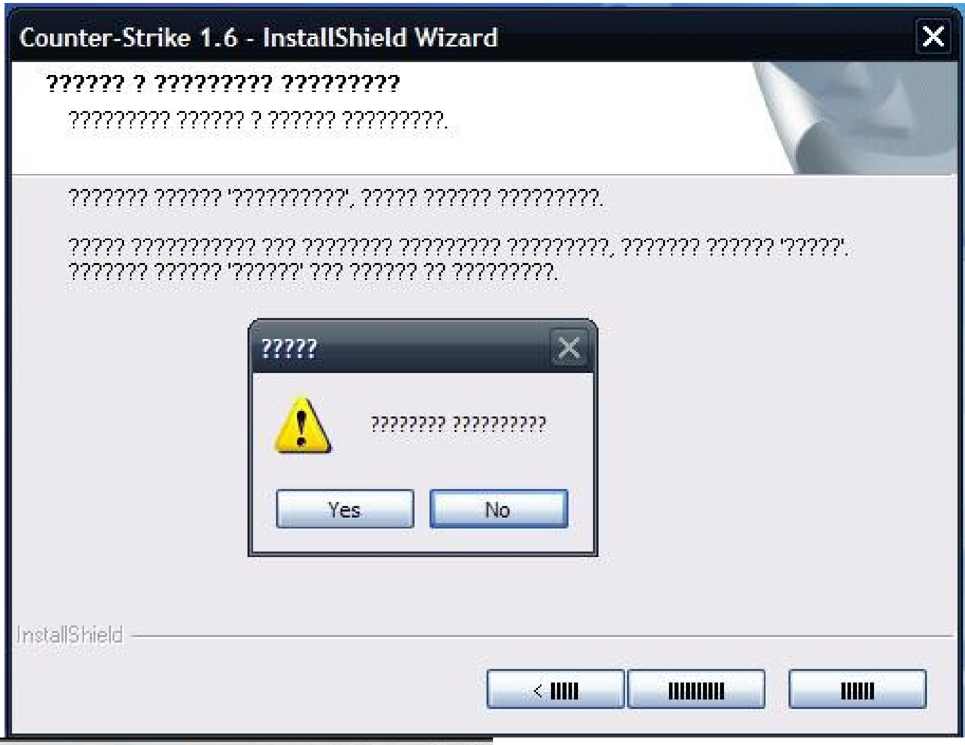
What are we trying to do again?



Installing components for Microsoft Office Word

Installing: Spelling and Grammar/Style Checker

Preparing to instal



Counter-Strike 1.6 - InstallShield Wizard

?????? ? ????????? ?????????

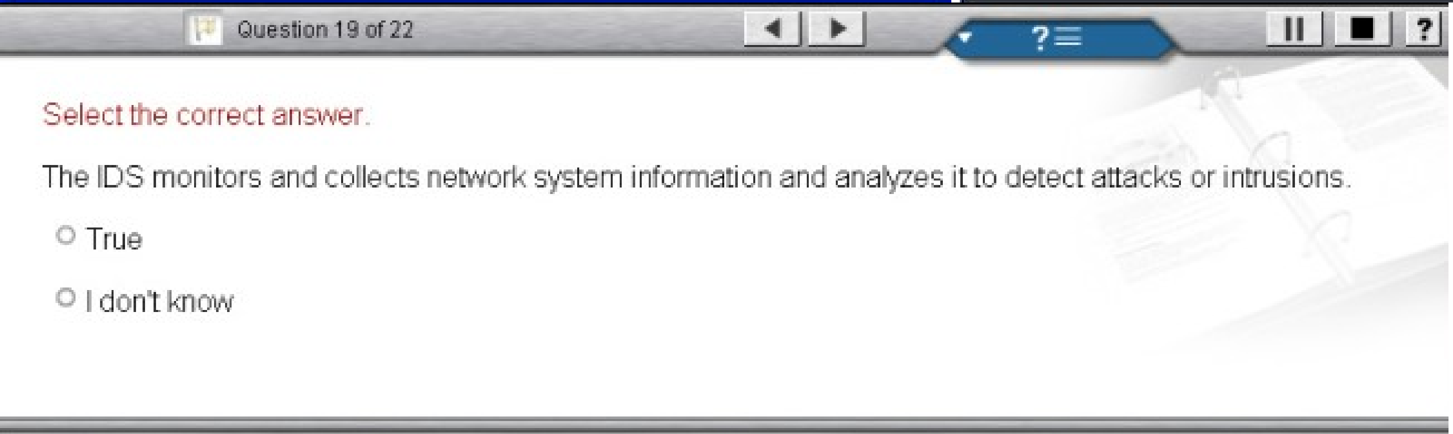
????????? ?????? ? ?????? ?????????.

??????? ?????? '??????????', ????? ?????? ?????????.

????? ??????????? ??? ???????? ????????? ?????????, ??????? ?????? '?????'.
??????? ?????? '??????' ??? ?????? ?? ?????????.

?????

???????? ?????????



Select the correct answer.

The IDS monitors and collects network system information and analyzes it to detect attacks or intrusions.

○ True

○ I don't know

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")

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

- This lets us write the polymorphic sort as

```
let
     s : ∀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 : 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")

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

- This lets us write the polymorphic sort as

```
let
    s : ∀t.τ = Λt. … 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 $\mathbf{f : \forall t{<}\tau.t \to t}$ and $\mathbf{g : \tau \to \tau}$ both be the identity function
  - Take $x : \tau'$ where $\tau' < \tau$
  - $\mathbf{f\ [\tau']\ x}$ has static type τ'
  - $\mathbf{g\ x}$ (using subsumption) has static type τ
  - Since both have dynamic type τ', we have lost information with **g**

Homework

- Project!