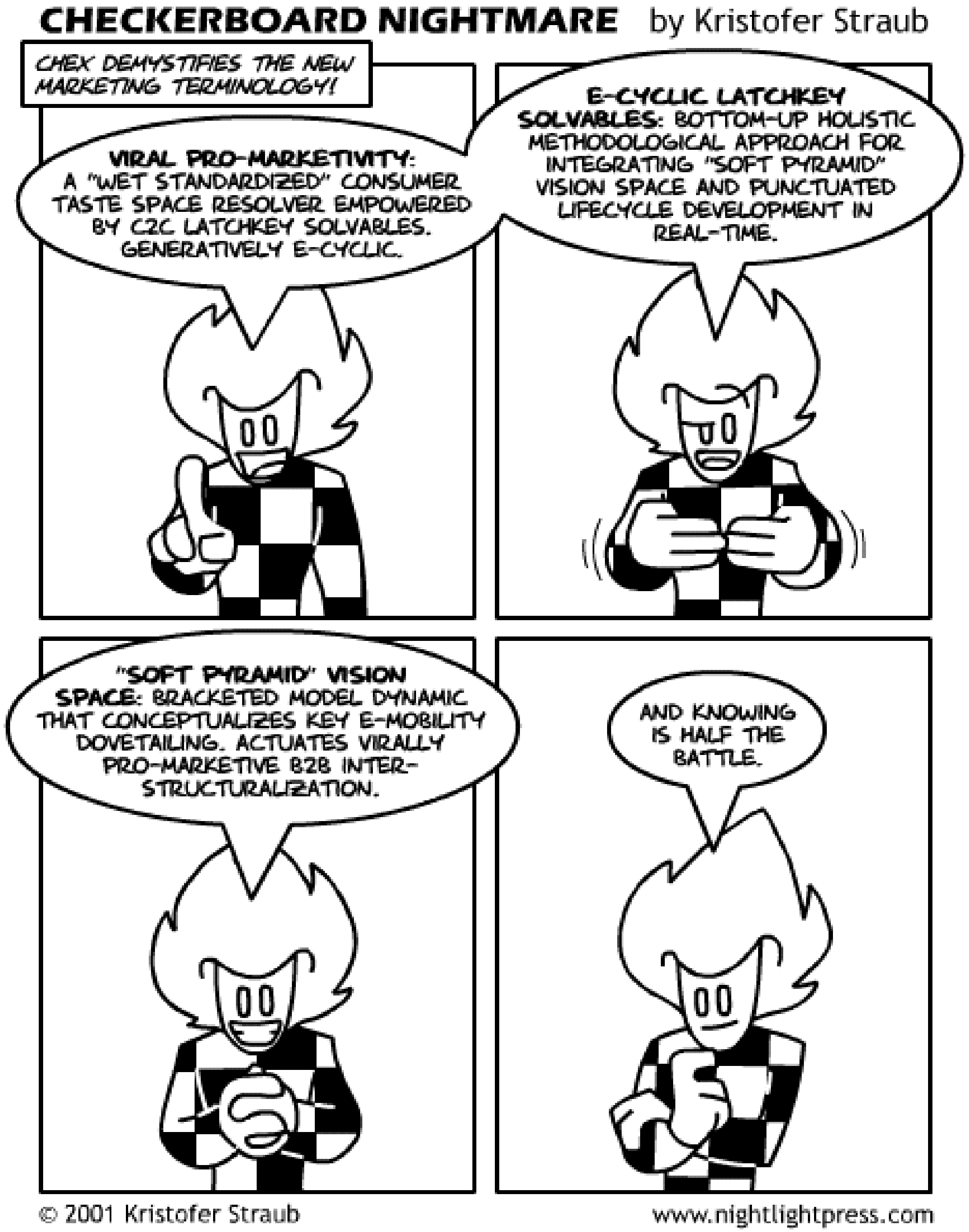


Second-Order Type Systems

Upcoming Lectures

- We’re now reaching the point where you have all of the tools and background to understand advanced topics.
- Upcoming Topics:
  - Automated Theorem Proving + Proof Checking
  - Model Checking
  - Software Model Checking
  - Types and Effects for Resource Management
  - Region-Based Memory Management
  - Object Calculi (OOP)

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

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:
  – $id = \Lambda t.\lambda x{:}t.\ x$: $\forall t.t \to t$
  – $id[int] = \lambda x{:}int.\ x$: $int \to int$
  – $id[bool] = \lambda x{:}bool.\ x$: $bool \to bool$
  – "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 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
  - 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 $\forall t.\ (\tau_1 \rightarrow t) \rightarrow (\tau_2 \rightarrow t) \rightarrow t$
  - $\tau_1 \times \tau_2$ as $\forall t.\ (\tau_1 \rightarrow \tau_2 \rightarrow t) \rightarrow t$
  - unit as $\forall t.\ t \rightarrow t$
- 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

What's Wrong with F_2

- Simple syntax but very complicated semantics
  - id can be applied to itself: "id [$\forall$t. t $\to$ 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$\to$ 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
- 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

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

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

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

  - 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 <u>not</u> a valid type

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

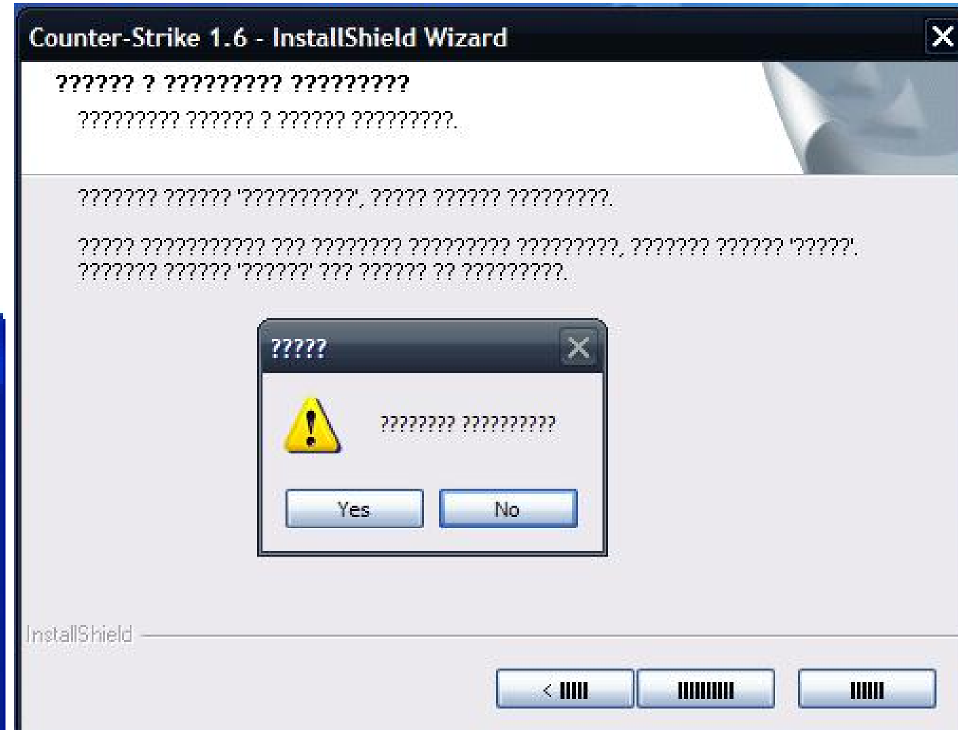
Expressiveness of Prenex Predicative F_2

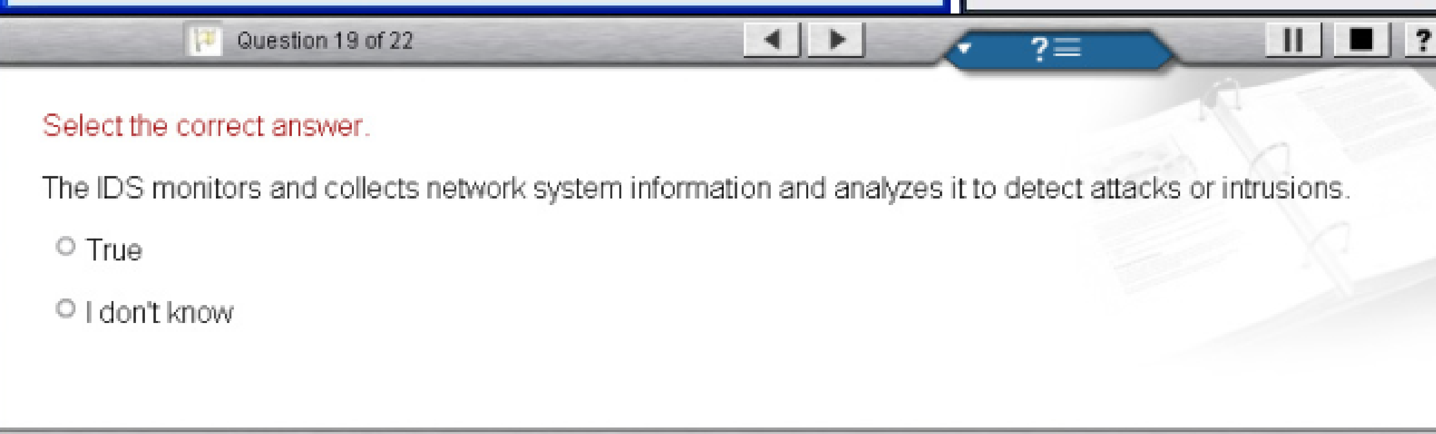
- We have simplified too much!
- Not expressive enough to encode nat, bool
  - But such encodings are only of theoretical interest anyway (cf. time wasting)
- Is it expressive enough in practice? Almost!
  - Cannot write something like

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

  $(\Lambda t.\ \ldots$ 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")

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

- 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 $= \Lambda t.\text{ref}\ (\lambda x : t.\ x)$
  in
  $x\ [\text{bool}] := \lambda x\!:\ \text{bool}.\ \text{not}\ x$;
  $(!\ x\ [\text{int}])\ 5$
  - 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 this different than $f' : \tau \to \sigma$
  - We can also invoke f' on any subtype of τ
- They are different if t appears in σ
  - e.g, $f : \forall t<\tau. t \to t$ and $f : \tau \to \tau$
  - Take $x : \tau' < \tau$
  - We have $f\ [\tau]\ x : \tau'$
  - And $f'\ x : \tau$
  - We have lost information with f'

Homework

- Project!