

Spammers are people too.  
Don't filter.

A MESSAGE FROM THE COUNCIL FOR THE ADVANCEMENT OF UNSOLICITED MAIL

Dependant  
Type Systems  
(saying what you are)

Data  
Abstraction  
(hiding what you are)

# Review

- We studied a variety of type systems
- We repeatedly made the type system **more expressive** to enable the type checker to catch more errors
- But we have steered clear of **undecidable** systems
  - Thus there must still be **many errors that are not caught**
- Now we explore more **complex type systems** that bring type checking closer to **program verification**

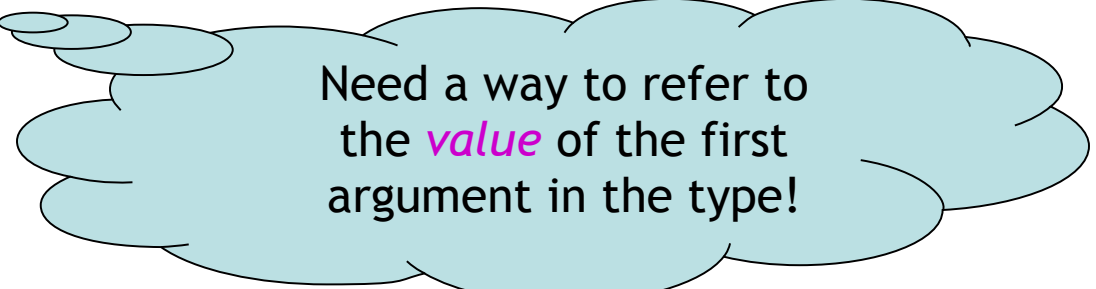
# Dependent Types

- Say that we have the functions  
     $\text{zero} : \text{nat} \rightarrow \text{vector}$  (creates vector of requested length)  
     $\text{dotprod} : \text{vector} \rightarrow \text{vector} \rightarrow \text{real}$  (dot product)
- The types do not prevent using dotprod on **vectors of different length**
  - If they did, we could catch more bugs!
- Idea: Make “vector” a type family annotated by a natural number

“vector n” is the type of vectors of length n

$\text{dotprod} : \text{vector } n \rightarrow \text{vector } n \rightarrow \text{real}$  (where is n bound?)

$\text{zero} : \text{nat} \rightarrow \text{vector} ?$



Need a way to refer to the *value* of the first argument in the type!

# Dependent Type Notation

- How to write the type of  $\text{zero} : \text{nat} \rightarrow \text{vector}$  ?
- Given two sets A and B verify the isomorphism

$$A \rightarrow B \simeq \prod_{x \in A} B$$

- The latter is the cartesian product of B with itself as many times as there are elements in A
- Also written as  $\prod x:A. B$  (x plays no role so far!)
- But now we can make B depend on x!
- Definition:  $\prod x:A. B$  is the type of functions with argument in A and with the result type B (possibly depending on the value of the argument x in A)
  - We write “ $\text{zero} : \prod x:\text{nat}. \text{vector } x$ ”
  - Special case when  $x \notin B$  we abbreviate as  $A \rightarrow B$
  - We play “fast and loose” with the binding of  $\prod$

# Dependent Typing Rules

$$\frac{\Gamma, x : \tau_2 \vdash e : \tau}{\Gamma \vdash \lambda x : \tau_2. e : \Pi x : \tau_2. \tau} \quad \frac{\Gamma \vdash e_1 : \Pi x : \tau_2. \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : [e_2/x]\tau}$$

- Note that **expressions are now part of types**
- Have types like “vector 5” and “vector (2 + 3)”
- We need **type equivalence**

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash e : \tau'}$$
$$\frac{\Gamma \vdash e_1 \equiv e_2}{\Gamma \vdash \text{vector } e_1 \equiv \text{vector } e_2}$$

# Dependent Types and Program Specifications

- **Types act as specifications**
- With dependent types we can specify *any property!*
- For example, define the following types:
  - “eq e” - the type of values equal to “e”.  
Also named “sng e” (the singleton type)
  - “ge e” - the type of values larger or equal to “e”
  - “lt e” - the type of values smaller than “e”
  - “and  $\tau_1 \tau_2$ ” - the type of values having both type  $\tau_1$  and  $\tau_2$
- Need appropriate typing rules for the new types
- The precondition for vector-accessing (cf. HW5)
  - read:  $\prod n:\text{nat}.\text{vector } n \rightarrow (\text{and } (\text{ge } 0) (\text{lt } n)) \rightarrow \text{int}$
- The **type checker must do program verification**

# Dependent Type Commentary

- Type checking with  $\Pi$  types can be *as hard as full program verification*
- Type equivalence can be **undecidable**
  - If types are dependent on expressions drawn from a powerful language (“powerful” = “arithmetic”)
  - Then even **type checking will be undecidable**
- Dependent types play an important role in the **formalization of logics**
  - Started with Per Martin-Lof
  - **Proof checking via type checking**
  - Proof-carrying code uses a dependent type checker to check proofs
  - There are program specification tools based on  $\Pi$  types

# Dependent Sum Types

- We want to pack **a vector** with **its length**
  - $e = (n, v)$  where “ $v : \text{vector } n$ ”
  - The type of an element of a pair depends on the *value* of another element
  - This is another form of dependency
  - The type of  $e$  is “ $\text{nat} \times \text{vector } ?$ ”

- Given two sets  $A$  and  $B$  verify the isomorphism

$$A \times B \simeq \sum_{x \in A} B$$

- The latter is the *disjoint union* of  $B$  with itself as many times as there are elements in  $A$
- Also written as  $\sum x:A. B$  ( $x$  here plays no role)
- But now we can make  $B$  depend on  $x$ !



# Dependent Sum Types

- Definition:  $\Sigma x:A.B$  is the type of **pairs** with first element of type A and second element of type B (*possibly depending on the value of first element x*)
  - Now we can write  $e : \Sigma x:\text{nat}. \text{vector } x$
- Old functions that compute the length of a vector
  - $\text{vlength} : \Pi n:\text{nat}. \text{vector } n \rightarrow \text{nat}$ 
    - (the result is not constrained)
  - $\text{length} : \Pi n:\text{nat}. \text{vector } n \rightarrow \text{sng } n$ 
    - “sng n” is a dependent type that contains only n
    - called the singleton type (recall from 3 slides ago ...)
- What if the vector is packed with its length?
  - $\text{pvlength} : \Sigma n:\text{nat}. \text{vector } n \rightarrow \text{nat}$
  - $\text{pslength} : \Sigma n:\text{nat}. \text{vector } n \rightarrow \text{sng } n$

# Dependent Sum Types

## Static Semantics

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : [e_1/x]\tau_2}{\Gamma \vdash (e_1, e_2) : \Sigma x : \tau_1. \tau_2}$$

$$\frac{\Gamma \vdash e : \Sigma x : \tau_1. \tau_2}{\Gamma \vdash \text{snd } e : [\text{fst } e/x]\tau_2}$$

- Note how this rule **reduces to the usual rules** for tuples when there is no dependency
- The evaluation rules are **unchanged**

# Weimeric Commentary

- Dependant types seem obscure: why care?
- Grand Unified Theory
  - Type Checking = Verification (= Model Checking = Proof Checking = Abstract Interpretation ...)
- CCured Project
  - Rumor has it this project was successful
  - The whole thing is dependant sum types
    - SEQ = (pointer + lower bound + upper bound)
    - FSEQ = (pointer + upper bound)
    - WILD = (pointer + lower bound + upper bound + rtti)

# Types for Data Abstraction

What's inside the implementation?  
We don't know!

QUESTIONS NOT EVEN 5+ YEARS OF GRAD SCHOOL WILL HELP YOU ANSWER



PHD IN ENVIRONMENTAL ENGINEERING



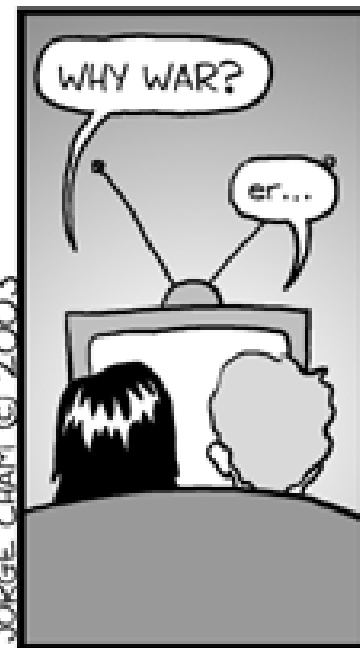
PHD IN PHYSICS



PHD IN BIOLOGY



PHD IN MECHANICAL ENGINEERING



PHD IN POLITICAL SCIENCE

# Data Abstraction

- Ability to **hide (abstract) concrete implementation details**
- **Modularity** builds on data abstraction
- Improves program structure and minimizes dependencies
- One of the most influential developments of the 1970's
- Key element for much of the success of object orientation in the 1980's

# Example of Abstraction

- Cartesian points (gotta love it!)
- Introduce the “**abstype**” language construct:
  - abstype** point **implements**
    - mk : real × real → point
    - xc : point → real
    - yc : point → real
  - is**
    - < point = real × real,
    - mk = λx. x,
    - xc = fst,
    - yc = snd >
  - Shows a concrete implementation
  - Allows the rest of the program to access the implementation *through an abstract interface*
  - Only the interface need to be publicized
  - Allows **separate compilation**

# Data Abstraction

- It is useful to **separate the creation of the abstract type and its use** (newsflash ...)
- Extend the syntax ( $t = \text{imp}$ ,  $\sigma = \text{interface}$ ):  
Terms ::= ... |  $\langle t = \tau, e : \sigma \rangle$  | open  $e_a$  as  $t, x : \sigma$  in  $e_b$   
Types ::= ... |  $\exists t. \sigma$
- The expression  $\langle t = \tau, e : \sigma \rangle$  takes the concrete implementation  $e$  and “**packs it**” as a value of an abstract type
  - Alternative notation: “pack  $e$  as  $\exists t. \sigma$  with  $t = \tau$ ”
  - “existential types” - used to model the stack, etc.
- The “**open**” expression allows  $e_b$  to access the abstract type expression  $e_a$  using the name  $x$ , the unknown type of the **concrete implementation “t”** and the **interface  $\sigma$**

# Example with Abstraction

- $C = \{mk = \lambda x.x, xc = fst, yc = snd\}$  is a *concrete implementation* of points as  $real \times real$
- We want to hide the type of the representation  $\sigma$  is the following type:  
$$\{ mk : real \times real \rightarrow point, \\ xc : point \rightarrow real, yc : point \rightarrow real \}$$
- Note that  $C : [real \times real / point] \sigma$
- $A = \langle point = real \times real, C : \sigma \rangle$  is an expression of the abstract type  $\exists point. \sigma$
- We want clients to access only the second component of  $A$  and just use the abstract name “point” for the first component:  
 $open\ A\ as\ point, P : \sigma\ in\ \dots P.xc(P.mk(1.0, 2.0)) \dots$



# Typing Rules for Existential Types

- We add the following typing rules:

$$\frac{\Gamma \vdash [\tau/t]e : [\tau/t]\sigma}{\Gamma \vdash \langle t = \tau, e : \sigma \rangle : \exists t.\sigma}$$

$$\frac{\Gamma \vdash e_a : \exists t.\sigma \quad \Gamma, t, p : \sigma \vdash e_b : \tau}{\Gamma \vdash \text{open } e_a \text{ as } t, p : \sigma \text{ in } e_b : \tau} \quad t \notin FV(\Gamma \cup \tau)$$

- The restriction in the rule for “open” ensures that **t does not escape its scope**

# Evaluation Rules for Abstract Types

- We add a new form of value

$$v ::= \dots \mid \langle t = \tau, v : \sigma \rangle$$

- This is **just like**  $v$  but with some type decorations that make it have an existential type

$$e_a \Downarrow \langle t = \tau, v : \sigma \rangle \quad [v/x][\tau/t]e_b \Downarrow v'$$

---

$$\text{open } e_a \text{ as } t, x : \sigma \text{ in } e_b \Downarrow v'$$

- At the time  $e_b$  is evaluated, abstract-type variables are replaced with concrete values
  - If we ignore the type issues “open  $e_a$  as  $t, x : \sigma$  in  $e_b$ ” is like “let  $x : \sigma = e_a$  in  $e_b$ ”
  - Difference:  $e_b$  *cannot know statically* what is the concrete type of  $x$  so it cannot take advantage of it

# Abstract Types as a Specification Mechanism

- Just like polymorphism, **existential types are mostly a type checking mechanism**
- A function of type  $\forall t. t \text{ List} \rightarrow \text{int}$  does not know *statically* what is the type of the list elements. Therefore no operations are allowed on them
  - But it will have at run-time the actual value of  $t$
  - “There are no type variables at run-time”
- Same goes for existentials
- These type mechanisms are a very powerful (and widely used!) **form of static checking**
  - Recall Wadler’s “Theorems for Free”

## Q: Games (540 / 842)

- This seminal 1991 turn-based strategy computer game by Sid Meier of Microprose spawned an entire genre about micromanaging exploration, expansion and conflict.

## Q: Games (543 / 842)

- His genre-spawning 1993 game, "affectionately" referred to as "crack for gamers", was later inducted into the **GAMES Magazine** and **Origins Halls of Fame**. Name this game designer, who also holds a doctorate in mathematics.

## Q: Books (754 / 842)

- Name the factory owner, the workers, and the newly-developed form of unending suckable candy in the 1964 children's book that features the title character finding a golden ticket and visiting the title chocolate factory.

# Data Abstraction and the Real World

- Example: **file descriptors**
- Solution 1:
  - Represent file descriptors as “**int**” and export the interface **{open:string→int, read:int→ data}**
- An untrusted client of the interface calls “read”
- How can we know that “read” is invoked with a file descriptor that was obtained from “open”? Anyone?

# Data Abstraction and the Real World

- Example: **file descriptors**
- Solution 1:
  - Represent file descriptors as “int” and export the interface `{open:string→int, read:int→data}`
- An untrusted client of the interface calls “read”
- How can we know that “read” is invoked with a file descriptor that was obtained from “open”?
  - We must **keep track of all integers that represent file descriptors**
  - We design the interface such that all such integers are small integers and we can essentially keep a bitmap
  - This becomes expensive with more complex (e.g. pointer-based) representations



# Data Abstraction, Static Checking

- Solution 2: Use the same representation but *export an abstraction* of it.
  - $\exists fd. \text{File}$  or
  - $\exists fd. \{\text{open} : \text{string} \rightarrow fd, \text{read} : fd \rightarrow \text{data}\}$
  - A possible value:
    - $\text{Fd} = \langle fd = \text{int}, \{\text{open} = \dots, \text{read} = \dots\} : \text{File} \rangle : \exists fd. \text{File}$
- Now the *untrusted* client  $e$ 
  - $\text{open Fd as fd, x : File in } e$
- At run-time “ $e$ ” can see that file descriptors are integers
  - But cannot cast 187 as a file descriptor.
  - Static checking with no run-time costs!
  - Catch: you must be able to type check  $e$ !

# Modularity

- A module is a program fragment along with *visibility constraints*
- Visibility of functions and data
  - Specify the function interface but hide its implementation
- Visibility of type definitions
  - More complicated because the type might appear in specifications of the visible functions and data
  - Can use data abstraction to handle this
- A module is represented as a *type component* and an *implementation component*
  - $\langle t = \tau, e : \sigma \rangle$  (where  $t$  can occur in  $e$  and  $\sigma$ )
  - even though the specification ( $\sigma$ ) refers to the implementation type we can still hide the latter
  - But there are problems ...

# Problems with Existentialists

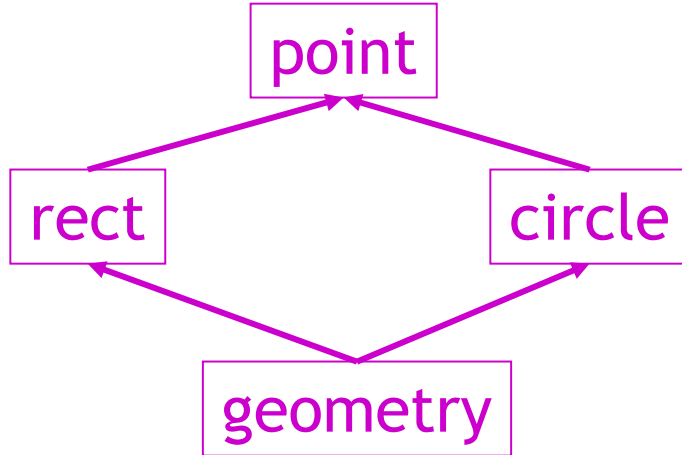
- Existentialist types
  - Assert that **truth is subjectivity**
  - **Oppose the rational tradition** and positivism
  - Are subject to an “**absurd**” universe
- Problems:
  - "In so far as Existentialism is a philosophical doctrine, it remains an idealistic doctrine: it hypothesizes specific historical conditions of human existence into ontological and metaphysical characteristics. **Existentialism thus becomes part of the very ideology which it attacks**, and its radicalism is illusory." (Herbert Marcuse, "Sartre's Existentialism", p. 161)

# Problems with Existentials

- Existential types
  - Allow **representation (type) hiding**
  - Allow **separate compilation**. Need to know only the type of a module to compile its client
  - **First-class modules**. They can be selected at run-time. (cf. OO interface subtyping)
- Problems:
  - **Closed scope**. Must open an existential before using it!
  - Poor support for **module hierarchies**

# Problems with Existentials (Cont.)

- There is an inherent tension between **handling modules in isolation** (good for **separate compilation**, interchangeability) and the need to **integrate them**



(the arrow means “depends on”)

- Solution 1: **open “point” at top level**
  - Inversion of program structure
  - The most basic construct has the widest scope

# Give Up Abstraction?

- Solution 2: incorporate point in rect and circle
$$R = \langle \text{point} = \dots, \langle \text{rect} = \text{point} \times \text{point}, \dots \rangle \dots \rangle$$
$$C = \langle \text{point} = \dots, \langle \text{circle} = \text{point} \times \text{real}, \dots \rangle \dots \rangle$$
- When we open R and C we get *two distinct notions of point!*
  - And we will *not be able to combine them*
- Another option is to allow the type checker to see the representation type
  - and thus give up representation hiding

# Strong Sums

- New way to open a package

Terms  $e ::= \dots \mid \text{Ops}(e)$

Types  $\tau ::= \dots \mid \Sigma t. \tau \mid \text{Typ}(e)$

- Use **Typ** and **Ops** to decompose the module
- Operationally, they are just like “fst” and “snd”
- $\Sigma t. \tau$  is the **dependent sum type**
- It is like  $\exists t. \tau$  except we can look at the type

$$\frac{\Gamma \vdash e : \Sigma t. \tau}{\Gamma \vdash \text{Ops}(e) : \tau[\text{Typ}(e)/t]}$$

# Modularity with Strong Sums

- Consider the R and C defined as before:

$Pt = \langle \text{point} = \text{real} \times \text{real}, \dots \rangle : \Sigma \text{point}. \tau_p$

$R = \langle \text{point} = \text{Typ}(Pt),$

$\quad \langle \text{rect} = \text{point} \times \text{point}, \dots \rangle : \Sigma \text{rect}. \tau_R$

$C = \langle \text{point} = \text{Typ}(Pt),$

$\quad \langle \text{circle} = \text{point} \times \text{real}, \dots \rangle : \Sigma \text{circle}. \tau_C$

- Since we use strong-sums the **type checker sees that the two point types are the same**



# Modules with Strong Sums

- ML's module system is based on strong sums

Problems:

- **Poorer data abstraction**
- Expressions appear in types ( $\text{Typ}(e)$ )
  - Types might not be known until at run time
  - **Lost separate compilation**
  - Trouble if  $e$  has side-effects (but we can use a value restriction - e.g., "IntSet.t")
- **Second-class modules** (because of value restriction)
- We can combine existentials with strong sums
  - Translucent sums: partially visible

# Homework

- Project!
  - You have ~21 days (including holidays) to complete it.
  - Need help? Stop by my office or send email.