1) Dependent Type Systems
2) Data Abstraction and modularity

Claire Le Goues
Grad PL, 11/15/11

Previously, in grad PL…

- We’ve studied a variety of type systems.
- We have made the type system more expressive over time: new errors, better programs, happier programmers.
  - Examples: exceptions, polymorphism, recursion…
- But! We have avoided undecidable systems.
  - Implication: *there are many errors that cannot be caught by the type systems we’ve discussed so far.*

And now, in grad PL…

- More complex type systems that bring type checking closer to program verification:
  1. Dependent types
  2. Types for data abstraction and modularity

And now, in grad PL…

- More complex type systems that bring type checking closer to program verification:
  1. **Dependent types**
  2. Types for data abstraction and modularity

Proximate cause/recent review

- Theorem proving is Wicked Useful and can determine if things are true or false: "your file system can seg fault", "this formula is satisfiable."
- However, we also want theorem provers to provide **checkable** proofs to back up what they decide.
- Fortunately, "proof checking is equivalent to type checking in a dependent type system."

Proximate cause/recent review

- Theorem proving is Wicked Useful and can determine if things are true or false: "your file system can seg fault", "this formula is satisfiable."
- However, we also want theorem provers to provide **checkable** proofs to back up what they decide.
- **Fortunately, "proof checking is equivalent to type checking in a dependent type system."**

A **dependent type** is a type that depends on a value.

(A somewhat-circular one-sentence spoiler)

(Simpler) Motivating Example

- Consider functions that manipulate vectors:

```
zero : nat → vector
dotprod : vector → vector → real
```

- Consider code that uses these functions:

```
let v1 = zero 5
let v2 = zero 15
  dotprod v1 v2
```

(Simpler) Motivating Example

- Consider functions that manipulate vectors:

```
zero : nat → vector
dotprod : vector → vector → real
```

- Consider code that uses thes

This is super wrong!

```
let v1 = zero 5
let v2 = zero 15
  dotprod v1 v2
```

(Simpler) Motivating Example

- Consider functions that manipulate vectors:

```
zero : nat → vector
dotprod : vector → vector → real
```

- Consider code that uses these fun

```
let v1 = zero 5
let v2 = zero 15
  dotprod v1 v2
```

Can't our type system stop us from doing this?

Dependent type-based solution

- Plan: make `vector` a type *family* that is annotated by a natural number corresponding to a length.
- `vector n` is the type of vectors of length `n`
  - Example: `<2,3,4> : vector 3`
- Now our example functions look like:

```
dotprod : vector n → vector n → real
zero : nat → vector ???
```

Dependent type-based solution

- Plan: make `vector` a type *family* that is annotated by a natural number corresponding to a length.
- `vector n` is the type of vecto
  - Example: `<2,3,4> : vec`
- Now our example functions

How do we refer to the value of the argument to `zero` in this type?

```
dotprod : vector n → vector n → real
zero : nat → vector ???
```

Notation FGJ Part 1
-or-
Dependent Product Types

Dependent Product Types

- Lets us model functions whose result type may vary based on the input value (like new-and-improved `zero`!).
- Given sets `A` and `B`:

$$\boxed{A \to B \cong \prod_{X \in A} B}$$

These two things are isomorphic!

- The latter is the cartesian product of B with itself as many times as there are elements in A.
  - Also written as:

$$\boxed{\Pi x : A.B}$$

Dependent Product Types

- Lets us model functions whose result type may vary based on the input value (like new-and-improved `zero`!).
- Given sets `A` and `B`:

$$\boxed{A \to B \cong \prod_{x \in A} B}$$

These two things are isomorphic!

an product of B with itself as many

ents in A.

as:

$$\boxed{\Pi x : A.B}$$

So far, we've done nothing with `x`!

Dependent Product Types

- Lets us model functions whose result type may vary based on the input value (like new-and-improved `zero`!).
- Given sets `A` and `B`:

$$\boxed{A \rightarrow B \cong \prod_{X \in A} B}$$

These two things are isomorphic!

- The latter is the cartesian product of B with itself as many times as there are elements in A.
  - Also written as:

$$\boxed{\Pi x : A.B}$$

But now we can make B depend on x!

Product type: Definition

$\boxed{\Pi x : A.B}$ is the type of functions with arguments in `A` and the result type `B` (where `B` possibly depends on type `x` in `A`).

- We can now write `zero` as:

```
dotprod : vector n → vector n → real
zero : nat → ∏x:nat.vector x
```

- When x is not in B we can just write A → B; we play "fast and loose" with the binding of $\prod$.

New Static Semantics

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

- Note that expressions are now part of types.
  - Ex: types like "`vector 5`" and "`vector (2+3)`"
- We need **type equivalence:**

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

Dependent Types and Program Specifications

- Types act as **specifications**
- We can specify *any property* (assuming appropriate typing rules for the new types). Examples:
  - `eq e` or `sng e`: the type of values equal to e (the singleton type).
  - `ge e, lt e:` the type of values $\geq$, < `e`.
  - `and` $\tau_1 \tau_2$: the type of values having both type τ_1 and τ_2
- Vector-accessing:

```
read: Πn:nat.vector n → (and (ge 0)(lt n)) → int
```

- The **type checker does program verification.**

Additional Commentary

- Type checking with Π types can be *as hard as full program verification*.
- Type equivalence – and checking – can be undecidable, if types depend on expressions drawn from a powerful language (e.g., arithmetic).
- 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.
  - There are program specification tools based on Π types

Notation FGJ Part 2
-or-
Dependent Sum Types

Dependent Sum Types: Vectors, Take 2

- Alternative: pack a vector with its length.
  - `e=(n,v)`, where `v : vector n`
  - Type: `e : nat x vector ??`
- **Dependent sum types**: the type of a pair where one element depends on the *value* of another element of the pair.
- Given sets A and B:
  (also an isomorphism!)

$$\boxed{A \times B \cong \Sigma_{x \in A} B}$$

- The latter is the ***disjoint* union** of `B` with itself as many times as there are elements in `A`.

$$\boxed{\Sigma x : A.B}$$

- Alternative notation. Again, x plays no role, but now we can make B depend on it.

Sum type: Definition

$\boxed{\Sigma x : A.B}$ 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`).

- We can now write `e`'s type as `e : Σx:nat.vector x`
- Old functions to compute the length of a vector:

```
vlength : Πn:nat.vector n → nat
slength: Πn:nat.vector n → sng n
```

- Packed with its length:

```
pvlength : Σn:nat.vector n → nat
pslength: Σn:nat.vector n → sng n
```

More Static Semantics

$$\boxed{\frac{\Gamma \vdash e_1 : \boldsymbol{\tau}_1 \quad \Gamma \vdash e_2 : [e_1 / x]\boldsymbol{\tau}_2}{\Gamma \vdash (e_1, e_2) : \Sigma x : \boldsymbol{\tau}_1.\boldsymbol{\tau}_2}}$$

$$\boxed{\frac{\Gamma \vdash e : \Sigma x : \boldsymbol{\tau}_1.\boldsymbol{\tau}_2}{\Gamma \vdash snd\ e : [fst\ e / x]\boldsymbol{\tau}_2}}$$

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

What did this all have to do with proofs, again?

Reminder: Proof Representation

- Proofs are trees; leaves are hypotheses/axioms; internal nodes are inference rules.
- **Problem:** `andel truei: pf` ← this only says that and elimination has type `proof`, but not whether it is proving something true in a valid way.
- *The **type** of the proof has nothing to do with the **values** of the thing that's being proven.*
  - Fortunately, we just discussed a solution to this problem.

Dependent type solution

- `pf` is now a family of types indexed by (or dependent on, if you prefer) formulas.
  - `f : Type` (type of encodings of formulas)
  - `e : Type` (type of encodings of expressions)
  - `pf : f → Type` (type of proofs indexed by formulas: a proof *that f is true*)
- Examples (that may make more sense now):

```
true : f
and : f → f → f
truei : pf true
andi : pf A → pf B → pf (and A B)
andi: ΠA:f.ΠB:f.pf A → pf B → pf (and A B)
```

Proof Checking

- We validate proof trees by type-checking them: given a proof tree `X` claiming to prove `A ∧ B`, we check `X : pf (and A B)`
- Thus **Type Checking = Proof Checking (…dependent types)**
  - Type checking your types involves additional fancy math (including kinds). I am helpfully eliding, though I assert it's fun.

"Weimeric Commentary"

- Dependent types seem obscure: why care?
- Grand Unified Theory: Type Checking = Verification (= Model Checking = Proof Checking = Abstract Interpretation …)
- Also, useful: rumor has it the CCured Project was successful. Turns out the whole thing is dependent sum types:
  - SEQ = (pointer + lower bound + upper bound)
  - FSEQ = (pointer + upper bound)
  - WILD = (pointer + lower bound + upper bound + rtti)

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, game designer, and/or the field in which the designer holds a doctorate.

And now, in grad PL…

- More complex type systems that bring type checking closer to program verification:
  1. **Dependent types**
  2. Types for data abstraction and modularity

And now, in grad PL…

- More complex type systems that bring type checking closer to program verification:
  1. Dependent types
  2. Types for data abstraction and modularity

And now, in grad PL…

- More complex type systems that bring type checking closer to program verification:
  1. Dependent types
  2. **Types for data abstraction and modularity**

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.
  - Note: abstract data types and objects are not actually the same thing, but the underlying concepts are similar.

An **abstract data type** has a public name, a hidden representation, and operations to create, combine, and observe values of the abstraction.

(Another circular one-sentence spoiler)

Example: Cartesian Points

- Introduce the **abstype** construct for creating abstract types.

```
abstype point implements
   mk : real x real → point
   xc : point → real
   yc : point → real
is
   <point = real x real,
    mk = λx.x,
    xc = fst,
    yc = snd >
```

- This is a concrete implementation.
- The rest of the program accesses the implementation *through an abstract interface*

- Only the interface need to be publicized; allows separate compilation

Example: Cartesian Points

- Introduce the **`abstype`** construct for creating abstract types.

```
abstype point implements
   mk : real x real → point
   xc : point → real
   yc : point → real
is
   <point = real x real,
    mk = λx.x,
    xc = fst,
    yc = snd >
```

- This is a **concrete implementation**.
- The rest of the program accesses the implementation *through an abstract interface*
- Only the interface need to be publicized; allows separate compilation

Example: Cartesian Points

- Introduce the **`abstype`** construct for creating abstract types.

```
abstype point implements
   mk : real x real → point
   xc : point → real
   yc : point → real
is
   <point = real x real,
    mk = λx.x,
    xc = fst,
    yc = snd >
```

- This is a **concrete implementation**.
- The rest of the program accesses the implementation *through an abstract interface*

- Only the interface need to be publicized; allows separate compilation

Example: Cartesian Points

- Introduce the **abstype** construct for creatin...

```
abstype point implements
   mk : real x real → point
   xc : point → real
   yc : point → real
is
   <point = real x real,
    mk = λx.x,
    xc = fst,
    yc = snd >
```

Can we hide the type of the concrete implementation C?

- ...
- Th... ...ogram accesses the implementation *through an abstract interface*

- Only the interface need to be publicized; allows separate compilation

Existential Types

- If σ is the type: `{ mk : real x real → point`
  `xc : point → real`
  `yc : point → real }`
- Notice! `C : [real x real/point]` σ
- Expression `A = <point = real x real, C :` σ`>` has **abstract type** $\exists$**`point.`**σ
- We want clients to access the second component of `A` and just use the abstract name `point` for the first component:

  `open` `A` **`as`** `point, P :` σ `in …`

Data Abstraction

- New syntax (`t` = implementation, σ = interface):

  Types ::= ... | $\exists$**t.σ**

  Terms ::= ... | **<t=**τ **,e:**σ **>**

  | **open** e_a **as** t, x:σ **in** e_b

- The expression `<t=`τ `, e :` σ`>` takes the concrete implementation `e` and "packs it" as a value of an abstract type.
  - Alternative notation: **pack** `e` **as** $\exists$`t.`σ **with** `t =` τ

- 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 σ .

Typing rules for existential types

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

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

- 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 ::= … | <t=`τ` , v:`σ`>`
  - This is *just like* `v`, but with type decorations that give it an existential type.

$$\boxed{\frac{e_a \Downarrow < t = \tau, v : \sigma > \quad [v/x][\tau/t]e_b \Downarrow v'}{open\ e_a\ as\ t, x : \sigma\ in\ e_b \Downarrow v'}}$$

- 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:`σ **in** e_b is just like **let** `x:`σ`=` e_a **in** e_b
  - Difference: e_b *cannot statically know* 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 `∀t.t List → int` does not *statically* know the type of the list elements; no operations are allowed on them.
- But the actual value of `t` is eventually available; "there are no type variables at run-time." **The same goes for existentials**.
- These type mechanisms are a very powerful (and widely used!) form of static checking
  - Recall Wadler's "Theorems for Free"

Real world example: file descriptors

- Solution 1: Represent file descriptors as `int` and export the interface

  `{open:string → int, read:int → data}`

- How can we know that `read` is invoked by untrusted clients with a file descriptor that was obtained from `open`?
  - We must track all integers representing file descriptors.
  - Design the interface such that all such integers are small so we can essentially keep a bitmap for run-time tracking.
- This becomes **expensive** with more complex (e.g. pointer-based) representations.

File descriptors with static checking

- Solution 2: Use the same representation but *export an abstraction* of it.

  `∃fd.File`, or `∃fd.{open : string → fd,`
  `read : fd → data}`

  - A possible value:

  `F=<fd=int,{open=…,read=…}:File>: ∃fd.File`

- The *untrusted* client can do: **`open`** `Fd` **`as`** `fd,x:File` **`in`** `e`
  - At run-time, `e` can see that file descriptors are integers, but still can't cast 7 as a file descriptor. Checking, but no run-time costs!
  - Catch: you must be able to type check `e`

A **module** is a program fragment along with visibility constraints.

(this one isn't circular, actually)

Modularity

- Visibility of:
  - **functions and data:** specify the function interface but hide its implementation.
  - **type definitions:** more complicated because the type might appear in specifications of the visible functions and data, but we can use data abstraction to handle this
- A module is represented as a type component and an implementation component $\texttt{<t=}\tau\texttt{,e:}\sigma\texttt{>}$ (where $\texttt{t}$ can occur in $\texttt{e}$ and σ)
  - We kind hide the implementation type even though the specification (σ) may refers to it.

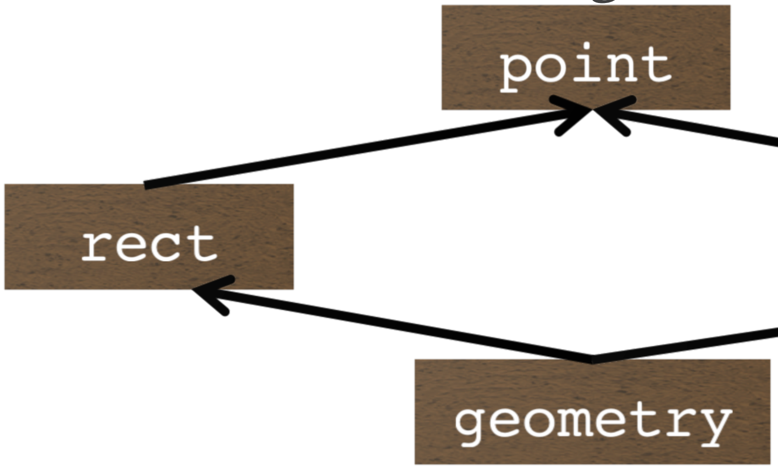
But there are problems…

Problems with existentials

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

Problems, continued

- 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 = <point = …, <rect = point x point,…> …>
  C = <point = …, <circle = point x real, …> …>
  ```

- BUT: when we open `R` and `C` we get *two distinct notions of point*! We will ***not* be able to combine them.**
  - No drawing circles around rectangles. Sad.
- Another option: allow the type checker to see the representation type.
  - (give up representation hiding)

Solution: strong sums

- New way to open a package:

```
Types ::= ... | Σt.τ | Typ(e)
Terms ::= ... | Ops(e)
```

- Use `Typ` and `Ops` to decompose modules.
  - Operationally just like `fst` and `snd`
  - Σ t. τ is the dependent sum type
- Like $\exists$`t.` τ , except we can look at the type:

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

Modularity with strong sums

- Consider the R and C defined as before:

  Pt = <point=real x real, …> : Σpoint. τ_P

  R = <point=Typ(Pt),

  <rect=point x point, …> : Σrect. τ_R

  C = <point=Typ(Pt),

  <circle=point x real, …> : Σcircle. τ_C

- The use of strong-sums means that the type checker sees that the two point types are the same.

Real-world strong sums modules

- ML's module system is based on strong sums.
  - Reasonable compromise in practice.
  - Downsides:
    - Poorer data abstraction
    - Expressions appear in types (`Typ(e)`)
    - Types might not be known until at run time
    - Lost separate compilation
- Trouble if `e` has side-effects. We get around this with value restrictions – e.g., `IntSet.t`) – but this means that modules are second-class.
- Translucent sums can combine existentials with strong sums: partially visible

And now, in grad PL…

- More complex type systems that bring type checking closer to program verification:
  1. Dependent types
  2. **Types for data abstraction and modularity**

And now, in grad PL…

- More complex type systems that bring type checking closer to program verification:
  1. Dependent types
  2. Types for data abstraction and modularity

From Wes: Homework

- Project!
- You have ~14 days (including Thanksgiving) to complete it.
- Need help? Stop by Wes’s office or send email.

Fireside Chat

Free Food

with …

Jason Lawrence

abhi shelat

Thursday, Nov 17

5:00 pm

Thornton D221

Meet and ask them questions in a non-academic setting.

Learn what they wish they had known when they were students, and what their lives are like outside of the office.

Ask them anything!
All are welcome.

One was a standup comic, the other runs marathons. Neither has had a haircut!

ACM-W

DIVERSITY

Computer Science
University of Virginia