Dependent Types

October 18th, 2014
Kristopher Micinski
What are they?

1 : \textit{Nat} \quad [1] : \textit{List(Nat)}

First order terms and types

Types depend on \textit{types}

(\lambda x \ y \ z. \ \text{if}0 \ x \ \text{then} \ y \ \text{else} \ z)

Terms depend on \textit{terms}

(\text{Animal} \ a).\text{speak}()

Terms depend on \textit{terms}

[1]@[2] : \textit{Vector}(1 + 1)

Types depend on \textit{terms}
Lambda cube

Terms depend on types

STLC

Types depend on terms

LF

Types depend on types

Coq
What about when our programs go wrong?
Therac 25

$60 Billion a year in bugs

Mars Climate Orbiter

Patriot missile launcher

“10 historical software bugs with extreme consequences” — Pingdom, March 19, 2009.
How do we verify software?
What do we need?

- **Program** — we want to talk about
- **Specification** — say when it’s correct
- **Verification** — show program meets spec
- **Validation** — show system meets end to end goals
Probably basis of modern PL

- This is why Robert Floyd and Tony Hoare are famous

```
{ a = m ∧ b = n ∧ n ≥ 0 }
{ a^b * 1 = m^n ∧ b ≥ 0 }
c := 1 ;
{ a^b * c = m^n ∧ b ≥ 0 }
while b > 0
do
\{ a^b * c = m^n ∧ b ≥ 0 ∧ b > 0 \}
while 2 * (b div 2) = b
do
\{ a^b * c = m^n ∧ b ≥ 0 ∧ b > 0 \}
\{ (a^b)^{b \text{ div} 2} * c = m^n ∧ (b \text{ div} 2) > 0 \}
a := a * a ;
\{ a^{b^2} + c = m^n ∧ (b \text{ div} 2) > 0 \}
b := b \text{ div} 2
\{ a^b * c = m^n ∧ b > 0 \}
{ a^{b+1} * a * c = m^n ∧ b + 1 ≥ 0 }
b := b - 1 ;
\{ a^b * a * c = m^n ∧ b ≥ 0 \}
c := a * c
\{ a^b * c = m^n ∧ b ≥ 0 \}
\{ a^0 * c = m^n \}
\{ c = m^n \}
```
Bunch of different techniques

- Program logic
  - Compositionally build programs with **pre** and **post** conditions
- Model checking
  - Write what program **should** and **shouldn’t** do
  - Check formula over **abstraction** of the program
- Program analysis
  - E.g., dataflow / control flow / abstract interpretation
  - “Is this pointer ever null?”
Going to study dependent types

- Type system give us compositional static static checks
- Key insight: stuff arbitrarily complex logic into types
- Now we can verify a program by type checking it
  - Type checking may be undecidable
- Verification: only one way to look at dependent types
  - Will discuss exactly what I mean
Preliminaries

• This is *intricate*

  • **Complex** systems can be *deceptively small*

• I’ll be working a lot with the natural numbers.

  • \(0 : \text{nat}\)

  • \(k : \text{nat}, S(k) : \text{nat}\)

  • \(S(k) \text{ is } k + 1\)

• Lots of follow up work if you’re interested
Use **types** as the specification

Program has type if it **satisfies** the property
Types are going to have to be more complex…
\[ \lambda f. (\lambda x. f(xx))(\lambda x. f(xx)) \]

Lambda calculus

Reduction relation

Right?
What is the lambda calculus

- A grammar for forming program terms
  - Tells what programs “look like”
- A reduction relation
  - Tells us what programs “do”
- Denotational semantics
  - What programs “mean” (wrt a mathematical domain)
What is the lambda calculus

• A grammar for forming program terms
  
  • Tells what programs “look like” \((\lambda x. x \ x)(\lambda x. x \ x)\)

• A reduction relation

  • Tells us what programs “do”

• Denotational semantics

  • What programs “mean” (wrt a mathematical domain)

Some programs do something that has no “meaning”

(By which, I mean an undesirable meaning…)
Type systems syntactically rule out "bad" programs

**Syntax**

$$
t ::= \\
\quad x \\
\quad \lambda x : T . t \\
\quad t \\

v ::= \\
\quad \lambda x : T . t \\

T ::= \\
\quad T \rightarrow T \\

\Gamma ::= \\
\quad \emptyset \\
\quad \Gamma , x : T \\
$$

**Evaluation**

$$
\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2} \quad \text{(E-APP1)}
\frac{t_2 \rightarrow t'_2}{\nu_1 \ t_2 \rightarrow \nu_1 \ t'_2} \quad \text{(E-APP2)}
\frac{(\lambda x : T_{11} . t_{12}) \nu_2 \rightarrow [x \rightarrow \nu_2] t_{12}} \quad \text{(E-APPABS)}
$$

**Typing**

$$
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}
\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2} \quad \text{(T-ABS)}
\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \quad \text{(T-APP)}
$$

---

Figure 9-1: Pure simply typed lambda-calculus ($\lambda -$)
Typed Lambda

Complain about type errors

Which would result in runtime errors…

\[ 2 \times \text{canine} = \text{Error: } * 2 \text{ applied to canine when expected Int} \]
type vector = int list

let rec add_vector a b =
  match (a, b) with
  | ([], []) -> []
  | ((h1::t1), (h2::t2)) ->
    h1 + h2 :: add_vector t1 t2
# add_vector [1;2;3] [1;2;3];;
- : int list = [2; 4; 6]
# add_vector [1;2;3] [1;2;3;4];;
Exception: Match_failure ("//toplevel//", 26, 2).

Exceptions are one thing we might wish to rule out...
Borrowing from our previous idea…

Use **types** as the **specification**
What’s the specification...?
add_vector : vector -> vector -> vector

These need to be the same length

forall n:nat,
a: vector(n) ->
b: vector(n) ->
c: vector(n)

For any number n, given vectors a b, of size n, I’ll give you c
Let’s define…

type (n : nat) vector =

Vectors of length n
A vector is a type family

\[ \text{vector} : \text{Nat} \rightarrow \text{Type} \]

Indexed by nat

“Give me a natural, and I’ll give you back a type”

Similar to list…
Similar to…

\[ \text{cons}(1, \text{nil}) : \text{list(nat)} \]

\[ 3 : \text{nat} \]

\[ \text{succ} : \text{nat} \rightarrow \text{nat} \]

All of the terms in our language have a type.

\[ [1, 2] : \text{Vector} \]

\[ [1, 2] : \text{Vector2} \]
Vector is a type `producer` operator

$$\text{vector} :: \text{Nat} \rightarrow *$$

This is an alternative notation that is sometimes used…
We call `Vector n` a “dependent type” because the type depends on `n`.

```
[1, 2] : Vector (1 + 1)
```

Depends on computation `1+1`

This computation must terminate...
Designing a vector API
What can go wrong?

Make it so “bad” programs can’t typecheck…
How Heartbleed transformed HTTPS security into the stuff of absurdist theater

Certificate revocation checking in browsers is "useless," crypto guru warns.

by Dan Goodin - Apr 21 2014, 6:44pm EDT

Ars subscribers get lots of benefits, one of which is they never see ads.

Learn more.
Preventing heartbleed bugs with safe programming languages

The Heartbleed bug in OpenSSL has resulted in a fair amount of damage across the internet. The bug itself was quite simple and is a textbook case for why programming in unsafe languages like C can be problematic.

As an experiment to see if a safer systems programming language could have prevented the bug I tried rewriting the problematic function in the ATS programming language. I've written about ATS as a safer C before. This gives a real world testcase for it. I used the latest version of ATS, called ATS2.

ATS compiles to C code. The function interfaces it generates can exactly match existing C functions and be callable from C. I used this feature to replace the dtls1_process_heartbeat and tls1_process_heartbeat functions in OpnSSL with ATS versions. These two functions are the ones that were patched to correct the heartbleed bug.

The approach I took was to follow something similar to that outlined by John Skaller on the ATS mailing list:

ATS on the other hand is basically C with a better type system. You can write very low level C like code without a lot of the scary dependent typing stuff and then you will have code like C, that will crash if you make mistakes.

If you use the high level typing stuff coding is a lot more work and requires more thinking, but you get much stronger assurances of program correctness, stronger than you can get in Ocaml or even Haskell, and you can even hope for *better* performance than C by elision of run time checks otherwise considered mandatory, due to proof of correctness from the type system. Expect over 50% of your code to be such proofs in critical software and probably 90% of your brain power to go into constructing them rather than just implementing the algorithm. It's a paradigm shift.
Things you might want to do with vectors...

- Create an empty vector
- Add an element to the end
- Add two vectors
- Take a vector to a list
- Get first element of vector
Use types as your guide

When you think about something you’d like to say about your program, you can bake it into your type system
Let’s define two constructors for vectors

• Empty vector
• Cons

```haskell
data list =
    []
  | cons of nat -> list

(* Equivalently ... *)
nil : list
cons : nat -> list -> list
```

Same as lists, but lists don’t carry around their length in their type
“I assert that there is an object named zero, and its type is Vector 0”

```
zero : Vector 0
```

cf.

```
nil : list
```
“Give me a vector of length $n$, and a number to add to the end of it, and I’ll give you a vector of length $S(n)$”

$$\text{cons} : \Pi n. \text{Vector}(n) \to \mathbb{N} \to \text{Vector}(S(n))$$

cf.

$$\text{cons} : \text{list} \to \text{nat} \to \text{list}$$
Now for a few functions over vectors…
“Give me a number n, and I’ll give you an empty vector of size n”

zero : \( \Pi n. \text{Vector}(n) \)

**Terms** always need kind \(*\), so we have to **apply** them until we get there! In this case \( n \).

Read \( \pi \) as \( \forall \)

Not quite right for logical reasons..
This *actually exists*, by the way...

```plaintext
Inductive natvec : nat -> Type :=
  | UnitVec : natvec 0
  | ConsVec : forall n, natvec n -> nat -> natvec (S(n)).

Let a := ConsVec 1 (ConsVec 0 UnitVec 2) 3.

Fixpoint zero_vec n : natvec n :=
  match n with
  | 0    => UnitVec
  | S(n) => ConsVec n (zero_vec n) 0
  end.
```
“For any \( n \), if you give me a vector of size \( S(n) \), I’ll give you back a natural.”

\[
\text{first} : \Pi n : \mathbb{N}. \ Vector(S(n)) \to \mathbb{N}
\]

\( S(n) \) because this guarantees they can’t give us an empty vector, \textit{that’s the magic!}
What happens if I try to add vectors of different lengths?
zero : \( \Pi n.\text{Vector}(n) \)

to_list : \( \Pi n.\text{Vector}(n) \rightarrow \text{list}(\mathbb{N}) \)

\text{cons} : \( \Pi n.\text{Vector}(n) \rightarrow \mathbb{N} \rightarrow \text{Vector}(n + 1) \)

first : \( \Pi n.\text{Vector}(n + 1) \rightarrow \mathbb{N} \)

\text{add} : \( \Pi n.\text{Vector}(n) \rightarrow \text{Vector}(n) \rightarrow \text{Vector}(n) \)
Types are Theorems
Programs are Proofs

–Curry Howard Isomorphism
Curry Howard Isomorphism

Types \textbf{are} Theorems

Programs \textbf{are} Proofs

–Curry Howard Isomorphism

Not all \textit{propositions} have proofs
Not all \textit{types} have programs
Every type is saying something...
23 : int

Proof that 23 is an integer

(Which is admittedly pretty boring...)
v : Vector 1

Proof that \( v \) has length 1

In other words, I don’t have to be afraid that my program is going to crash if I run first \( v \)
Pretty cute
Let’s define something else...
\[ \leq : \mathbb{N} \rightarrow \mathbb{N} \rightarrow * \]
How we define $\leq$

\[
\text{Inductive } \leq \ (n: \text{nat}) : \text{nat} \rightarrow \text{Prop} \ := \\
| \quad \text{le}_n : n \leq n \\
| \quad \text{le}_S : \forall m: \text{nat}, \\
\quad \quad \quad n \leq m \rightarrow n \leq S \ m
\]
“Give me a number $n$, and I’ll give you a proof it’s $\leq$ itself”

$$\text{le}_n : \Pi n. n \leq n$$
“For any $n$ and $m$, if you can give me a proof that $n \leq m$, then I’ll give you a proof that $n \leq S(m)$”

\[ \text{le}_S : \prod n, m. \ n \leq m \rightarrow n \leq S(m) \]

I produce proofs
How do I prove that $0 \leq 1$?
What if I took a logic class IRL (I have)

- “Well, zero is less than or equal to itself (le\_n 0)
  
  - Let’s call that proof pf
- for any n less than or equal to itself, we have this rule that says S(n) is less than or equal to that thing (le\_S)
- So now 0 is less than or equal to S(0) = 1 too (le\_S 0 1 pf)
- So now we know 0 <= 1”
  
  - Can repeat for any finite n >= 0.
Curry Howard Isomorphism

• Not that complicated: read types as theorems

• In math we have modus ponens

\[(A \implies B) \implies A \implies B\]

• In programming we have this function

  \[(A \rightarrow B) \rightarrow A \rightarrow B\]

• Which we usually just call “apply” : )
What's a program that has this type?
\[(A \to B \to C) \to (A \to B) \to A \to C\]
Seriously, *modus ponens* is *really just apply* \(\lambda / \text{app can build the entire universe}\)
Two steps

- Prove 0 \leq 0
- Then use that proof to prove 0 \leq 1

Definition zero_leq_one : 0 \leq 1 := le_S 0 0 (le_n 0).

Computer is going to check proof for us
What about $0 \leq 2$

Definition `zero_leq_two` : $0 \leq 2 :=$

```plaintext
le_S 0 1
(le_S 0 0 (le_n 0)).
```
Automating it...
How do I prove that \( n \leq k \)

- Start with \( n \),

- keep adding \( \text{le}_S \)

  - a lot... \((10 \leq 1000)\) — Going to need hundreds of apps of \( \text{le}_S \)

  - keep going

- give up eventually
The computer can do magic

• Prove a theorem: guess proofs and see if they work
  • Slightly more complicated in reality (search strategy?)
  • Some decision procedures (e.g., omega test)

Theorem zero_leq_two' : 0 <= 2.
Proof.
  auto.
Qed.

Just guess for proofs
\((A \to B \to C) \to (A \to B) \to A \to C\)

Theorem a\_b\_c :
    
    \[
    \forall A B C, \quad (A \to B \to C) \to (A \to B) \to A \to C.
    \]

Proof.
    
    auto.
    
    Qed.

Here’s the proof!

\[ a\_b\_c = \text{fun} \ (A B C : \text{Type}) (X : A \to B \to C) (X0 : A \to B) (X1 : A) \Rightarrow X X1 (X0 X1) \]

: \forall A B C : \text{Type},
    
    \((A \to B \to C) \to (A \to B) \to A \to C\)
Define winning boards as proofs of moves to get to 2048.

Automatic proof corresponds to automatic search strategy for winning 2048 game...
Scale up to real projects: CompCert

- Write logical specs for functions
- E.g., write a spec for a compiler
- Write specs for each pass
  - Prove translation of C to IR preserves semantics
- Chain together a bunch of small steps
  - Prove a compiler correct
Theorem transf_c_program_is_refinement:
forall p tp,
transf_c_program p = OK tp ->
(forall beh, exec_C_program p beh -> not_wrong beh) ->
(forall beh, exec_asm_program tp beh -> exec_C_program p beh).

- 50kloc Coq source
- 8koc source — others are proofs
- No errors
Where can I learn about this stuff!?

Software Foundations
Benjamin C. Pierce
Chris Casinghino
Marco Gaboardi
Michael Greenberg
Catã¬lin Hriçteu
Vilhelm Sjöberg
Brent Yorgey

with Loris D’Antoni, Andrew W. Appel, Arthur Azevedo de Amorim, Arthur Chargueraud, Anthony Cowley, Jeffrey Foster, Michael Hicks, Ranjit Jhala, Greg Morrisett, Mukund Raghothaman, Chung-chieh Shan, Leonid Spesivtsev, and Andrew Tolmach

Certified Programming with Dependent Types
A Pragmatic Introduction to the Coq Proof Assistant
Adam Chlipala

Suitable for beginners

“Real” proof engineering

http://www.cis.upenn.edu/~bcpierce/sf/
http://adam.chlipalala.net/cpdt/
Auxiliary slides...
Just a quick run-through of LF...
### Syntax

**t ::=**

- `x` variable
- `λx:T.t` abstraction
- `t t` application

**T ::=**

- `X` type/family variable
- `Πx:T.T` dependent product type
- `T t` type family application

**K ::=**

- `*` kind of proper types
- `Πx:T.K` kind of type families

**Γ ::=**

- `∅` empty context
- `Γ, x:T` term variable binding
- `Γ, X:::K` type variable binding

Well-formed kinds

\[
\Gamma ⊢ * \quad \text{(WF-STAR)}
\]

\[
\Gamma ⊢ T :::: * \quad \Gamma, x:T ⊢ K
\]

\[
\Gamma ⊢ Πx:T.K \quad \text{(WF-Pi)}
\]

### Kinding

\[
\begin{align*}
& \Gamma ⊢ X :::: K \quad \Gamma ⊢ K \\
& \text{(K-VAR)}
\end{align*}
\]

\[
\begin{align*}
& \Gamma ⊢ T_1 :::: * \quad \Gamma, x:T_1 ⊢ T_2 :::: * \\
& \Gamma ⊢ Πx:T_1.T_2 :::: * \quad \text{(K-Pi)}
\end{align*}
\]

\[
\begin{align*}
& \Gamma ⊢ S :::: Πx:T.K \quad \Gamma ⊢ t :::: T \\
& \Gamma ⊢ S t :::: [x → t]K \quad \text{(K-App)}
\end{align*}
\]

\[
\begin{align*}
& \Gamma ⊢ T :::: K \quad \Gamma ⊢ K ≡ K' \\
& \Gamma ⊢ T :::: K' \quad \text{(K-Conv)}
\end{align*}
\]

### Typing

\[
\begin{align*}
& \Gamma ⊢ t :::: T \quad \text{(Γ-T)}
\end{align*}
\]

\[
\begin{align*}
& x:T ∈ Γ \quad \Gamma ⊢ T :::: * \\
& \Gamma ⊢ x :::: T \quad \text{(T-Var)}
\end{align*}
\]

\[
\begin{align*}
& \Gamma ⊢ S :::: * \quad \Gamma, x:S ⊢ t :::: T \\
& \Gamma ⊢ λx:S.t :::: Πx:S.T \quad \text{(T-Abs)}
\end{align*}
\]

\[
\begin{align*}
& \Gamma ⊢ t_1 :::: Πx:S.T \quad \Gamma ⊢ t_2 :::: S \\
& \Gamma ⊢ t_1 t_2 :::: [x → t_2]T \quad \text{(T-App)}
\end{align*}
\]

\[
\begin{align*}
& \Gamma ⊢ t :::: T \quad \Gamma ⊢ T ≡ T' :::: * \\
& \Gamma ⊢ t :::: T' \quad \text{(T-Conv)}
\end{align*}
\]

---

**Figure 2-1:** First-order dependent types (λLF)
**Syntax**

\[
\begin{align*}
\text{terms:} & \quad x \\
& \quad \lambda x : T . t \\
& \quad t \ t \\
\text{variable} & \quad \text{abstraction} \\
& \quad \text{application}
\end{align*}
\]

\[
\begin{align*}
\text{types:} & \quad X \\
& \quad \Pi x : T . T \\
& \quad T \ t \\
\text{type/family variable} & \quad \text{dependent product type} \\
& \quad \text{type family application}
\end{align*}
\]

\[
\begin{align*}
\text{kinds:} & \quad * \\
& \quad \Pi x : T . K \\
\text{kind of proper types} & \quad \text{kind of type families}
\end{align*}
\]

\[
\begin{align*}
\text{contexts:} & \quad \emptyset \\
& \quad \Gamma , x : T \\
& \quad \Gamma , X :: K \\
\text{empty context} & \quad \text{term variable binding} \\
& \quad \text{type variable binding}
\end{align*}
\]

**Well-formed kinds**

\[
\begin{align*}
\Gamma \vdash \star & \quad \text{(WF-STAR)} \\
\Gamma \vdash \Gamma , x : T & \quad \Gamma \vdash \Pi x : T . K & \quad \text{(WF-Pi)}
\end{align*}
\]

**Kinding**

\[
\begin{align*}
\Gamma \vdash X :: K & \quad \Gamma \vdash x :: K & \quad \text{(K-VAR)} \\
\Gamma \vdash T_1 :: * & \quad \Gamma , x : T_1 \vdash T_2 :: * & \quad \text{(K-PI)} \\
\Gamma \vdash \Pi x : T_1 . T_2 :: * & \quad \text{(K-APP)} \\
\Gamma \vdash S :: \Pi x : T . K & \quad \Gamma \vdash t :: T & \quad \text{(K-CONV)}
\end{align*}
\]

**Typing**

\[
\begin{align*}
\Gamma \vdash \Gamma \vdash x :: T & \quad \text{(T-VAR)} \\
\Gamma \vdash S :: * & \quad \Gamma , x : S \vdash t :: T & \quad \text{(T-ABS)} \\
\Gamma \vdash \Lambda x : S . t :: \Pi x : S . T & \quad \text{(T-ABS)} \\
\Gamma \vdash t_1 :: \Pi x : S . T & \quad \Gamma \vdash t_2 :: S & \quad \text{(T-APP)} \\
\Gamma \vdash t_1 t_2 :: [x \mapsto t_2] T & \quad \text{(T-APP)} \\
\Gamma \vdash t :: T & \quad \Gamma \vdash T :: T' & \quad \Gamma \vdash T :: T' & \quad \text{(T-CONV)}
\end{align*}
\]

**Figure 2-1: First-order dependent types (\(\lambda LF\))**
Figure 2-1: First-order dependent types ($\lambda$LF)
In pure LF types depend on terms, but *can’t depend on types* (we can’t have \texttt{List(A)} )