Dependent Types

October 18th, 2014
Kristopher Micinski

What are they?

$$1 : Nat$$

First order terms and types

$$[1] : List(Nat)$$

Types depend on **types**

$$(\lambda x\ y\ z.\ \mathrm{if0}\, x\ \mathrm{then}\ y\ \mathrm{else}\ z)$$

Terms depend on **terms**

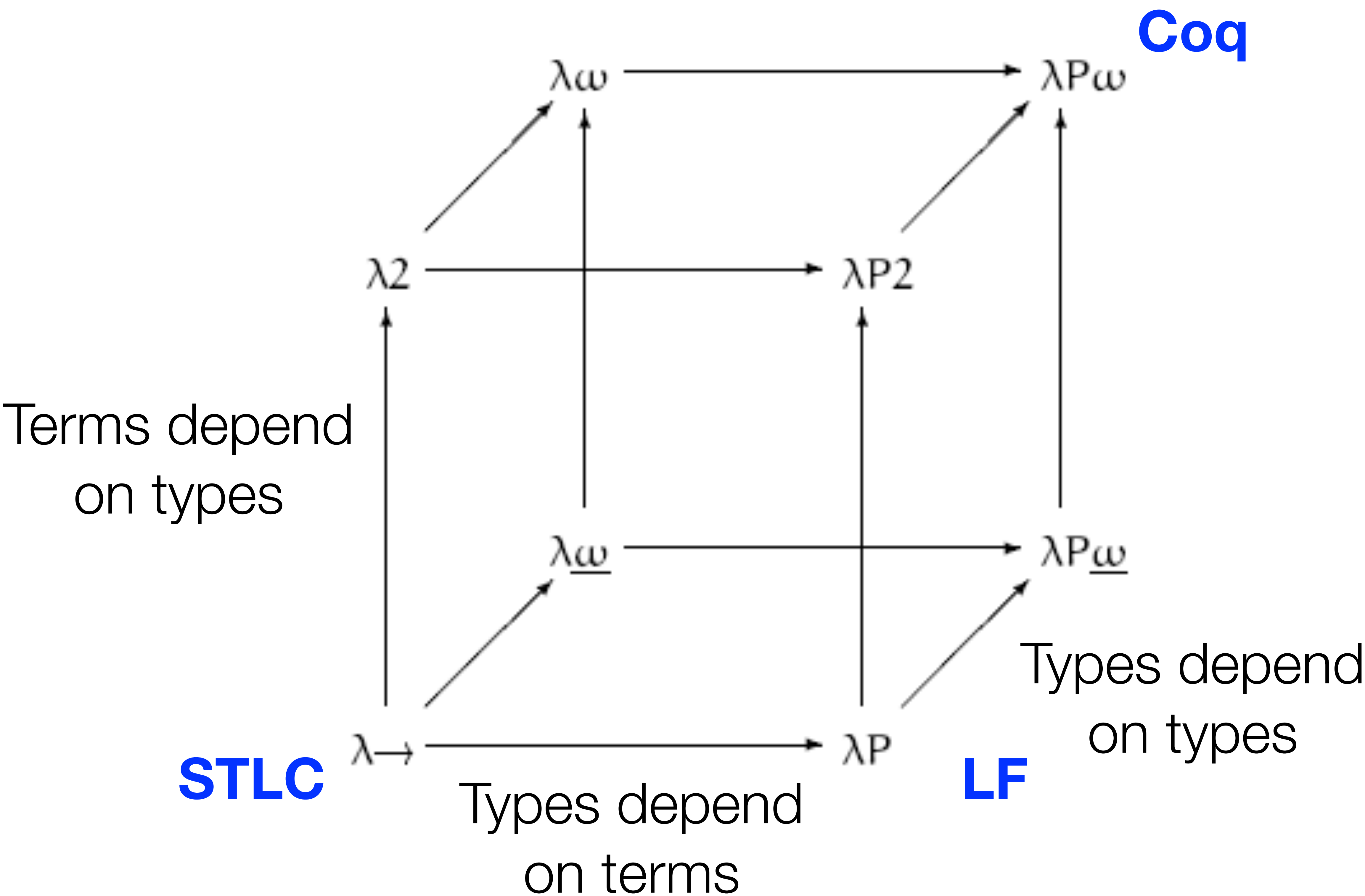
$$(\mathrm{Animal\ a}).\mathrm{speak}()$$

Terms depend on **types**

$$[1]@[2] : Vector(1+1)$$

Types depend on **terms**

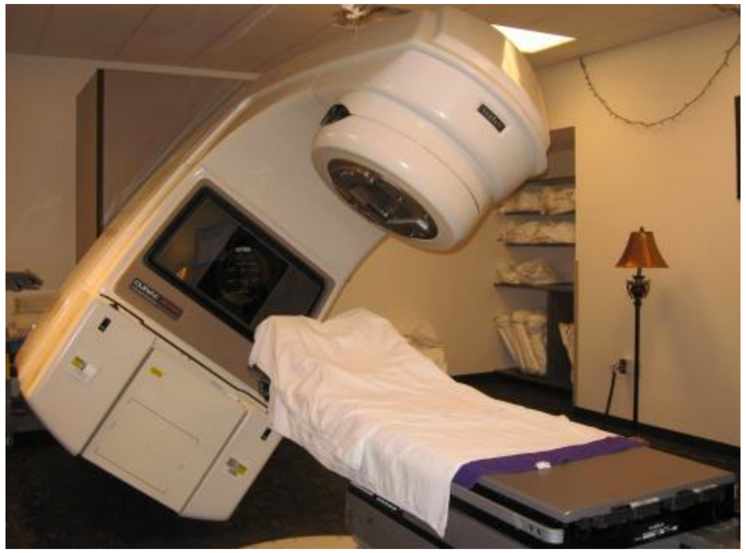
Lambda cube



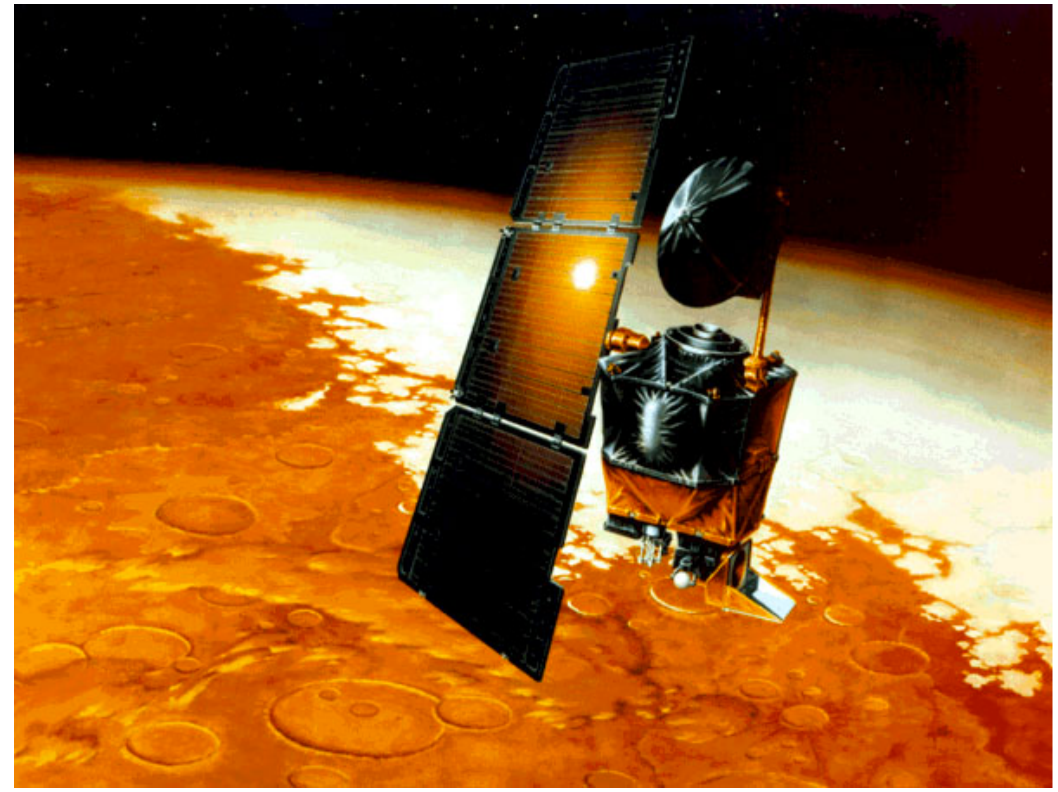
Coq
$\lambda\omega$
$\lambda P\omega$
$\lambda 2$
$\lambda P2$
Terms depend on types
$\lambda\underline{\omega}$
$\lambda P\underline{\omega}$
Types depend on types
STLC
$\lambda\to$
λP
LF
Types depend on terms

What about when our programs **go wrong**?

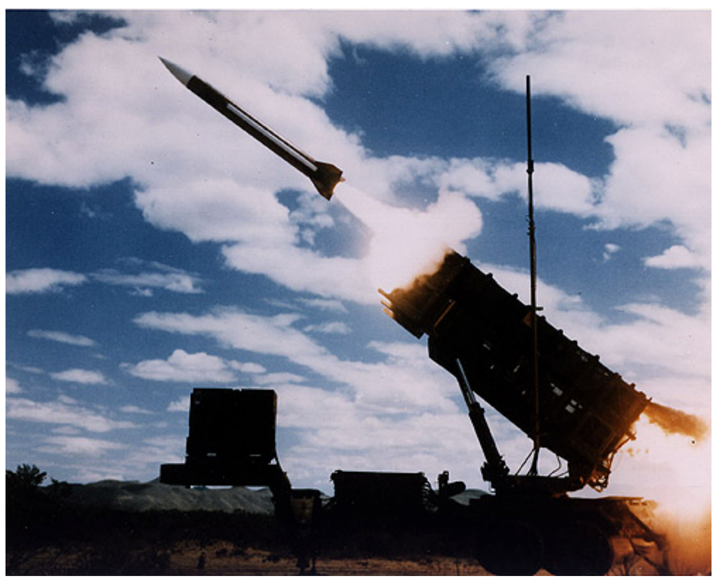
$60 Billion a year in bugs



Therac 25



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

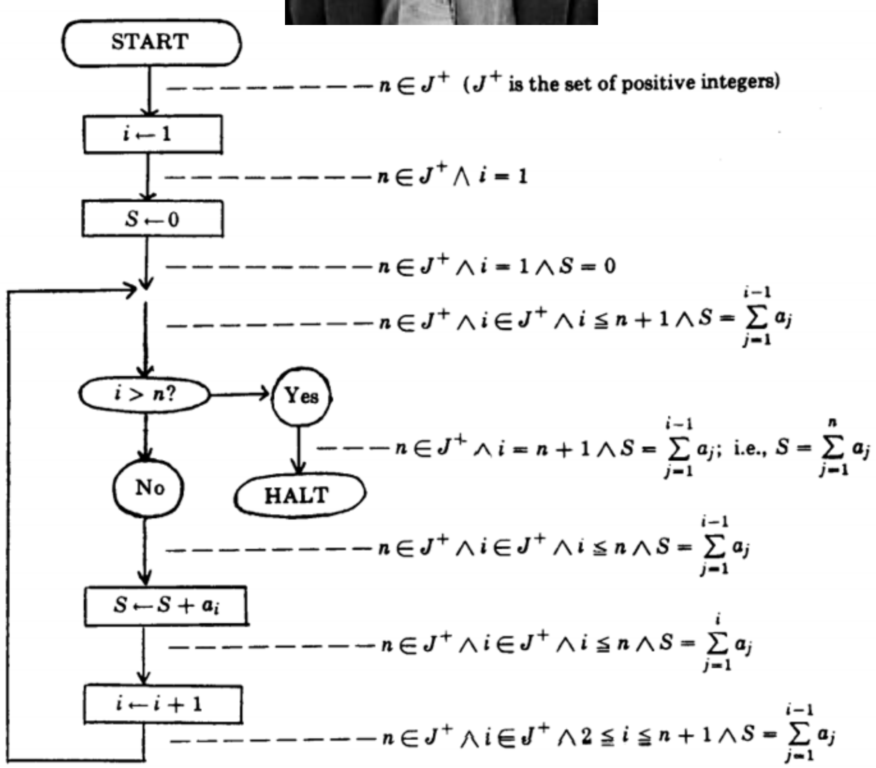
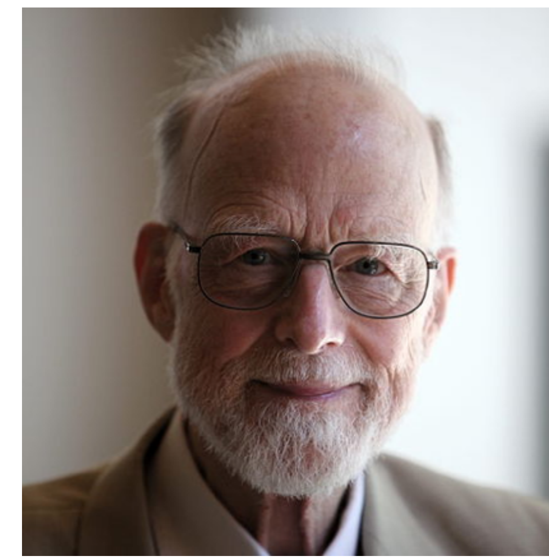


FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ $(n \geqq 0)$

```
{ 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  ∧  2*(b div 2) = b }
        { a^(2*(b div 2)) * c = m^n  ∧  (b div 2) > 0 }
        { (a*a)^(b div 2) * c = m^n  ∧  (b div 2) > 0 }
        a := a * a ;
        { a^(b div 2) * c = m^n  ∧  (b div 2) > 0 }
        b := b div 2
        { a^b * c = m^n  ∧  b > 0 }
    { a^b * c = m^n  ∧  b > 0  ∧  2*(b div 2) ≠ b }
    { 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^b * c = m^n  ∧  b ≥ 0  ∧  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 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 : nat`
  - `k : nat, S(k) : nat`
  - `S(k) 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

2 *  = 

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"
- A reduction relation
  - Tells us what programs "do"
- Denotational semantics
  - What programs "mean" (wrt a mathematical domain)

$$(\lambda x.\, x\ x)(\lambda x.\, x\ x)$$

Some programs *do* something that has no "meaning"

(By which, I mean an undesirable meaning…)

Type systems syntactically rule out "bad" programs

→ *(typed)* *Based on λ (5-3)*

Syntax

$\texttt{t}$	::=		*terms:*
		$\texttt{x}$	*variable*
		$\lambda\texttt{x:T.t}$	*abstraction*
		$\texttt{t t}$	*application*
$\texttt{v}$	::=		*values:*
		$\lambda\texttt{x:T.t}$	*abstraction value*
$\texttt{T}$	::=		*types:*
		$\texttt{T}\to\texttt{T}$	*type of functions*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma,\texttt{x:T}$	*term variable binding*

Evaluation $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

Figure 9-1: Pure simply typed lambda-calculus ($\lambda_\to$)

Typed Lambda

Complain about type errors

Which would result in runtime errors…

2 *

=

```
Error: * 2
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**

$$vector : Nat \to Type$$

Indexed by nat

"Give me a natural, and I'll give you back a type"

Similar to list…

Similar to… $cons(1, nil) : list(nat)$

$$3 : nat$$

$$succ : nat \to nat$$

All of the **terms** in our language have a **type**.

$$[1, 2] : Vector$$

WRONG

$$[1, 2] : Vector2$$

Vector is a type **producer**
operator

$$vector :: Nat \to *$$

This is an alternative notation that is sometimes used…

$$[1, 2] : Vector2$$

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…

Register | Log in

ars technica

MAIN MENU MY STORIES: 25 FORUMS SUBSCRIBE JOBS

RISK ASSESSMENT / SECURITY & HACKTIVISM

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

HACKING 58

LATEST FEATURE STORY

FEATURE STORY (3 PAGES)

BLUISH CODER

PROGRAMMING LANGUAGES, MARTIALS ARTS AND COMPUTERS. THE WEBLOG OF CHRIS DOUBLE.

2014-04-11

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.
```


Tags

acme	1
ajax	7
alice	1
ats	25
audio	2
b2g	4
backbase	1
bitcoin	3
bjj	1
blackdog	3
commonlisp	10
concurrency	4
continuations	10
cyclone	1
dojo	1
eee	1
erlang	19
facebook	2
factor	60
firefox	6
flash	1
forth	2
fxos	1
git	6
gstreamer	5
happs	2
haskell	14
hyperscope	1
inferno	6

Things you might want to do with vectors…

Create an empty vector

Add an element to the end

Take a vector to a list

Add two vectors

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

```
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`"

$$\mathrm{zero} : Vector\, 0$$

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)”

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

cf.

$$cons : list \rightarrow nat \rightarrow list$$

Now for a few functions over vectors…

“Give me a number n, and I’ll give you an empty vector of size n”

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

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

Read π as ∀

Not quite right for logical reasons..

This *actually exists*, by the way…

```
Inductive natvec : nat -> Type :=
  | UnitVec : natvec O
  | 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.”

$$\mathrm{first} : \Pi n : \mathbb{N}.\, Vector(S(n)) \to \mathbb{N}$$

S(n) because this guarantees they can’t give us an empty vector, *that’s the magic*!

This **isn’t** the strongest spec possible

$$add : \Pi n : \mathbb{N}. Vector(n) \to Vector(n) \to Vector(n)$$

What happens if I try to add vectors of different lengths?

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

$$\text{to\_list} : \Pi n.Vector(n) \to list(\mathbb{N})$$

$$cons : \Pi n.Vector(n) \to \mathbb{N} \to Vector(n+1)$$

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

$$add : \Pi n.Vector(n) \to Vector(n) \to Vector(n)$$

Types are **Theorems**

Programs are **Proofs**

–Curry Howard Isomorphism

Propositions

Types are ~~**Theorems**~~

Programs are **Proofs**

–Curry Howard Isomorphism

Not all propositions have proofs

Not all 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…

$$\leq :: \mathbb{N} \to \mathbb{N} \to *$$

How we define <=

```
Inductive <= (n:nat) : nat -> Prop :=
  | le_n : n <= n
  | le_S : forall m:nat,
        n <= m -> n <= S m
```

“Give me a number n, and I’ll give you a proof it’s <= itself”

$$\mathrm{le\_n} : \Pi n. n \leq n$$

“For any n and m, if you can give me a proof that n <= m, then I’ll give you a proof that n <= S(m)”

$$\mathrm{le\_S} : \Pi n, m.\, n \leq m \to n \leq S(m)$$

I *produce proofs*

How do I prove that 0 <= 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 -> B) -> A -> B`
- Which we usually just call “apply” :)

$$A \to A$$

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***

λ / app can **build the entire universe**

Two steps

- Prove 0 <= 0
- Then **use** that proof to prove 0 <= 1

```
Definition zero_leq_one : 0 <= 1 :=
  le_S 0 0 (le_n 0).
```

Computer is going to **check** proof for us

What about 0 <= 2

```
Definition zero_leq_two : 0 <= 2 :=
  le_S 0 1
    (le_S 0 0 (le_n 0)).
```

Automating it…

How do I prove that n <= k

- Start with n,
- keep adding `le_S`
  - a lot… (10 <= 1000) — Going to need hundreds of apps of 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,
       (A -> B -> C) -> (A -> B) -> A -> C.
   Proof.
    auto.
   Qed.
```

Here's the proof!

```
a_b_c = fun (A B C : Type)(X : A -> B -> C)
(X0 : A -> B) (X1 : A) =>
   X X1 (X0 X1)

: forall A B C : Type,
  (A -> B -> C) -> (A -> B) -> A -> C
```

```
Require Import Ascii String List EqNat NArith.

Open Scope N_scope.

(*****************************************************************************)
(*                                                                           *)
(*          A  Coq version of the 2048 game                                  *)
(*              tested with 8.4pl2                                           *)
(*                                         Laurent.Thery@inria.fr            *)
(*                                                                           *)
(*****************************************************************************)

(* Possible moves *)
Inductive move := Rm (* right *) | Lm (* left *) | Um (* up *) | Dm (* down *).

(* Remove all the elements a of l such that p a holds *)
Fixpoint strip {A : Type} (p : A -> bool) l :=
      match l with
      | nil => l
      | a :: l1 => if p a then strip p l1 else a :: strip p l1
      end.

(* Cumulative action on a line *)
Fixpoint cumul (n : nat) (l : list N) {struct n} : list N :=
  match n with
   | 0%nat => nil | 1%nat => hd 0 l :: nil
   | S (S _ as n1) =>
      let a := hd 0 l in
      let l1 := tl l in
      let b := hd 0 l1 in
          if a =? b then (a + b) :: cumul n1 (tl l1)
          else a :: cumul n1 l1
  end.

(* Cumulative action + strip on lines *)
Definition icumul n := map (fun x => cumul n (strip (N.eqb 0) x)).

(* Count the number of occurrences of p on a line *)
Definition count (p : N -> bool) :=
  fold_right (fun n => if p n then N.succ else id) 0.

(* Count the number of occurrences of p on lines *)
Definition icount p := fold_right (fun l => N.add (count p l)) 0%N.
```

Define winning boards as proofs of moves to get to 2048

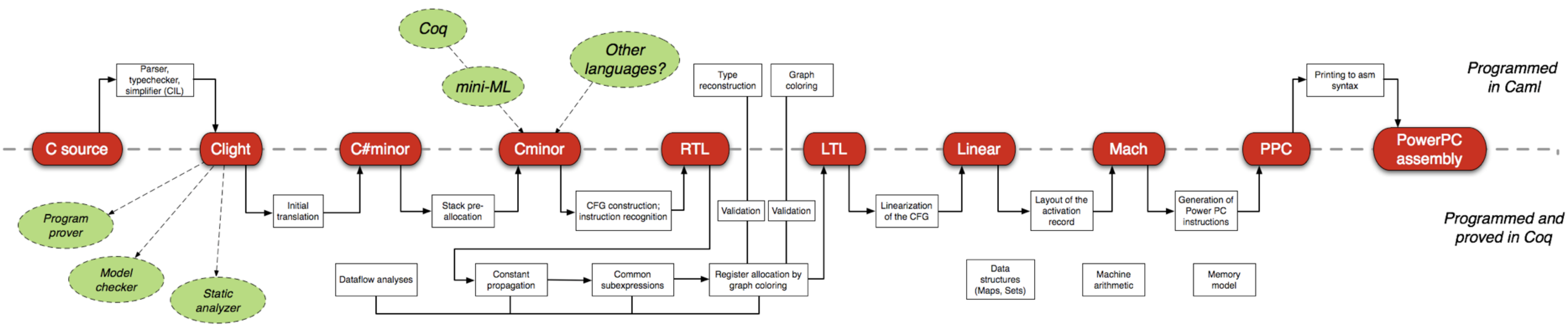
Automatic proof corresponds to automatic search strategy for winning 2048 game…

Scale up to real projects: CompCert

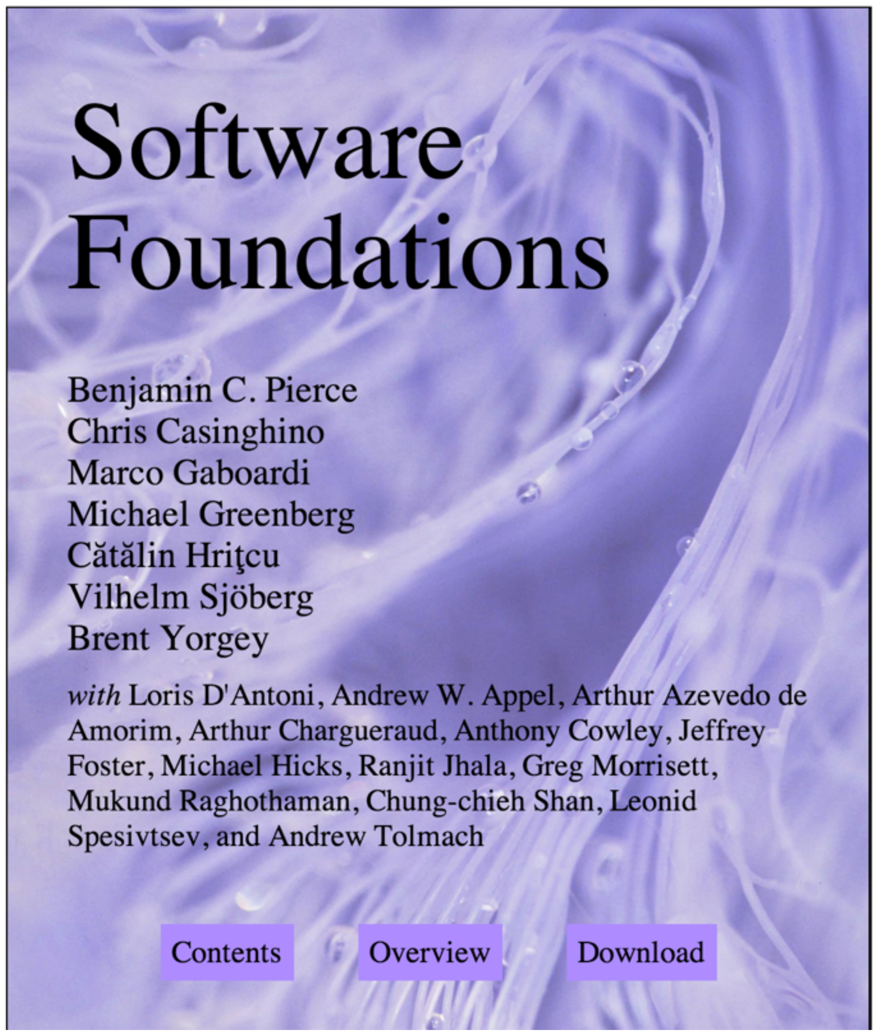
- Write logical specs for functions
- E.g., write a spec for a compiler
- Write specs for each *pass*
    - Prove translation of C to IR preserves semantics
- Chain together a bunch of small steps
    - **Prove a compiler correct**

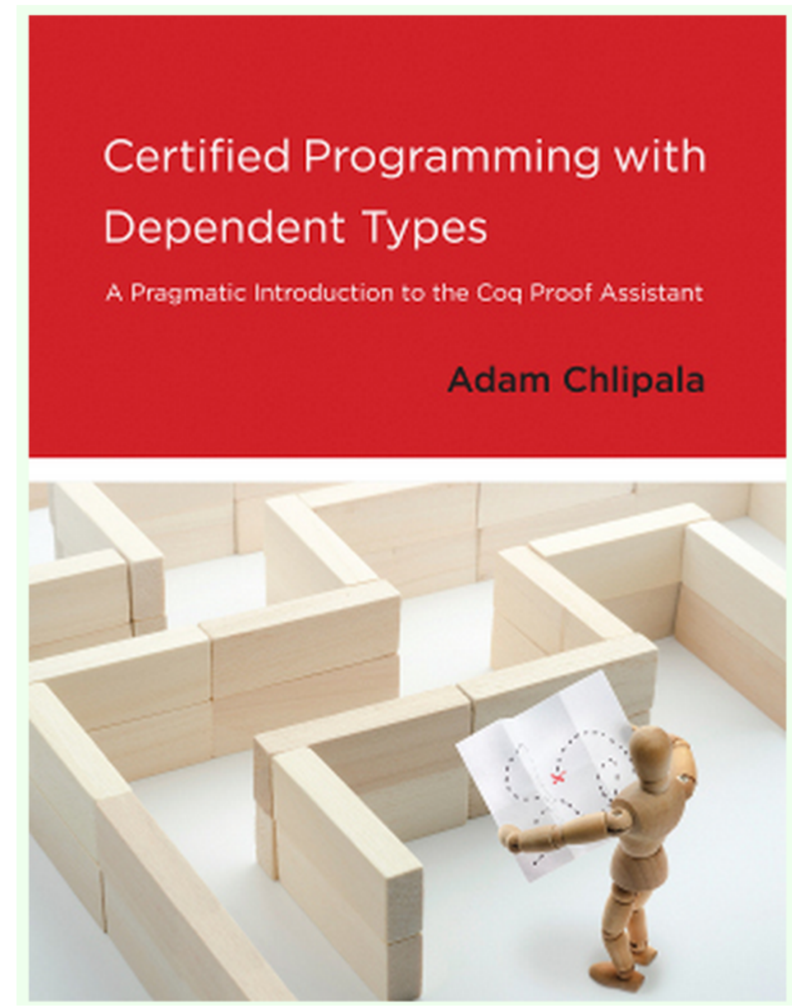
```
Theorem transf_c_program_is_refinement:
forall p tp,
transf_c_program p = OK tp ->
(forall beh, exec_C_program p beh -> not_wrong beh) ->
(forall beh, exec_asm_program tp beh -> exec_C_program p beh).
```

- 50kloc Coq source
- 8koc source — others are proofs
- **No errors**



Where can I learn about this stuff!?





Suitable for beginners

“Real” proof engineering

`http://www.cis.upenn.edu/~bcpierce/sf/`

`http://adam.chlipala.net/cpdt/`

Auxiliary slides…

Just a quick run-through of LF…

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x{:}T.t}$	*abstraction*
		$\mathtt{t\ t}$	*application*
T	::=		*types:*
		X	*type/family variable*
		$\Pi\mathtt{x{:}T.T}$	*dependent product type*
		$\mathtt{T\ t}$	*type family application*
K	::=		*kinds:*
		$*$	*kind of proper types*
		$\Pi\mathtt{x{:}T.K}$	*kind of type families*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, \mathtt{x{:}T}$	*term variable binding*
		$\Gamma, \mathtt{X{::}K}$	*type variable binding*

Well-formed kinds $\boxed{\Gamma \vdash \mathsf{K}}$

$$\Gamma \vdash * \qquad \text{(WF-STAR)}$$

$$\frac{\Gamma \vdash \mathsf{T} :: * \qquad \Gamma, \mathtt{x{:}T} \vdash \mathsf{K}}{\Gamma \vdash \Pi\mathtt{x{:}T.K}} \qquad \text{(WF-PI)}$$

Kinding $\boxed{\Gamma \vdash \mathsf{T} :: \mathsf{K}}$

$$\frac{\mathsf{X} :: \mathsf{K} \in \Gamma \qquad \Gamma \vdash \mathsf{K}}{\Gamma \vdash \mathsf{X} :: \mathsf{K}} \qquad \text{(K-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 :: * \qquad \Gamma, \mathtt{x{:}T_1} \vdash \mathsf{T}_2 :: *}{\Gamma \vdash \Pi\mathtt{x{:}T_1.T_2} :: *} \qquad \text{(K-PI)}$$

$$\frac{\Gamma \vdash \mathsf{S} :: \Pi\mathtt{x{:}T.K} \qquad \Gamma \vdash \mathtt{t} : \mathsf{T}}{\Gamma \vdash \mathsf{S}\ \mathtt{t} : [\mathtt{x} \mapsto \mathtt{t}]\mathsf{K}} \qquad \text{(K-APP)}$$

$$\frac{\Gamma \vdash \mathsf{T} :: \mathsf{K} \qquad \Gamma \vdash \mathsf{K} \equiv \mathsf{K}'}{\Gamma \vdash \mathsf{T} :: \mathsf{K}'} \qquad \text{(K-CONV)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathsf{T}}$

$$\frac{\mathtt{x{:}T} \in \Gamma \qquad \Gamma \vdash \mathsf{T} :: *}{\Gamma \vdash \mathtt{x} : \mathsf{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{S} :: * \qquad \Gamma, \mathtt{x{:}S} \vdash \mathtt{t} : \mathsf{T}}{\Gamma \vdash \lambda\mathtt{x{:}S.t} : \Pi\mathtt{x{:}S.T}} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \Pi\mathtt{x{:}S.T} \qquad \Gamma \vdash \mathtt{t}_2 : \mathsf{S}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_2]\mathsf{T}} \qquad \text{(T-APP)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathsf{T} \qquad \Gamma \vdash \mathsf{T} \equiv \mathsf{T}' :: *}{\Gamma \vdash \mathtt{t} : \mathsf{T}'} \qquad \text{(T-CONV)}$$

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

(Mostly) Lambda

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x{:}T.t}$	*abstraction*
		$\mathtt{t\ t}$	*application*
T	::=		*types:*
		X	*type/family variable*
		$\Pi\mathtt{x{:}T.T}$	*dependent product type*
		$\mathtt{T\ t}$	*type family application*
K	::=		*kinds:*
		$*$	*kind of proper types*
		$\Pi\mathtt{x{:}T.K}$	*kind of type families*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, \mathtt{x{:}T}$	*term variable binding*
		$\Gamma, \mathtt{X{::}K}$	*type variable binding*

Well-formed kinds $\boxed{\Gamma \vdash \mathsf{K}}$

$$\Gamma \vdash * \qquad \text{(WF-STAR)}$$

$$\frac{\Gamma \vdash \mathsf{T} :: * \qquad \Gamma, \mathtt{x{:}T} \vdash \mathsf{K}}{\Gamma \vdash \Pi\mathtt{x{:}T.K}} \qquad \text{(WF-PI)}$$

Kinding $\boxed{\Gamma \vdash \mathsf{T} :: \mathsf{K}}$

$$\frac{\mathsf{X} :: \mathsf{K} \in \Gamma \qquad \Gamma \vdash \mathsf{K}}{\Gamma \vdash \mathsf{X} :: \mathsf{K}} \qquad \text{(K-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 :: * \qquad \Gamma, \mathtt{x{:}T_1} \vdash \mathsf{T}_2 :: *}{\Gamma \vdash \Pi\mathtt{x{:}T_1.T_2} :: *} \qquad \text{(K-PI)}$$

$$\frac{\Gamma \vdash \mathsf{S} :: \Pi\mathtt{x{:}T.K} \qquad \Gamma \vdash \mathtt{t} : \mathsf{T}}{\Gamma \vdash \mathsf{S}\ \mathtt{t} : [\mathtt{x} \mapsto \mathtt{t}]\mathsf{K}} \qquad \text{(K-APP)}$$

$$\frac{\Gamma \vdash \mathsf{T} :: \mathsf{K} \qquad \Gamma \vdash \mathsf{K} \equiv \mathsf{K}'}{\Gamma \vdash \mathsf{T} :: \mathsf{K}'} \qquad \text{(K-CONV)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathsf{T}}$

$$\frac{\mathtt{x{:}T} \in \Gamma \qquad \Gamma \vdash \mathsf{T} :: *}{\Gamma \vdash \mathtt{x} : \mathsf{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{S} :: * \qquad \Gamma, \mathtt{x{:}S} \vdash \mathtt{t} : \mathsf{T}}{\Gamma \vdash \lambda\mathtt{x{:}S.t} : \Pi\mathtt{x{:}S.T}} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \Pi\mathtt{x{:}S.T} \qquad \Gamma \vdash \mathtt{t}_2 : \mathsf{S}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_2]\mathsf{T}} \qquad \text{(T-APP)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathsf{T} \qquad \Gamma \vdash \mathsf{T} \equiv \mathsf{T}' :: *}{\Gamma \vdash \mathtt{t} : \mathsf{T}'} \qquad \text{(T-CONV)}$$

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

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x}\!:\!\mathtt{T.t}$	*abstraction*
		$\mathtt{t\ t}$	*application*
T	::=		*types:*
		X	*type/family variable*
		$\Pi\mathtt{x}\!:\!\mathsf{T}.\mathsf{T}$	*dependent product type*
		$\mathsf{T}\ \mathtt{t}$	*type family application*
K	::=		*kinds:*
		$*$	*kind of proper types*
		$\Pi\mathtt{x}\!:\!\mathsf{T}.\mathsf{K}$	*kind of type families*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, \mathtt{x}\!:\!\mathsf{T}$	*term variable binding*
		$\Gamma, \mathsf{X}\!::\!\mathsf{K}$	*type variable binding*

Well-formed kinds $\boxed{\Gamma \vdash \mathsf{K}}$

$$\Gamma \vdash * \qquad \text{(WF-STAR)}$$

$$\frac{\Gamma \vdash \mathsf{T} :: * \qquad \Gamma, \mathtt{x}\!:\!\mathsf{T} \vdash \mathsf{K}}{\Gamma \vdash \Pi\mathtt{x}\!:\!\mathsf{T}.\mathsf{K}} \qquad \text{(WF-PI)}$$

Kinding $\boxed{\Gamma \vdash \mathsf{T} :: \mathsf{K}}$

$$\frac{\mathsf{X} :: \mathsf{K} \in \Gamma \qquad \Gamma \vdash \mathsf{K}}{\Gamma \vdash \mathsf{X} :: \mathsf{K}} \qquad \text{(K-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 :: * \qquad \Gamma, \mathtt{x}\!:\!\mathsf{T}_1 \vdash \mathsf{T}_2 :: *}{\Gamma \vdash \Pi\mathtt{x}\!:\!\mathsf{T}_1.\mathsf{T}_2 :: *} \qquad \text{(K-PI)}$$

$$\frac{\Gamma \vdash \mathsf{S} :: \Pi\mathtt{x}\!:\!\mathsf{T}.\mathsf{K} \qquad \Gamma \vdash \mathtt{t} : \mathsf{T}}{\Gamma \vdash \mathsf{S}\ \mathtt{t} : [\mathtt{x} \mapsto \mathtt{t}]\mathsf{K}} \qquad \text{(K-APP)}$$

$$\frac{\Gamma \vdash \mathsf{T} :: \mathsf{K} \qquad \Gamma \vdash \mathsf{K} \equiv \mathsf{K}'}{\Gamma \vdash \mathsf{T} :: \mathsf{K}'} \qquad \text{(K-CONV)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathsf{T}}$

$$\frac{\mathtt{x}\!:\!\mathsf{T} \in \Gamma \qquad \Gamma \vdash \mathsf{T} :: *}{\Gamma \vdash \mathtt{x} : \mathsf{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{S} :: * \qquad \Gamma, \mathtt{x}\!:\!\mathsf{S} \vdash \mathtt{t} : \mathsf{T}}{\Gamma \vdash \lambda\mathtt{x}\!:\!\mathsf{S}.\mathtt{t} : \Pi\mathtt{x}\!:\!\mathsf{S}.\mathsf{T}} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \Pi\mathtt{x}\!:\!\mathsf{S}.\mathsf{T} \qquad \Gamma \vdash \mathtt{t}_2 : \mathsf{S}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_2]\mathsf{T}} \qquad \text{(T-APP)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathsf{T} \qquad \Gamma \vdash \mathsf{T} \equiv \mathsf{T}' :: *}{\Gamma \vdash \mathtt{t} : \mathsf{T}'} \qquad \text{(T-CONV)}$$

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

In pure LF types depend on terms,
but *can't depend on types*
(we can't have `List(A)`)