A Unified View Of Some Theories (PhD Oral Exam Presentation)

Yin Wang May 7, 2012

Topics

- Type Inference
- Intersection Types
- Control Flow Analysis
- Hoare Logic
- Linear Logic
- Type Theory
- Automated Deduction
- Supercompilation

Problem 1: Tom ate my sandwich

Problem 1: Tom ate my sandwich

Solution 1:

Problem 1: Tom ate my sandwich

Solution 1:

1. Prove theorem: “Tom ate my sandwich”

Problem 1: Tom ate my sandwich

Solution 1:

1. Prove theorem: “Tom ate my sandwich”
2. Beat Tom

Problem 1: Tom ate my sandwich

Solution 1:

1. Prove theorem: “Tom ate my sandwich”
2. Beat Tom
3. Tom ate my sandwich again!

Problem 1: Tom ate my sandwich

Solution 1:

1. Prove theorem: “Tom ate my sandwich”
2. Beat Tom
3. Tom ate my sandwich again!
4. Goto 1

Problem 1: Tom ate my sandwich

Solution 1:

1. Prove theorem: “Tom ate my sandwich”
2. Beat Tom
3. Tom ate my sandwich again!
4. Goto 1

Solution 2:

1. Ask Tom: “Why you ate my sandwich?”

Problem 1: Tom ate my sandwich

Solution 1:

1. Prove theorem: “Tom ate my sandwich”
2. Beat Tom
3. Tom ate my sandwich again!
4. Goto 1

Solution 2:

1. Ask Tom: “Why you ate my sandwich?”
2. Get rid of the reason that makes Tom eat my sandwich.

Approach “Evidence and proofs are not enough. Everything happens for a reason.”

- Reason with first principles
- Deliberately reinvent things
- Implement and experiment
- Collapse duplicated concepts
- Soundness by construction
- Recheck by reading literature

Approach "Evidence and proofs are not enough. Everything happens for a reason."

- Reason with first principles
- Deliberately reinvent things
- Implement and experiment
- Collapse duplicated concepts
- Soundness by construction
- Recheck by reading literature

Goal: A Simple Unified Theory

Things Built

1. A type inferencer with most things we want from:
   – ML
   – Parametric polymorphism combined with subtyping
   – MLF, HML etc.
   – System I, System E (Kfoury & Wells intersection types)
   – System P (Trevor Jim “A Polar Type System”)
   – BidirecNonal Typechecking (Dunfield & Pfenning)
2. A “control flow analysis” as powerful as CFA2, but much simpler
3. A register allocator which manipulates a “model” of the real machine
4. They turn out to be highly related

Criteria of a Good Concept

Criteria of a Good Concept

- ScoZ: “A good concept is one that is closed
  1) under arbitrary composition
  2) under recursion”

Criteria of a Good Concept

- ScoZ: “A good concept is one that is closed

  1) under arbitrary composition

  2) under recursion”

- Examples of violation:

  1. Let-polymorphism (Rule 1)
  2. Intersection Types without Idempotence (Rule 2)

Type Inference, Intersection Types, Control Flow Analysis

What is Type Inference?

- Given an untyped term, infer its type
- Example:

  λf.λg.λx.(f x) (g x)

  =>

  (a --> b --> c) --> (a --> b) --> (a --> c)

- Also called: “type reconstruction”

Major Concepts

- Let-polymorphism, Algorithm W, Milner, 1978
- Value Restriction, Wright 1995
- MLF, Botlan and Rémy 2003
- Bidirectional Typechecking, Dunfield and Pfenning 2004
- Intersection types, Coppo and Dezani--Ciancaglini 1980
- Principal Typings, Jim 1996
- Expansion, Kfoury and Wells 1999

Unified Type Inference System

- A type inference system with all the desirable features
- Without arbitrary restrictions
- Without creating a mess by piling features upon features
- Many features overlap. There are only very few important ones which subsume all others.

Intuitions

Intuitions

- Every lambda term is a circuit

Intuitions

- Every lambda term is a circuit
- Types flow through wires

Intuitions

- Every lambda term is a circuit
- Types flow through wires
- Names and return points are ends of wires

Intuitions

- Every lambda term is a circuit
- Types flow through wires
- Names and return points are ends of wires
- Applications are connection points

Intuitions

- Every lambda term is a circuit
- Types flow through wires
- Names and return points are ends of wires
- Applications are connection points
- Substitutions maintain:
  - overall “connection state”
  - equivalence relation (“union--find”)

Intuitions

- Every lambda term is a circuit
- Types flow through wires
- Names and return points are ends of wires
- Applications are connection points
- Substitutions maintain:
  - overall “connection state”
  - equivalence relation (“union--find”)
- Unification extends substitutions, adds more connections

Intuitions

- Every lambda term is a circuit
- Types flow through wires
- Names and return points are ends of wires
- Applications are connection points
- Substitutions maintain:
  - overall “connection state”
  - equivalence relation (“union--find”)
- Unification extends substitutions, adds more connections
- Type inference feels like logic programming in the domain of types

The Only Trouble: Polymorphism

- WANT: apply some function to different types
- Examples:
  - $\lambda x.x$
  - $\lambda f.\lambda x.f(f\ x)$
  - $\lambda f.(f\ 1, f\ \text{true})$

Let-polymorphism is weird

Let-polymorphism is weird

- For any value x of type A, By intuition, we expect [x] to have type [A]

Let-polymorphism is weird

- For any value x of type A, By intuition, we expect [x] to have type [A]
- let x = [1] x is typed [int] (okay)

Let-polymorphism is weird

- For any value x of type A, By intuition, we expect [x] to have type [A]
- let x = [1] x is typed [int] (okay)
- let x = [ ] x is typed $\forall a.[a]$

Let-polymorphism is weird

- For any value x of type A, By intuition, we expect [x] to have type [A]
- let x = [1] x is typed [int] (okay)
- let x = [ ] x is typed $\forall a.[a]$
- let x = [[ ]] expect: $[\forall a.[a]]$ but get: $\forall a.[[a]]$

Let-polymorphism is weird

- For any value x of type A, By intuition, we expect [x] to have type [A]
- let x = [1] x is typed [int] (okay)
- let x = [ ] x is typed ∀a.[a]
- let x = [[ ]] expect: [∀a.[a]] but get: ∀a.[[a]]
- let id = λx.x x is typed ∀a.a-->a

Let-polymorphism is weird

- For any value x of type A, By intuition, we expect [x] to have type [A]
- let x = [1] x is typed [int] (okay)
- let x = [ ] x is typed ∀a.[a]
- let x = [[ ]] expect: [∀a.[a]] but get: ∀a.[[a]]
- let id = λx.x x is typed ∀a.a-->a
- let x = [id] expect: [∀a.a --> a] but get: ∀a.[a --> a]

Let-polymorphism is weird

- For any value x of type A, By intuition, we expect [x] to have type [A]
- let x = [1] x is typed [int] (okay)
- let x = [ ] x is typed ∀a.[a]
- let x = [[ ]] expect: [∀a.[a]] but get: ∀a.[[a]]
- let id = λx.x x is typed ∀a.a-->a
- let x = [id] expect: [∀a.a --> a] but get: ∀a.[a --> a]

Violates Rule 1: Not closed under arbitrary composition

Unsoundness With Effects

- let r = ref (λx.x) in (r := λx.x+1; (!r)true);

Unsoundness With Effects

- ∀a.(ref (a-->a))

  let r = ref (λx.x) in (r := λx.x+1; (!r)true);

Unsoundness With Effects

- ∀a.(ref (a-->a))

  let r = ref (λx.x) in (r := λx.x+1; (!r)true);
- Each gets a fresh copy of ∀a.(ref (a-->a))
- Passes type check!

ML With Value Restrictions

- let r = ref (λx.x) in (r := λx.x+1; (!r)true);

ML With Value Restrictions

- $\forall$a.(ref (a-->a))

  let r = ref (λx.x) in (r := λx.x+1; (!r)true);

ML With Value Restrictions

- ∀a.(ref (a-->a))

  let r = ref (λx.x) in (r := λx.x+1; (!r)true);

ML With Value Restrictions

- $\forall$a.(ref (a-->a))

  let r = ref (λx.x) in (r := λx.x+1; (!r)true);

- constrains the type to int-->int

ML With Value Restrictions

- ∀a.(ref (a-->a))

  let r = ref (λx.x) in (r := λx.x+1; (!r)true);
- constrains the type to int-->int
- type error because constrains the type r now has type int--to int-->int >int

ML With Value Restrictions

- ∀a.(ref (a-->a))

  let r = ref (λx.x) in (r := λx.x+1; (!r)true);
- constrains the type to int-->int
- type error because constrains the type r now has type int--to int-->int >int
- ML error "unable to unify int with bool"

Beat Tom

Beat Tom

- Value Restriction:

  Only values are generalized at LET

  – variables: YES

  – functions: YES

  – Applications: NO

  – Constructor calls: YES

  – Ah wait… except ref Tom: “Meow… do you expect me to remember all these?”

Beat Tom

- Value Restriction:

  Only values are generalized at LET

  – variables: YES

  – functions: YES

  – Applications: NO

  – Constructor calls: YES

  – Ah wait… except ref Tom: “Meow… do you expect me to remember all these?”
- Restored soundness

Beat Tom

- Value Restriction:

  Only values are generalized at LET

  – variables: YES

  – functions: YES

  – Applications: NO

  – Constructor calls: YES

  – Ah wait… except ref Tom: “Meow… do you expect me to remember all these?”
- Restored soundness
- But introduced awkwardness

Beat Tom

- Value Restriction:

  Only values are generalized at LET

  – variables: YES

  – functions: YES

  – Applications: NO

  – Constructor calls: YES

  – Ah wait… except ref Tom: “Meow… do you expect me to remember all these?”
- Restored soundness
- But introduced awkwardness
- Supported not by reason, but by empirical study of 250,000 lines of ML code (“nobody complained”)

Beat Tom

- Value Restriction:

  Only values are generalized at LET

  – variables: YES

  – functions: YES

  – Applications: NO

  – Constructor calls: YES

  – Ah wait… except ref Tom: “Meow… do you expect me to remember all these?”
- Restored soundness
- But introduced awkwardness
- Supported not by reason, but by empirical study of 250,000 lines of ML code (“nobody complained”)
- Turned out to be pain (e.g. during my interview with Jane Street ;--))

Beat Tom Again, and Again…

- Interfere with first-class continuations
- Interfere with intersection types
- …

What is the Real Problem?

- let r = ref (λx.x) in (r := λx.x+1; (!r)true);
- ML error “unable to unify int with bool”

What is the Real Problem?

- real problem: this type should be ref (∀a.a-->a), but let-polymorphism infers
  - ∀a.(ref (a-->a)) let r = ref (λx.x) in (r := λx.x+1; (!r)true);
- ML error “unable to unify int with bool”

What is the Real Problem?

- real problem: this type should be ref (∀a.a-->a), but let-polymorphism infers
  - ∀a.(ref (a-->a)) let r = ref (λx.x) in (r := λx.x+1; (!r)true);
- real error:
  - trying to assign int-->int ML error “unable to unify int with bool” into ref (∀a.a-->a)
  - int-->int is not a subtype of ∀a.a-->a

What is the Real Problem?

- Similar to the problem of real problem: this type should be ref dynamic scoping, the scope (∀a.a-->a), but let-polymorphism of ∀a changes in non--infers ∀a.(ref (a-->a)) composable ways
  - let r = ref (λx.x) in (r := λx.x+1; (!r)true);
- real error:
  - trying to assign int-->int
  - ML error “unable to unify int with bool” into ref (∀a.a-->a)
  - int-->int is not a subtype of ∀a.a-->a

What is the Real Problem?

- Similar to the problem of real problem: this type should be ref Can dynamic scoping, the scope we type ref (λx.x) (∀a.a-->a), but let-polymorphism as of ∀a changes in non-- infers ∀a.(ref (a-->a)) ref (∀a.a-->a) ? composable ways
  - let r = ref (λx.x) in (r := λx.x+1; (!r)true);
- real error:
  - trying to assign int-->int
  - ML error “unable to unify int with bool” into ref (∀a.a--->a)
  - int-->int is not a subtype of ∀a.a-->a

Generalization at λ

- Essence of type constraints: record how parameters are used in the function body
- If a parameter is not used in the function body, then it should be generalized
- The caller cannot constrain the parameter type, only the function definition can
- Universal quantification means: "I will just pass it on"
- We should probably generalize at λ, and not LET

Example

$\lambda f.\lambda g.\lambda x.\ f\ (g\ x)$

Example

$\lambda f.\lambda g.\lambda x.\ f\ (g\ x)$

a

Example

$\lambda f.\lambda g.\lambda x.\ f\ (g\ x)$

$(g\ a$

Example

$\lambda f.\lambda g.\lambda x.\ f\ (g\ x)$

(g a --> a b

Example

λf.λg.λx. f (g x)

f (g a --> a b

Example

λf.λg.λx. f (g x)

f (g a --> -->b c a b

Example

λf.λg.λx. f (g x)

f (g a --> --> generalize b c a b

Example

λf.λg.λx. f (g x)

f (g a --> --> generalize b c a b generalize

Example

λf.λg.λx. f (g x)

f (g a --> --> generalize b c a b generalize generalize

How to Generalize at λ ?

- Method 1:
    1. Keep track of parents of type variables
    2. When finishing typing λx, generalize all type variables whose ancestor is x
- Method 2:
    1. Bottom-up type checking
    2. If x isn't constrained in function body, assign it a fresh type variable
    3. Otherwise, use type variable already assigned to x
    4. Easy transition into intersection types

The Rest of The Story

- Unify parametric polymorphism with subtyping
- Polar/Bidirectional Typechecking
- Union types
- Function types treated as lambdas
- Unification as pattern binding for beta-reduction
- …
- Many things in one thing, but doesn’t blow up

Intersection Types

Intersection Types

- Crossing point: MLF

Intersection Types

- Crossing point: MLF
- MLF requires type annotations for all polymorphically used parameters
- Example, f must be annotated: λf.(f 1, f true)
- If we hope to do without any annotations, we must use intersection types
- The above term can be typed with intersection type: (int --> a ^ bool --> b) --> (a,b) “takes a function which is both int-->a and bool-->b”
- Application (λf.(f 1, f true)) (λx.x) is then typed (int, bool).

Example

$\lambda f.(f\ 1, f\ \mathrm{true})$

Example

λf.(f 1, f true) --> --> int a bool b

Example

λf.(f 1, f true) --> --> int a bool b

Example

λf.(f 1, f true) --> --> (int --> a ^ bool --> b) int a bool b

Example

- Must use bottom--up typing
  - λf.(f 1, f true) --> --> (int --> a ^ bool --> b) int a bool b

Example

- Must use bottom--up typing
- Reason:

    1. Intersection operation happens at multi-threaded positions

    2. Usual abstract interpretation is single threaded λf.(f 1, f true)

    3. Side--effect in subs1tution creates interference --> --> among multiple occurrences (int --> a ^ bool --> b) int a bool b

Trouble with Intersection Types

- idempotence: “a^a = a ?”
- With idempotence, can’t type higher ranked terms like λx.xxx (because can’t encode control flow)
- Without idempotence, type inference is equivalent to normalization
- Example:
  - (λf.λx.f(f x)) (λf.λx.f(f x)) has type:
    - ((a-->b ^ b-->c) ^ (c-->d ^ d-->e)) --> (a-->e)
    - exactly the type of λf.λx.f(f(f(f x)))

Trouble with Intersection Types

- idempotence: “a^a = a ?”
- With idempotence, can’t type higher ranked terms like λx.xxx (because can’t encode control flow)
- Without idempotence, type inference is equivalent to normalization
- Example:
  - (λf.λx.f(f x)) (λf.λx.f(f x)) has type:
    - ((a-->b ^ b-->c) ^ (c-->d ^ d-->e)) --> (a-->e)
    - exactly the type of λf.λx.f(f(f(f x)))
    - Violates Rule 2: Not closed under recursion

Trouble with Intersection Types

- idempotence: “a^a = a ?”
- With idempotence, can’t type higher ranked terms like λx.xxx (because can’t encode control flow)
- Without idempotence, type inference is equivalent to normalization
- Example:
  - (λf.λx.f(f x)) (λf.λx.f(f x)) has type:
    - ((a-->b ^ b-->c) ^ (c-->d ^ d-->e)) --> (a-->e)
    - exactly the type of λf.λx.f(f(f(f x)))
    - Violates Rule 2: Not closed under recursion
    - Lesson: Type checking cannot be fully modular unless using some annotations

Observations

Observations

- Type inference is in essence puong parts of the program itself into types

Observations

- Type inference is in essence puong parts of the program itself into types
- Example:

  λf.f 1 => (int --> a) --> a

  "f will be applied to int"

  λf.(f 1, f true) => (int --> a ^ bool --> b) --> (a,b)

  "f will be applied to int and bool"

  These are encodings of "what"

Observations

- Type inference is in essence puong parts of the program itself into types
- Example:

  λf.f 1 => (int --> a) --> a

  "f will be applied to int"

  λf.(f 1, f true) => (int --> a ^ bool --> b) --> (a,b)

  "f will be applied to int and bool"

  These are encodings of "what"
- What information is lost? control flow information: "When?" "Where?"

Observations

- Type inference is in essence puong parts of the program itself into types
- Example:

  λf.f 1 => (int --> a) --> a

  "f will be applied to int"

  λf.(f 1, f true) => (int --> a ^ bool --> b) --> (a,b)

  "f will be applied to int and bool"

  These are encodings of "what"
- What information is lost? control flow information: "When?" "Where?"
- Intersection types can contain control flow information

Intersection Types ==> CFA

- λu.uu   ==> (a ^ a-->b) --> b
- λu.(uu)u ==> ((b ^ (a ^ (a --> (b --> c)))) --> c)
- λu.u(uu) ==> (((a ^ (a --> b)) ^ (b --> c)) --> c)
  Conjecture:
  - Intersection types has encoded control flow information
  - Intersection type inference is equivalent to control flow analysis

Hoare Logic
Linear Logic

Hoare Logic (Separation Logic)

- Hoare/Separation Logic formula looks like an encoding of the program itself with extra information about the model
- Formulas are just symbolic encoding of the model
- We can probably achieve the same thing with a software model checker

Linear Logic

- Correspondence between Linear Logic connectives and types:
  1. & == intersection type
  2. $\oplus$ == union type
  3. $\otimes$ == product type
- The only thing lep: ephemeral formulas
- But that can be easily implemented with a “ephemeral model” (as used in my register allocator)

Type Theory
Automated Deduction
Supercompilation

Why we have Curry--Howard Correspondence

- Howard: “The formulae--as--types notion of construction”
- Observations: 1. Everything that can be named can be called a “type” 2. We can refer to it using the name 3. We can manipulate it using the name
- So it seems that we have Curry--Howard simply because we can bind things to names?
- This explains Martin-Löf Type Theory, Hoare Logic, etc.

Automated Deduction and Supercompilation

- A proposition is a program which evaluates to a boolean value
- A theorem prover is an “supercompiler” which takes shortcuts (induction hypotheses) and tell you the answer without actually running the program
- If the program is not of type boolean, then the theorem prover is just a normal type checker
- Type checking and theorem proving unified

Turchin’s View

- “We do not think in terms of rules of formal logic. We create mental and linguistic models of the reality we observe.” Girard: “Locus Solum: From the rules of logic to the logic of rules”
- “The essence of supercompilation is in always moving in the direction of time, and never against it.”
- “… the persistent problem of transformation systems: how to know which rules to apply and in which order to apply them.”

A Simple View

- We can capture the essence of formalisms by thinking about the transition of models in the direction of time
- We can design or implement logics using this way of thinking

Verifying Turchin’s View with Coq

- Mindlessly proved all theorems in first chapter of Pierce’s Software Foundations using a spartan set of tactics which emulates a supercompiler, using no lemmas
- Mindlessly generated a necessary lemma for proving a proposition which contains an accumulating argument (as in Hamilton’s Poison prover paper)

Structural Induction and Recursion Induction

- Experiments on Coq shows that recursion induction is more powerful than structural induction
- Coq’s structural induction gets in the way in one of the theorems, making it less mindless (needed “ingenuity”)
- If using recursion induction, the theorem will be proved straightforwardly
- This matches the view of McCarthy (as noted in Burstall’s 1968 paper on structural induction) “… in a sense structural induction is merely a special case of recursion induction, presented in a rather different manner.“
- Automatic theorem provers using recursion induction can probably prove more theorems

Example

- Theorem evenb_negb :
  - forall n : nat, evenb n = negb (evenb (S n)).

1. Prove these two base cases:
   1. Fixpoint evenb (n:nat) : bool := match n with
   2. | O => true evenb O = negb (evenb (S O)).
   3. | S O => false evenb (S O) = negb (evenb (S (S O))).
   4. | S (S n) => evenb n end.
2. Prove the inductive case:
   1. Definition negb (b:bool) : bool := n : nat match b with
   2. | true => false
   3. IHn : evenb n = negb (evenb (S n))
   4. | false => true ============================ end.
   5. evenb (S (S n)) = negb (evenb (S (S (S n)))) => evenb n = negb (evenb (S n))

TODO List

- Build a supercompiler and a theorem prover
- Experiment more “mindless” theorem proving with Coq and Agda
- Write something about:
  - type inference
  - control flow analysis
  - theorem proving