Outline

Introduction

Types and Polymorphism
Outline

Introduction
Types and Polymorphism

Exp and Hindley-Milner
Outline

Introduction
Types and Polymorphism

Exp and Hindley-Milner

Denotational Semantics
Crash Course in Domain Theory
Semantics Of Exp
Semantics of Types
Outline

Introduction
  Types and Polymorphism

Exp and Hindley-Milner

Denotational Semantics
  Crash Course in Domain Theory
  Semantics Of Exp
  Semantics of Types

Inference
A Theory of Type Polymorphism in Programming
by Robin Milner
1978
Background

- Robin Milner, Turing Award Winner
- Invented Pi-Calculus, ML, Bi-Graphs, etc
- In the 70s he was working on proof assistants

picture from theguardian.com
LCF

▶ “Logic of Computable Functions”
LCF

- “Logic of Computable Functions”
- Logic as Abstract Data Type
LCF

- “Logic of Computable Functions”
- Logic as Abstract Data Type
- **Proof Script** as Program
• “Logic of Computable Functions”
• Logic as Abstract Data Type
• **Proof Script** as Program
• LCF was implemented in Lisp...*But Lisp doesn’t have types!*
ML!

- *MetaLanguage* for LCF
- Typed Functional Programming Language
- Direct ancestor of Ocaml, StandardML (which Milner co-designed), Haskell, etc
Types: To Be or Not To Be

- Types catch errors early
Types: To Be or Not To Be

- Types catch errors early
- Essential to the purpose of ML
Types: To Be or Not To Be

- Types catch errors early
- Essential to the purpose of ML
- But are limiting
Types: To Be or Not To Be

- Types catch errors early
- Essential to the purpose of ML
- But are limiting
- And can be annoying
The ML Way

- polymorphism: make types as flexible as possible
- inference: make the compiler write the types, not the user
Multiple Types, same code

\[
\text{intId} : \text{Int} \rightarrow \text{Int} \\
\text{intId} x = x
\]

\[
\text{boolId} : \text{Bool} \rightarrow \text{Bool} \\
\text{boolId} x = x
\]

\[
\text{stringId} : \text{String} \rightarrow \text{String} \\
\text{stringId} x = x
\]
Polymorphism

use type variables

\texttt{id} : a \rightarrow a

\texttt{id} \ x = x
Polymorphism

use type variables

\textbf{id} : \ a \to\ a \\
\textbf{id} \ x = x

called \textit{generics} in languages such as Java
This Paper

- Milner describes a simple functional language “Exp” with polymorphism and type inference
- Type Safety
  - Semantic Soundness: “well typed programs do not go wrong”.
  - Syntactic Soundness: if type inference succeeds, program is well typed
This Paper

- Milner describes a simple functional language “Exp” with polymorphism and type inference
- Type Safety
  - Semantic Soundness: “well typed programs do not go wrong”.
  - Syntactic Soundness: if type inference succeeds, program is well typed
- Multiple Breakthroughs here, but presentation dated
This Paper

- Milner describes a simple functional language “Exp” with polymorphism and type inference
- Type Safety
  - Semantic Soundness: “well typed programs do not go wrong”.
  - Syntactic Soundness: if type inference succeeds, program is well typed
- Multiple Breakthroughs here, but presentation dated
- I’m updating it slightly
Exp is a simple functional programming language

\[ e_i \in \text{Exp} ::= x \mid \lambda x. e \mid e_1 e_2 \mid \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \]
\[ \mid \text{true} \mid \text{false} \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{fix } x \text{ e} \]

\[ t_i \in \text{MonoTypes} ::= \alpha \mid \text{bool} \mid t_1 \rightarrow t_2 \]

\[ p \in \text{PolyTypes} ::= t \mid \forall \alpha. p \]
Type System 1

Basic Type System

\[ \Gamma \vdash e_0 : \text{bool} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T \]
\[ \Gamma \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T \]

\[ \Gamma \vdash \text{true} : \text{bool} \quad \Gamma \vdash \text{false} : \text{bool} \]
Type System 2

Functions, recursion, and let

\[ \Gamma, x : T_1 \vdash e : T_2 \]
\[ \Gamma \vdash \lambda x. e : T_1 \rightarrow T_2 \]
\[ \Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 : T_1 \]
\[ \Gamma \vdash e_1 \ e_2 : T_2 \]
\[ \Gamma, x : T \vdash e : T \]
\[ \Gamma \vdash \text{fix} \ x \ e : T \]
\[ \Gamma \vdash e_1 : P_1 \quad \Gamma, x : P_1 \vdash e_2 : P_2 \]
\[ \Gamma \vdash \text{let} \ x = e_1 \ \text{in} \ e_2 : P_2 \]
Substitution

We define the type substitutability relation $\sqsubseteq$

$$p[t_2/\alpha \sqsubseteq t_1]$$

$$\forall \alpha. p \sqsubseteq t_1$$

$$t \sqsubseteq t$$

intuitively, $p \sqsubseteq t$ means that $t$ is a *specialization* of $p$. You might also think about it in terms of subtypes.
The variable rule *specializes* the type

\[ P \subseteq T \]

\[ \Gamma, x : P \vdash x : T \]

we also need to allow generalization of types

\[ \Gamma \vdash e : P \quad \alpha \notin FV(\Gamma) \]

\[ \Gamma \vdash \forall \alpha. P \]
A *typing* of a term $e$ is a pair $(\Gamma, P)$ such that $\Gamma \vdash e : P$ is derivable. A *principal typing* is a typing that is most general.
Typings and Principle Types

A typing of a term $e$ is a pair $(\Gamma, P)$ such that $\Gamma \vdash e : P$ is derivable. A principal typing is a typing that is most general. On the other hand, a type of a term $e$ is a $P$ such that $\emptyset \vdash e : P$ is derivable. A principal type is a most general type: that is a $P$ such that for any $T$ if $\emptyset \vdash e : T$ is derivable then $P \sqsubseteq T$.
Denotational Semantics?!?

Give meaning to programs by translating into math
Denotational Semantics?!?

In any partial order \((S, \leq)\) a *chain* is a sequence \(x_0 \leq x_1 \leq x_2 \leq \ldots\)
In any partial order \((S, \leq)\) a *chain* is a sequence \(x_0 \leq x_1 \leq x_2 \leq \ldots\)

A *Complete Partial Order* (CPO) is a four tuple \((S, \leq, \bot, \sqcup)\) where \((S, \leq)\) is a partial order, \(\bot\) is the least element, and \(\sqcup\) assigns a lowest upper bound to each chain.

Intuition:

- \(S\) is a set of possible program meanings
Denotational Semantics?!?

In any partial order \((S, \leq)\) a *chain* is a sequence \(x_0 \leq x_1 \leq x_2 \leq \ldots\). A *Complete Partial Order* (CPO) is a four tuple \((S, \leq, \perp, \sqcup)\) where \((S, \leq)\) is a partial order, \(\perp\) is the least element, and \(\sqcup\) assigns a lowest upper bound to each chain.

Intuition:

- \(S\) is a set of possible program meanings
- \(\leq\) is the “information order”
In any partial order \((S, \leq)\) a *chain* is a sequence \(x_0 \leq x_1 \leq x_2 \leq \ldots\). A *Complete Partial Order* (CPO) is a four tuple \((S, \leq, \perp, \sqcup)\) where \((S, \leq)\) is a partial order, \(\perp\) is the least element, and \(\sqcup\) assigns a lowest upper bound to each chain.

**Intuition:**

- \(S\) is a set of possible program meanings
- \(\leq\) is the “information order”
- \(\perp\) is non-termination
In any partial order \((S, \leq)\) a \textit{chain} is a sequence \(x_0 \leq x_1 \leq x_2 \leq \ldots\). A \textit{Complete Partial Order} (CPO) is a four tuple \((S, \leq, \bot, \sqcup)\) where \((S, \leq)\) is a partial order, \(\bot\) is the least element, and \(\sqcup\) assigns a lowest upper bound to each chain.

Intuition:

- \(S\) is a set of possible program meanings
- \(\leq\) is the “information order”
- \(\bot\) is non-termination
- \(\sqcup\) solves loops
Continuous Functions

A *continuous functions* is a structure preserving map on CPOs.
Continuous Functions

A *continuous functions* is a structure preserving map on CPOs. $f : A \rightarrow B$ is continuous if

1. It is monotonic: $a \leq a' \Rightarrow f(a) \leq f(a')$
Continuous Functions

A *continuous function* is a structure preserving map on CPOs. $f : A \rightarrow B$ is continuous if

1. It is monotonic: $a \leq a' \Rightarrow f(a) \leq f(a')$ **Intuition: Halting Problem**
A continuous functions is a structure preserving map on CPOs. $f : A \to B$ is continuous if

1. It is monotonic: $a \leq a' \Rightarrow f(a) \leq f(a')$  **Intuition: Halting Problem**

2. It is “Scott Continuous”: $f(\bigsqcup x_i) = \bigsqcup (f(x_i))$

**Aside**
if $f \perp_A = \perp_B$ then $f$ is strict
A fix point of a function $f : A \rightarrow A$ is a value $x$ such that $f(x) = x$. Every continuous function $A \rightarrow A$ has a least fixpoint.

Proof.
Let $x_0 = \bot$ and $x_{n+1} = f(x_n)$. $x_0, x_1, x_2, \ldots$ is a chain (induction). The least fixpoint is $x = \bigcup x_i$. 
Fixpoints

A fix point of a function $f : A \to A$ is a value $x$ such that $f(x) = x$.

Every continuous function $A \to A$ has a least fixpoint.

Proof.
Let $x_0 = \bot$ and $x_{n+1} = f(x_n)$. $x_0, x_1, x_2, \ldots$ is a chain (induction).
The least fixpoint is $x = \bigsqcup x_i$.

- $x$ is a fixpoint: $f(x) = f(\bigsqcup x_i) = \bigsqcup f(x_i) = \bigsqcup x_{i+1} = x$
Fixpoints

A fix point of a function \( f : A \to A \) is a value \( x \) such that \( f(x) = x \). Every continuous function \( A \to A \) has a least fixpoint

Proof.
Let \( x_0 = \bot \) and \( x_{n+1} = f(x_n) \). \( x_0, x_1, x_2, \ldots \) is a chain (induction). The least fixpoint is \( x = \bigsqcup x_i \).

- \( x \) is a fixpoint: \( f(x) = f(\bigsqcup x_i) = \bigsqcup f(x_i) = \bigsqcup x_{i+1} = x \)
- \( x \) is the least: let \( y = f(y) \) then \( \bot \leq y \) and \( f\bot \leq fy \). By induction \( x_i \leq y \) so \( \bigsqcup x_i \leq y \)
Constructing CPOs

- Primitive Domains (such as the domain of booleans) can be formed by attaching a least element. That is we have the set \{true, true, ⊥\} order by \(x \leq y \iff (x = ⊥ \lor x = y)\).
Constructing CPOs

- PrimitiveDomains(suchasthe domain of booleans) can be formed by attaching a least element. That is we have the set \{true, true, \bot\} order by \( x \leq y \iff (x = \bot \lor x = y) \)
- The set of function between two CPOs (\(\rightarrow\)) is itself a CPO: under the *information ordering*
  \( f \leq g \iff (\forall x \in A, f(x) \leq_B g(x)) \)
Constructing CPOs

- Primitive Domains (such as the domain of booleans) can be formed by attaching a least element. That is we have the set \{true, true, ⊥\} ordered by \( x \leq y \iff (x = ⊥ \lor x = y) \)

- The set of functions between two CPOs (\( \rightarrow \)) is itself a CPO under the information ordering
  \( f \leq g \iff (\forall x \in A, f(x) \leq_B g(x)) \)

- Cartesian Products (\( \times \))
Constructing CPOs

- Primitive Domains (such as the domain of booleans) can be formed by attaching a least element. That is we have the set \{true, true, ⊥\} order by \( x \leq y \iff (x = ⊥ \lor x = y) \)
- The set of function between two CPOs (\( \rightarrow \)) is itself a CPO: under the *information ordering*
  \( f \leq g \iff (\forall x \in A, f(x) \leq_B g(x)) \)
- Cartesian Products (\( \times \))
- Co-Products (\( + \))
Any set of *Domain Equations* formed out of Primitive Domains, $\times$, $+$, and $\rightarrow$ has a unique “least” solution (up to isomorphism)

**Proof Sketch**

We just need to construct a CPO of (small) CPOs such that each of the type constructors is continuous. We say $A \leq B$ if there exists a pair of continuous maps $f : A \rightarrow B$, $g : B \rightarrow A$ such that $g \cdot f = id$ and $f \cdot g \leq id$. The least CPO is $\{\bot\}$. $\bigsqcup X_i$ consists of sequences where $x_i \in X_i \land x_i = g_i(x_{i+1})$ ($g_i$ from the witness that $X_i \leq X_{i+1}$)—this is known as “Scott’s Inverse Limit Construction”

That this works is surprising, but you don’t need to know the details to use it
Milner gives an *untyped* semantics for Exp.
Milner gives an *untyped* semantics for Exp.

Does this by treating Exp as a *dynamically typed language* with *runtime type errors*.
Exp’s Semantic Domain

The least solution to the domain equations

\[ V = B_0 + B_1 + \ldots + B_n + F + W \]
\[ F = V \rightarrow V \]
Exp’s Semantic Domain

The least solution to the domain equations

\[
V = B_0 + B_1 + \ldots + B_n + F + W
\]

\[
F = V \rightarrow V
\]

- Parameterized over a set of base types/domains \( B_0, B_1, B_2, \ldots, B_n \) where \( B_0 \) is the primitive domain of booleans.
Exp’s Semantic Domain

The least solution to the domain equations

\[ V = B_0 + B_1 + \ldots + B_n + F + W \]
\[ F = V \rightarrow V \]

- Parameterized over a set of base types/domains $B_0, B_1, B_2, \ldots, B_n$ where $B_0$ is the primitive domain of booleans.

- A special value \textbf{wrong} is the only inhabitant of $W$. This is the semantics of a runtime type failure.
The least solution to the domain equations

\[ V = B_0 + B_1 + \ldots + B_n + F + W \]

\[ F = V \rightarrow V \]

- Parameterized over a set of base types/domains \( B_0, B_1, B_2, \ldots, B_n \) where \( B_0 \) is the primitive domain of booleans.
- A special value **wrong** is the only inhabitant of \( W \). This is the semantics of a runtime type failure.
- **minor technical detail**: The coproduct is *lifted*
Some Notation

- For $d \in D$, the notation \( d \in V \) is the image of $d$ in the injection map from $D \rightarrow V$.

- $v \in D$ is a runtime check if $\exists d \cdot v = d \in V$.

- $v \in D = \text{true}$ if $\exists d \cdot v = d \in V$ and $\text{false}$ otherwise.

- $\perp \in D$.

- $v | D$ is a coercion into $D$.

- $v | D = d$ if there exists $d \in V$.

- $\perp | D = \perp$.

- The ternary operator $a ? b : c$ is defined as $\text{false} \ ? b \ : c$.

- $\text{true} \ ? b \ : c$.

- $\perp \ ? b \ : c$.

Philip Johnson-Freyd
A Theory of Type Polymorphism in Programming
Some Notation

- For $d \in D$, the notation "$d$ in $V$" is image of $d$ in the injection map from $D \rightarrow V$.
- $v \in D$ is a runtime check if $\exists d. v = d$ in $V$.
  - $v \in D = \text{true}$ if $\exists d. v = d$ in $V$ and $\not= \text{false}$ otherwise.
  - $\perp \in D = \perp_{B_0}$
Some Notation

- For $d \in D$, the notation “$d$ in $V$” is image of $d$ in the injection map from $D \to V$.
- $v \in D$ is a runtime check if $\exists d. v = d$ in $V$.
  - $v \in D = \text{true}$ if $\exists d. v = d$ in $V$ and $= \text{false}$ otherwise.
  - $\bot \in D = \bot_{B_0}$.
- $v \mid D$ is a coercion into $D$.
  - $v \mid D = d$ if there exists $d$ in $V$.
  - $v \mid D = \bot_D$ otherwise.
Some Notation

- For $d \in D$, the notation “$d \text{ in } V$” is image of $d$ in the injection map from $D \rightarrow V$.
- $v \mathbf{E} D$ is a runtime check if $\exists d. v = d$ in $V$.
  - $v \mathbf{E} D = \text{true}$ if $\exists d. v = d$ in $V$ and $\neq \text{false}$ otherwise.
  - $\bot \mathbf{E} D = \bot_{B_0}$.
- $v|D$ is a coercion into $D$.
  - $v|D = d$ if there exists $d$ in $V$.
  - $v|D = \bot_D$ otherwise.
- The ternary operator $a?b : c$ is defined.
  - $\bot ? b, c = \bot$
  - $\text{true} ? b, c = b$
  - $\text{false} ? b, c = c$.
Interpreting Expressions

The semantic function $E[\cdot]: \text{Exp} \to (\text{Var} \to V) \to V$ takes a term and an environment and produces a value in the semantics domain

$$E[\mathit{x}] \eta = \eta(x)$$
$$E[\mathit{true}] \eta = \text{true} \in V$$
$$E[\mathit{false}] \eta = \text{false} \in V$$
$$E[\lambda x. e] \eta = \lambda y. E[e](\eta, x \mapsto x) \in V$$
Function Application

We ensure our function is really a function and that its argument is not an exception before calling it

\[ E[e_1 e_1] \eta = \nu_1 \ E \ F?(\nu_2 \ E \ W?\text{wrong}, (\nu_1 | F) \nu_2), \text{wrong} \]

where \( \nu_i = E[e_i] \eta \)

Note: this is call-by-value.
If Then Else

\[ E[\text{if } e_0 \text{ then } e_1 \text{ else } e_2] \eta = v_0 \ E \ B_0?(v_0 | B_0?v_1, v_2)_{\text{wrong}} \]

where \( v_i = E[e_i] \eta \)
Let

\[
E[\text{let } x = e_1 \text{ in } e_2] \eta = \nu_1 \ E \ W?\text{wrong}, (E[e_2](\eta, x \mapsto E[e_1] \eta))
\]

where \(\nu_1 = E[e_1] \eta\)

Note, this is the same as \((\lambda x. e_2)e_1\ldots\) but the typing rule is less restrictive
Because we are in a CPO we have fix points. Let $Y$ be the fix point operator.

$$E[\textbf{fix} \times e_1]\eta = Y(\lambda v.E[e](\eta, x \mapsto v))$$
We have given a *syntax* to types, a rules for determining if syntax in Exp has a given syntactic type. To prove type safety, we need a *semantics* for types.
 MonoTypes

- For the base types $\text{bool}, b_1, b_2, \ldots, b_n$ we say
  $T[b_i] = \{ v \mid v = \bot \lor (v \ E B_i = \text{true}) \}$
MonoTypes

- For the base types bool, $b_1, b_2, \ldots, b_n$ we say
  \[ T[b_i] = \{ v \mid v = \bot \lor (v \text{ E } B_i = \text{true}) \} \]

- Function types are totally extensional
  \[ T[t_1 \rightarrow t_2] = \{ f \mid f = \bot \lor (f \text{ E } F = \text{true} \land \forall x \in [t_1].(f|F)x \in [t_2]) \} \]
Properties of Types

Monotypes satisfy two properties making the sub CPOs. If $t$ is a type then

- It is downward closed: $\forall x \in T[t], \forall y \in T, y \leq x \Rightarrow y \in T[t]$
- It is chain complete: for any chain such that $x_i \in T[t], \bigcup x_i \in T[t]$
PolyTypes

Easy (predicative)

\[ T[\forall \alpha.p] = \bigcap_{t \in \text{MonoType}} T[p[t/\alpha]] \]

Observe that if \( p \sqsubseteq t \) is derivable then \( T[p] \subseteq T[t] \)
Semantic Types are Not Wrong

*wrong* has no type!
We define $\llbracket \Gamma \rrbracket = \{ f : \text{Var} \to V | \forall x \in \Gamma, f(x) \in T\llbracket \Gamma(x) \rrbracket \}$

Now we can state the semantic soundness theorem: if $\Gamma \vdash e : P$ is derivable then for any $\eta \in \llbracket \Gamma \rrbracket$ the semantic and syntactic types of $e$ agree $E[e] \eta \in T\llbracket P \rrbracket$

Proof.
By induction on the type derivation. \qed
Type Safety

Well Typed Programs Do Not Go Wrong!

Proof.
By semantic soundness no well typed program has a denotation of wrong
Imperative Features

- ML also has imperative features in the form of references
Imperative Features

- ML also has imperative features in the form of references.
- This breaks soundness:

```ml
let x = mkRef (fun y => y)
in x := (fun y => y + 1);
if x true then 1 else 2
```
Imperative Features

- ML also has imperative features in the form of references
- This breaks soundness:
  
  ```ml
  let x = mkRef (fun y => y)
in x := (fun y => y + 1);
if x true then 1 else 2
  ```

- The original ML had a check against this, but I’m not sure it is sound
Imperative Features

- ML also has imperative features in the form of references.
- This breaks soundness:

  ```
  let x = mkRef (fun y => y)
  in x := (fun y => y + 1);
  if x true then 1 else 2
  ```

- The original ML had a check against this, but I’m not sure it is sound.
- Modern MLs (Ocaml, SML) use the “value restriction”
Substitutions

A type substitution is a mapping from type variables to monotypes.
Substitutions

A type substitution is a mapping from type variables to monotypes. The unification algorithm computes substitutions.
Unification

\[ U : \text{Type} \times \text{Type} \rightarrow \text{Substitution} \text{ such that} \]

- if \( S = U(\sigma, \tau) \) then \( S\sigma = S\tau \)
- if \( R\sigma = R\tau \) then there exists some \( S' \) such that \( R = S' \cdot U(\sigma, \tau) \)
- \( U(\sigma, \tau)\alpha = \alpha \) for all \( \alpha \notin FV(\sigma, \tau) \)
Algorithm $W$ takes a pair of an environment $\Gamma$ and a term $e$ and returns a pair $(S, P)$ where $S$ is a substitution and $P$ is a type.
Algorithm W

- Algorithm $W$ takes a pair of an environment $\Gamma$ and a term $e$ and returns a pair $(S, P)$ where $S$ is a substitution and $P$ is a type.
- The intended behavior is that $S\Gamma \vdash e : P$ is derivable.
Inferring Variables

\[ W(\Gamma, x) = (I, \Gamma(x)) \text{ if } \Gamma(x) \text{ is a monotype} \]
\[ W(\Gamma, x) = \text{let } \beta := \text{fresh;} \]
\[ S := \{\alpha \mapsto \beta\}; \]
\[ (S', T) := W(S\Gamma, x); \]
\[ \text{in } (S \cdot S', ST) \]
Function Application

\[ W(\Gamma, e_1 \ e_2) = \text{let } \beta := \text{fresh}; \]
\[ (R, p_1) := W(\Gamma, e_1); \]
\[ (S, p_2) := W(\Gamma, e_2); \]
\[ S' := U(Sp_1, p_2 \to \beta); \]
\[ \text{in } (S' \cdot S \cdot R, S'\beta) \]
Abstraction

\[ W(\Gamma, \lambda x. e) = \text{let } \beta := \text{fresh}; \]
\[ (S, p) := W((\Gamma, x : \beta), e) \]
\[ \text{in } (S, S(\beta \rightarrow p)) \]
Let

\[ W(\Gamma, \text{let } x = e_1 \text{ in } e_2) = \text{let } (R, p) := W((\Gamma), e_1) \]
\[ a := \text{generalize}(\Gamma, p) \]
\[ (S, b) := W((S\Gamma, x : a), e_2) \]
\[ \text{in } (S \cdot R, b) \]


\[ W(\Gamma, \text{fix } x \ e) = \text{let } \beta := \text{fresh} \]

\[ (R, p) := W((\Gamma, x : \beta), e) \]

\[ S := U(R\beta, p) \]

\[ \text{in } (S \cdot R, Sp) \]
What we end up with

- Algorithm $W$ can be easily modified to return the entire derivation, not just the typing
What we end up with

- Algorithm W can be easily modified to return the entire derivation, not just the typing
- W can be proven correct by induction on the term (easy but tedious)
What we end up with

- Algorithm W can be easily modified to return the entire derivation, not just the typing
- W can be proven correct by induction on the term (easy but tedious)
- Many extensions and optimizations (such as J)