ML Type Inference and Unification

Arlen Cox

Research Goals

- Easy to use, high performance parallel programming
- Primary contributions in backend and runtime
- Need a front end to target backend
- ML offers ease of use and safety

ML Type Inference

- Hindley/Milner Type Inference
- Statically typed language with no mandatory annotations
- Three phases to determining types
  - Constraint generation
  - Unification
  - Annotation

An Example

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

An Example

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi

val apply: ('a->'a)->'a->int->'a
```

Constraint Generation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Variables

Constraints

Constraint Generation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Variables
apply: 'a

Constraints

Constraint Generation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Variables
apply: 'a
f: 'b
v: 'c
t: 'd

Constraints

Constraint Generation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Constraints

'a = 'b → 'e

Variables

apply: 'a

f: 'b

v: 'c

t: 'd

Constraint Generation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Variables
apply: 'a
f: 'b
v: 'c
t: 'd

Constraints
'a = 'b → 'e
'b = 'c → 'f

Constraint Generation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Variables
apply: 'a
f: 'b
v: 'c
t: 'd

Constraints
'a = 'b → 'e
'b = 'c → 'f
'e = 'f → 'g

Constraint Generation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Constraints

'a = 'b → 'e
'b = 'c → 'f
'e = 'f → 'g
'd = int

Variables

apply: 'a
f: 'b
v: 'c
t: 'd

Constraint Generation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Constraints
'a = 'b → 'e
'b = 'c → 'f
'e = 'f → 'g
'd = int
'g = int → 'h

Variables
apply: 'a
f: 'b
v: 'c
t: 'd

Constraint Generation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Variables
apply: 'a
f: 'b
v: 'c
t: 'd

Constraints
'a = 'b → 'e
'b = 'c → 'f
'e = 'f → 'g
'd = int
'g = int → 'h
'd = int
bool = bool

Constraint Generation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Constraints
'a = 'b → 'e
'b = 'c → 'f
'e = 'f → 'g
'd = int
'g = int → 'h
'd = int
bool = bool
'c = 'h

Variables
apply: 'a
f: 'b
v: 'c
t: 'd

Constraint Generation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Variables
apply: 'a
f: 'b
v: 'c
t: 'd

Constraints
'a = 'b → 'e
'b = 'c → 'f
'e = 'f → 'g
'd = int
'g = int → 'h
'd = int
bool = bool
'c = 'h
'a = 'b → 'c → 'd → 'c

Constraint Solving - Unification

Constraints

'a = 'b → 'e
'b = 'c → 'f
'e = 'f → 'g
'd = int
'g = int → 'h
'd = int
bool = bool
'c = 'h
'a = 'b → 'c → 'd → 'c

Mapping

Constraint Solving - Unification

Constraints

'b = 'c → 'f
'e = 'f → 'g
'd = int
'g = int → 'h
'd = int
bool = bool
'c = 'h
'b → 'e = 'b → 'c → 'd → 'c

Mapping

'a = 'b → 'e

Constraint Solving - Unification

Constraints

'e = 'f → 'g
'd = int
'g = int → 'h
'd = int
bool = bool
'c = 'h
('c → 'f) → 'e = ('c → 'f) → 'c → 'd → 'c

Mapping

'a = ('c → 'f) → 'e
'b = 'c → 'f

Constraint Solving - Unification

Constraints

'd = int
'g = int → 'h
'd = int
bool = bool
'c = 'h
('c → 'f) → 'f → 'g = ('c → 'f) → 'c → 'd → 'c

Mapping

'a = ('c → 'f) → 'f → 'g
'b = 'c → 'f
'e = 'f → 'g

Constraint Solving - Unification

Constraints

'g = int → 'h
int = int
bool = bool
'c = 'h
('c → 'f) → 'f → 'g = 'c → ('f → 'c) → int → 'c

Mapping

'a = ('c → 'f) → 'f → 'g
'b = 'c → 'f
'e = 'f → 'g
'd = int

Constraint Solving - Unification

Constraints

int = int
bool = bool
'c = 'h
('c → 'f) → 'f → int → 'h = ('c → 'f) → 'c → int → 'c

Mapping

'a = ('c → 'f) → 'f → int → 'h
'b = 'c → 'f
'e = 'f → int → 'h
'd = int
'g = int → 'h

Constraint Solving - Unification

Constraints

'c = 'h

('c → 'f) → 'f → int → 'h = ('c → 'f) → 'c → int → 'c

Mapping

'a = ('c → 'f) → 'f → int → 'h

'b = 'c → 'f

'e = 'f → int → 'h

'd = int

'g = int → 'h

Constraint Solving - Unification

Constraints

('c → 'f) → 'f → int → 'c = ('c → 'f) → 'c → int → 'c

Mapping

'a = ('c → 'f) → 'f → int → 'c

'b = 'c → 'f

'e = 'f → int → 'c

'd = int

'g = int → 'c

'h = 'c

Constraint Solving - Unification

Constraints

'c → 'f = 'c → 'f
'f = 'c
int = int
'c = 'c

Mapping

'a = ('c → 'f) → 'f → int → 'c
'b = 'c → 'f
'e = 'f → int → 'c
'd = int
'g = int → 'c
'h = 'c

Constraint Solving - Unification

Constraints

'c = 'c

'f = 'f

'f = 'c

int = int

'c = 'c

Mapping

'a = ('c → 'f) → 'f → int → 'c

'b = 'c → 'f

'e = 'f → int → 'c

'd = int

'g = int → 'c

'h = 'c

Constraint Solving - Unification

Constraints

'f = 'c

int = int

'c = 'c

Mapping

'a = ('c → 'f) → 'f → int → 'c

'b = 'c → 'f

'e = 'f → int → 'c

'd = int

'g = int → 'c

'h = 'c

Constraint Solving - Unification

Constraints

int = int

'c = 'c

Mapping

'a = ('c → 'c) → 'c → int → 'c

'b = 'c → 'c

'e = 'c → int → 'c

'd = int

'g = int → 'c

'h = 'c

'f = 'c

Constraint Solving - Unification

Constraints

Mapping

'a = ('c → 'c) → 'c → int → 'c
'b = 'c → 'c
'e = 'c → int → 'c
'd = int
'g = int → 'c
'h = 'c
'f = 'c

Type Annotation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Variables
apply: 'a
f: 'b
v: 'c
t: 'd

Mapping
'a = ('c → 'c) → 'c → int → 'c
'b = 'c → 'c
'e = 'c → int → 'c
'd = int
'g = int → 'c
'h = 'c
'f = 'c

Type Annotation

```
let rec apply = fun f v t ->
  if t = 0 then
    v
  else
    apply f (f v) (t - 1)
  fi
```

Variables
apply: ('c → 'c) → 'c → int → 'c
f: 'c → 'c
v: 'c
t: int

Mapping
'a = ('c → 'c) → 'c → int → 'c
'b = 'c → 'c
'e = 'c → int → 'c
'd = int
'g = int → 'c
'h = 'c
'f = 'c

Type Annotation

```
let rec apply : ('c -> 'c) -> 'c -> int -> 'c
 = fun (f:'c -> 'c) (v:'c) (t:int) ->
   if t = 0 then
     v
   else
     apply f (f v) (t - 1)
   fi
```

Variables

apply: ('c → 'c) → 'c → int → 'c
f: 'c → 'c
v: 'c
t: int

Difficulties

- Polymorphic function application
- Matching
- Reference Types

Polymorphic Function Application

```
let f : 'a->'a = fun (x: 'a) -> x

let t1 = f true
let t2 = f 3
```

Polymorphic Function Application

```
let f : 'a->'a = fun (x: 'a) -> x

let t1 = f true
let t2 = f 3
```

'a = bool

'a = int

Solution

- Copy the type of f every time f is used

```
let f : 'a->'a = fun (x: 'a) -> x

let t1 = f true
let t2 = f 3
```

'b = bool

'c = int

Matching

- Different types for expression being matched and that used with unions:

```
type 'a list =
   | Nil
   | Cons of 'a * 'a list

let map = fun f l ->
  case l
   | Nil -> Nil
   | Cons(h,t) -> Cons(f h, map f t)
  esac
```

Matching

- Different types for expression being matched and that used with unions:

```
type 'a list =
   | Nil
   | Cons of 'a * 'a list

let map = fun f l ->
   case l
   | Nil -> Nil
   | Cons(h,t) -> Cons(f h, map f t)
   esac
```

`l` : 'a list

Matching

- Different types for expression being matched and that used with unions:

```
type 'a list =
   | Nil
   | Cons of 'a * 'a list

let map = fun f l ->
   case l
   | Nil -> Nil
   | Cons(h,t) -> Cons(f h, map f t)
   esac
```

`Cons(h,t)` :
'a *'a list

Solution

- Folding and Unfolding
- `l` is folded
- `Cons(h,t)` is unfolded
- Implicit in ML

Reference Types

- Classical ML Bug:

```
let r = ref (fun x -> x)
r := (fun x -> x + 1)
!r true
```

Solution

- Value Restriction
  - SML
  - Only allow values
- Modified Value Restriction
  - OCaml
  - Value assigned at first use
  - Monomorphic in use, polymorphic at initial definition

Conclusion

- In restricted type systems, full inference can be performed through unification
  - Allows code compactness and static type safety
- Type rules contain constraint generation
- Unification uses constraints to reduce potential solutions to the one correct one

References

- Krishnamurthi, Shriram, Programming Languages: Application and Interpretation, http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/
- Benjamin C. Pierce, Types and Programming Languages
- SML/NJ Type Checking Documentation, http://www.smlnj.org/doc/Conversion/types.html
- Francois Pottier, A modern eye on ML type inference, September 2005, http://gallium.inria.fr/~fpottier/publis/fpottier-appsem-2005.pdf