Introduction to Type Inference

Brian Hurt

22 April 2015

Why you care

- Forms the basis for most static type systems (Haskell, Ocaml, Scala, Swift, etc.)
- Increasingly used in optional type systems (Typed Racket, Clojure's core.typed, C++'s auto, static analysis tools like Coverity, etc.)
- The basis for more advanced type system features
- Helps in understanding otherwise cryptic error messages

What this talk does

- Introduces a simple subset of Haskell called SiML
- Creates a type inference/type checking algorithm for it
- Concurrently introduces the notation used by the research (the "Natural Deduction Style")

This code is not how GHC really works, however...

Technobabble

SiML is an enriched lambda caculus on which we perform a modified and simplified Hindley-Milner-Damas algorithm.

The SiML Language

```
data Expr a =
    Const Const
  | Var Var
  | If (Expr a) (Expr a) (Expr a)
  | Apply (Expr a) (Expr a)
  | Lambda Var (Expr a)
  | Let Var (Expr a) (Expr a)
  | LetRec [ (Var, (Expr a)) ]
           (Expr a)
  | Typed (Expr a) (Type a)
  deriving (Read, Show, Eq, Ord)
```

The SiML Language

```
instance Functor Expr where ...

instance Foldable Expr where ...
```

The SiML Language

```
type Var = String

data Const =
    ConstInt Integer
    | ConstBool Bool
    deriving (Read, Show, Eq, Ord)
```

The SiML Language

```
data Type a =
    TBool
  | TInt
  | TFun Type Type
  | TVar a
    deriving (Read, Show, Eq, Ord)

instance Functor Type where ...

instance Foldable Type where ...
```

The SiML Language

```
data Stmnt =
    LetStmnt Var Expr
    | LetRecStmnt [ (Var, Expr) ]
```

The basic intuition:

The structure of the code itself imposes constraints on the types.

We can use these constraints to determine the type.

For example, given:

if e_1 then e_2 else e_3

Where:

$e_1 : t_1 \qquad e_2 : t_2 \qquad e_3 : t_3$

Basic Intuition

Then we know that:

$$t_1 = Bool$$

$$t_2 = t_3 = \text{type of the if statement}$$

Natural Deduction Style

The “natural deduction style” of logical systems.

Natural Deduction Style

$$\frac{\Gamma, x : t_1 \dashv e : t_2}{\Gamma \dashv \lambda x.e : t_1 \to t_2}$$

Don't Panic

Natural Deduction Style

$$\frac{\text{If this is true}}{\text{Then this is true}}$$

Natural Deduction Style

$$\frac{e_1 : t_1 \quad e_2 : t_2 \quad e_3 : t_3 \quad t_1 = \mathsf{Bool} \quad t_2 = t \quad t_3 = t}{(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) : t}$$

What is meant by:

$$t_1 = t_2$$

The mathematician means:

You can replace all occurrences of t_1 with t_2.

Type Unification

So you can replace t_1 with Bool, and t_2 and t_3 with t:

$$\frac{e_1 : \text{Bool} \qquad e_2 : t \qquad e_3 : t}{(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : t}$$

The programmer means:

Change the state of the system so that $t_1 = t_2$.

Type Unification

```
type Matching a = ...

instance Monad Matching where
    ...

unify :: Type Var -> Type Var
         -> Matching (Type Var)
unify = undefined
```

So $t_1 = t_2$ becomes `unify t1 t2`.

Type Inference: If

```
typeInfer :: Expr Var
                -> Matching (Type Var)

typeInfer (If e1 e2 e3) = do
    t1 <- typeInfer e1
    t2 <- typeInfer e2
    t3 <- typeInfer e3
    _ <- unify t1 TBool
    unify t2 t3
```

Type Inference: Constants

Integer and boolean constants have their obvious types.

$$\frac{}{\mathsf{True} : \mathsf{Bool}} \qquad \frac{}{\mathsf{False} : \mathsf{Bool}}$$

$$\frac{}{0 : \mathsf{Int}} \quad \frac{}{1 : \mathsf{Int}} \quad \ldots$$

Type Inference: Constants

```
typeInfer (Const (ConstInt _)) =
    return TInt
typeInfer (Const (ConstBool _)) =
    return TBool
```

Type Inference: Constants

Type application is also obvious:

$$\frac{f : t_1 \to t_2 \qquad x : t_1}{(f\ x) : t_2}$$

Type Inference: Application

```
typeInfer (Apply f x) = do
    tf <- typeInfer f
    tx <- typeInfer x
    case tf of
        TFun t1 t2 -> do
            _ <- unify t1 tx
            return t2
        _ -> fail "Not a function!"
```

Type Inference: Typed Expressions

Typed expressions are also obvious:

$$\frac{x : t}{(x \;\; :: \;\; t) : t}$$

Type Inference: Typed Expressions

```
typeInfer (Typed x t) = do
    tx <- typeInfer x
    unify tx t
```

Type Inference: Var and Let

Consider...

```
typeInfer (Var x) = ...
typeInfer (Let x e1 e2) = ...
```

Type Inference: Var and Let

We need to pass around a map of variables to their types.

Type Inference: Var and Let

```
typeInfer :: [ (Var, Type Var) ]
               -> Expr Var
               -> Matching (Type Var)

typeInfer ctx (Var x) =
    case (lookup x ctx) of
        Just t -> return t
        Nothing -> fail "Unknown variable"

typeInfer ctx (Let x e1 e2) = do
    t1 <- typeInfer ctx e1
    typeInfer ((x, t1) : ctx) e2
```

Type Inference: If and Constants (v2.0)

```
typeInfer ctx (If e1 e2 e3) = do
    t1 <- typeInfer ctx e1
    t2 <- typeInfer ctx e2
    t3 <- typeInfer ctx e3
    _ <- unify t1 TBool
    unify t2 t3
typeInfer _ (Const (ConstInt _)) =
    return TInt
typeInfer _ (Const (ConstBool _)) =
    return TBool
```

Natural Deduction Style: Contexts

We use Γ to represent our context, and $\dashv$ to mean "evaluate the right hand side with the context on the left":

$$\frac{\Gamma \dashv e_1 : Bool \qquad \Gamma \dashv e_2 : t \qquad \Gamma \dashv e_3 : t}{\Gamma \dashv (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : t}$$

Natural Deduction Style: Contexts

$$\frac{x : t \in \Gamma}{\Gamma \dashv x : t}$$

$$\frac{\Gamma \dashv e_1 : t_1 \qquad \Gamma, x : t_1 \dashv e_2 : t_2}{\Gamma \dashv (\text{let } x = e_1 \text{ in } e2) : t_2}$$

Type Inference: Lambda

This works great for let (where we know the value being bound, and therefor the type).

But what about lambda?

Type Inference: Lambda

We need to know the type of the argument before we can infer the type of the body- except it's how the argument is used which determines it's type! For example:

```
(\x -> x + 1)
```

Type Inference: Lambda

With math, we can just assume it exists:

$$\frac{\Gamma, x : t_1 \dashv e : t_2}{\Gamma \dashv (\lambda x.e) : t_1 \to t_2}$$

With code, we can't pull this stunt.

Type Inference: Lambda

```
typeInfer ctx (Lambda x e) = do
    t1 <- what goes here?
    t2 <- typeInfer ((x, t1) : ctx) e
    return (TFun t1 t2)
```

Type Inference: Lambda

By the way, does this expression look familiar?

$$\frac{\Gamma, x : t_1 \dashv e : t_2}{\Gamma \dashv \lambda x.e : t_1 \rightarrow t_2}$$

(hint: Don't Panic)

The Two Types of Type Variables

Type Variables: Universal

Definition

A universal type variable can be any type.
Also known as: rigid type variables, skolem type variables.

Type Variables: Universal

Universal type variables are the “normal” type variables:

```
map :: (a -> b) -> [a] -> [b]
```

Type Variables: Existential

Definition

A existential type variable represents a specific type that is not yet known. The type is known when the type variable is unified with some other type.

Also known as: flexible type variables.

Type Variables: Universal

You don't see existential type variables in Haskell, but other languages do display them:

```
> ocaml
        OCaml version 4.01.0

# let r = ref None;;
val r : '_a option ref = {contents = None}
# r := Some 1;;
- : unit = ()
# r;;
- : int option ref = {contents = Some 1}
#
```

Type Variables

Universal type variable: this type could be any type in the whole wide universe.

Existential type variable: this type exists, but we don't know what it is yet.

Type Inference: Lambda

Existential type variables solve our Lambda problem.

```
typeInfer ctx (Lambda x e) = do
    t1 <- allocExistVar
    t2 <- typeInfer ((x, t1) : ctx) e
    return (TFun t1 t2)
```

Type Inference: Lambda

Of course, this requires some type signature changes:

```
type EVar = ...

type TVar = Either Var EVar

allocExistVar :: Matching (Type TVar)
allocExistVar = undefined

unify :: Type TVar -> Type TVar
         -> Matching (Type TVar)
unify = undefined

typeInfer :: [ (Var, Type TVar) ]
                -> Expr Var
                -> Matching TVar
...
```

Type Inference: Lambda

Existential type variables also solves the problem with let rec:

```
typeInfer ctx (LetRec defns e) = do
        ts <- mapM getEVar defns
        let ctx' = ts ++ ctx
        mapM_ (inferBody ctx') defns
        typeInfer ctx' e
    where
        getEVar (v, _) = do
            t <- allocExistVar
            return (v, t)
        inferBody c (_, e1) =
            typeInfer c e1
```

Two Problems

Type Variables: Problem 1: Typed Expressions

With Typed expressions, the AST gives us `Type Var`, but we need `Type TVar` to pass in to unify.

Note: the types given in the AST can only contain universal type variables!

Type Variables: Problem 1: Typed Expressions

Using `fmap Left` converts a `Type Var` into a `Type TVar` (making all variables universal):

```
typeInfer (Typed x t) = do
    tx <- typeInfer x
    unify tx (fmap Left t)
```

Type Variables: Problem 2: Using Variables

Consider map:

```
map :: (a -> b) -> [a] -> [b]
```

Inside map, a and b are universal type variables, and can not be unified with any other type.

But when we call map, they can be any type we want.

Type Variables: Problem 2: Using Variables

When we use a variable whose type has universal type variables, the universal type variables need to be converted into existential type variables.

But, all instances of the same universal type variable need to map to the same existential type variable.

Type Variables: Problem 2: Using Variables

```
import Data.List(nub)
import Data.Foldable(toList)
import Data.Maybe(fromJust)

typeInfer ctx (Var x) =
    case (lookup x ctx) of
        Just t -> do
            let uvars = nub (toList t)
            evars <- mapM (const allocExistVar)
                                            uvars
            let varMap = zip uvars evars
            return (fmap (fixVar varMap) t)
        Nothing -> fail "Unknown variable"
    where
        fixVar varMap v =
            fromJust (lookup v varMap)
```

Unify

Unify

```
data Type a =
    TBool
    | TInt
    | TFun Type Type
    | TVar a

unify :: Type TVar -> Type TVar
         -> Matching (Type TVar)
unify = undefined
```

Unify

The easy cases:

```
unify TBool TBool = return TBool
unify TInt  TInt  = return TInt
```

Unify

$$t_1 \to t_2 = t_3 \to t_4$$

implies:

$$t_1 = t_3 \qquad \&\& \qquad t_2 = t_4$$

```
unify (TFun t1 t2) (TFun t3 t4) = do
    t5 <- unify t1 t3
    t6 <- unify t2 t4
    return (TFun t5 t6)
```

Unify

Two universal type variables only unify if they're the same type variable.

```
unify (TVar (Left a)) (TVar (Left b))
    | a == b = return (TVar (Left a))
    | otherwise = fail "Type error"
```

Unify

Interesting Question:

What is the scope of a universal type variable? That is: when does one a in one type expression match represent the same (polymorphic) type as another a in some other type expression?

Existential Types

Unify: Existential Types

The rules for unifying existential types are:

- Existential type variables can be assigned another type at most once.
- If an existential type variablehas been assigned another type previously, we unify with that type instead.
- Otherwise, we assign the other type to the existential type variable.
- It is possible for both types to be existential type variables which have not been assigned previously, in which case we assign one to the other.

Unify: Existential Types

We need some way to set an existential type variable to a given type:

```
setEVar :: EVar -> Type TVar -> Matching ()
setEVar = undefined
```

And we need a way to get the value it was set to (if it was set previously):

```
getEVar :: EVar -> Matching (Maybe (Type TVar))
getEVar = undefined
```

Unify: Existential Types

```
unify (TVar (Right a)) t2 = do
    mt1 <- getEVar a
    case mt1 of
        Some t1 -> unify t1 t2
        None -> do
            setEVar a t2
            return t2
```

Unify: Existential Types

```
unify t1 (TVar (Right b)) = do
    mt2 <- getEVar b
    case mt2 of
        Some t2 -> unify t1 t2
        None -> do
            setEVar b t1
            return t1
```

Unify: Existential Types

All other patterns are type errors!

```
unify _ _ = fail "Type error"
```

Unify: Matching Utils

```
import Data.Default

type EVar = Int

data MState = MState {
    evarCounter :: Int,
    evarMappings :: [ (EVar, Type TVar) ]
}

type Matching a = State MState a

instance Default MState where
    def = MState 0 []
```

Unify: Matching Utils

```
allocExistVar :: Matching (Type TVar)
allocExistVar = do
    mstate <- get
    let evar = evarCounter mstate
    put (mstate { evarCounter = evar + 1 })
    return (Type (Right evar))
```

Unify: Matching Utils

```
setEVar :: EVar -> Type TVar -> Matching ()
setEVar evar typ = do
    mstate <- get
    let mappings =
        (evar, typ) : evarMappings mstate
    put (mstate { evarMappings = mappings })
```

Unify: Matching Utils

```
getEVar :: EVar -> Matching (Maybe (Type TVar))
getEVar evar = do
    mstate <- get
    let mappings = evarMappings mstate
    return (lookup evar mappings)
```

One Last Problem

How do you prevent existential type variables from "leaking" into a global type?

One Last Problem

Before a type can be promoted to the global scope:

- If an existential type variable has been assigned another type, replace the existential type variable with the assigned type.
- If an existential type variable has not been assigned another type, generate a new, unique universal type variable and assign it to the existential type.

Repeat the above until the type no longer has any existential type variables in it.

Type Variables: Universal

Definition

The act of replacing an unassigned existential type variable with a new, unique universal type variable is called Skolemization (named after Thoralf Skolem).

Summary

We now have function:

```
typeInfer :: [ (Var, Type Var) ]
                -> Expr Var
                -> Matching (Type Var)
```

Which can be used for both type inference and type checking.

Comming soon: working code in my github repo.

Summary

In addition, formula like the following aren't so scary any more:

$$\frac{\Gamma, x : t_1 \dashv e : t_2}{\Gamma \dashv \lambda x.e : t_1 \to t_2}$$

Summary

Where to go from here:

"Types and Programming Languages" Benjamin C. Pierce

Then start reading papers.

fini