Interpreting types as abstract values

Oleg Kiselyov (FNMOC)
Chung-chieh Shan (Rutgers University)

Formosan Summer School on Logic, Language, and Computation
July 9–10, 2008
Outline

- Introduction: Untyped $\lambda$-calculus with constants

  Untyped $\lambda$-calculus embedded in Haskell: EvalN.hs

  What is a type?

  Type reconstruction in Church style: TEvalNC.hs

  Type reconstruction in Curry style: TInfT.hs

  Sharing and polymorphism
    - Implicit TVE and State monad: TInfTM.hs
    - Let-bound polymorphism as inlining: TInfLetI.hs
    - Polymorphic type $\approx$ CSE denotation: TInfLetP.hs

  Type-checking an object language by type-checking the metalanguage
Untyped \( \lambda \)-calculus with constants

Terms \[ E, F ::= i \mid x \mid \lambda x. E \mid FE \mid E_1 + E_2 \mid \text{ifz} EE_1E_2 \]

Transitions
\[
(\lambda x. E)F \rightsquigarrow E\{x \mapsto F\} \quad (\beta)
\]
\[
(\lambda x. Ex) \rightsquigarrow E, \ x \notin \text{FVE} \quad (\eta)
\]
\[
i_1 + i_2 \rightsquigarrow i_1 + i_2 \quad (\delta_1)
\]
\[
\text{ifz } iE_1E_2 \rightsquigarrow \begin{cases} E_1 & i = 0 \\ E_2 & i \neq 0 \end{cases} \quad (\delta_2)
\]
\[
E[E_1] \rightsquigarrow E[E_2], \text{ if } E_1 \rightsquigarrow E_2 \quad (\text{congr})
\]
Hygiene

- Notation $E\{x \mapsto F\}$
- $\alpha$-conversion: $(\lambda x. \lambda y. xy(\lambda x. x))y$
Reduction strategies

- Normal order strategy: leftmost outermost rule application (‘redex’) first
- Applicative order: leftmost innermost first
- Call-by-name (CBN): leftmost outermost in a non-value; no $\eta$-rule
- Call-by-value (CBV): leftmost innermost in a non-value; no $\eta$-rule
Untyped $\lambda$-calculus with constants

Terms $E, F ::= i \mid x \mid \lambda x. E \mid FE \mid E_1 + E_2 \mid \text{ifz } EE_1E_2$

Transitions

$(\lambda x. E)F \rightsquigarrow E\{x \mapsto F\}$ \hspace{1cm} ($\beta$)

$(\lambda x. Ex) \rightsquigarrow E, \ x \notin \text{FVE}$ \hspace{1cm} ($\eta$)

$i_1 + i_2 \rightsquigarrow i_1 + i_2$ \hspace{1cm} ($\delta_1$)

$\text{ifz } iE_1E_2 \rightsquigarrow \begin{cases} E_1 & i = 0 \\ E_2 & i \neq 0 \end{cases}$ \hspace{1cm} ($\delta_2$)

$E[E_1] \rightsquigarrow E[E_2], \text{ if } E_1 \rightsquigarrow E_2$ \hspace{1cm} (congr)

$(\lambda y. \lambda z. 1 + 2)((\lambda x. 3x) + 4)$
Denotational semantics

A partial map from terms to something else

Domain of denotation
Mathematical integers of functions; Haskell integers or functions

Compositionality
The meaning of a compound syntactic phrase is a mathematical combination of the meanings of its immediate subphrases.
Outline

Introduction: Untyped \( \lambda \)-calculus with constants

- **Untyped \( \lambda \)-calculus embedded in Haskell: EvalN.hs**
  
  What is a type?

  Type reconstruction in Church style: TEvalNC.hs

  Type reconstruction in Curry style: TInfT.hs

Sharing and polymorphism
  
  Implicit TVE and State monad: TInfTM.hs
  
  Let-bound polymorphism as inlining: TInfLetI.hs
  
  Polymorphic type \( \approx \) CSE denotation: TInfLetP.hs

Type-checking an object language by type-checking the metalanguage
Representation of terms

type VarName = String
data Term = V VarName
  | L VarName Term
  | A Term Term
  | I Int
  | Term :+: Term
  | IFZ Term Term Term

deriving (Show, Eq)
infixl 9 'A'

Sample term

(\x. ifz x1(x + 2))10

(L "x" (IFZ (V "x") (I 1) ((V "x") :+: (I 2)))) 'A' (I 10)
data Value = VI Int | VC (Value -> Value)

instance Show Value where
  show (VI n) = "VI " ++ show n
  show (VC _) = "<function>"

Not quite: L "x" (V "x")
Environment

Meanings of (free) variables

type Env = ...

env0 :: Env
lkup :: Env -> VarName -> Value
ext :: Env -> (VarName,Value) -> Env

lkup x (ext env (y,v)) ≡ v  \quad x = y
lkup x (ext env (y,v)) ≡ lkup x env  \quad x \neq y
lkup x env0  \quad ≡ \bot
The evaluator

The type of eval

\[ \text{eval} :: \text{Env} \rightarrow \text{Term} \rightarrow \text{Value} \]

Sample terms

\[
(L \ "x" (L \ "y" (V \ "x" :+ V \ "y"))) \ 'A' (I \ 1) \\
(L \ "x" (L \ "y" (V \ "x" :+ V \ "y"))) \ 'A' (I \ 1) \ 'A' (I \ 2)
\]
Questions about the evaluator

- Why the result of evaluating \((L \times e)\) is called closure?
- Why `eval` expresses denotational semantics? Is `eval` compositional?
- Is `eval` a partial function? What kind of partiality?
- Does `eval` correspond to CBN or CBV? Or call-by-need?
- Can we see the correspondence with \(\beta\), \(\eta\) and \(\delta\)-rules?
- Where are the substitutions? In which sense one may say that the environment is a delayed substitution?
- Should we worry about hygiene? Where is the \(\alpha\)-conversion?
Discussing the evaluator

- Taking advantage of the meta-language
- More examples, of good and of bad terms
- Hidden errors: test3, test4, test6
- Multiplication: cheating
- Multiplication: Y-combinator
Exercises

- add `Fix Term` as a primitive
- add multiplication as a primitive
- add comparisons and booleans (again, as derived operations and as primitives; compare)
- implement Fibonacci, factorial
- How to make sure the interpreter does CBN or CBV? (using ad hoc and principled approaches)
- Extra credit: write a custom Show instance for terms
Outline

Introduction: Untyped $\lambda$-calculus with constants

Untyped $\lambda$-calculus embedded in Haskell: EvalN.hs

- What is a type?

  Type reconstruction in Church style: TEvalNC.hs

  Type reconstruction in Curry style: TInfT.hs

Sharing and polymorphism

  Implicit TVE and State monad: TInfTM.hs
  Let-bound polymorphism as inlining: TInfLetI.hs
  Polymorphic type $\approx$ CSE denotation: TInfLetP.hs

Type-checking an object language by type-checking the metalanguage
What is a type

“Types arise informally in any domain to categorize objects according to their usage and behavior” (Cardelli, Wegner, 1985)

- A type is a set of values
- A type is a set of values and operations on them
- Types as ranges of significance of propositional functions (Bertrand Russell, ‘Mathematical Logic as based on the theory of types’, 1908). In modern terminology, types are domains of predicates
- Type structure is a syntactic discipline for enforcing levels of abstraction (John Reynolds)
- A type system is a syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute (Benjamin Pierce, ‘Types and Programming Languages’)
Telling when a proposition ‘makes sense’

- \( \forall n. (\text{PerfectNumber}(n) \rightarrow \text{Even}(n)) \)
- \( p(0) \land (\forall x. p(x) \rightarrow p(s(x))) \rightarrow \forall x. p(x) \)

- Each propositional function \( P(x) \) makes sense only for some collection of \( x \) – the range of significance
- We can tell the ranges of significance just ‘by looking’
- We can tell \textit{syntactically} if or when the formula makes sense, but not if it is true
Type as an approximation

Thus a type is an approximation of a dynamic behavior that can be derived from the form of an expression

- If the type system is sound, the approximation is correct
- Types are useful usually not in what they tell but in what they do not tell
Outline

Introduction: Untyped $\lambda$-calculus with constants

Untyped $\lambda$-calculus embedded in Haskell: EvalN.hs

What is a type?

► **Type reconstruction in Church style: TEvalNC.hs**

Type reconstruction in Curry style: TInfT.hs

Sharing and polymorphism

- Implicit TVE and State monad: TInfTM.hs
- Let-bound polymorphism as inlining: TInfLetI.hs
- Polymorphic type $\approx$ CSE denotation: TInfLetP.hs

Type-checking an object language by type-checking the metalanguage
data Typ = TInt | Typ :> Typ
infixr 9 :>

- TInt :> TInt :> TInt
- (TInt :> TInt) :> TInt
- TInt :> (TInt :> TInt)
Church-style calculus

\((\lambda x: \text{int} \to \text{int}. (x: \text{int} \to \text{int})(1: \text{int})): (\text{int} \to \text{int}) \to \text{int}\)

- type checking
- type inference
- type reconstruction
Type inference as evaluation with different rules

- Type annotations on bound variables:
  \[ \text{L VarName Typ Term} \]
- Variable environment becomes type environment
- \[ \text{teval} :: \text{TEnv} \rightarrow \text{Term} \rightarrow \text{Typ} \]
- Abstract vs. concrete integers
- Checking both branches of a conditional
- Evaluation under lambda

Sample term: \( \lambda x : \text{int} \rightarrow \text{int}. \lambda y : \text{int}. (x y) + (y + 1) \)
Type check earlier terms
Soundness of the type system

The problematic code, test3, test4, test6 does not type check.

Soundness

- Well-typed terms don’t get stuck
  (formally: the progress property).
- If the term yields a value, it will be of the statically predicted type
  (formally: type preservation property, or subject reduction).

Soundness and the similarity of eval and teval
Decidability and recursion

- The problem with $tmul1$
- The problem with term$Y$ and delta
- Adding Fix and the evaluation and typing rules
- New expression for $tmul1$ and its application
- Is teval *always* terminating?
Decidability and recursion

- The problem with $tmul_1$
- The problem with $term_Y$ and $delta$
- Adding $Fix$ and the evaluation and typing rules
- New expression for $tmul_1$ and its application
- Is $teval$ always terminating?

Types are quickly decidable static approximations of dynamic behavior
Outline

Introduction: Untyped $\lambda$-calculus with constants

Untyped $\lambda$-calculus embedded in Haskell: EvalN.hs

What is a type?

Type reconstruction in Church style: TEvalNC.hs

- Type reconstruction in Curry style: TInfT.hs

Sharing and polymorphism

  - Implicit TVE and State monad: TInfTM.hs
  - Let-bound polymorphism as inlining: TInfLetI.hs
  - Polymorphic type $\approx$ CSE denotation: TInfLetP.hs

Type-checking an object language by type-checking the metalanguage
λ-calculus, Curry style

Types are well-formedness constraints on terms

- \((\lambda x:\text{int}\to \text{int}. (x: \text{int} \to \text{int})(1: \text{int})): (\text{int} \to \text{int}) \to \text{int}\)
- \(\lambda x: \text{int} \to \text{int}. x 1\)
- \(\lambda x. x 1\)

- Type checking, inference, reconstruction
- Church: \(Y\)-expression is meaningless
  Curry: \(Y\)-expression is meaningful but ill-typed
λ-calculus, Curry style

Types are well-formedness constraints on terms

- $(\lambda x: \text{int} \to \text{int}. (x: \text{int} \to \text{int})(1: \text{int})): (\text{int} \to \text{int}) \to \text{int}$
- $\lambda x: \text{int} \to \text{int. } x 1$
- $\lambda x. x 1$

- Type checking, inference, reconstruction
- Church: $Y$-expression is meaningless
  - Curry: $Y$-expression is meaningful but ill-typed

The whole value of type system is in what programs they *reject*

cf. security
A barrel has several apples. If we take five apples, there will remain half as many as there were before. How many apples were in the barrel?
Type variables

Examples

\( \lambda x. \lambda y. (xy) + (y + 1) \)

(Seen before, with annotations) Can we recover annotations and find the type of the term?

\( \lambda x. \lambda y. \text{if} z(xy)x(\lambda z. z) \)
Techniques

1. Fresh type variables
2. Keep track of the equations
3. Chase the binding of variables at the end
4. Solve equations: unification
5. Incremental solution; solved equations: substitution
Implementing techniques

- TVE, differences from TEnv
- Chasing through TVE
- unify: given tve, t1, t2, solve t1=t2, adding new equations to tve if any. On failure, return the reason.
- Trace unification when solving the last equation for \( \lambda x. \lambda y. \text{ifz}(xy)x(\lambda z. z) \)

Unification exercises:

- \( r\rightarrow r = (t\rightarrow u)\rightarrow(v\rightarrow w) \)
- \( (r\rightarrow r)\rightarrow(s\rightarrow t) = (b1\rightarrow b2)\rightarrow (b1\rightarrow b2) \)
- \( r\rightarrow s = (s\rightarrow s)\rightarrow(s\rightarrow s) \)
- \( ((r\rightarrow r)\rightarrow(r\rightarrow r))\rightarrow(s\rightarrow s) = (s\rightarrow s)\rightarrow ((r\rightarrow r)\rightarrow(r\rightarrow r)) \)
- The equations arising in our examples
Type inference algorithm

- Essentially the same as the type checking
- Hypothetical reasoning
- Constraint generation, propagation, resolution
- More declarative (cases for A and Fix)
- Tests: test0, test10, test1, testm21, test3
- Blaming a term rather than an annotation (test4, delta)
  Curry-style is stronger
Top-level parametric polymorphism

- Types with type variables: term2a, termid
- Meta-language let: term2id, termlet
- Sharing terms, but not types
Outline

Introduction: Untyped $\lambda$-calculus with constants

Untyped $\lambda$-calculus embedded in Haskell: EvalN.hs

What is a type?

Type reconstruction in Church style: TEvalNC.hs

Type reconstruction in Curry style: TInfT.hs

Sharing and polymorphism

Implicit TVE and State monad: TInfTM.hs
Let-bound polymorphism as inlining: TInfLetI.hs
Polymorphic type $\approx$ CSE denotation: TInfLetP.hs

Type-checking an object language by type-checking the metalanguage
Outline

Introduction: Untyped $\lambda$-calculus with constants

Untyped $\lambda$-calculus embedded in Haskell: EvalN.hs

What is a type?

Type reconstruction in Church style: TEvalNC.hs

Type reconstruction in Curry style: TInfT.hs

▶ Sharing and polymorphism
  Implicit TVE and State monad: TInfTM.hs
  Let-bound polymorphism as inlining: TInfLetI.hs
  Polymorphic type $\approx$ CSE denotation: TInfLetP.hs

Type-checking an object language by type-checking the metalanguage
The TVEM monad

- Simplifying the notation for the threading of tve
  Single-threading ⇒ beta-expansion, abstraction of (>>=)
- The general type of (>>=), TVEM ‘abbreviation’
- The do-notation as let
- Fully sugared version: TInfLetI.hs
- Updated definitions: newtv, unifyM, teval’, teval
- Pros and cons of monads
Outline

Introduction: Untyped \(\lambda\)-calculus with constants

Untyped \(\lambda\)-calculus embedded in Haskell: EvalN.hs

What is a type?

Type reconstruction in Church style: TEvalNC.hs

Type reconstruction in Curry style: TInfT.hs

Sharing and polymorphism

Implicit TVE and State monad: TInfTM.hs
Let-bound polymorphism as inlining: TInfLetI.hs
Polymorphic type \(\approx\) CSE denotation: TInfLetP.hs

Type-checking an object language by type-checking the metalanguage
let in meta- and object languages

term2id = let termid = L "x" vx
            in L "f" (L "y" ((I 2) :
                        ((termid 'A' (V "f"))
                          'A' ((termid 'A' vy) :+: (I 1)))))

term2id = Let ("termid", L "x" vx)
            (L "f" (L "y" ((I 2) :
                          ((V "termid" 'A' (V "f"))
                            'A' ((V "termid" 'A' vy) :+: (I 1))))))

- Evaluating Let: sharing, CSE
- Type checking Let: problematic
- Inlining before type checking, CSE
Type checking let

- New type for TEnv
- Modified cases: \((V \ x), (L \ x \ e)\)
- New case: \((\text{Let} (x,e) \ eb)\)
- Old tests pass
- New tests: test1*, termlet
- Let- and lambda-bound variables: test166, test167 vs. test176, test177
- Value restriction; sharing vs. copying in the presence of effects
Outline

Introduction: Untyped $\lambda$-calculus with constants

Untyped $\lambda$-calculus embedded in Haskell: EvalN.hs

What is a type?

Type reconstruction in Church style: TEvalNC.hs

Type reconstruction in Curry style: TInfT.hs

- **Sharing and polymorphism**
  - Implicit TVE and State monad: TInfTM.hs
  - Let-bound polymorphism as inlining: TInfLetI.hs
  - Polymorphic type $\approx$ CSE denotation: TInfLetP.hs

Type-checking an object language by type-checking the metalanguage
Hindley-Milner type inference

- Type check a CSE where it is defined rather than mentioned
- ‘Pre-compute’ the type of CSE: derive a ‘pre-type’
- A polymorphic type as a pre-type
- Non-canonical presentation
Truly free type variables

\[
data \text{TypS} = \text{TypS} \ [\text{TVarName}] \ \text{Typ}
\]

- Examples: \(\text{TypS} \ [\] \ \text{TInt},\)
  \(\text{TypS} \ [0] \ (\text{TV} \ 0 :> \ \text{TV} \ 0),\)
  \(\text{TypS} \ [3] \ ((\text{TV} \ 0 :> \ \text{TV} \ 3) :> \ \text{TV} \ 3)\)

- Terminology: generic or generalizable type variables, generalization

- Free and bound type variables in a type scheme

\[
\text{terml51} = \text{L} \ "x" \ (\text{Let} \ ("y", \ \text{L} \ "f" \ ((\text{V} \ "f") \ ‘A’ \ \text{vx})) \ \text{vy})
\]
\[
\text{terml52} = \text{terml51} \ ‘A’ \ (\text{I} \ 10)
\]
**Instantiation**

```haskell
type TEnv =
    (VarName, TVEM Typ)]

type TEnv =
    [(VarName, TypS)]

teval' env (V x) =
    lookup env x

nteval' env (V x) =
    instantiate(lookup env x)

teval' env (L x e) = do
    tv <- newtv
    te <- flip nteval' e $ ext env (x,
        return tv)
    return (tv :> te)

teval' env (L x e) = do
    tv <- newtv
    te <- flip nteval' e $ ext env (x,
        TypS [] tv)
    return (tv :> te)
```
Instantiation as an effect

- TVEM Typ and TypS as pre-types
- The nature of the effect
- An instance of a type scheme
- The code of instantiate

```plaintext
instantiate (TypS [1] TInt)
  -- TInt

instantiate (TypS [3] ((TV 0 -> TV 3) -> TV 3))
  -- (TV 0 -> TV 42) -> TV 42
```
Generalization

\begin{align*}
\text{teval'}\ env \quad (\text{Let}\ (x,e)\ eb) & = \text{do} \\
\_ & \leftarrow \text{teval'}\ env\ e \\
\text{let}\ t = \text{teval'}\ env\ e \\
\text{teval'} \quad (\text{ext}\ env\ (x, t)) & eb
\end{align*}

The two columns should be equivalent: specification for generalize
Generalization in detail

- Find the generic type variables in $t$, make it a type scheme
- A type variable is generic if it does not depend on the initial state used to reconstruct the type
- Goal: Execution of $\text{teval'} \ env \ e \equiv$ instantiation of the type scheme
- The same number of fresh variables in both cases

$\text{tve}_{\text{before}}$ \hspace{2cm} $\text{tve}_{\text{after}}$

\[ \text{teval'} \ env \ e \]

A free tv in a type produced by executing $\text{teval'} \ env \ e$ at $\text{tve}_{\text{before}}$ is generic if it would still be free if $\text{tve}_{\text{before}}$ were extended with arbitrary bindings.
Generalization in detail

A free tv in a type produced by executing `teval` \(\text{env e at tve}_{\text{before}}\) is generic if it would still be free if \(tve_{\text{before}}\) were extended with arbitrary bindings.

1. tv is unbound in \(tve_{\text{after}}\)
2. \(\forall tvb \in tve_{\text{before}}. tv \not\in (\text{tvsub tve}_{\text{after}} tvb)\) (or, for any tvb unbound in \(tve_{\text{before}}\))
Polymorphism

- Multiple instantiations of a type scheme $\equiv$ multiple executions of $\texttt{teval'}\ env\ e$ (only faster).
- The essence of polymorphism is inlining.
- A polymorphic type is an abstract interpretation of an expression that can be inlined in many places.
Examples

- Generalization in the top-level `teval`
- Inferred types are the same, but presented differently
- Inferred types of `termid` and `term2a`
- Our presentation is non-canonical
Outline

Introduction: Untyped λ-calculus with constants

Untyped λ-calculus embedded in Haskell: EvalN.hs

What is a type?

Type reconstruction in Church style: TEvalNC.hs

Type reconstruction in Curry style: TInfT.hs

Sharing and polymorphism

  Implicit TVE and State monad: TInfTM.hs
  Let-bound polymorphism as inlining: TInfLetI.hs
  Polymorphic type $\approx$ CSE denotation: TInfLetP.hs

► Type-checking an object language by type-checking the metalanguage
Untyped terms make no sense
The language should not permit even writing them, let alone evaluating them.
All expressible terms must be well-typed.

\[
\begin{align*}
term3 &= L \ "x" \ (IFZ \ vx \ (I \ 1) \ vy) \\
term4 &= L \ "x" \ (IFZ \ vx \ (I \ 1) \ (vx \ 'A' \ (I \ 1)))
\end{align*}
\]

- Terms ill-typed in the object language are ill-typed in the metalanguage
- Ill-typed object terms become inexpressible
- No need to write our own teval: use the ‘teval’ of Haskell
- The ‘teval’ of the metalanguage is more powerful, gives better error messages.