IMPLEMENTING SYSTEM T IN HASKELL

Jingren Wang

May 3rd, 2022

School of Computing Science

INTRODUCTION

How to enable a programming language that supports different types and function?

WHY SYSTEM T

System T is the simply typed λ -calculus, with natural numbers, booleans and recursion

Haskell is a popular functional programming language,

- pattern matching
- Data types
- GADTs
- property based testing

Features in the PL we care about:

- expressivity
- robustness
- efficiency

We design a toy language based on System T, and implement it using Haskell. Our language has

- natural numbers
- booleans
- higher-order functions
- recursion

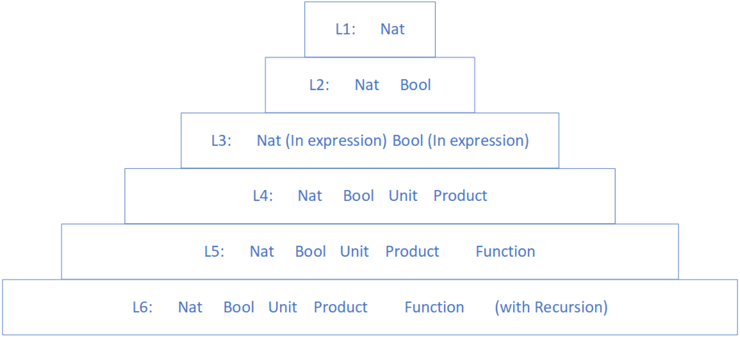

Figure 1: Design structure

BACKGROUND

What is a programming language?

- Grammar: The syntax of language, the expressivity
- Type system: Judgement and inference rules
- Operational Semantics: How to run programs

A judgement is a relation, connecting expressions to types. For example,

$$e : \tau$$

means e has the type τ. In **bidirectional type checking**, we split $e : \tau$ into check:

$$e \Leftarrow \tau$$

and infer/synthesis:

$$e \Rightarrow \tau$$

Frank Pfenning's Bidirectional checking rules, e.g.

Type check

$$\frac{e_1 \Leftarrow \mathsf{Bool} \qquad e_2 \Leftarrow T \qquad e_3 \Leftarrow T}{\vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Leftarrow T}\ \textsc{EIf}$$

Type Infer/Synthesis

$$\frac{e_1 \Leftarrow \mathsf{Bool} \qquad e_2 \Rightarrow T \qquad e_3 \Rightarrow T' \qquad T = T'}{\vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow T}\ \textsc{EIf}$$

Big-step:

$$\frac{e_3 \Downarrow \mathsf{Zero} \qquad e_1 \Downarrow e_1' \qquad e_1'\ val}{\mathsf{Iter}(e_1, e_2, e_3) \Downarrow e_1'}\ \textsc{EIter-Zero}$$

$$\frac{e_3 \Downarrow \mathsf{Suc}(e_3') \qquad e_2\ \mathsf{Iter}(e_1, e_2, e_3') \Downarrow e_4 \qquad e_4\ val}{\mathsf{Iter}(e_1, e_2, e_3) \Downarrow e_4}\ \textsc{EIter}$$

$$(1+1)+1 \Downarrow 3$$

Small-step:

$$\frac{}{\mathsf{Iter}(e_1, e_2, \mathsf{Zero}) \mapsto e_1}\ \textsc{EIter} \qquad \frac{}{\mathsf{Iter}(e_1, e_2, \mathsf{Suc}(e_3)) \mapsto e_2\ \mathsf{Iter}(e_1, e_2, e_3)}\ \textsc{EIter}$$

$$\frac{e_3 \mapsto e_3'}{\mathsf{Iter}(e_1, e_2, e_3) \mapsto \mathsf{Iter}(e_1, e_2, e_3')}\ \textsc{EIter}$$

$$(1+1)+1 \mapsto 2+1 \mapsto 3$$

L3: A LANGUAGE WITH NUMBERS AND BOOLEANS

Grammar of L3:

$$
\begin{array}{llll}
\textsc{Types} & T & ::= & \mathsf{Nat} \mid \mathsf{Bool} \\
\textsc{Expressions} & e & ::= & \mathsf{Zero} \mid \mathsf{Suc}(e) \mid \mathsf{true} \mid \mathsf{false} \mid \\
 & & & \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid e_1 + e_2 \mid e_1 * e_2 \\
\textsc{Values} & v & ::= & \mathsf{Suc}^n(\mathsf{Zero}) \mid \mathsf{true} \mid \mathsf{false} \\
\textsc{Judgements} & \mathcal{J} & ::= & \vdash e : T
\end{array}
$$

```
data Exp
    = EZero
    | ESucc Exp
    | ETrue
    | EFalse
    | EAdd Exp Exp
    | EMul Exp Exp
    | EIf Exp Exp Exp

data Val
  = VSuccN Nat
  | VTrue
  | VFalse
```

```
newtype TC a = TC {runTC :: Either TCError a}
  deriving (Eq, Show, Functor, Applicative, Monad)

tccheck :: Exp → Ty → TC ()

tcinfer :: Exp → TC Ty
```

Generalized Algebraic Data Types (GADTs)

- Encode invariants about a data structure in its type
- Enforce in a "type-safe" way

EXTRINSIC AND INTRINSIC

```
data Exp
    = EZero
    | ESucc Exp
    | ETrue
    | EFalse
    | EAdd Exp Exp
    | EMul Exp Exp
    | EIf Exp Exp Exp
```

```
data Exp :: Ty → Type where
   EZero :: Exp 'TNat
   ESucc :: Exp 'TNat → Exp 'TNat
   ETrue :: Exp 'TBool
  EFalse :: Exp 'TBool
    EAdd :: Exp 'TNat → Exp 'TNat → Exp 'TNat
    EMul :: Exp 'TNat → Exp 'TNat → Exp 'TNat
     EIf :: Exp 'TBool → Exp ty → Exp ty → Exp ty
```

EXTRINSIC AND INTRINSIC

```
eval (EIf e1 e2 e3) =
  do
    b1 ← eval e1
    case b1 of
      VTrue → eval e2
      VFalse → eval e3
      _ → fail (show e1 ++ "has a type of" ++ show b1)
```

```
eval (EIf e1 e2 e3) =
  case eval e1 of
    VTrue → eval e2
    VFalse → eval e3
```

Extrinsic bi-directional checking exmaple:

```
tccheck (EIf e1 e2 e3) ty =
  do
    _ ← tccheck e1 TBool
    _ ← tccheck e2 ty
    _ ← tccheck e3 ty
    return ()
tccheck e ty = tcfail ("check: "
++ show e ++ " is not an expression
of type " ++ show ty ++ "!")
```

```
tcinfer (EIf e1 e2 e3) =
  do
    _ ← tccheck e1 TBool
    tcin2 ← tcinfer e2
    tcin3 ← tcinfer e3
    if tcin2 == tcin3
      then return tcin2
      else
        tcfail
          ( "infer: " ++
"EIf has different type in last two expression:" ++
show e2 ++ "has type of" ++ show tcin2
              ++ show e3
              ++ "has type of"
              ++ show tcin3
          )
```

L6: A LANGUAGE WITH MANY TYPES

SYNTAX OF L6

Grammar of L6:

$$
\begin{array}{llll}
\textsc{Types} & T & ::= & \mathsf{Nat} \mid \mathsf{Bool} \mid \mathsf{Unit} \mid T \times T \mid T \to T \\
\textsc{Expressions} & e & ::= & \mathsf{Zero} \mid \mathsf{Suc}(e) \mid \mathsf{true} \mid \mathsf{false} \mid \\
& & & * \mid \mathsf{Fst}(e) \mid \mathsf{Snd}(e) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \\
& & & x \mid \lambda(x : T).e \mid e_1\ e_2 \mid (e_1, e_2) \mid \\
& & & \mathsf{Iter}(e_1, e_2, e_3) \\
\textsc{Context} & \Gamma & ::= & \bullet \mid \Gamma, x : T \\
\textsc{Judgements} & \mathcal{J} & ::= & x : T \in \Gamma \mid \Gamma \vdash e : T \mid e\ val
\end{array}
$$

BIDIRECTIONAL CHECKING RULES OF L6

Type check

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda(x : A).e \Leftarrow A \to B}\ \textsc{Lam} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B' \qquad \Gamma \vdash e_2 \Leftarrow A \qquad B' = B}{\Gamma \vdash e_1 e_2 \Leftarrow B}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow A \qquad \Gamma \vdash e_2 \Leftarrow A \to A \qquad \Gamma \vdash e_3 \Leftarrow \mathsf{Nat}}{\Gamma \vdash \mathsf{Iter}(e_1, e_2, e_3) \Leftarrow A}\ \textsc{Iter}$$

Type Infer/Synthesis

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda(x : A).e \Rightarrow A \to B}\ \textsc{Lam} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1 e_2 \Rightarrow B}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Leftarrow A \to A \qquad \Gamma \vdash e_3 \Leftarrow \mathsf{Nat}}{\Gamma \vdash \mathsf{Iter}(e_1, e_2, e_3) \Rightarrow A}\ \textsc{Iter}$$

Small-step semantics of L6

$$\frac{e \mapsto e'}{\mathsf{Suc}(e) \mapsto \mathsf{Suc}(e')}\ \textsc{Suc}$$

$$\frac{e_1 \mapsto e_1'}{(e_1, e_2) \mapsto (e_1', e_2)}\ \textsc{Tuple-Left} \qquad \frac{e_2 \mapsto e_2'}{(e_1, e_2) \mapsto (e_1, e_2')}\ \textsc{Tuple-Right}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mapsto \mathsf{if}\ e_1'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3}\ \textsc{EIf}$$

$$\frac{}{\mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3 \mapsto e_2}\ \textsc{EIf-True}$$

$$\frac{}{\mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3 \mapsto e_3}\ \textsc{EIf-False}$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \textsc{EApp} \qquad \frac{e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \textsc{EApp} \qquad \frac{}{(\lambda x.e_1)e_2 \mapsto [e_2/x]e_1}\ \textsc{EApp}$$

Small-step semantics of L6

$$\frac{}{\textsf{Fst}((e_1, e_2)) \mapsto e_1}\ \textsc{EFst} \qquad \frac{}{\textsf{Snd}((e_1, e_2)) \mapsto e_2}\ \textsc{ESnd}$$

$$\frac{e \mapsto e'}{\textsf{Fst}(e) \mapsto \textsf{Fst}(e')}\ \textsc{EFst} \qquad \frac{e \mapsto e'}{\textsf{Snd}(e) \mapsto \textsf{Snd}(e')}\ \textsc{ESnd}$$

$$\frac{}{\textsf{Iter}(e_1, e_2, \textsf{Zero}) \mapsto e_1}\ \textsc{EIter}$$

$$\frac{}{\textsf{Iter}(e_1, e_2, \textsf{Suc}(e_3)) \mapsto e_2\ \textsf{Iter}(e_1, e_2, e_3)}\ \textsc{EIter}$$

$$\frac{e_3 \mapsto e_3'}{\textsf{Iter}(e_1, e_2, e_3) \mapsto \textsf{Iter}(e_1, e_2, e_3')}\ \textsc{EIter}$$

Big-step semantics of L6

$$\frac{}{\textsf{Zero} \Downarrow \textsf{Zero}}\ \textsc{Zero} \qquad \frac{e \Downarrow e' \qquad e'\ val}{\textsf{Suc}(e) \Downarrow \textsf{Suc}(e')}\ \textsc{Suc}$$

$$\frac{}{\textsf{true} \Downarrow \textsf{true}}\ \textsc{True} \qquad \frac{}{\textsf{false} \Downarrow \textsf{false}}\ \textsc{False}$$

$$\frac{e_1 \Downarrow \textsf{true} \qquad e_2 \Downarrow e_2' \qquad e_2'\ val}{\textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 \Downarrow e_2'}\ \textsc{EIf-True}$$

$$\frac{e_1 \Downarrow \textsf{false} \qquad e_3 \Downarrow e_3' \qquad e_3'\ val}{\textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 \Downarrow e_3'}\ \textsc{EIf-False}$$

$$\frac{e_3 \Downarrow \textsf{Zero} \qquad e_1 \Downarrow e_1' \qquad e_1'\ val}{\textsf{Iter}(e_1, e_2, e_3) \Downarrow e_1'}\ \textsc{EIter-Zero}$$

$$\frac{e_3 \Downarrow \textsf{Suc}(e_3') \qquad e_2\ \textsf{Iter}(e_1, e_2, e_3') \Downarrow e_4 \qquad e_4\ val}{\textsf{Iter}(e_1, e_2, e_3) \Downarrow e_4}\ \textsc{EIter}$$

Big-step semantics of L6

$$\frac{}{* \Downarrow *}\ \textsc{EUnit} \qquad \frac{e_1 \Downarrow e_1' \quad e_2 \Downarrow e_2' \quad e_1'\ val \quad e_2'\ val}{(e_1, e_2) \Downarrow (e_1', e_2')}\ \textsc{ETuple}$$

$$\frac{e \Downarrow (e_1, e_2) \quad e_1\ val}{\mathsf{Fst}(e) \Downarrow e_1}\ \textsc{Fst} \qquad \frac{e \Downarrow (e_1, e_2) \quad e_2\ val}{\mathsf{Snd}(e) \Downarrow e_2}\ \textsc{Snd}$$

$$\frac{}{\lambda(e_1 : A).e_2 \Downarrow \lambda(e_1 : A).e_2}\ \textsc{ELam} \qquad \frac{e_1 \Downarrow \lambda(x : A).e}{e_1\ e_2 \Downarrow [e_2/x]e}\ \textsc{EApp}$$

EXAMPLE PROGRAM

EXAMPLE PROGRAM

Addition function:

```
addHs :: Nat → Nat → Nat
addHs Zero n = n
addHs (Succ n) m = Succ (n + m)
```

```
addL6 :: Nat → (Nat → Nat)
addL6 = λ(n :: Nat).λ(m :: Nat).
        EIter(m, λ(t :: Nat).Suc(t), n)
```

Main idea:

Iterate m for n times.

EXAMPLE PROGRAM

Fibonacci function:

```
fibHs :: Nat → Nat
fibHs Zero = Zero
fibHs (Succ Zero) = Succ Zero
fibHs (Succ (Succ n)) = fibHs (Succ n) + fibHs n
```

```
(λ(fib_m :: Nat).
fst(((λ(fib_n :: Nat). Iter((0, S(0)),
(λ(fib_t : (Nat × Nat)). (snd(fib_t),
(((λ(nat_n :: Nat). (λ(nat_m :: Nat).
Iter(nat_m,
(λ(nat_t :: Nat). S(nat_t)), nat_n)))
fst(fib_t)) snd(fib_t)))),
fib_n))
fib_m)))
```

Main idea: Save the current result as the first item in the current tuple.

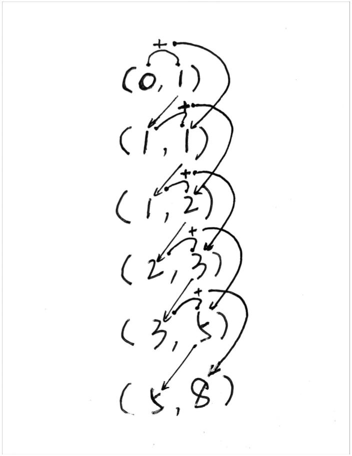

Figure 2: Fibonacci function

EXAMPLE PROGRAM

Factorial function:

```
facHs :: Nat → Nat
facHs Zero =
  Succ Zero
facHs (Succ n) =
  Succ n ∗ facHs n
```

```
facL6 :: Nat → Nat
facL6 =
(λ(m : Nat). snd(((λ(n : Nat).
 Iter((0, S(0)),
     (λ(t : (Nat × Nat)).
       (S(fst(t)),
        (((λ(n : Nat). (λ(m : Nat).
          Iter(0,
            ((λ(n : Nat). (λ(m : Nat).
              Iter(m, (λ(t : Nat). S(t)),
            n))) m),
        n)))
    S(fst(t))) snd(t)))), n)) m)))
```

Main idea: Save the current result as the second item in a tuple.

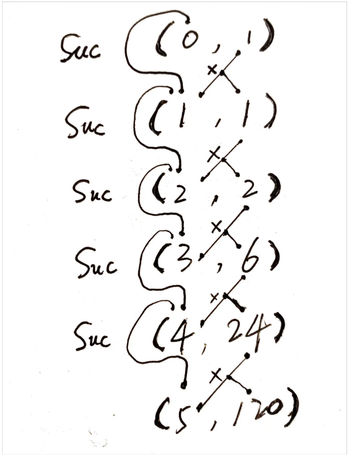

Figure 3: Factorial function

Ackermann function:

$$\mathsf{A}(0, n) = n + 1$$

$$\mathsf{A}(m+1, 0) = \mathsf{A}(m, 1)$$

$$\mathsf{A}(m+1, n+1) = \mathsf{A}(m, \mathsf{A}(m+1, n))$$

Ackermann function:

```
ackerHs :: Nat → Nat → Nat
ackerHs Zero n = Succ n
ackerHs (Succ m) Zero = ackerHs m (Succ Zero)
ackerHs (Succ m) (Succ n) = ackerHs m (ackerHs (Succ m) n)
```

EXAMPLE PROGRAM

Ackermann function:

```
compExp :: (Nat → Nat) × (Nat → Nat) → (Nat → Nat)

itExp :: (Nat → Nat) → Nat → (Nat → Nat)

sExp :: Nat → Nat

rExp :: (Nat → Nat) → ((Nat → Nat) → (Nat → Nat)) → (Nat → Nat)

ackerExp :: Nat → (Nat → Nat)
ackerExp = λ(n :: Nat). Iter(sExp, rExp, n)
```

EVALUATION

Tasty framework
What we have tested:

- every inferable expression can be checked for its inferred type.
- every well-typed expression can be inferred
- Progress: Well-typed expressions always reduce to a value.
- Type-preservation: Well-typed expressions reduce to a value of the same type.

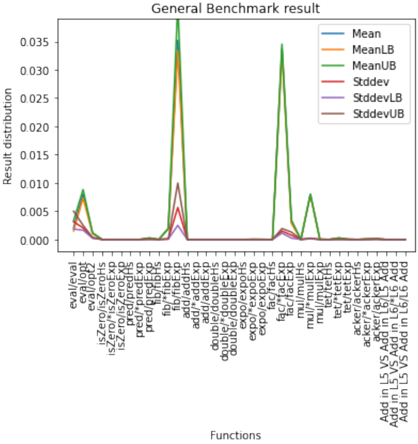

Figure 4: Function general benchmark result

CONCLUSION

Features in the language we designed and built:

- expressivity
- robustness
- efficiency

Get the source of this project and the thesis from

`https://github.com/wjrforcyber/SystemT`

Thank you!

Questions?