Intrinsically-Typed Interpreters

Syouki Tsuyama
Tokyo Institute of Technology

Intrinsically-Typed AST : Expr

Intrinsically-typed AST is data structure for well-typed terms and representation of type system of object language.

```
data Expr : Ty -> Set where
  true  : Expr Bool
  false : Expr Bool
  zero  : Expr Nat
  succ  : Expr Nat -> Expr Nat

-- well-typed terms type-check in host language
succ zero : Expr Nat

-- ill-typed terms don't type-check
succ true  👈 type error!
```

Intrinsically-Typed Interpreter : eval

Intrinsically-typed interpreter is executable and type-safe specification of semantics of a language.

Type safety of object language follows from the fact that the function `eval` type-checks in the dependently-typed host language.

```
eval : Expr T -> Val T  👈 Statement of type safety
eval true = true        👈 Proof
. . .
```

My research

I'm trying to develop technique & library for implemenation of intrinsically-typed AST & interpreter for language supporting advanced features.

In concrete, algebraic effects & handlers

Contents

Contents

Definition of Typed Language

- Terms , Values , Types , etc ..
- Typing Rules
- Evaluation Rules

· Structure of Rules

$$\frac{\texttt{Assumption}}{\texttt{Conclusion}}$$

· Values

```
v ::=
  true
  false
  zero
  succ v
```

· Terms

```
t ::=
  true
  false
  if t then t else t
  zero
  succ t
  iszero t
```

· Type

```
T ::=
  Bool
  Nat
```

· Typing rules

$$\texttt{true : Bool}$$

$$\texttt{false : Bool}$$

$$\frac{\texttt{c : Bool, t : T, e : T}}{\texttt{if c t e : T}}$$

· Evaluation rules

$$\texttt{true} \Downarrow \texttt{true}$$

$$\texttt{false} \Downarrow \texttt{false}$$

$$\frac{\texttt{c} \Downarrow \texttt{true, t} \Downarrow \texttt{v}}{\texttt{if c t e} \Downarrow \texttt{v}}$$

$$\frac{\texttt{c} \Downarrow \texttt{false, e} \Downarrow \texttt{v}}{\texttt{if c t e} \Downarrow \texttt{v}}$$

Writing Typed Language Specification

1. Implement AST , type checker and interpreter according to the definition using host language
2. Test type checker and interpreter
3. Prove type-safety

```
typeOf e is T =>
eval e is value of type T
```

```
data Ty   = . . .
data Expr = . . .
data Val  = . . .

typeOf : Expr -> Ty
eval : Expr -> Val
```

Problems

- Manual proofs takes time and effort.
- Modifying definition requires new proofs.
- Interpreter needs to handle ill-typed terms (e.g. 1 + true).

Dependent Type Approach

Implement typing and evaluation rules
in a dependently-typed language.

→

- Automatic proof of type safety
- Implementation and proof are modified at the same time
- No need to think about ill-typed terms

Contents

Dependent Types

Dependent types can refer to values, not just to other types.

```
-- List of n elements of type T
data List : Set -> ℕ -> Set where
  []  : List T zero
  _::_ : T -> List T n -> List T (succ n)

[]            : List T 0
1 :: []       : List ℕ 1
2 :: 1 :: []  : List ℕ 2
```

Dependent Types ~Advanced Verification~

Execution of head [] in plain language raises runtime error,

but dependently-Typed language can detect that with type checker.

```
-- Get the first element from a non-empty list
-- Never fails at runtime
head : List T (succ n) -> T
head (x :: xs) = x

head []    👈 type error!
head (1 :: []) = 1
```

The Curry-Haward Correspondence

Logic		Program
Proposition	⟷	Types
Proofs	⟷	Terms

e.g. Proposition of Equality for values x and y (x ≡ y)

```
data _≡_ : A -> A -> Set where   👈 Signature of Equality
  refl : x ≡ x                   👈 Rule of Proof

theorem : (1 + 1) ≡ 2   👈 Statement
theorem = refl          👈 Proof
```

Automatic Proof

If the program passes type checking,

it means that the proposition is valid.

```
-- correct
theorem : (1 + 1) ≡ 2
theorem = refl

-- no way to prove 0 ≡ 1
wrong : 0 ≡ 1
wrong = refl    👈 type error!
```

Contents

Intrinsically-Typed AST : Expr T

Data structure for well-typed terms and representation of typing rules.

```
data Ty : Set where
  Bool : Ty
  Nat  : Ty

-- e : Expr T  corresponds to the proposition,
-- e is an expression of type T.
data Expr : Ty -> Set where
  true  : Expr Bool
  false : Expr Bool
  if    : Expr Bool -> Expr T -> Expr T -> Expr T
  zero  : Expr Nat
  succ  : Expr Nat -> Expr Nat
```

$$\frac{c : \mathrm{Bool},\ t : T,\ e : T}{\mathrm{if}\ c\ t\ e : T}$$

Testing Typing Rules

The type checker of the host language automatically performs type derivation of the object language

```
-- well-typed terms type-check
t1 : Expr Nat
t1 = if true (succ zero) zero

-- ill-typed terms doesn't type-check
fail1 = succ true     👈 type error! => failure

fail2 : Expr Nat
fail2 = iszero zero  👈 type error! => failure
```

Intrinsically-Typed Interpreter : eval

The function `eval` is representation of evaluation rule.

as well as proof of type safety.

`eval` type-checks => Type safety is valid.

Expression of type T is always evaluated into value of type T

```
eval : Expr T -> Val T    👈 Statement of type safety
eval true = true          👈 Proof
. . .
```

Testing Evaluation Rules

We can easily test the interpreter using Equality (_≡_) type

test passes type check => The result of evaluation is correct

test case :

```
test : eval (if true 1 0) ≡ 1
test = refl  👈 Correct

fail : eval (iszero zero) ≡ false
fail = refl  👈 type error! => failure
```

Contents

Previous Reasearch

Intrinsically-typed interpreters supporting advanced features.

- Mutable state[Poulsen POPL'18] 👉 Mutable store and pointer types
- Middleweight Java[Poulsen POPL'18] 👉 Object oriented programming
- Linear types[Rouvoet CPP'20] 👉 Verify that a particular variable is used exactly one

Contents

Future work

I'm trying to implement intrinsically-typed interpreters for languages with algebraic effect & handlers.

To do so, I need to integrate delimited continuation operation into object language.

Because there is a deep relationship between effect systems and continuation.

Continuation

Continuation : Rest of the computation

e.g. `5 * (2 + 3) - 4` `result : 21`

When the interpreter evaluates `2 + 3`,

the continuation is `5 * [·] - 4` .

`[·]` will be replaced by the result of `2 + 3`

Delimited Continuation Operation : reset

`reset e` : Limit the range of continuation of e

e.g. `5 * reset((2 + 3) - 4)`

`result : 21`

When the interpreter evaluates `2 + 3`,

the continuation is `[ · ] - 4`

Delimited Continuation Operation : shift

`shift(fun k -> e)`:

evaluate e with binding the current continuation to k

e.g. `5 * shift(fun k -> (k 5) + 1) - 4`

`result : 22`

When the interpreter evaluates `shift(.)`,
the continuation is `5 * [ · ] - 4` .

So `fun x -> 5 * x - 4` is binded to k ,
thus k `5 = 21` and the result is 22 .

Program example using shift/reset

Non-deterministic computation

```
coin () = shift (fun k -> [k true , k false])

reset( if coin() then 1 else 0 )
-> [k true , k false] 👈 k is fun x -> if x then 1 else 0
-> [if true then 1 else 0 , k false]
-> [1 , k false]
-> [1 , if false then 1 else 0]
-> [1 , 0]
```

Algebraic Effect & Handler

An approach to manage computational effects.

```
-- user defined effect
effect NDet {coin () : Bool}

-- handler : Determine the behavior of the operation.
-- k is exactly the continuation that was just introduced
with handler {
  coin () k -> [k true , k false] }
{
  if coin() then 1 else 0
}

>>[1 , 0]
```

My work so far & future

I have already implemented intrinsically-typed AST & CPS interpreter for shift/reset .

I’m trying to try to apply these techniques to implement a language with Algebraic effects and handlers.

My Current Short-term Goal

To implement intrinsically AST & CPS interpreter
for λ_{eff}, a small language with algebraic effect & handlers.

Summary

- Intrinsically-typed AST & Interpreter is an approach to automatically proveing type safety of the object language.
- I'm trying to develop Intrinsically-typed AST & CPS Interpreter for a language with effects system .

End.

CPS Interpreter

CPS : Continuation Passing Style

CPS interpreter recieves rest of evaluation.

```
eval : Expr -> Cont -> Val
eval (Num i) k = k (Num i)
eval (n + m) k =
  eval n $ \(Num i) ->
  eval m $ \(Num j) ->
  k $ Num (i + j)

eval ((Num 2) + (Num 3)) id
= eval (Num 2) $ \(Num i) ->
```

CPS Interpreter

CPS interpreter is appropriate for shift/reset language,

because it always recieves continuation at the evaluation.

```
eval : Expr -> Cont -> Val

eval (reset e) k =
  k $ eval e id

eval (shift f) k =
  eval (f k) id
```

Effect types

The function type knows
what computational effect the expressions may cause.

```
-- user defined effect
effect NDet {coin () : Bool}

              Effect caused by this function

fun causeNDet() : < NDet > int {
  if coin() then 1 else 0
}
```

Effect types

You can’t cause effects which the type doesn’t know .

This function causes no effects

```
👇 type error!
fun pure() : <> int {
  let results = causeNDet() ;
  results[0]
}
```

This causes the effect NDet