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

Abstract

We expound a view of type checking as evaluation with 'abstract values'. Whereas dynamic semantics, evaluation, deals with (dynamic) values like 0, 1, etc., static semantics, type checking, deals with approximations like int. A type system is sound if it correctly approximates the dynamic behavior and predicts its outcome: if the static semantics predicts that a term has the type int, the dynamic evaluation of the term, if it terminates, will yield an integer.

As object language, we use simply-typed and let-polymorphic lambda calculi with integers and integer operations as constants. We use Haskell as a metalanguage in which to write evaluators, type checkers, type reconstructors and inferencers for the object language.

We explore the deep relation between parametric polymorphism and 'inlining'. Polymorphic type checking then is an optimization allowing us to type check a polymorphic term at the place of its definition rather than at the places of its use.

1 Introduction: Untyped λ-calculus with constants

This is the background material the students are supposed to know already. It is here for completeness only.

Our object language in this course is the simply typed λ-calculus with integer constants, addition and branching on zero. When extended with booleans, the language is often called *PCF*.

Figure 1 is the 'standard presentation' of such a language, as typically found in programming language books and articles. The syntax is given by a context-free grammar with the non-terminal 'Terms'; i stands for an integer literal; x stands for one out of the countably infinitely-many *variables*. The semantics is given by transition rules.

Variables are either free or bound. The α-conversion (aka, hygiene) specifies that two lambda terms are equivalent if they differ only in the naming of bound

$$
\begin{array}{lll}
\text{Terms} & E, F ::= i \mid x \mid \lambda x.\, E \mid FE \mid E_1 + E_2 \mid \mathrm{ifz}\, E E_1 E_2 & \\
\text{Transitions} & (\lambda x.\, E)F \rightsquigarrow E\{x \mapsto F\} & (\beta) \\
 & (\lambda x.\, Ex) \rightsquigarrow E,\ x \notin \mathrm{FV}\, E & (\eta) \\
 & i_1 + i_2 \rightsquigarrow i_1 \dot{+} i_2 & (\delta_1) \\
 & \mathrm{ifz}\, i E_1 E_2 \rightsquigarrow \begin{cases} E_1 & i = 0 \\ E_2 & i \neq 0 \end{cases} & (\delta_2) \\
 & E[E_1] \rightsquigarrow E[E_2], \text{if } E_1 \rightsquigarrow E_2 & (\text{congr})
\end{array}
$$

Figure 1: Basic calculus: λ-calculus with constants

variables. For example, these two terms are equivalent: $(\lambda x.\, \lambda y.\, xy(\lambda x.\, x))y \equiv_\alpha (\lambda x.\, \lambda z.\, xz(\lambda x.\, x)) \equiv_\alpha (\lambda u.\, \lambda z.\, uz(\lambda x.\, x))$

In the β-rule, the notation $E\{x \mapsto F\}$ means a capture-avoiding substitution of F for x in E. Here is an example where the capture-avoidance, hygiene, matters: $\lambda x.(\lambda z.\, \lambda x.\, z)x$.

It may happen that none of the transitions apply to a term: e.g., 1 or $\lambda x.\, x$. Such a term is in *normal form.* It may happen that several rules may apply: $(\lambda y.\, \lambda z.\, 1{+}2)((\lambda x.\, 3x){+}4)$. A *reduction strategy* determines which of the several possible rule applications to do first.

- Normal order strategy: leftmost outermost rule application ('redex') first;
- Applicative order: leftmost innermost first;
- Call-by-name (CBN): leftmost outermost in a non-*value*; the η-rule does not apply;
- Call-by-value (CBV): leftmost innermost in a non-*value*; the η-rule does not apply;

In CBV and CBN, certain terms are declared *values* based on their syntactic form: integers, variables and lambda-abstractions. Values are not further reducible. The transition rules only performed in certain contexts: the rules are context-sensitive.

With normal or applicative order strategies, we either reach a normal form, or we keep applying transition rules forever. Church-Rosser theorem; the normal order reduction strategy will always lead to the normal form if there is one. With CBV or CBN, we have three possibilities: reaching a value, keep applying the rules, or *getting stuck.*

Exercise: show the reductions in the above term using the four strategies. Why the η-rule is not used in CBV and CBN?

The reduction semantics is an *operational* semantics: a term *means* what it eventually reduces to. An operational semantics is a (partial) mapping from terms to terms. Why is it partial?

The *denotational* semantics gives meaning to program phrases by mapping or relating them to some other objects; the meaning of the latter is considered 'obvious'. When giving denotational semantics, we have to specify these other objects. Often they are *mathematical* objects such as numbers and *mathematical* functions. In our case, the *domain of denotation* is Haskell integers and Haskell functions.

How does the CBV/CBN distinction manifest itself in the denotational semantics?

To quote from [3], denotational semantics has three basic principles:

1. Every syntactic phrase in a program has a meaning, or denotation.
2. Denotations are well-defined mathematical objects, such as mathematical functions (often higher-order functions).
3. The meaning of a compound syntactic phrase is a mathematical combination of the meanings of its immediate subphrases.

The last assumption is often called compositionality or, according to Stoy, the denotational assumption.

Let us now see how we implement this semantics in Haskell.

2 Untyped λ-calculus embedded in Haskell

Code file: `EvalN.hs`.

We must decide first how to represent terms of the object language in Haskell, our metalanguage. We represent variables as `String`s, integers as `Int`s, and object terms as the values of the following data type:

```
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`
```

Here in a sample term:

```
  (L "x" (IFZ (V "x") (I 1) ((V "x") :+ (I 2)))) `A` (I 10)
```

In denotational approach, giving meaning to terms is relating them to some other objects. In our case, these 'other objects' are Haskell values, elements of the following data type:

```
data Value = VI Int | VC (Value -> Value)
instance Show Value where
    show (VI n) = "VI " ++ show n
    show (VC _) = "<function>"
```

The necessity of environments: to give a meaning to `L "x" (V "x")` in terms of its sub-terms' meanings, we should give a meaning to `(V "x")`. But what is the meaning of a free variable?

An environment is a finite map from variables to their meanings, from `VarName` to `Value`:

```
type Env = ...

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

Here is one implementation of the environment:

```
type Env = [(VarName, Value)]

env0 = []

lkup env x = maybe err id $ lookup x env
 where err = error $ "Unbound variable " ++ x

ext env xt = xt : env
```

Other choices: functions, `Data.Map`.

Denotational semantics relates a `Term` to a `Value` in the environment specifying the meaning of free variables in the `Term`. In Haskell, the denotational semantics of the object language is given as the following function.

```
eval :: Env -> Term -> Value
eval env (V x)   = lkup env x
eval env (L x e) = VC (\v -> eval (ext env (x,v)) e)
eval env (A e1 e2) =
    let v1 = eval env e1
        v2 = eval env e2
    in case v1 of
       VC f  -> f v2
       v     -> error $
                "Trying to apply a non-function: " ++ show v
eval env (I n) = VI n
eval env (e1 :+ e2) =
    let v1 = eval env e1
        v2 = eval env e2
    in case (v1,v2) of
       (VI n1, VI n2) -> VI (n1+n2)
       vs     -> error $
                 "Trying to add non-integers: " ++ show vs
eval env (IFZ e1 e2 e3) =
    let v1 = eval env e1
    in case v1 of
```

```
       VI 0 -> eval env e2
       VI _ -> eval env e3
       v    -> error $
                "Trying to compare a non-integer to 0: " ++ show v
```

Our sample term, in the empty environment, evaluates to `VI 12`:

```
eval env0 (L "x" (IFZ (V "x") (I 1) ((V "x") :+ (I 2)))
          `A` (I 10))
```

Questions about `eval`:

- Is `eval` a partial function? Partiality as divergence (we see later) or as getting stuck? What 'getting stuck' means in our `eval`?
- Why `eval` expresses denotational semantics? Is `eval` compositional?
- Does `eval` correspond to CBN or CBV? Or call-by-need? Note the `A`-case, the type of the environment, the fact the code is in Haskell. Can we write a test that gives the conclusive answer?
- Can we see the correspondence with β, η and δ-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 α-conversion?
- Why can we say that we evaluate `(L x e)` to a *closure*? What exactly is it closed over?

Taking advantage of the metalanguage:

```
(vx,vy) = (V "x", V "y")
term1 = L "x" (IFZ vx (I 1) (vx :+ (I 2)))
```

We use `vx` as a *shortcut* for `V "x"`, taking advantage of Haskell's facility to *name* expressions, so that complex expressions can be succinctly referred to by their names. We see the benefit of defining a language by embedding it in a metalanguage (as an embedded DSL): we can take great advantage of the metalanguage facilities such as definitions, module system, etc.

More examples of terms and their meanings:

```
test11 = eval env0 term1
test12 = eval env0 (term1 `A` (I 2))    -- VI 4
test13 = eval env0 (term1 `A` (I 0))    -- VI 1
test14 = eval env0 (term1 `A` vx)       -- *** Exception:
                                        --     Unbound variable x
term2 = L "x" (L "y" (vx :+ vy))
test21 = eval env0 term2
test22 = eval env0 (term2 `A` (I 1))
test23 = eval env0 (term2 `A` (I 1) `A` (I 2)) -- VI 3
```

We observe that not all terms have meanings: `eval` is indeed partial. Here are a few more problematic terms. Some problems are 'hidden', showing up only in some 'contexts'.

```
term3 = L "x" (IFZ vx (I 1) vy)
test31 = eval env0 term3
test32 = eval env0 (term3 ‘A‘ (I 0))    -- VI 1
test33 = eval env0 (term3 ‘A‘ (I 1))    -- *** Exception:
                                        --     Unbound variable y
term4 = L "x" (IFZ vx (I 1) (vx ‘A‘ (I 1)))
test41 = eval env0 term4
test42 = eval env0 (term4 ‘A‘ (I 0))    -- VI 1
test43 = eval env0 (term4 ‘A‘ (I 1))    -- applying a non-function

term6  = (L "x" (I 1)) ‘A‘ vy
test61 = eval env0 term6
```

What does the result of `term6` tell us about our language being CBV or CBN?

Let's define more interesting terms: e.g., multiplication, using the familiar equation $(x+1)y = xy + y$. Here is the first try:

```
tmul1 = L "x" (L "y"
          (IFZ vx (I 0)
            ((tmul1 ‘A‘ (vx :+ (I (-1))) ‘A‘ vy) :+ vy)))
testm1 = eval env0 (tmul1 ‘A‘ (I 2) ‘A‘ (I 3))
```

It does work. Is it, however, cheating and taking advantage of the metalanguage too far? Try showing the term. We observe that infinitary terms may be finitely represented.

The honest solution uses the object language itself for non-trivial operations such as recursion. We rely on the Y combinator, `termY`, which has the property `termY f x` being equal to `f (termY f) x`. This property suggests the implementation:

```
termY = L "f"
        (delta ‘A‘ L "y" (L "x" (vf ‘A‘ (delta ‘A‘ vy) ‘A‘ vx)))
 where delta = L "y" (vy ‘A‘ vy)
       vf = V "f"

tmul = termY ‘A‘ (L "self" (L "x" (L "y"
          (IFZ vx (I 0)
            (((V "self") ‘A‘ (vx :+ (I (-1))) ‘A‘ vy) :+ vy)))))

testm2 = eval env0 (tmul ‘A‘ (I 2) ‘A‘ (I 3)) -- VI 6
```

Now we can observe divergence: try multiplying (-1) by (-1).

Exercises:

- add the recursion operator as a primitive in the object language:
  `data Term = ... | Fix Term` (needed for the next chapter)

- 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 and for values. For more extra credit: use the pretty-printing library.

Next: adding types.

3 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')

Russell's view is most unifying: On one hand, it suggests that a type denotes a collection of objects. On the other hand, Russell introduced types specifically as a *syntactic restriction* on logical formulas, to rule out formulas expressing Russell's paradox and formulas exhibiting circularity. Russell came to his view by considering universally quantified propositions like $\forall n.(\mathrm{PerfectNumber}(n) \rightarrow \mathrm{Even}(n))$. Although formally n ranges over the whole universe, it really does not make sense to evaluate the *propositional function*, or predicate – the formula in the parentheses parameterized by n – when n is anything but a natural number. When n is a color or a proposition, $\mathrm{PerfectNumber}(n)$ is not just true or false – it is meaningless. Likewise, the proposition $p(0) \wedge (\forall x.\, p(x) \rightarrow p(s(x))) \rightarrow \forall x.\, p(x)$ makes sense only when p is a predicate over natural numbers. Thus each propositional function makes sense only when its argument is within a particular 'subuniverse' – the *range of significance*. We stress that we can tell these ranges of significance just by the (syntactic) look of the formula: in the first proposition, we see that n must be a natural number. Although we can tell, by the look of it, whether or when a proposition 'makes sense,' we cannot likewise tell if the proposition is true or false – the latter requires evaluation. In fact, whether the first proposition holds is not known to this day.

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

Types are usually quite imprecise approximations: if an expression has the type `Int`, we are sure that if the evaluation of the expression terminates, it will yield some particular integer. Just by looking at the form of the expression (using the simple type system of this course), we cannot tell which integer. Types are useful usually not in what they tell but in what they do not tell. For example, if an expression has the type `Int`, we know that evaluating the expression *never* yields a function (given *any* dynamic inputs). Furthermore, in the type system of this lecture, if an expression has a type, its evaluation will never get stuck. Types approximate the dynamic behavior, abstracting away many details. If the type system is sound, the specific concrete behavior is certainly within the approximation. Thus if the approximation does not include 'being stuck' behavior, then *any* concrete behavior won't get stuck.

4 Type reconstruction in Church style

Code file: `TEvalNC.hs`.

In this course, we will be concerned with simple types:

```
data Typ = TInt | Typ :> Typ
infixr 9 :>
```

The type `TInt` is an abstraction over a set of integers; the type `TInt :> TInt` is an abstraction over a set of integer-valued functions on integers. One concrete member of that set is 'increment'. What are some concrete members of

- `TInt :> TInt :> TInt`,
- `(TInt :> TInt) :> TInt` and
- `TInt :> (TInt :> TInt)`?

In Church's original typed λ-calculus, *all* terms and variables were annotated with their types. If we don't know to which objects a predicate may meaningfully apply, then what is the purpose of writing this predicate. That is a reasonable point, leading, however, to a very heavy notation: $(\lambda x{:}\mathrm{int}\to\mathrm{int}.(x{:}\,\mathrm{int}\to\mathrm{int})(1{:}\,\mathrm{int})){:}(\mathrm{int}\to\mathrm{int})\to\mathrm{int}$

The grammar of terms (formation rules) in the original Church calculus were all typed, so we can determine the type of a composite expression from the type of its components. This determination is a *deduction*, which permits us (originally, Girard and Reynolds) to lighten the notation. We should only annotate binders with types, and *deduce* the annotations on all other subterms and on the whole term. If we succeed in determining the type of the term, we declare it well-typed. One may prove that evaluation of a well-typed term does not get stuck, and, if terminates, gives a value that is a concrete instance of the deduced type.

The topic of this part therefore is this deduction of the type of the term.

Type checking, type inference, type reconstruction These terms are used in specific and more general meanings, and so may be hard to distinguish. Suppose, like in the original Church λ-calculus, we are given a term where everything – every bound variable, every subterm, and the term itself – is annotated with types. The problem is then to check if the term is well-typed, i.e., if it indeed has the type given by the annotation. In this case we merely check the consistency of the annotations. This is the problem of *type checking* in the strict sense. This problem arises for example when type-checking proof terms of Coq or type-checking programs in the intermediate language of a compiler, to assure the correctness of compilation (GHC core and core-lint).

Normally we are given a term with some type annotations missing. For example, only bound variables are annotated with types; the type for the whole term is not given. Again we want to check if the term is well-typed, and if so, recover the missing annotations. This problem is generally called *type inference*; logically, it is a decision procedure for axiomatic theory. Some [4, Section 11] still call this problem type checking when the well-typedness check and the accompanying recovery of the missing annotations is straightforward, with no 'guessing' (hypothetical reasoning) required. This is the case of the present section.

As we said, binding occurrences of variables must be annotated with types. So, we have to change the syntax of our object language, however slightly: only `L`-terms are changed.

```
data Term = ...
          | L VarName Typ Term
          | ...
```

We change the type of the environment, which previously (during evaluation, file `EvalN.hs`) associated variables with their meanings, Values: `type Env = [(VarName, Value)]`. In type checking, or *type evaluation*, the environment still associates variables with their meanings, which are now types:

```
type TEnv = [(VarName, Typ)]
```

Types, being approximations of values, are also meanings. The environment manipulation functions – `lkup`, `ext`, `env0` – remain the same, as expected.

Here is our function `teval`, which compared to `eval` gives a different denotation of a term. Now we map terms not to values but to approximations of values, in other words, types. The signature of `teval` reflects this difference:

```
teval :: TEnv -> Term -> Typ
teval env (V x) = lkup env x
teval env (L x t e) = t :> teval (ext env (x,t)) e
teval env (A e1 e2) =
    let t1 = teval env e1
        t2 = teval env e2
    in case t1 of
       t1a :> t1r | t1a == t2 -> t1r
       t1a :> t1r -> error $
                     unwords ["Applying a function of arg type",
                              show t1a, "to argument of type",
                              show t2]
       t1 -> error $ "Trying to apply a non-function: " ++ show t1
teval env (I n) = TInt
teval env (e1 :+ e2) =
    let t1 = teval env e1
        t2 = teval env e2
    in case (t1,t2) of
       (TInt, TInt) -> TInt
       ts -> error $ "Trying to add non-integers: " ++ show ts
teval env (IFZ e1 e2 e3) =
    let t1 = teval env e1
        t2 = teval env e2
        t3 = teval env e3
    in case t1 of
       TInt | t2 == t3 -> t2
       TInt -> error $
               unwords ["Branches of IFZ have different types:",
                        show t2, "and", show t3]
       t -> error $
            "Trying to compare a non-integer to 0: " ++ show t
```

Although the code of `teval` has the same form as that of `eval`, there are differences. The `I`-case in `teval` makes it clear that we are computing approximations: `teval env (I n)` returns `TInt` for any `n`, abstracting away the available concrete information. We thus call `teval` an *abstract* interpreter. In the case of `IFZ e1 e2 e3`, we no longer know the concrete value of `e1`, and so cannot choose the branch. We thus evaluate both branches of the conditional and make sure they have the same type. Only then we can statically assure that no matter how the conditional is evaluated by `eval`, the result will have the same type. In stark contrast with `eval`, the lambda-case of `teval` evaluates under lambda. Since we know the type of the bound variable from the type annotation, we can determine the type of the function's body.

We may regard this code as an executable *specification* of the type reconstruction algorithm. Let us follow the specification and manually process the term $\lambda x{:}\mathrm{int}\to\mathrm{int}.\, \lambda y{:}\mathrm{int}.(xy) + (y+1)$.

Is `teval` still partial and if so, what causes the partiality: divergence, or getting stuck, or both?

We can try to type-check the terms we have evaluated earlier.

```
term1 = L "x" TInt (IFZ vx (I 1) (vx :+ (I 2)))

test1 = teval env0 term1 -- TInt :> TInt

term2a = L "x" TInt (L "y" TInt (vx ‘A‘ vy))
test2a = teval env0 term2a

term2b = L "x" (TInt :> TInt) (L "y" TInt (vx ‘A‘ vy))
test2b = teval env0 term2b -- (TInt :> TInt) :> (TInt :> TInt)
```

Can we give different type annotations to `term2b` and would it still type-check?

Type-checking problematic code:

```
term3 = L "x" TInt (IFZ vx (I 1) vy)
test3 = teval env0 term3

term4a = L "x" TInt (IFZ vx (I 1) (vx ‘A‘ (I 1)))
test4a = teval env0 term4a

term4b = L "x" (TInt :> TInt) (IFZ vx (I 1) (vx ‘A‘ (I 1)))
test4b = teval env0 term4b

term6  = (L "x" TInt (I 1)) ‘A‘ vy
test61 = teval env0 term6
```

These terms hid evaluation problems, which are now apparent. Types, as static approximations of program behavior, give us assurance for *all* possible program inputs. We type-check under lambda, and type-check both branches of conditionals, so problems like unbound variables cannot hide any more.

We emphasize the similarity between `eval` and `teval`, and recall the notion of soundness of a type system:

- 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).

The fact the type checking relation is so similar an approximation of the evaluation relation makes soundness (or unsoundness) easier to see and grasp.

Can divergence still occur?

```
tmul1 = L "x" TInt (L "y" TInt
          (IFZ vx (I 0)
            ((tmul1 ‘A‘ (vx :+ (I (-1))) ‘A‘ vy) :+ vy)))
testm1 = teval env0 tmul1 -- is type-checking really decidable?
```

Is type-checking really decidable? Yes, if terms are finite. Why `eval` of `testm1` succeeded but `teval` diverged? Because `teval` abstractly interprets all branches, it has to check the whole term, whereas `eval` does not evaluate the whole term.

So, our cheating with recursion no longer works; to write recursive terms, we have to use the fixpoint operation in the object language. Let us type-check that operation, `termY`, used earlier. We start by type-checking the 'core' subterm of `termY`, the term `delta`, or $\lambda x.\, xx$. How do we write `delta` with type annotations?

```
delta = L "y" (TInt :> TInt) (vy `A` vy)
testd = teval env0 delta
```

Solution: introduce `Fix` as a primitive of the language

```
data Term = ...
          | Fix Term    -- fix f, where f : (a->b)->(a->b)
```

and specify how to type-check it (add a clause to `teval`):

```
teval env (Fix e) =
    let t = teval env e
    in case t of
       (ta1 :> tb1) :> (ta2 :> tb2) | ta1 == ta2 && tb1 == tb2
         -> ta1 :> tb1
       t -> error $ "Inappropriate type in Fix: " ++ show t
```

Example of using and type-checking Fix:

```
tmul = Fix (L "self" (TInt :> TInt :> TInt) (L "x" TInt (L "y" TInt
          (IFZ vx (I 0)
            (((V "self") `A` (vx :+ (I (-1))) `A` vy) :+ vy)))))

testm21 = teval env0 tmul                    -- TInt :> TInt :> TInt
testm22 = teval env0 (tmul `A` (I 2))                -- TInt :> TInt
testm23 = teval env0 (tmul `A` (I 2) `A` (I 3))              -- TInt
testm24 = teval env0 (tmul `A` (I (-1)) `A` (I (-1)))   -- TInt
```

Is `teval` still terminating? What is the corresponding `eval` rule? Why is `eval` with `Fix` not terminating in general but `teval` always terminating?

The type checking of `Fix` illustrates an important point: we want type checking to be decidable and finish quickly, even if evaluation does not terminate. We observe that `tmul` and `testm24` type-check, even though evaluating the term in `testm24` diverges.

Types are quickly decidable static approximation of dynamic behavior.

Does it matter for `teval` if the dynamic semantics is CBV vs CBN?

Bonus exercise: convert `teval` to be a total function (with the result type `Either String Typ`. One could use the error monad.

Bonus code file `TEvalNR.hs` – reporting the reconstructed types for all subterms of a term. So error messages can be made better: even if the full type-checking failed, we can still show what types have been found for subterms before the error has occurred. The bonus exercise can be reformulated so that the result type of `teval` is `Either (ErrMsg,Typs) Typs`. Very big bonus exercise: show the type of each subterm in an Emacs buffer; see the Tuareg Emacs mode for OCaml and the subterm typing feature of OCaml (`ocamlc -dtypes`: Save type information in `filename.annot`).

Next we discuss how to get by without the annotations on the bound variables: how to infer them too.

5 Type inference in Curry style

Code file: `TInfT.hs`.

Haskell B. Curry thought that terms may make sense without types; one should be permitted to evaluate a term even if we don't have any idea of the term's type. We get some value at the end (if we are lucky).

But we can use types to impose additional restrictions on terms. In return, we obtain some guarantees – such as an approximation of the term's evaluation result and, mainly, the assurance of not getting stuck. Incidentally, Curry's view permits multiple type systems for the same language (with various trade-offs of what the type system rejects vs. what it assures).

Types are well-formedness constraints on terms.

People complain that the type checker rejects their seemingly correct program. Keep in mind that the whole purpose of types is to restrict the set of programs; the whole value of type system is in what programs they *reject.* Analogy with security: the whole purpose of security is to create inconvenience.

The problem of finding a type for a term within a given type system is called type inference problem (Cardelli, 'Type systems'). One can pose this problem for a Church-style calculus (which includes type annotations) or a Curry-style calculus. Formally, given the type environment Γ and a term e, find if there is a type t so that the judgment $\Gamma \vdash e : t$ is valid (alternatively, Γ can be deduced rather than given, see `TInfTEnv.hs`). In the case when no annotations are given (i.e., we start with an 'untyped' term), the problem of inferring all annotations including the type of the term is sometimes called *type reconstruction.* Some [4] call this (proper) type inference.

5.1 Logic variables and their unification

To build intuition, we handle two terms by hand. We start with the term we did by hand earlier, after erasing the type annotations: $\lambda x.\, \lambda y. (xy) + (y + 1)$. Can we still reconstruct the type of the term (and recover the annotations along the way)? We use the same type deduction algorithm as before, but interpret it a bit differently, using *logic variables.* In elementary school, we called them just 'variables'. Sample problem: "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?" We use x as an indeterminate number, derive another indeterminate number $x - 5$, and solve the equality constraint $x - 5 = x/2$ to infer that $x = 10$. We infer the type of our term just as we solve the elementary-school problem.

Once we finish the processing of the term, the blackboard should contain the equations:

```
x:t1  y:t2
      t1=t3->t4  t3=t2  t2=Int  t4=Int
```

We then determine that `x` should have the type `Int->Int` and `y` has the type `Int`. We got the same result as before and recovered the annotations on the bound variables. In our manual type inference, we had to: (i) introduce fresh type variables; (ii) keep track of the equations relating two types; and (iii) chase the binding of variables when finally writing down the types.

Let us consider one more example: $\lambda x.\,\lambda y.\,\mathrm{ifz}(xy)x(\lambda z.\,z)$.

```
x:t1  y:t2                    (z:t5)
      t1=t3->t4  t3=t2  t4=Int     t1=t5->t5
```

Now we have to solve the equations: notice two equations for `t1`. Solving for `t1` produces more equations (for `t3` and `t4`), which we have to solve again. This last step illustrates that solving the equations requires *unification*.

The code below implements our hand-written process with small optimizations. We wish to keep our equations in *solved* form all the time; that is, each equation should be in the form `tvar = type` (such as `t1=t3->t4` but not `t5->t5=t3->t4`), and there should not be multiple equations for the same type variable. In the last example, we do not add `t1=t5->t5`. We should solve it right away (do it by hand). Thus we unify types incrementally. A set of equations in solved form is called a *substitution*. In other words, a substitution consists of equations associating type variables with types.

Let us now write the code implementing our manual process. First, we change back the syntax of our terms:

```
data Term = ...
          | L VarName Term
          | ...
```

There are no longer any types in terms. Second, we modify the syntax of types to account for type variables:

```
data Typ = TInt | Typ :> Typ | TV TVarName
type TVarName = Int
```

We label type variables with integers, as when inferring types by hand.

We need to generate fresh type variables and to store the substitution. We use the following data structure to do both. The Haskell type `M.IntMap Typ` stores finite maps from `Int` to `Typ`, with library operations such as `M.empty`, `M.lookup` and `M.insert`.

```
data TVE = TVE Int (M.IntMap Typ)
```

We obviously need operations to generate a yet-unused type variable (by incrementing the first component of `TVE`), to create an empty substitution, to look up the equation for a given type variable (if any), and to add a new equation.

```
newtv :: TVE -> (Typ,TVE)
newtv (TVE n s) = (TV n, TVE (succ n) s)
```

```
tve0 :: TVE
tve0 = TVE 0 M.empty

tvlkup :: TVE -> TVarName -> Maybe Typ
tvlkup (TVE _ s) v = M.lookup v s

tvext :: TVE -> (TVarName,Typ) -> TVE
tvext (TVE c s) (tv,t) = TVE c (M.insert tv t s)
```

We also need the operation to chase through the equations, as we did by hand:

```
tvsub :: TVE -> Typ -> Typ
tvsub tve (t1 :> t2) = tvsub tve t1 :> tvsub tve t2
tvsub tve (TV v) | Just t <- tvlkup tve v = tvsub tve t
tvsub tve t = t
```

The second-to-last line above uses *pattern guards*, a form of syntactic sugar that GHC adds to Haskell.

Finally, we need the operation to solve the equations. That is, given the existing substitution `tve` and two types `t1` and `t2`, we wish to solve `t1=t2`, adding new solved equations to `tve`, if any. The solution may of course fail; for example, the equation `Int=t1->t2` has no solutions. Hence the type of `unify`.

```
unify :: Typ -> Typ -> TVE -> Either String TVE
unify t1 t2 tve = unify' (tvchase tve t1) (tvchase tve t2) tve

-- chase through a substitution ‘shallowly’:
-- stop at the last equivalent type variable
tvchase :: TVE -> Typ -> Typ
tvchase tve (TV v) | Just t <- tvlkup tve v = tvchase tve t
tvchase _ t = t
```

In the case of no solutions, we return a string describing the problem. We start the unification by a shallow chase.

```
-- If either t1 or t2 are type variables, they must be unbound
unify' :: Typ -> Typ -> TVE -> Either String TVE
unify' TInt TInt = Right
unify' (t1a :> t1r) (t2a :> t2r) = either Left (unify t1r t2r)
                                   . unify t1a t2a
unify' (TV v1) t2 = unifyv v1 t2
unify' t1 (TV v2) = unifyv v2 t1
unify' t1 t2 = const (Left $ unwords ["constant mismatch:",
                                      show t1, "and", show t2])

-- Unify a free variable v1 with t2
unifyv :: TVarName -> Typ -> TVE -> Either String TVE
unifyv v1 (TV v2) tve =
    if v1 == v2 then Right tve
       else Right (tvext tve (v1,TV v2)) -- record new constraint
```

```
unifyv v1 t2 tve = if occurs v1 t2 tve
                      then Left $ unwords ["occurs check:",
                                           show (TV v1), "in",
                                           show (tvsub tve t2)]
                      else Right (tvext tve (v1,t2))

-- The occurs check: if v appears free in t
occurs :: TVarName -> Typ -> TVE -> Bool
occurs v TInt _ = False
occurs v (t1 :> t2) tve = occurs v t1 tve || occurs v t2 tve
occurs v (TV v2) tve =
    case tvlkup tve v2 of
         Nothing -> v == v2
         Just t  -> occurs v t tve
```

The algorithm is basically the same we used in our manual derivations. Only two aspects are new: First, the equation `t1=t1` is trivially true, so we do not add it to our set of solved equations. Second, the equation `t1=t1->Int` has no solutions: a type cannot contain itself; hence the occurs check.

5.2 Type inference

We modify the type deduction algorithm in the previous section to account for type variables. We split `teval` into two functions: `teval'` does all the work, and `teval` does the final chase – as we did on the blackboard. The type of `teval'` shows that we have to thread `TVE` all the way through. `TVE` is the 'blackboard' with the equations.

```
teval' :: TEnv -> Term -> (TVE -> (Typ,TVE))
teval' env (V x)   = \tve0 -> (lkup env x, tve0)
teval' env (L x e) = \tve0 ->
    let (tv,tve1) = newtv tve0
        (te,tve2) = teval' (ext env (x,tv)) e tve1
    in (tv :> te,tve2)
teval' env (A e1 e2) = \tve0 ->
    let (t1,tve1) = teval' env e1 tve0
        (t2,tve2) = teval' env e2 tve1
        (t1r,tve3)= newtv tve2
    in case unify t1 (t2 :> t1r) tve3 of
       Right tve -> (t1r,tve)
       Left err  -> error err
teval' env (I n) = \tve0 -> (TInt,tve0)
teval' env (e1 :+ e2) = \tve0 ->
    let (t1,tve1) = teval' env e1 tve0
        (t2,tve2) = teval' env e2 tve1
    in case either Left (unify t2 TInt) . unify t1 TInt $ tve2 of
       Right tve -> (TInt,tve)
       Left err  -> error $ "Trying to add non-integers: " ++ err
teval' env (IFZ e1 e2 e3) = \tve0 ->
    let (t1,tve1) = teval' env e1 tve0
        (t2,tve2) = teval' env e2 tve1
        (t3,tve3) = teval' env e3 tve2
    in case unify t1 TInt tve3 of
       Right tve -> case unify t2 t3 tve of
        Right tve -> (t2,tve)
        Left err -> error $ unwords ["Branches of IFZ have",
                                     "different types.",
                                     "Unification failed:", err]
       Left err -> error $
                   "Trying to compare a non-integer to 0: " ++ err
teval' env (Fix e) = \tve0 ->
    let (t,tve1)  = teval' env e tve0
        (ta,tve2) = newtv tve1
        (tb,tve3) = newtv tve2
    in case unify t ((ta :> tb) :> (ta :> tb)) tve3 of
       Right tve -> (ta :> tb,tve)
       Left err  -> error ("Inappropriate type in Fix: " ++ err)

-- Resolve all type variables, by chasing as far as possible
teval :: TEnv -> Term -> Typ
teval tenv e = let (t,tve) = teval' tenv e tve0 in tvsub tve t
```

The code is basically the same `teval` in `TEvalNC.hs` – so our new code specifies the same type system. There are a few small but interesting differences, especially in the cases of application and of `Fix`. We replaced pattern matching with unification, so that the specification become more ‘declarative’.

Logic variables thus bring a form of *hypothetical reasoning.* The code clearly has the flavor of constraint programming: generating, propagating and solving constraints. Unification is the resolution (pun intended) of equality constraints.

Let us run a few old tests:

```
test0 = teval' env0 ((L "x" (vx :+ (I 2))) `A` (I 1)) tve0
-- (TV 1,TVE 2 (fromList [(0,TInt),(1,TInt)]))

term1 = L "x" (IFZ vx (I 1) (vx :+ (I 2)))
test10 = teval' env0 term1 tve0
-- (TV 0 :> TInt,TVE 1 (fromList [(0,TInt)]))
```

When using `teval'` we see all the accumulated equations, our ‘blackboard’. If we want to see only the final result, the deduced type with all type variables chased through, we use the function `teval`:

```
test1 = teval env0 term1 -- TInt :> TInt
```

The result is the same as before. The other tests, such as

```
tmul = Fix (L "self" (L "x" (L "y"
          (IFZ vx (I 0)
            (((V "self") `A` (vx :+ (I (-1))) `A` vy) :+ vy)))))
testm21 = teval env0 tmul               -- TInt :> TInt :> TInt
testm22 = teval env0 (tmul `A` (I 2))           -- TInt :> TInt
testm23 = teval env0 (tmul `A` (I 2) `A` (I 3))         -- TInt
testm24 = teval env0 (tmul `A` (I (-1)) `A` (I (-1)))   -- TInt
```

also give the same results. The terms (such as `tmul`) no longer have any type annotations however.

Terms that could not be type-checked before still fail the type-checking, as expected. There is an interesting variation: in `TEvalNC.hs` we had these two terms

```
term4a = L "x" TInt (IFZ vx (I 1) (vx `A` (I 1)))
test4a = teval env0 term4a
-- *** Exception: Trying to apply a non-function: TInt

term4b = L "x" (TInt :> TInt) (IFZ vx (I 1) (vx `A` (I 1)))
test4b = teval env0 term4b
-- *** Exception: Trying to compare a non-integer to 0:
--     TInt :> TInt
```

which are essentially the same term but with different type annotations. Type-checking of both terms failed, with the above error messages. Perhaps the term is 'good' though, we merely have to find the right type annotation? Now we write the term with no annotations:

```
term4 = L "x" (IFZ vx (I 1) (vx `A` (I 1)))
test4 = teval env0 term4
-- *** Exception: Trying to compare a non-integer to 0:
--     constant mismatch: TInt :> TV 1 and TInt
```

The error message is differently formulated; we can see right away it is produced by the unifier. The failure to type-check `term4` now tells us more: no matter how we may annotate the bound variables in `term4`, it is still an ill-typed term. A clearer example is the type-checking of the `delta` term. In `TEvalNC.hs`, we had to write type annotations on the bound variable in `delta`, but we could not find the right annotation. Perhaps we should have tried harder? Now we write `delta` as it was given in `EvalN.hs`, with no need to think up annotations:

```
delta = L "y" (vy `A` vy)
testd = teval env0 delta
-- *** Exception: occurs check: TV 0 in TV 0 :> TV 1
```

And it still fails. So we blame the term rather than faulty annotations. The typing of `delta` fails due to the occurs check. Historically, Russell and Church introduced types to rule out terms like `delta` and circular definitions (which often don't actually define anything and lead to paradoxes).

In TEvalNC.hs, we had one more sample term, term2a. Let us type-check it, with no annotations on bound variables.

```
term2a = L "x" (L "y" (vx ‘A‘ vy))
test2a = teval env0 term2a
-- (TV 1 :> TV 2) :> (TV 1 :> TV 2)
```

It is a bit odd to see type variables in the inferred type. Here is a simpler example:

```
termid = L "x" vx
testid = teval env0 termid -- TV 0 :> TV 0
```

The inferred type has type variables. What does that mean? We can guess from the following term

```
term2id = L "f" (L "y" ((I 2) :+
            ((termid ‘A‘ (V "f")) ‘A‘ ((termid ‘A‘ vy) :+ (I 1)))))
test2id = teval env0 term2id -- (TInt :> TInt) :> (TInt :> TInt)
```

which contains two occurrences of termid. We again rely on the metalanguage, to name the term L "x" vx as termid and then refer to it by the short name. This ‘referring to’ is *sharing* – the two occurrences of the name termid refer to the same term. Do the shared terms have the same inferred type? The two occurrences of termid are both applications, but to terms of different types. Work out which types are those. So, what do the type variables in the inferred type of termid tell us about the use of termid? We have just encountered top-level *parametric polymorphism.*

Here is a more elaborate example of using binding in the metalanguage to express sharing of object terms.

```
termlet = let c2  = L "f" (L "x" (V "f" ‘A‘ (V "f" ‘A‘ vx)))
              inc = L "x" (vx :+ (I 1))
              compose = L "f" (L "g" (L "x"
                          (V "f" ‘A‘ (V "g" ‘A‘ vx))))
          in c2 ‘A‘ (compose ‘A‘ inc ‘A‘ inc) ‘A‘ (I 10) :+
             ((c2 ‘A‘ (compose ‘A‘ inc) ‘A‘ termid) ‘A‘ (I 100))
testlet = teval env0 termlet
```

Exercise: what is the value of termlet? Why is c2 so named?

6 Sharing and polymorphism

We just saw how to express sharing and polymorphism by naming object terms in the metalanguage. This chapter extends our object language to express sharing in itself. Would it work at all with types? After all, although shared *untyped* terms are identical, the corresponding typed terms are not the same.

6.1 Implicit type variable environment: State monad

Code files: `TInfTM.hs`, `TInfLetI.hs`.

We said before that the new `teval` in `TInfT.hs` specifies the type system more declaratively than in `TEvalNC.hs`. One may retort that the new code is messier, due to `tve1`, `tve2`, `tve3` in the `let` forms. One can easily make a numbering mistake and use `tve2` where `tve3` is called for (which I did).

Let us simplify the notation by applying a few algebraic transformations to the code. We start with the `A`-case in `teval'`:

```
teval' env (A e1 e2) = \tve0 ->
    let (t1,tve1) = teval' env e1 tve0
        (t2,tve2) = teval' env e2 tve1
        (t1r,tve3)= newtv tve2
    in case unify t1 (t2 :> t1r) tve3 of
       Right tve -> (t1r,tve)
       Left err  -> error err
```

The body of the definition has the form

```
\tve0 -> let (t1,tve1) = teval' env e1 tve0 in <body>
```

where `<body>` may refer to `t1` and `tve1` but not `tve0`. That is the essence of single-threading of `tve`: `tve0` is 'consumed' by `teval' env e1 tve0` and may no longer be used. To highlight that fact, we re-write the expression in a more explicit form

```
\tve0 -> let (t1,tve1) = teval' env e1 tve0 in E t1 tve1
```

where `E` is an expression where none of `t1`, `tve1`, and `tve0` occur free. That fact enables a β-expansion:

```
(\f -> \tve0 -> let (t1,tve1) = teval' env e1 tve0 in f t1 tve1) E
```

Likewise `teval' env e1` obviously does not have free occurrences of `t1`, `tve0`, and `tve1`. We may β-expand further:

```
(\m -> \f -> \tve0 -> let (t1,tve1) = m tve0 in f t1 tve1)
  (teval' env e1) E
```

The expression in parenthesis is a pure combinator; we may as well give it a name, for example:

```
m >>= f = \tve0 -> let (t1,tve1) = m tve0 in f t1 tve1
```

Let us look into the type of `(>>=)`. From the type annotation to `teval'` we know that `teval' env e1 :: (TVE -> (Typ,TVE))`; `E` as the representation of the body of the `let` form with `t1` and `tve` explicitly abstracted should have the type `Typ -> (TVE -> (Typ,TVE))`. Let us define a convenient type abbreviation

```
type TVEM a = TVE -> (a,TVE)
```

We could then assign to `(>>=)` the type `TVEM Typ -> (Typ -> TVEM Typ) -> TVEM Typ`. However, the Haskell type checker infers a more general type for the `(>>=)` expression

```
(>>=) :: TVEM a -> (a -> TVEM b) -> TVEM b
```

which we shall adopt.

After our re-writing of the first `let`-form in the `A` clause, we obtain:

```
teval' env (A e1 e2) =
    (teval' env e1) >>= (\t1 tve1 ->
    let (t2,tve2) = teval' env e2 tve1
        (t1r,tve3)= newtv tve2
    in case unify t1 (t2 :> t1r) tve3 of
       Right tve -> (t1r,tve)
       Left err  -> error err)
```

Re-writing the other `let`-forms in the same spirit gives us

```
teval' env (A e1 e2) =
    teval' env e1 >>= (\t1 ->
    teval' env e2 >>= (\t2 ->
    newtv         >>= (\t1r tve3 ->
    case unify t1 (t2 :> t1r) tve3 of
       Right tve -> (t1r,tve)
       Left err  -> error err)))
```

We observe that unification operation does update `tve` but does not produce any 'direct' result. We reflect this fact by introducing several β, η expansions:

```
teval' env (A e1 e2) =
    teval' env e1 >>= (\t1 ->
    teval' env e2 >>= (\t2 ->
    newtv         >>= (\t1r tve3 ->
    case unify t1 (t2 :> t1r) tve3 of
       Right tve -> ((\tve -> ((),tve)) >>= (\_ tve -> (t1r,tve))) tve
       Left err  -> error err)))
```

which we then distribute over the case statement. After introducing an auxiliary function

```
unifyM t1 t2 = \tve3 ->
      case unify t1 t2 tve3 of
       Right tve -> ((),tve)
       Left err  -> error err
```

noting that `(\t1r tve -> (t1r,tve))` is a useful combinator deserving a name

```
return :: a -> TVEM a
return v = \tve3 -> (v,tve3)
```

we finally obtain

```
teval' env (A e1 e2) =
    teval' env e1 >>= (\t1 ->
    teval' env e2 >>= (\t2 ->
    newtv         >>= (\t1r->
    unifyM t1 (t2 :> t1r) >>= (\_ ->
       return t1r))))
```

There are no longer any traces of `tve`: the threading of `tve` throughout the computation is hidden in the combinator `(>>=)`. There is no longer need to introduce `tve1`, `tve2`, etc., there is no longer a danger of mis-numbering them. We stress that the new form of the `A`-clause for `teval'` is the result of algebraic transformations, enabled by the facts that `tve0` does not occur free in the body of the first `let`, `tve1` is not free in the body of the second `let`, etc – in short, `tve` is used single-threadedly, *linearly*. The transformations are meaning-preserving: in `TInfTM.hs`, we re-wrote only the `A`-clause of `teval'` and left the other clauses as they were in `TInfT.hs`. The new code type checks and works as before.

The rest of `teval'` clauses can be re-written in a similar fashion. One cannot help but notice that

```
teval' env (A e1 e2) =
    teval' env e1 >>= (\t1 ->
    teval' env e2 >>= (\t2 -> ...
```

looks quite similar to the original `let` notation, only with the left- and the right-hand sides of `let` reversed. This reversal, which puts the binders on the right-hand side, looks unnatural. Fortunately Haskell has a convenient abbreviation for such a pattern, the `do`-notation, which places the binders to the left of the expressions to be bound:

```
teval' env (A e1 e2) = do
    t1 <- teval' env e1
    t2 <- teval' env e2
    ...
```

This new definition for `teval'` is equivalent to the one with the explicit `(>>=)`; in fact, the Haskell compiler rewrites the `do` form to the `(>>=)` form during the compilation. Finally, we observe that the combinators `(>>=)` and `return` are already defined in Haskell; furthermore, our type abbreviation `TVEM` is `State TVE` where `State` is a type constructor defined in `Control.Monad.State`. The result of our re-writing to hide `tve` is the file `TInfLetI.hs`.

We observe that `teval'` had to thread `TVE`, which consists of the current substitution and the next fresh type variable, all the way through. Thus, we can think of the `teval'` computation as incurring the side effect of reading and writing a piece of mutable state, of type `TVE`. This way of thinking prompts us to use the `State` monad in the Haskell standard library. (Incidentally, the implementation of the `State` monad uses the type `s->(a,s)` – compare with the result type of `teval'`.)

```
type TVEM = State TVE
```

The monadic version of `newtv`, the function to allocate a fresh type variable, has a new signature and a slightly different implementation. The `TVE` type explicit in the old signature is now hidden inside the type constructor `TVEM`.

```
newtv :: TVEM Typ
newtv = do
  TVE n s <- get
  put (TVE (succ n) s)
  return (TV n)
```

We also need a monadic version of `unify`:

```
unifyM :: Typ -> Typ -> (String -> String) -> TVEM ()
unifyM t1 t2 errf = do
  tve <- get
  case unify t1 t2 tve of
       Right tve -> put tve
       Left  err -> fail (errf err)
```

Finally, we rewrite `teval'` in the monadic style (we omit some cases). Again, the return type now hides `TVE` inside the monadic type constructor `TVEM`.

```
teval' :: TEnv -> Term -> TVEM Typ
teval' env (A e1 e2) = do
  t1  <- teval' env e1
  t2  <- teval' env e2
  t1r <- newtv
  unifyM t1 (t2 :> t1r) id
  return t1r
teval' env (I n) = return TInt
teval' env (e1 :+ e2) = do
  t1 <- teval' env e1
  t2 <- teval' env e2
  unifyM t1 TInt ("Trying to add non-integers: " ++)
  unifyM t2 TInt ("Trying to add non-integers: " ++)
  return TInt
teval' env (IFZ e1 e2 e3) = do
  t1 <- teval' env e1
  t2 <- teval' env e2
  t3 <- teval' env e3
  unifyM t1 TInt ("Trying to compare a non-integer to 0: " ++)
  unifyM t2 t3 (\err -> unwords ["Branches of IFZ have",
                                 "different types.",
                                 "Unification failed:", err])
  return t2
```

The code of the final `teval` changes slightly, to explicitly run the monadic value produced by `teval' tenv e` given the initial state `tve0`:

```
teval :: TEnv -> Term -> Typ
teval tenv e = let (t,tve) = runState (teval' tenv e) tve0
               in tvsub tve t
```

The cases for application and addition here contrast with those in the non-monadic version, `TInfT.hs`. The monad relieves us from threading `tve` explicitly, so we no longer worry about misnumbering `tve`. This worry is specific to Haskell; in OCaml or Clean one can use the same name `tve` throughout. Why?

On the other hand, the monadic code is more implicit: it is imperative! The dependencies are no longer apparent (e.g., the composition of two unifications in the type checking of addition). Do the exercise to see what problems that may bring:

Exercise: Comment out `Control.Monad.State.Strict` and uncomment `Control.Monad.State.Lazy`. Now run `test62` and explain the result. Is it different from running `test62` in `TInfT.hs`? Why?

6.2 Let-bound polymorphism as inlining

Code file: `TInfLetI.hs`.

Let us come back to polymorphic terms like `termid`. Before, we took advantage of binding in the metalanguage to name these terms and use them in variously typed contexts:

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

We would like to write this term without relying on the metalanguage to share (the values of) common sub-expressions such as `termid`. We extend the syntax of our language with the `Let` form:

```
data Term = ...
          | Let (VarName,Term) Term
```

so that `term2id` could now be written as

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

Exercise: extend `eval` in `EvalN.hs` to handle `Let`.

To evaluate this expression, we observe that `Let` is sort of an optimization: it lets us replace a common sub-expression that occurs in several places in a larger term by a much shorter name. `Let` may also speed up evaluation, because we can evaluate a common sub-expression only once and use the result several times. Type checking of `Let` presents a problem however. Although syntactically the same term appears in several places (so we can 'lift' these occurrences), the inferred type may differ from occurrence to occurrence. (This is the case for `term2id`. Why?) So, we can't assign a single type to the term. That is, `teval` may not return the same type for multiple occurrences of the same term. This observation suggests that, before type-checking a `Let` term, we 'undo' the optimization and inline the common term wherever it is mentioned, replacing the `Let`-bound variable with the term itself. Thus, before applying `teval` to the term `term2id`, we preprocess the term to yield

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

and proceed with type inference as before. We can do better however, relying on the compositionality of `teval`. Indeed, `teval' env term2id'` has the form

```
teval' env term2id' = do
  ...
  t1 <- teval' env' (L "x" vx)
  ...
  t2 <- teval' env' (L "x" vx)
  ...
```

The Haskell expression `teval' env' (L "x" vx)` appears twice above, so we can lift it at the metalanguage level, along with its side effect on the mutable state. Thus, we can write type checking of `term2id` with the `Let`-form as follows:

```
teval' env term2id = do
  let env' = ext env ("termid", teval' env (L "x" vx))
  ...
  t1 <- lkup env'' (V "termid")
  ...
  t2 <- lkup env'' (V "termid")
  ...
```

We stress that the computation `teval' env' (L "x" vx)` associated with the variable `(V "termid")` is a *monadic action.* It is executed several times in the above code, generally at different states `TVE`. Thus the types resulting from the executions, `t1` and `t2`, may differ. This explains how the same term occurring in different places may be given different types.

To implement this approach formally, we need to change surprisingly little. First of all, our `TEnv` no longer associates term variables with types. Rather, it associates term variables with monadic actions that produce a type, when executed in the `State` monad hiding mutable state of type `TVE`:

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

We slightly change the cases for variable and lambda, and add a new case for the `Let` form:

```
teval' env (V x)   = lkup env x
teval' env (L x e) = do
  tv <- newtv
  te <- teval' (ext env (x, return tv)) e
  return (tv :> te)
teval' env (Let (x,e) eb) = do
  _  <- teval' env e           -- what is the point of this line?
  teval' (ext env (x, teval' env e)) eb
```

We observe that in the lambda-case, we associate the variable `x` with a monadic action `return tv` – which returns the same type `tv` no matter how many times the action is executed. In contrast, the `Let`-case associates the variable `x` with the action `teval' env e`, which may create and return new type variables every time it is executed.

The changes do not affect the type inference for expressions without `Let`. All the examples from `TInfT.hs` can be used as they are and give the same result. We can write new examples, with `Let`:

```
testl1 = teval env0 $ Let ("x",vx) vx            -- error
testl2 = teval env0 $ Let ("x",vy) (I 1)         -- error
testl3 = teval env0 $ Let ("x",(I 1)) (vx :+ vx) -- TInt
```

We can also re-write `termlet` of `TInfT.hs` using the object-language `Let` form:

```
termlet =
  Let ("c2", L "f" (L "x" (V "f" `A` (V "f" `A` vx)))) (
  Let ("inc", L "x" (vx :+ (I 1))) (
  Let ("compose", L "f" (L "g" (L "x"
                  (V "f" `A` (V "g" `A` vx))))) (
  Let ("id",L "x" vx) (
   V "c2" `A` (V "compose" `A` V "inc" `A` V "inc") `A` (I 10) :+
    (V "c2" `A` (V "compose" `A` V "inc") `A` V "id" `A` (I 100))
  ))))
testlet = teval env0 termlet -- TInt
```

Exercise: the following tests illustrate the difference between lambda-bound and let-bound variables. Explain the different results of the tests.

```
testl66 = teval env0 $ L "x" (Let ("y",vx)
                               (Let ("z", (vy `A` (I 1) :+ (I 2)))
                                 vy))
-- (TInt :> TInt) :> (TInt :> TInt), monomorphic

testl67 = teval env0 $ L "x" (Let ("y",vx) ((vy `A` (I 1)) :+
                                             (vy `A` (L "x" vx))))
-- *** Exception: constant mismatch: TInt and TV 2 :> TV 2

testl76 = teval env0 $ Let ("x", L "y" (I 10))
                          (Let ("y",vx)
                                (Let ("z", vy `A` (I 1) :+ (I 2))
                                  vy))
-- TV 4 :> TInt, polymorphic
```

```
test177 = teval env0 $ Let ("x", L "y" (I 10))
                              (Let ("y",vx) ((vy `A` (I 1)) :+
                                              (vy `A` (L "x" vx))))
-- TInt, OK.
```

We thus allow `Let`-bound variables to take polymorphic types without restricting the expressions they are bound to. In particular, we do not impose the so-called value restriction and require `Let`-bound polymorphic expressions to be values. However, if the object language has effects such as mutation, then we should not support polymorphism by executing the type-checking computation `teval' env e` multiple times, or the abstract result of type checking may not correctly approximate the concrete result of dynamic evaluation. Why? Hint: the contrast between sharing and replication/inlining is unobservable in a pure language but observable in a language with effects (exercise: give an example using assignment). If replicating an expression may affect the result of dynamic evaluation, then type checking must refrain from replicating the expression, or it may not correctly approximate the result of dynamic evaluation with sharing.

6.3 A polymorphic type as an approximate denotation of a common sub-expression

Code file: `TInfLetP.hs`.

In this section, we optimize the type inference algorithm of `TInfLetI.hs` to avoid repeated execution of the `teval'` action every time a `Let`-bound variable is used. We would like to type-check a common sub-expression once, where it is defined, rather than at each point where it is mentioned. Informally, we would like to 'precompile' the `Let`-bound expression as far as possible, associating the `Let`-bound variable with the resulting 'pre-type'. The goal is to do much of the work of type reconstruction up front, so that each use of the `Let`-bound variable is fast to type-check. This is Hindley-Milner type-checking, presented in a non-canonical way.

The function `teval` of `TInfT.hs` inferred for the term `L "x" vx` the type `TV 0 :> TV 0`, containing a free type variable. Since the term stands alone, the type variable `TV 0` is 'truly free', with nothing to constrain it. To describe such types with 'truly free' type variables, we introduce *type schemes*:

```
data TypS = TypS [TVarName] Typ
```

explicitly enumerating the truly free type variables. Examples: `TypS [] TInt` (which is essentially `TInt`), `TypS [0] (TV 0 :> TV 0)` (the type scheme for the identity function), `TypS [3] ((TV 0 :> TV 3) :> TV 3)`. These truly free type variables are called *generalizable* or *generic*; the act of forming a type scheme out of a type by explicitly specifying generic variables of the type is called *generalization.* The type variables explicitly named in the type scheme are considered bound (or, 'quantified') in the scheme. For example, whereas the type ((TV 0 :> TV 3) :> TV 3) has two free type variables, TV 0 and TV 3, only TV 0 is free in the type scheme TypS [3] ((TV 0 :> TV 3) :> TV 3). The latter type scheme will be inferred, for example, for the Let-bound variable y in the term terml51

```
terml51 = L "x" (Let ("y", L "f" ((V "f") `A` vx)) vy)
terml52 = terml51 `A` (I 10)
```

Indeed, the inferred type for the expression L "f" ((V "f") `A` vx) bound to y is (TV 0 :> TV 3) :> TV 3. The type variable TV 0 is not generic because it is constrained, namely by the type of x. When terml51 is used in a larger term, terml52, the inferred type for L "f" ((V "f") `A` vx) becomes (TInt :> TV 3) :> TV 3. There is no longer TV 0, but the free type variable TV 3 remains. It will remain in the type of the expression for y no matter how terml51 is used. The type variable TV 3 is thus generic, and we can infer for y the type scheme TypS [3] ((TV 0 :> TV 3) :> TV 3).

We make the notion of generic type variables precise a bit later. Right now we stress that Hindley-Milner type inference associates term variables with type schemes rather than with types:

```
type TEnv = [(VarName, TypS)]
```

It is instructive to compare this type environment with that of TInfLetI.hs, which associated term variables with monadic actions of the type TVEM Typ. To obtain the type, we had to execute the action in the monad TVEM. To obtain the type from TypS, we have to *instantiate* it. In both cases, TVEM Typ and TypS may be regarded as 'pre-types' or 'would-be' types, requiring certain actions to obtain the type, which each time may return a type with fresh type variables. To be precise, the instantiation of a type scheme, containing the type t and the list of type variables tvs, is a TVEM action. When executed, it returns an *instance* of t: for each type variable in tvs, its occurrences in t are replaced with a fresh type variable. The name 'instantiation' comes from logic, where universally quantified formulas are instantiated.

```
instantiate :: TypS -> TVEM Typ
instantiate (TypS tvs t) = do
  tve <- associate_with_freshvars tvs
  return $ tvsub tve t
 where
 associate_with_freshvars [] = return tve0
 associate_with_freshvars (tv:tvs) = do
   tve     <- associate_with_freshvars tvs
   tvfresh <- newtv
   return $ tvext tve (tv,tvfresh)
```

Examples:

```
instantiate (TypS [1] TInt)          -- TInt
instantiate (TypS [3] ((TV 0 :> TV 3) :> TV 3))
                                     -- (TV 0 :> TV 42) :> TV 42
```

The optimized `teval'` differs from the unoptimized `teval'` in `TInfLetI.hs` in only three cases:

```
teval' env (V x)   = instantiate (lkup env x)

teval' env (L x e) = do
    tv <- newtv
    te <- teval' (ext env (x, TypS [] tv)) e
    return (tv :> te)

teval' env (Let (x,e) eb) = do
    t  <- generalize (teval' env e)
    teval' (ext env (x,t)) eb
```

The variable case now instantiates a type scheme to obtain the type; this is faster than executing the action associated with the variable, as was the case in `TInfLetI.hs`. The lambda-case creates an association of a variable with a 'would-be' type. In `TInfLetI.hs` that association was `return tv`, which is the action that gives the same `tv` no matter how many times it is executed. Now, we associate the lambda-bound variable with the likewise trivial type scheme `TypS [] tv`, which returns the same `tv` no matter how many times it is instantiated.

The remaining difference is in the `Let`-case, which reconstructs the type of the `Let`-bound expression and generalizes it to the type scheme. The `Let`-case is the contribution of Robin Milner to Hindley's type system (which Milner re-discovered).

The function `generalize` finds the generic type variables in a type and turns the type into a type scheme. Informally, a type variable in a type is generic if it does not depend on the initial state used to reconstruct the type. Our goal is to reproduce the behavior in `TInfLetI.hs` when handling `Let (x,e) eb`. There, generalization was the identity – that is, our 'type scheme' was the unexecuted type-reconstruction action `teval' env e` – and instantiation was the execution of that action. Now, we would like to execute the action `teval' env e` and reconstruct the type of the `Let`-bound expression only once, before processing the body `eb`.

We denote by `tve_before` the type checker's initial state before executing the action `teval' env e`, and denote by `tve_after` the final state after executing the action. At any point in the `Let`-body where the `Let`-bound variable is mentioned – that is, at any state that extends `tve_before` – we want executing `teval' env e` to be equivalent to instantiating the type scheme. That is, the instantiation should produce exactly as many fresh type variables as does the execution of `teval' env e`. A free type variable in a type produced by executing `teval' env e` at `tve_before` is generic if it would still be free if `tve_before` were extended with arbitrary bindings. To be more precise, a type variable `tv` is generic if

1. tv is unbound in tve_after; and
2. tv does not appear in tvsub tve_after tvb for any type variable tvb in tve_before (or equivalently, for any type variable tvb unbound in tve_before).

We implement generalization using this notion of generic type variables.

```
generalize :: TVEM Typ -> TVEM TypS
generalize ta = do
  tve_before <- get
  t          <- ta
  tve_after  <- get
  let t' = tvsub tve_after t
  let tvdep = tvdependentset tve_before tve_after
  let fv = filter (not . tvdep) (nub (freevars t'))
  return (TypS fv t')
```

The function freevars used above lists the type variables unbound in a type.

```
freevars :: Typ -> [TVarName]
freevars TInt       = []
freevars (t1 :> t2) = freevars t1 ++ freevars t2
freevars (TV v)     = [v]
```

Of the two conditions above for a type variable to be generic, the first is satisfied automatically in generalize by applying tvsub tve_after to t. The second condition is checked using the function tvdependentset.

```
tvdependentset :: TVE -> TVE -> (TVarName -> Bool)
tvdependentset tve_before tve_after =
    \tv -> any (\tvb -> occurs tv (TV tvb) tve_after) tvbs
 where tvbs = tvfree tve_before
```

Finally, the function tvfree lists the type variables that are allocated but not bound:

```
tvfree :: TVE -> [TVarName]
tvfree (TVE c s) = filter (\v -> not (M.member v s)) [0..c-1]
```

We reiterate that multiple instantiations of a type scheme are exactly like multiple executions of a type-reconstruction action – 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.

Exercise: can the computation of tvdependentset be optimized?

We modify teval to generalize the inferred type of the input term to a type scheme: we assume that the input term stands alone at the top level, so generalization is appropriate.

```
teval :: TEnv -> Term -> TypS
teval tenv e = let (ts,tve) =
   runState (generalize (teval' tenv e)) tve0 in ts
```

All examples from previous files run as they are. The inferred type is presented slightly differently:

```
term1 = L "x" (IFZ vx (I 1) (vx :+ (I 2)))
test1 = teval env0 term1 -- TypS [] (TInt :> TInt)
```

```
testlet = teval env0 termlet -- TypS [] TInt
```

Now, the the identity function has the inferred type `TypS [0] (TV 0 :> TV 0)`, which is commonly written as $\forall a.\, a \to a$.

The difference is more noticeable in

```
term2a = L "x" (L "y" (vx `A` vy))
test2a = teval env0 term2a
-- TypS [1,2] ((TV 1 :> TV 2) :> (TV 1 :> TV 2))
```

The inferred type is the type scheme, no longer containing 'free' type variables. That makes it clear the inferred type for `term2a` is polymorphic.

Our presentation of Hindley-Milner type-checking is non-canonical in that we only talk about mappings from type variables to types: our notions of generic type variables and generalization never refer to a mapping from term variables to types.

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

We come back to Church's point of view: untyped terms make no sense, so the language should not permit even writing them, let alone evaluating them. All expressible terms must be well-typed.

In `EvalN.hs`, we could write terms such as

```
term3 = L "x" (IFZ vx (I 1) vy)
term4 = L "x" (IFZ vx (I 1) (vx `A` (I 1)))
```

and even successfully evaluate them and their applications to some values. One may argue though that a term (such as `term3`) with a reference to an unbound variable makes no sense, no matter how well the problem is hidden.

Thus we wish to embed an object language in the metalanguage in such a way so that terms ill-typed in the object language become ill-typed in the metalanguage. Therefore, the metalanguage itself will not let us successfully enter and use ill-typed object terms. That saves us the trouble of writing `teval` – we piggy-back on the (much more powerful) '`teval`' built in the metalanguage. One immediate advantage is that type error messages become better, with location information, hints about the error, etc.

7.1 Representing an object language as a data type in the metalanguage

Code file: `EvalTaglessI.hs`.

We have to use GADT – this file is not in Haskell98.

The problematic terms mentioned earlier are now rejected by the Haskell type checker, with good error messages:

```
-- term3 = L (\vx -> IFZ vx (I 1) vy) -- Not in scope: ‘vy’

-- term4 = L (\vx -> IFZ vx (I 1) (vx ‘A‘ (I 1)))

    Couldn’t match expected type ‘t1 -> Int’
           against inferred type ‘Int’
      Expected type: Term (t1 -> Int)
      Inferred type: Term Int
    In the first argument of ‘A’, namely ‘vx’
    In the third argument of ‘IFZ’, namely ‘(vx ‘A‘ (I 1))’

-- delta = L (\vy -> vy ‘A‘ vy)
    Occurs check: cannot construct the infinite type: t1 = t1 -> t2
      Expected type: Term t1
      Inferred type: Term (t1 -> t2)
    In the second argument of ‘A’, namely ‘vy’
    In the expression: vy ‘A‘ vy
```

Compare the tests of `TInfLetP.hs` with those of `EvalTaglessI.hs`. In the latter case, the Haskell type-checker has inferred the types. Thus we can compare our Hindley-Milner type-checker against Haskell’s.

7.2 Tagless-final approach

Code file: `EvalTaglessF.hs`.

The concrete syntax for object terms changes from `EvalTaglessI.hs` only in *case*: `(i 1)` vs. `(I 1)`, etc. Please compare the tests of `EvalTaglessI.hs` and `EvalTaglessF.hs`. The tests were automatically converted by the downcase-region function of Emacs.

It is challenging to show higher-order abstract syntax terms. Yet, in contrast to the GADT-based approach in `EvalTaglessI.hs`, we can show terms in the tagless-final approach, without extending our language with auxiliary syntactic forms. After all, showing terms is just another way to *evaluate* them, to strings.

For more details, see Carette et al. [2].

8 Further reading

- Cardelli [1]
- the substitution treatment of polymorphism: [4, Section 11].
- Jones and Nielson [3, Sections 1, 2.1, 2.7, and 3.1]
- Reynolds [5, Sections 1–5]; Reynolds [6]

References

[1] Cardelli, Luca. 1987. Basic polymorphic typechecking. *Science of Computer Programming* 8(2):147–172. `http://lucacardelli.name/Papers/BasicTypechecking.pdf`.

[2] Carette, Jacques, Oleg Kiselyov, and Chung-chieh Shan. 2007. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. In *Proceedings of APLAS 2007: 5th Asian symposium on programming languages and systems*, ed. Zhong Shao, 222–238. Lecture Notes in Computer Science 4807. `http://okmij.org/ftp/papers/tagless-final-APLAS.pdf`.

[3] Jones, Neil D., and Flemming Nielson. 1994. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of logic in computer science*, 527–629. Oxford University Press. `http://www.diku.dk/forskning/topps/bibliography/1994.html#D-58`.

[4] Mitchell, John C. 1996. *Foundations for programming languages.*

[5] Reynolds, John C. 1998. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation* 11(4):363–397. `ftp://ftp.cs.cmu.edu/user/jcr/defint.ps.gz`.

[6] ———. 1998. Definitional interpreters revisited. *Higher-Order and Symbolic Computation* 11(4):355–361. `ftp://ftp.cs.cmu.edu/user/jcr/defintintro.ps.gz`.