An exercise in dependent types: A well-typed interpreter

Lennart Augustsson
Magnus Carlsson

Department of Computing Sciences
Chalmers University of Technology
S-412 96 Göteborg, Sweden
Email: {augustss,magnus}@cs.chalmers.se
WWW: http://www.cs.chalmers.se/~{augustss,magnus}

Abstract. The result type of an interpreter written in a typed language is normally a tagged union. By using depent types, we can be more precise about the type of values that the intepreter returns. There is no need for tagging these values with their type, something which opens the door to more efficient interpreters.

1 An annoying problem

We are ignoring $\perp$ for now.

Writing an interpreter for a language which can produce results of different types requires some kind of encoding of the result, as in figure 1. The encoding is captured in the tagging in the data type `Value`. This tagging is inefficient since each return value must be tagged. In addition, the tags must be tested and the real values extracted again when they are used.

```
data Expr = EBool Bool | EInt Int | EAdd Expr Expr

data Value = VBool Bool | VInt Int | VError

interp :: Expr -> Value
interp (EBool b) = VBool b
interp (EInt i)  = VInt i
interp (EAdd e1 e2) =
    case (interp e1, interp e2) of
    (VInt i1, VInt i2) -> VInt (i1+i2)
    _                  -> VError
```

Fig. 1. A very simple language and its interpreter.

The tags serve two purposes: first, they turn the result into a union type so that the result is a *single* type; second, they are used to make sure that the interpreted program is type correct. However, if the interpreted program has already been type checked and it is known to be type correct so using the union is really unnecessary.

Assuming a type correct program, the interpreter we would like to write is shown in figure 2. This interpreter can be written in a dynamically typed langue like Lisp [Fox60] or Erlang [AV90], but unfortunately not in strongly typed languages like Haskell [HPJWe92] or SML [MTH90].

```
interp (EBool b) = b
interp (EInt i)  = i
interp (EAdd e1 e2) = interp e1 + interp e2
```

Fig. 2. An untyped interpreter.

2 Dependent types to the rescue

The problem with the untagged interpreter (figure 2) is that the type of the result *depends* on the value of the argument. To handle that, we need a more powerful type system, one that can handle *dependent types*. We are going to continue this example by using Cayenne, [Aug98], a Haskell-like language which has dependent types.

The interpreter written in Cayenne is shown in figure 3. One difference from a Haskell version is the type of `interp`. Dependent function types are written "`(x :: t) -> s`", where the variable `x` may occur in `s`. In this way the result type of a function can depend on the value of the argument. A second difference from Haskell is manifested in the function `TypeOf`. This function computes a type. The type of types is written "`#`" in Cayenne and it is treated mostly like other types.

```
interp :: (e :: Expr) -> TypeOf e
interp (EBool b) = b
interp (EInt i)  = i
-- EAdd doesn't work

TypeOf :: Expr -> #
TypeOf (EBool _) = Bool
TypeOf (EInt _) = Int
TypeOf (EAdd _ _) = Int
```

Fig. 3. A dependently typed interpreter.

This interpreter is well typed, but the `EAdd` case is missing. The reason is that the expression "`interp e1 + interp e2`" is not well typed. The type of `interp` states that it sometimes returns `Int` and sometimes `Bool`, but we cannot know that both (or any) operands will be of type `Int`. Something more is needed; when interpreting an expression we need some tangible evidence that it is actually well typed, otherwise we cannot know that the `EAdd` clause cannot fail.

Before we add this we will make the language more interesting by adding some more cases to the expression type, figure 4, and by formalizing the typing rules, table 5. The typing rules are exactly what you can expect for this little language.

```
data Expr = EBool Bool | EInt Int
          | EAdd Expr Expr | EAnd Expr Expr | ELE Expr Expr
```

Fig. 4. A slightly extended expression type.

$$\frac{}{i \in \mathit{int}} \qquad \frac{}{b \in \mathit{bool}}$$

$$\frac{e_1 \in \mathit{int} \quad e_2 \in \mathit{int}}{e_1 + e_2 \in \mathit{int}} \quad \frac{e_1 \in \mathit{bool} \quad e_2 \in \mathit{bool}}{e_1 \&\& e_2 \in \mathit{bool}} \quad \frac{e_1 \in \mathit{int} \quad e_2 \in \mathit{int}}{e_1 <= e_2 \in \mathit{bool}}$$

Fig. 5. Typing rules for the slightly extended expression type.

3 The well typing predicate

We want to capture the typing rules in such a way that we can know when an expression is well typed. One way to do this to encode the typing rules as a predicate. A predicate is encoded as a function that yields a type. The trick of the encoding is that the type it yields should be empty when the predicate is false, and non-empty when the predicate is true. All of this is, of course, part of the Curry-Howard isomorphism.

More about predicates

The typing rules have been encoded as predicate, `HasType` in figure 6, together with some auxillary definitions.[1] Since it is impossible to pattern match on types in Cayenne we need to introduce a concrete type, `Type`, to encode the

[1] As can be seen from the example Cayenne does not place the same restrictions as Haskell on type and constructor names.

possible types of an Expr. The types Truth and Absurd are used to encode logical truth and falsity respectively, but any non-empty and empty type would do equally well. More

The /\ type is simply pairs (with an unusual name). It is used to represent conjunction. Pairs represent conjunction simply because if either of the component types of a pair is an empty type then the whole pair is an empty type as well.

```
data Type = TBool | TInt

data Truth = truth

data Absurd =      -- the empty type, no constructors

data (/\) a b = (&) a b

HasType :: Expr -> Type -> #
HasType (EBool _)    (TBool) = Truth
HasType (EInt _)     (TInt)  = Truth
HasType (EAdd e1 e2) (TInt)  = HasType e1 TInt  /\ HasType e2 TInt
HasType (EAnd e1 e2) (TBool) = HasType e1 TBool /\ HasType e2 TBool
HasType (ELE  e1 e2) (TBool) = HasType e1 TInt  /\ HasType e2 TInt
HasType _            _       = Absurd
```

Fig. 6. The well typing predicate.

Why are we using a predicate instead of a boolean function? The HasType function could as easily have been written to yield a boolean instead of a type, but as we will soon see it is really the type we need. Furthermore, as we extend the type system there will be cases when the predicate is easy to write, but the boolean function could be very difficult (or even impossible).

A complete interpreter for our little language is presented in figure 7. The interpreter is now more complicated. It takes the expression to interpret, the expected type of the expression, and an object in the well-typing predicate above. The Decode function simply takes the encoded representation of the type into its real type.

The well-typing predicate is an empty type when the expression is not well typed. This means that in, e.g., the first clause of interp the p argument has type Truth. The most interesting clause is the last one. It is there to make the patterns exhaustive, without it the Cayenne compiler would complain that not all cases were covered in interp. In this clause the type of the argument p must be Absurd. This can be easily checked by examining the HasType predicate; the cases where it is not Absurd were covered by the other clauses in interp. This means that if we reach the last clause of interp then p must be an element of the empty type, but this is of course absurd. The absurdity of it is handled by the case analysis of `p`. The case expression has no arms, since the empty type has no constructors; still all constructors are covered (since there are none) and all the right hand sides of the case have the same type as we want (`Decode t`).

We can now see the reason for having the `HasType` function be a predicate. This way the ill-typed case ends up as a clause in the interpreter that has an argument in the empty type, and which can thus never be reached. The fact that `p` is an element of the empty type makes it possible to construct an expression of the right type on the right hand side of the last clause. A boolean value would have been of no use, and we would have had to resort to calling, e.g., the `error` function, despite the fact that the clause can never be reached.[2]

Note how the pair type used for the operators is taken apart and the constituent proofs are passed along in the recursive calls to `interp`. It is easily verified that they actually have the right type by comparing with the definition of `HasType`.

```
interp :: (e :: Expr) -> (t :: Type) -> HasType e t -> Decode t
interp (EBool b)    (TBool) p         = b
interp (EInt  i)    (TInt)  p         = i
interp (EAdd e1 e2) (TInt)  (p1 & p2) = interp e1 TInt  p1 +  interp e2 TInt p2
interp (EAnd e1 e2) (TBool) (p1 & p2) = interp e1 TBool p1 && interp e2 TBool p2
interp (ELE  e1 e2) (TBool) (p1 & p2) = interp e1 TInt  p1 <= interp e2 TInt p2
interp _            _       p         = case p of { }

Decode :: Type -> #
Decode (TBool) = Bool
Decode (TInt)  = Int
```

Fig. 7. An interpreter for the slightly extended expression type.

4 Adding variables

The interpreted language in the previous sections is very simple. We will now extend it to λ-calculus by adding variables, application, and λ-expressions. The λs will be explicitely typed to simplify things.

The extended expression type is shown in figure 8. The concrete type expressions and the decoding function are also extended to handle the function type.

4.1 The type system

The typing rules must be extended with an environment (or assumptions) to handle the variables. These rules, figure 9, are the standard ones.

[2] A better type system might be able to determine that the last clause cannot be reached and can therefore be omitted, see eg. [Coq92].

```
data Expr = EBool Bool | EInt Int
          | EAdd Expr Expr | EAnd Expr Expr | ELE Expr Expr
          | EVar Symbol | ELam Type Symbol Expr | EAp Expr Expr

type Symbol = String

data Type = TBool | TInt | TArrow Type Type

Decode :: Type -> #
Decode (TBool) = Bool
Decode (TInt)  = Int
Decode (TArrow a b) = Decode a -> Decode b
```

Fig. 8. The expression type.

$$\frac{}{\Gamma \vdash i \in \mathit{int}} \qquad \frac{}{\Gamma \vdash b \in \mathit{bool}}$$

$$\frac{\Gamma \vdash e_1 \in \mathit{int} \quad \Gamma \vdash e_2 \in \mathit{int}}{\Gamma \vdash e_1 + e_2 \in \mathit{int}} \quad \frac{\Gamma \vdash e_1 \in \mathit{bool} \quad \Gamma \vdash e_2 \in \mathit{bool}}{\Gamma \vdash e_1 \&\& e_2 \in \mathit{bool}} \quad \frac{\Gamma \vdash e_1 \in \mathit{int} \quad \Gamma \vdash e_2 \in \mathit{int}}{\Gamma \vdash e_1 <= e_2 \in \mathit{bool}}$$

$$\frac{}{\Gamma, x \in t \vdash x \in t} \qquad \frac{\Gamma \vdash f \in s \to t \quad \Gamma \vdash a \in s}{\Gamma \vdash f\ a \in t} \qquad \frac{\Gamma, x \in s \vdash e \in t}{\Gamma \vdash \lambda x : s.e \in s \to t}$$

Fig. 9. Typing rules for expressions.

The `HasType` predicate (see figure 10) must be extended with an environment, `g`, as well. The environment is represented simply by a function from symbols to types, and the function `extendT` is used to add a binding. We assume that the environment is a total function (it can easily be made total).

The `EVar` clause simply looks up the variable in the type environment and "checks" that it is equal to the type it should have. The `EqType` predicate is used to test if two types are equal. It is written in the obvious way, but yields a type instead of the usual boolean result.

The `ELam` clause follows the typing rule closely and checks that the body has the right type in the extended environment.

The most interesting clause is `EAp`. According to the rule an application is well typed if there is some type s, that the argument has, which makes the types fit together. The type s is not present in the conclusion of the rule, which means that it is not present on the left hand side of `HasType`. Instead, it is a component of a record type on the right hand side. The `sig` keyword simply indicates a record type; it is followed by the signatures for its components. The record has three parts, two of them correspond to the two premises (and could be represented by a pair). The first component of the record is the type `s` which makes the other two components true. The use of a record type corresponds to using an existential quantifier. MORE

Note how the definition of the `HasType` predicate follows the typing rules closely. The conclusions correspond to patterns in the predicate, and the premises are the right-hand-side expressions, using `Truth` for the axioms. The `Absurd` clause expresses that there are no other rules in the system but these. There are some minor differences, though:

- Cayenne does not allow variables to occur more than once in a pattern, but in the rule for λ-expressions, the type s occurs twice. The `ELam` clause encodes this by using two variables, and an additional premise stating that these are equal. The same goes for the variable clause.
- We need to be explicit about variables introduced in the premises, such as s in the `EAp` clause.

4.2 Representing the environment

The interpreter for the full language needs to use an environment to look up the values of variables. The representation of this environment is not totally obvious since the variables can have different types. This problem can be solved by using both a type environment and a value environment. The type of the latter can then be expressed using the former, as shown in figure 11. As can be seen from this definition the return type of the environment lookup depends on the value (encoding a type) found in the type environment.

The implementation of both the type and value environments are straightforward, but the type of `extendV` is rather complicated. The function `extendV` takes not only the expected arguments `r`, `x`, and `v`, but also the type environment, `g`, and the type of the value, `t`. Without these it would be impossible to

```
type TEnv = Symbol -> Type
extendT :: TEnv -> Symbol -> Type -> TEnv

HasType :: TEnv -> Expr -> Type -> #
HasType g (EBool _)    (TBool) = Truth
HasType g (EInt _)     (TInt)  = Truth
HasType g (EAdd e1 e2) (TInt)  = HasType g e1 TInt  /\ HasType g e2 TInt
HasType g (EAnd e1 e2) (TBool) = HasType g e1 TBool /\ HasType g e2 TBool
HasType g (ELE  e1 e2) (TBool) = HasType g e1 TInt  /\ HasType g e2 TInt
HasType g (EVar x)     t       = EqType (g x) t
HasType g (EAp f a)    t       =
    sig s  :: Type
        pf :: HasType g f (TArrow s t)
        pa :: HasType g a s
HasType g (Lam x s' e) (TArrow s t) =
    HasType (extendT g x s) e t /\ EqType s' s
HasType _ _            _       = Absurd

EqType :: Type -> Type -> #
EqType (TInt) (TInt) = Truth
EqType (TBool) (TBool) = Truth
EqType (TArrow a1 r1) (TArrow a2 r2) = EqType a1 a2 /\ EqType r1 r2
EqType _ _ = Absurd
```

Fig. 10. The well typing predicate.

express the type since the `VEnv` type is indexed by the type environment. The `IF` function used in `entendV` is the if function that allows the two branches to have different types.

The `extendV` function really has the most amazing type of any function in this example, so let us examine why it is type correct a little closer. The type of the `IF` function is

```
IF ::
  (tx :: #) |-> (ty :: #) |->
  (b :: Bool) -> tx -> ty -> case b of (True) -> tx; (False) -> ty
```

The function arrow `|->` indicates that the argument does not normally have to be given; the type checker deduces it instead.

The variable `v` has type "`Decode t`" and the expression "`r x`" has type "`Decode (g x)`" according to the type of `r`. So looking at the right hand side of `extendV` we can see that it has type "`(x' :: Symbol) -> case x == x' of (True) -> Decode t; (False) -> Decode (g x)`".

According to the signature the type should be "`VEnv (extendT g x t)`", expand this and we get "`(x :: Symbol) -> Decode (if (x == x') t (g x))`", furthermore, expand the definition of `if` and we get "`(x :: Symbol) -> Decode (case x == x' of (True) -> t; (False) -> g x`". Using the definition of `Decode` (which is really a case expression) and the rule of commuting case expressions the type checker can verify that this expression is indeed equal to the inferrred type above.

```
type VEnv (g :: TEnv) = (x :: Symbol) -> Decode (g x)

-- implementation of the environments
emptyT :: TEnv
emptyT = \ (s :: Symbol) -> TInt  -- make it total

extendT :: TEnv -> Symbol -> Type -> TEnv
extendT g x t = \ (x' :: Symbol) -> if (x == x') t (g x)

emptyV :: VEnv emptyT
emptyV = \ (x :: Symbol) -> 0     -- make it total

extendV :: (g :: TEnv) ->
           (r :: VEnv g) ->
           (x :: Symbol) ->
           (t :: Type) ->
           (v :: Decode t) ->
           (VEnv (extendT g x t))
extendV g r x t v = \ (x' :: Symbol) -> IF (x == x') v (r x)
```

Fig. 11. Representing the environments.

5 The interpreter

We now have all the building blocks to make the complete interpreter, figure 12. The application and λ clauses simply end up as an application and a λ-expression respectively. For the application clause it is simply a matter of calling the interpreter recursively with the right arguments; note the use of `p.s` which is the type of the argument, `a`. The type of the argument is not apparant from the value or type argument to `interp`, but can be found in the "proof". In the λ clause the type and value environments must be extended in the obvious way.

The most complicated clause is really the variable access. The variable is looked up by "`(r x)`". This expression has the (encoded) type "`(g x)`", but the expected return type is `t`. These two types are not obviously equal, but `p` is a proof that they actually are equal since it has type "`EqType (g x) t`". The function `convert` converts a value belonging to one type into the same value of another type given a proof that the types are actually equal. The `convert` functions is thus nothing but an identity function and its only purpose is to placate the type checker.

```
interp :: (e :: Expr) -> (t :: Type) -> (g :: TEnv) ->
          VEnv g -> HasType g e t -> Decode t
interp (EBool b)    (TBool) g r p         = b
interp (EInt  i)    (TInt)  g r p         = i
interp (EAdd e1 e2) (TInt)  g r (p1 & p2) = interp e1 TInt  g r p1 +  interp e2 TInt  g r p2
interp (EAnd e1 e2) (TBool) g r (p1 & p2) = interp e1 TBool g r p1 && interp e2 TBool g r p2
interp (ELE  e1 e2) (TBool) g r (p1 & p2) = interp e1 TInt  g r p1 <= interp e2 TInt  g r p2
interp (Var x)      t       g r p         = convert (g x) t p (r x)
interp (App f a)    t       g r p         = (interp f (TArrow p.s t) g r p.pf)
                                            (interp a p.s           g r p.pa)
interp (Lam x _ e)  (TArrow s t) g r (p & _) =
     \ (v :: Decode s) -> interp e t (extendT g x s) (extendV g r x s v) p
interp _            _       g r p         = case p of { }

-- An identity function.
convert :: (t :: Type) -> (t' :: Type) -> EqType t t' -> Decode t -> Decode t'
```

Fig. 12. The interpreter.

MORE

6 Type checking

Whenever the `interp` function is called it has to be given a value that represents the proof that the expression is well typed. This can, of course, be constructed by hand for any particular (well typed) expression. On the other hand, if the expression comes from a parser we would like it to be generated dynamically. In this section we will take a look at a type checker that given an expression will either return an indication that it is ill typed, or the type of the expression together with the proof that it is well typed. Its type is shown in figure 13.

```
tcheck :: (e :: Expr) -> (g :: TEnv) ->
          Maybe (sig { t :: Type; p :: HasType g e t })
```

Fig. 13. The type of the type checker.

MORE

7 Putting it all together

Assuming that we have a parser and a printing function we can now put all the pieces together into a program that take a string, parses it, type checks the expression, interprets it, and finally prints the result. All of this is show in figure 14.

As shown the interpreter assumes that all free variables have value `0`, but it would be easy to add a check that there are no free variables on the top level. The type of this check can be refined to include a proof that there are no free variables, but this is left as an exercise for the reader.

```
parse :: String -> Maybe Expr
parse s = ...

print :: (t :: Type) -> Decode t -> String
print (TInt) i = System$Int.show i
print (TBool) b = System$Bool.show b
print (TArrow _ _) f = "<<function>>"

run :: String -> String
run s =
    case parse s of
    (Nothing) -> "Parse error!"
    (Just e) ->
        case tcheck e emptyT of
        (Nothing) -> "Type error!"
        (Just tp) -> print tp.t (interp e tp.t emptyT emptyV tp.p)
```

Fig. 14. A complete program.

8 Efficiency

The interpreter we have presented is more efficient than one using a tagged representation because it can avoid the repeated tagging and untagging. On the other hand, our version of the interpreter has three extra arguments (the result type, the type environment, and the well typing proof). These extra arguments are really superflous from a computational point of view. [?] describes a method where the Cayenne programmer uses type annotations to flag expressions and arguments that are only needed during type checking. The compiler is then able to remove all arguments except the expression to interpret, and the value environment. For some expressions, the resulting interpreter runs considerably faster than a corresponding interpreter written in Haskell, due to the lack of tagging and untagging.

More on how to get rid of them

A further potential source of inefficiency is the `convert` function. It is really an identity function, but the most straightforward implementation of it will result in an function the will completely traverse the two `Type` arguments before returning the value. This problem can also be remedied.

More on how to solve this problem

9 Conclusion

Using dependent types it is possible to write an interpreter that avoids tagging, but there are several other benefits.

- The typing rules were almost trivially captured as the `HasType` predicate and since the correspondance is so close it is unlikely that the definition is wrong.
- The type checker must be faithful to the typing rules since it delivers a type for the expression that is correct according to `HasType`.
- We do not know if the interpreter respects the operational semantics of the language (since the semantics has not been formalized), but we know that the interpreter respects the types of the language since it uses the `HasType` predicate.

This means that certain kind of errors in the type check and/or interpreter, e.g., claiming that integer constants have type bool, will be caught by the type checker.

More

10 Acknowledgments

Thanks to Jim Hook for getting one of us (Lennart) started on the interpreter and for showing how to handle environments for a typed interpreter. Thanks also to Jessica who helped improving the English of this paper.

References

[Aug98] Lennart Augustsson. Cayenne — a language with dependent types. In *Proc. of the International Conference on Functional Programming (ICFP'98)*. ACM Press, September 1998.

[AV90] J. L. Armstrong and S. R. Virding. Erlang - an experimental telephony programming language. In *XIII International Switching Symposium*, June 90.

[Coq92] T. Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[Fox60] Phyllis A. Fox. *LISP 1 Programmer's Manual*. March 1960.

[HPJWe92] P. Hudak, S. L. Peyton Jones, and P. Wadler (editors). Report on the programming language haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, Mar, 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.