Dependent Type Checkers made Simple

Jared Roesch and Tristan Konolige

Introduction

You may have just opened this document with the intent of learning something about dependent types. This is not a straight forward journey as the landscape of dependent type theory and implementations are in constant flux.

Much of the wisdom about implementing dependent types is scattered across the literature without an overall sense of how it fits together. This document is an attempt at making high level ideas about dependent type checkers clear for someone with very little experience in the academic community. We will begin with a high level discussion about type theory, type systems, and the evolution of dependent types.

Dependent Type Theory

Dependent types are often viewed as an esoteric or obscure idea from programming language community. This is not too far from the mark but it doesn't mean they are not useful or without merit.

Many language's type systems are evolving in ways that allow them to emulate constructs from dependent types and knowledge about dependent types can enable programmers to use their day to day type systems more effectively.

Traditionally when programming in statically typed languages we learn about a very important distinction early on: the one between values and types. In these traditional type systems each term lives in its own world and are unable to interact.

If we look to C style declaration we explicitly declare a value o, with a type `Object`.

```
Object o;
```

Very quickly programmers in typed languages learn that there are two sub-languages embedded into their programming language, the language of types, and the language of terms or programs.

In this world we are not able to intermix terms and types so our ability to prove interesting invariants in the type system is limited.

We can view this if we interpret our types as formulas in a logical framework. We ideally want a very expressive language and by moving to dependent types we allow our propositions to be parametrized by proofs.

They key idea is to generalize function types to dependent function spaces which allow for our co-domain to be determined by the value of the domain.

This subsumes the typical function type as a special case where `(_ : A) -> B`, and quantifiers as `(A : Type) => (_ : A) => A`.

Hubris

We now describe a small dependently typed language that is not much more than a dependently typed lambda calculus with some syntactic additions.

We will omit full formal presentation here, as we would like to not cloud the tutorial with details about the presentation of type theory and instead focus on a discussion of the implementation. The typing rules we used are adapted from the typing rules for LambdaPi in [1].

Syntax

```
data Term a = Ascribe (Term a) (Term a)        -- e : T
            | Type                             -- Type
            | Pi (Term a) (Scope () Term a)    -- (pi x : A. e)
            -- scope abstracts away variable binding in a Term
            | Var a                            -- x
            | Apply (Term a) (Term a)          -- e e'
            | Lam (Scope () Term a)            -- \x -> e
            | Let a (Term a) (Maybe (Term a)) -- let e = x ?(in t)
            deriving (Eq, Ord, Show, Read)
```

Our syntax is simple. One of the positives of implementing a dependently typed lambda calculus is we can collapse all of our syntax into a single class of pseudo-terms. If you are paying close attention this means we have completely erased the distinction between types and terms.

In this case each term can be attributed a “type” or “sort” and these terms live at different levels, but it is the type checker’s job to ensure that we only construct terms in a well formed way.

Type Checking and Evaluation

In dependently typed languages it is the case that type checking and evaluation are intermixed. In order for us to type check a term we may need to do evaluation. You will see this in the typing judgments expressed as ⇓.

We have a set of definitions that are necessary to perform type checking. The language we use is Haskell.

[1] http://www.andres-loeh.de/LambdaPi/LambdaPi.pdf

```haskell
-- The type of name we use for type checking.
type TyName = String

-- A convenience type for documentation purposes.
type Type = Term TyName

-- A term parametrized by TyName.
type TyTerm = Term TyName

-- A mapping of names to types (terms).
data Context = Context { nameMap :: M.Map TyName Type, counter :: Integer }

instance Show Context where
    show (Context nm _) = show nm

-- A context with no names, and types.
emptyContext :: Context
emptyContext = Context { nameMap = M.empty, counter = 0 }

-- A type error
data TypeErr = InferenceErr
             | NameErr String
             | MismatchErr TyTerm TyTerm
             | MiscErr String
             | UnimplementedErr String
             deriving (Show)

-- The monad we will use for type checking, we support state (StateT) and failure (Either).
type TypeCheck a = StateT Context (Either TypeErr) a
```

Now that we have the basic scaffolding we will define our type checking function. We define two versions one that checks a single term in isolation, and another that allows us to build up context between calls to the type checker.

```haskell
-- Typechecking takes a context Var -> Type, a Term to check
-- and will either return the type of the term, or an error.
typeCheck :: Context -> TyTerm -> Either TypeErr TyTerm
typeCheck ctxt tm = evalStateT (infer tm) ctxt

-- Returns both the typechecked term and the final context.
typeCheckWithContext :: Context -> TyTerm -> Either TypeErr (TyTerm, Context)
typeCheckWithContext ctxt tm = runStateT (infer tm) ctxt
```

You can see that our type checker just calls to infer. Our implementation is that of a bidirectional type checker. We have two functions, `infer` and `check`, which work together to perform type checking. The central idea is that we can mark what information is inferable or checkable.

This is different from normal type system presentations where the rules are specified in a less algorithmic manner and it is up to the implementor to correctly work out.

First we specify the checking judgment which checks a term against a given type. There are two cases.

First if we see a lambda term we check it against the given type. The second case attempts to synthesize a type for the term and check it against the given type, if these don't match the type checker will fail.

```haskell
-- We implement a bidirectional typechecker meaning we
-- have two forms of judgement.

-- This first of which `check` is a checking judgement
-- which asserts that a term checks to a certain type.

-- If we read the rules from LambdaPi we have two cases
-- that we must check, either we are executing the judgment
-- that a term must check with a certain type or we are checking
-- a lambda term annotated with a type.
check :: TyTerm -> TyTerm -> TypeCheck TyTerm
check (Lam scope) p @ (Pi t t') = do -- LAM
    x <- freshName
    bindName x t (check (instantiate1 (Var x) scope) (instantiate1 (Var x) t'))
    return p
check e t = do -- CHK
   infered <- infer e
   case infered == t of
     False -> tyError $ MismatchErr e t
     True  -> return t
```

Now the meat is in the inference judgment which attempts to synthesize a type given a term. A key thing to notice here is how we move between the checking and inference judgment. In order for us to synthesize a term's type we may need to check sub-component's types. In some places we need to call `eval`. This is because we have a dependency typed language where types maybe be terms.

```haskell
-- The second judgement is a inference judgement which
-- attempts to compute a type for term based on information.
-- We intermix these two judgements to type check a term.
infer :: TyTerm -> TypeCheck TyTerm
infer (Ascribe e p) = do -- ANN
    check p Type
    let t = eval p
    check e t
    return t
infer Type = return Type -- STAR
infer (Pi argT body) = do -- PI
  check argT Type
  let t = eval argT
  check (instantiate1 t body) Type
  return Type
infer (Var x) = lookupT x -- VAR
infer (Apply fun arg) = do -- APP
   funT <- infer fun
   (argT, body) <- case funT of
             Pi argT body -> return (argT, body)
             _ -> tyError $ MiscErr "failed in typing app"
   check arg argT
   return $ eval (instantiate1 arg body)
infer x = tyError $ MiscErr $ show x
```

Finally we can examine evaluation which is one of the simplest parts of the type checker, we implement the evaluation rules directly from [2].

The important cases are `Apply`, `Lam`, and `Pi`.

Pi types represent dependent function spaces. Pi types can contain a term which can be a program fragment that must be evaluated. In this case we enter the body of the Pi type and perform beta-reduction on the contained term.

`Apply` is function application we simply evaluate the first term to a function, and then invoke it with argument, if we can't evaluate the function we simply reduce the argument.

`Lam` we move inside the lambda abstraction and attempt to reduce the body further. The other cases are straight forward and are not of much interest.

```
-- Typechecking and evaluation are intertwined so we must
-- defined the evaluation relation here.
eval :: Term a -> Term a
eval (Ascribe e _) = eval e
eval Type = Type
eval (Pi p scope) = Pi (eval p) (toScope $ eval $ fromScope scope)
eval v @ (Var _)  = v
eval (Apply e e') =
    case eval e of
      Lam scope -> eval (instantiate1 e' scope)
      n         -> Apply n (eval e')
eval (Lam scope) = Lam (toScope $ eval $ fromScope scope)
```

[2]http://www.andres-loeh.de/LambdaPi/LambdaPi.pdf

Obviously this language is very simple and not useful for practical programming, but we hope that it makes it apparent that implementing and reasoning about a small type checker is feasible and relatively easy. This language with a couple extensions could be used as the core language for a proof assistant of dependently typed language in the style of Idris or Coq.

Appendix

Helper code for the type checker:

```haskell
-- Create a monadic type error.
tyError :: TypeErr -> TypeCheck a
tyError e = lift $ Left e

-- Generate a fresh name for type checking.
freshName :: TypeCheck TyName
freshName = do
   s <- get
   let tyName = "freshName___" ++ (show (counter s))
   put (s { counter = (counter s) + 1 })
   return tyName

-- Lookup the type corresponding to a name.
lookupT :: TyName -> TypeCheck TyTerm
lookupT n = do
    ctxt <- get
    case M.lookup n (nameMap ctxt) of
        Nothing -> tyError $ NameErr n
        Just t  -> return t

-- Bind a name `n` with type `ty` in scope for `action`.
bindName :: TyName -> Type -> TypeCheck a -> TypeCheck a
bindName n ty action = do
    ctxt <- get
    put $ ctxt { nameMap = M.insert n ty (nameMap ctxt) }
    result <- action
    put ctxt
    return result
```