Dependent Types: Easy as PIE

Work-In-Progress Project Description

Dimitrios Vytiniotis and Stephanie Weirich

University of Pennsylvania

Abstract

Dependent type systems allow for a rich set of program properties to be expressed and mechanically verified via type checking. However, despite their significant expressive power, dependent types have not yet advanced into mainstream programming languages. We believe the reason behind this omission is the large design space for dependently typed functional programming languages, and the consequent lack of experience in dependently-typed programming and language implementations. In this newly-started project, we lay out the design considerations for a general-purpose, effectful, functional, dependently-typed language, called PIE. The goal of this project is to promote dependently-typed programming to a mainstream practice.

1 INTRODUCTION

The goal of static type systems is to identify programs that contain errors. The type systems of functional languages, such as ML and Haskell, have been very successful in this respect. However, these systems can ensure only relatively weak safety properties—there is a wide range of semantic errors that these systems cannot eliminate. As a result, there is growing consensus [13, 15, 28, 27, 26, 6, 7, 20, 21, 16, 11, 2, 25, 19, 1, 22] that the right way to make static type systems more expressive in this respect is to adopt ideas from *dependent type theory* [12]. Types in dependent type systems can carry significant information about program values. For example, a datatype for network packets may describe the internal byte alignment of variable-length subfields. In such precise type systems, programmers can express many of complex properties about programs—e.g. that only packets that obey the internal alignment will ever be read from the network.

However, day-to-day programming with dependent types has yet to become mainstream. We think that the problem has been a lack of practical experience in the design space of such languages. Although there have been many previous and ongoing efforts in dependently-typed programming language design, none has stood out. Static type checking and type inference for dependent types is a technically challenging problem with a large design space that has been only moderately explored—especially in the context of a general-purpose language.

Our language, called PIE, serves as our vehicle for exploring the design space of dependently-typed languages. PIE is a general-purpose, functional programming

language with dependent types and dependent pattern matching. Important features for the design of PIE are:

- A unified term and type language. In PIE, expressions that index types are written in the computation language. This decision is important for two reasons: (i) it provides flexibility for programmers as the same expressions may be used both for dynamic execution and static verification, and (ii) it provides a uniform view of computation, so that programmers need not learn a new syntax or computational model to take advantage of expressive type checking.
- The application of a simple effect-type system [23] to ensure the soundness of PIE in the presence of computational effects. We intend PIE to be a general purpose language, but sound and decidable type checking requires that we restrict the set of terms that may appear in types to those that are effect-free. This distinction is made by the typing rules of PIE.

- The semi-automatic exploitation of knowledge gained from pattern matching during type checking. We adapt existing technology used in the implementations of *Generalized Algebraic Datatypes* (GADTs) found in ATS, Ωmega and GHC [26, 21, 17].
- A *phase distinction* between static and dynamic terms that is separate from the distinction between computations and types. Distinguishing dynamic terms (which may affect the final result of the program) from static terms (which only affect type checking) allows for efficient compilation and proof irrelevance.
- Aggressive inference, based on a combination of global and local term and type inference [24, 17]. Inference infers the descriptors of local binders, implicit arguments, and automatic coercions between types.

The design of the PIE language is still in its early stages. We have settled with respect to the first three points, and in this paper we report this initial design. However, we are considering the choices for the last two, and hence present them as future work here. We intersperse comparisons with related work throughout this document.

2 DEPENDENT TYPES FOR VERIFIED AND EFFICIENT CODE

Before describing the basic features of PIE (Section 3), we demonstrate some of the issues that must be considered in the design of a dependently-typed language with a small example of a scope-safe evaluator for the λ-calculus. This example is written in PIE, modulo some syntactic conventions. In the following, we assume that readers are familiar with de Bruijn representations of λ-terms. Using a de Bruijn representation in a typical functional language, an environment-based

```
data Nat :: * where
  Z :: Nat
  S :: Nat -> Nat

data Bool :: * where
  True  :: Bool
  False :: Bool

leq :: #Nat -> Nat -> Bool
leq Z _           = True
leq (S x0) Z      = False
leq (S x0) (S y0) = leq x0 y0

-- reflexive equality on booleans
data Eq :: Bool => Bool => * where
  Refl :: (b :: Bool) -> Eq b b
```

FIGURE 1: Natural numbers and Booleans

evaluator must perform runtime checks that ensure that indices of variables are not "out of scope" during execution. Here we demonstrate how dependent types can ensure that the lexical depth of λ-terms matches the length of the environment during evaluation, thus eliminating the need for these runtime checks. Not only does the scope-safe interpreter run faster, but we can be assured that dynamic failures will not occur.

Figure 1 displays part of the standard prelude of PIE. There, datatypes `Nat` and `Bool` are the types of natural numbers, and boolean expressions respectively. The function `leq` (less than or equal) is defined recursively on its first argument, hence the indicator # on its type. `Eq` is a dependent type that is indexed by two boolean expressions, hence its kind `(Bool => Bool => *)`, inhabited by proofs of reflexive equality of its indices.

The `Term` datatype in Figure 2 encodes terms of the object language, so that any term of type `(Term n)` may contain only variable references (de Brujn indices) smaller than `n`. The `Var` constructor takes an index `m` and a proof that `m < n`, which is expressed as `(Eq (leq (S m) n) True)`. The constructor `Lam` binds a new variable, so the body of the abstraction is one lexical level deeper.

Our evaluator models a call-by-name semantics, so environments `Env` are lists of `Thunks`, `Terms` paired with the environments they should be evaluated in. Environments are indexed by their size. In this pure λ-calculus, values `Val` include only closures, given by pairs of environments and expressions with one free index.

Now, consider the definition of a lookup function that maps a variable in the current environment to a suspended computation. Looking up a variable in the environment relies on the invariant that the index that we are accessing is less than the length of the environment (otherwise a match failure results). In ML or Haskell, the programmer must informally check that this invariant holds, but in a dependently-typed language she can *enforce* this invariant with the type of the function. To look

```
-- Terms indexed by the bound on their free indices
data Term :: Nat => * where
  Var :: (m :: Nat) -> (n :: Nat) ->
                Eq (leq (S m) n) True -> Term n
  Lam :: (n :: Nat) -> Term (S n) -> Term n
  App :: (n :: Nat) -> Term n -> Term n -> Term n

-- Suspended computation
type Thunk = {m :: Nat, Term m, Env m}

-- Environments
data Env :: Nat => * where
  Nil :: Env Z
  Cons :: (n :: Nat) -> Thunk -> Env n -> Env (S n)

-- Values are closures: terms with one free index
data Val :: * where
  Closure :: (m :: Nat) -> Env m -> Term (S m) -> Val
```

FIGURE 2: Scope-safe representation of a λ-calculus

up a variable `n` in an environment of length `m` requires a proof that `n` is less than `m`.

```
lkup :: (n :: Nat) -> (m :: Nat) -> Eq (leq (S n) m) True ->
            Env m -> Thunk
lkup (Z)    _ prf (Cons _ thunk _) = thunk
lkup (S n0) _ prf (Cons m0 _ hs)   = lkup n0 m0 prf hs
```

The `lkup` function proceeds by case analysis on the environment. However, there is no case for when the environment is `Nil` as this branch is inaccessible (it would mean that $\mathtt{m} = \mathtt{Z}$, and `(leq (S Z) Z)` is convertible to `False`). In the case of `Cons`, we have a further case analysis on `n`—whether we should return the current head of the list or another thunk stored in the tail. In the case where `n` is `S n0`, *exactly the same* proof object `prf` can be reused. The reason is that

```
(Eq (leq (S (S n0)) (S m0)) True)
```

can be converted to be equal to `(Eq (leq (S n0) m0) True)` by simply reducing the function `leq`.

Now, writing an evaluator for the well-scoped representation of λ-terms is simple:

```
eval :: (n :: Nat) -> Env n -> Term n -> Val
eval n env (Var n0 m0 prf) =
   open { n1, t1, env1 } = lkup n0 m0 prf env
   in eval n1 env1 t1
eval n env (Lam _ t1) = Closure n env t1
eval n env (App _ t1 t2) =
   let thunk = {n, t2, env} in
   case eval n env t1 of
       Closure n′ env′ t1′ ->
          eval (S n′) (Cons n′ thunk env′) t1′
```

The proof `prf` that comes packed in variable nodes is passed to `lkup`. Additionally, note that proof objects `prf` do not affect the result of the program. Hence, after type checking we can safely erase them for a more efficient implementation. Proof objects `prf` belong to an earlier *phase* of the computation. We believe that phase determination should be done manually through programmer annotations, but have not yet added this feature to PIE. This issue is one part of the design space that we intend to explore with our implementation.

Post-design-time invariants Above, environments are indexed by their size, but it is not the case in general that the properties that should be expressed by the dependent types are *a priori* known to the programmer. For example, when designing a general-purpose data type for binary trees, what property should index the tree type? The number or the elements in a tree, or perhaps the depth of the tree? Something else? Since different applications may require exposure of different properties, a practical dependently-typed language must also facilitate after-the-fact enforcing of invariants.

To make the above precise, imagine that the programmer has been given an `Env` datatype that is not indexed by size, but he nevertheless wishes to write a verified version of a `lkup` function. He may wish to give the function a type that uses an auxiliary function `length` to calculate the length of the environment.

```
lkup :: (n :: Nat) -> (env :: Env) ->
        (Eq (leq (S n) (length env)) True) -> Thunk
```

The function `lkup` may be now called in the following fashion:

```
... let env = ... potentially effectful/diverging computation ...
          y = length env
    in case (leq (S x) y) of
         True prf -> lkup x env prf
         False _ -> ...
```

where the variable `prf` binds to a proof of `Eq (leq (S x) y) True` and is part of our extended pattern matching construct that we outline in Section **??**.

To facilitate this style of programming not only we need to extend the pattern matching construct but we have to allow expressions such as `env` to appear inside types. In particular, observe that the type of `lkup` is now instantiated with a dependent type containing `env`, a potentially effectful computation. When the type checker encounters such effectful expressions during type equivalence checking, it must not reduce them further. In our particular example, if the type checker attempts to reduce `env` then it could potentially diverge. Fortunately the solution is easy: Names of effectful computations can indeed appear inside types if they are kept abstract. We return to this issue in the next section.

3 TYPE CHECKING IN PIE

A fundamental difficulty in the design of dependently-typed languages is the presence of computational effects. In the presence of computational effects deciding type equivalence becomes more complicated. Consider for example the type equivalence problem:

$$\texttt{T (y := 0; let z = ref 1 in !z)} \overset{?}{\equiv} \texttt{T (y := 1; 1)}$$

It is not clear that it is sound to consider these two types as equivalent. Additionally, in the presence of the simplest effect, non-termination, type checking can become undecidable.

To resolve this problem Epigram [13] and the proof assistant Coq [3] completely disallow any form of effects (including non-termination) from the language and admit only certain forms of recursive functions—even when these functions are not indices of some dependent type in the program. Alternatively, Ωmega [21], ATS [26], and Harper and Licata's version of DML [11] allow general recursion by completely separating the term from the type language, and allowing type level computations to be written in a way that ensures confluence and strong normalization when these type-level functions are viewed as rewrite rules. RSP1 [25] is the only, to our knowledge, monolithic dependently-typed language that allows general recursion. RSP1 imposes syntactic restrictions on the terms appearing in types, so that no reducible terms can appear inside types.

In PIE, we do not wish to sacrifice the possibility for effects and unrestricted recursion for computations, but simultaneously we want to allow (i) effect-free computations to appear inside types and (ii) even *names of effectful computations* that are treated opaquely. The last example of the previous section demonstrates the importance of (ii). We now demonstrate the importance of (i) by exhibiting how restrictive the language becomes when reducible terms are disallowed from types. Consider the definition of lkup when leq is not allowed to appear inside types: instead the programmer must encode a *relation* Leq as a datatype and pass an inhabitant of Leq to lkup, as follows:

```
data Leq :: Nat => Nat => * where
  Leq_base :: (n :: Nat) -> Leq Z n
  Leq_ind  :: (n :: Nat) -> (m :: Nat) ->
                 Leq n m -> Leq (S n) (S m)

lkup :: (n :: Nat) -> (m :: Nat) -> Leq (S n) m -> Env m -> ...
lkup = ...
```

This style results in excessive manipulation of proof objects, as it becomes the user's responsibility to pattern match against the passed proof terms, and construct others. Contrast this to the first definition of lkup, where absolutely no user intervention was necessary.

Finally we do not wish to separate the term from the type language because this results in duplication of (i) semantic rules from the compiler's side, and (ii) program

logic from the programer's side. It additionally requires familiarity of programmers with two (potentially different) styles of programming for term and type-level computations.

To satisfy all these requirements, PIE implements a type-based analysis mechanism to restrict the computations appearing inside types to be effect-free. This analysis is reminiscent of *effect-type* systems [23] (also suggested but not explored by the authors of Epigram [15]). Instead of imposing syntactic restrictions on the indices of dependent types we propose the usage of a semantic analysis, based on types, to ensure soundness and decidability of type checking. Our main typing judgement becomes:

$$\Gamma \vdash e :^{\phi} \tau$$

where ϕ is the effect of the expression e. Effects are propagated up the syntax tree during type checking and particular type and kind rules ensure that expressions that produce an effect do not appear inside types.

Variables that are λ-bound are pushed in the environment with the assumption that they stand for effect-free computations and the effect of a λ-abstraction is simply the effect of its body. The reason is that we may have to reduce the bodies of λ-abstractions in order to determine equivalence, and hence we need to know their effects. Applications at the type and term level are handled as follows:

$$\frac{\begin{array}{c}\Gamma \vdash \tau : \Pi x{:}\sigma \,.\, \kappa \\ \Gamma \vdash e :^{\varepsilon} \sigma\end{array}}{\Gamma \vdash \tau\ e : \kappa\{e/x\}}\ \text{T-APP} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 :^{\phi_1} \Pi x{:}\tau_1 \,.\, \tau_2 \\ \Gamma \vdash e_2 :^{\phi_2} \tau_1 \quad \Gamma \vdash \tau_2\{e_2/x\} : \star\end{array}}{\Gamma \vdash e_1\ e_2 :^{\phi_1 \sqcup \phi_2} \tau_2\{e_2/x\}}\ \text{APP}$$

where ε indicates the absense of effects. In rule T-APP, an application of a type to an expression is well-formed only if the expression yields no effect. In rule APP, the effect of the expression $e_1\ e_2$ is just the union of the effects of each expression, denoted with $\phi_1 \sqcup \phi_2$. Note that we have to check that the returned type is well formed. These rules allow a function to be applied to a potentially effectful argument, as long as that argument does not appear in the return type.

For recursive definitions (such as top-level definitions, or `let`-bound definitions), after we type check and produce an effect ϕ for the body of the definition, if that effect is ε then we can push the binder and its definition in the environment, so that the definition may be later used during type equivalence checking. If, on the contrary, we detect a non-trivial effect ϕ, we push the binder *without its definition*, so that it may still appear inside types, but will be treated opaquely. We thus ensure sound and decidable type checking.

In what follows we present a specialization of our approach to non-termination as the sole effect in the language, although the above discussion applies to more general settings as well.

Staged types and effectful coercions for termination checking In the case where non-termination is the effect of interest, ϕ can either be $\Uparrow$ or $\Downarrow$ ($\Downarrow$ can be viewed as equivalent to ε). We indicate with $\Uparrow$ the possibility of divergence, while with $\Downarrow$ the provable convergence of a term.

The actual type-based termination checking employed in PIE is based on the work of Barthe *et al.* [4]. The key element of termination checking is that types are internally decorated with *stage annotations*. Stages s, are constructed from stage variables ι, a "next-stage" operator $s+$, and a limit stage ∞. Intuitively, stage annotations keep track of the approximations of datatypes by their unfoldings, and give rise to a subsumption relation, for example:

$$\texttt{Term}^{s}\, n \prec \texttt{Term}^{s+}\, n \quad \textbf{and} \quad \texttt{Term}^{s}\, n \prec \texttt{Term}^{\infty}\, n$$

Recursion indicators on types, such as `#Nat` are used by this termination checking mechanism. For example, consider again the function `leq` from the previous section. Initially, a fresh stage variable ι is picked and we get the following problem (non-annotated types enter environment with a stage annotation of ∞):

$$\begin{array}{l} \texttt{leq:Nat}^{\iota} \rightarrow \texttt{Nat}^{\infty} \rightarrow \texttt{Bool}^{\infty} \vdash \\ \qquad \texttt{\textbackslash x y.(}\ldots\texttt{leq}\ \textbf{body}\ldots\texttt{)} :^{\Downarrow} \texttt{Nat}^{\iota+} \rightarrow \texttt{Nat}^{\infty} \rightarrow \texttt{Bool}^{\infty} \end{array}$$

achieving thus an inductive proof of the termination of `leq`. Pattern matching on `x` of type $\texttt{Nat}^{\iota+}$ in the case of the `S` constructor, results in `x0` getting the type $\texttt{Nat}^{\iota}$, and now `leq` can indeed be called on `x0`.

Provably terminating programs are those that typecheck without violating the subsumption relation, yielded by the stage ordering. These programs, such as `leq` above, produce the $\Downarrow$ effect. However, when a stage subsumption is violated, we can view it as an *effectful coercion*, which outputs $\Uparrow$. Effects produced by stage subsumption checks are propagated up the syntax tree to our main typing relation.

Soundness and termination checking Soundness, in our setting, implies that no branch which was statically discovered to be inaccessible may be accessed operationally. Lack of soundness may result in unpredictable pattern match failures, and diminishes the value of using dependent types for program verification alltoghether.

In the presence of general recursion, objects inhabiting types that stand for propositions may diverge. Hence, if the semantics of our interpreter is call-by-value, we can only get a soundness modulo-termination guarantee: If a program terminates then no statically inaccessible branch can be reached.

On the other hand, in a call-by-name semantics "proof objects" may not be evaluated. This results in lack of soundness. For example the following code leads to a pattern match failure in the definition of `lkup`:

```
foo = lkup (S Z) Z diverge Nil
```

Operationally (in a call-by-name semantics) `diverge` is not evaluated, but `lkup` does not include a case for `Nil` environments, as that branch is statically inacessible. Consequently the program crashes with a pattern match failure.

Fortunately termination checking comes to the rescue. We could require in the definition of `lkup` that the argument corresponding to the particular equality proof is terminating, as follows:

```
lkup :: ... -> Terminating (Eq (leq (S n) m) True) -> ...
```

The type checker in this case has to enforce that the argument is indeed a terminating object. This requires an easy modification to the APP rule to enforce the $\Downarrow$ effect when the argument lives in `Terminating`, but nothing changes in the way that the body of `consumer` type checks (as λ-bound variables enter the environment with the assumption that they terminate anyway). This example also indicates that a precise annotation of types with effects may be desirable, an issue that we return to in Section 3.1.

Termination checking alone is not enough to ensure soundness, since in general soundness must also involve coverage checking when constructing proof terms: We must ensure that functions that return proof objects cover all possible inputs and are terminating—only then can such proofs be considered constructively valid. The following variation of the previous example demonstrates a (slightly contrived) situation where coverage checking must be applied to ensure soundness:

```
bogus :: (n,m::#Nat) -> Eq (leq n m) True
bogus = case n of { }

foo = lkup (S Z) Z (bogus (S Z) Z) Nil
```

Notice that `bogus` does not cover all its inputs, and a coverage check could have detected the problem.

3.1 Precise effect analysis

Types such as the improved type of the `lkup` function above provide a precise view of the effects of computations. Our current implementation only can assign types such as

$$\Gamma \vdash e :^{\phi} \texttt{Nat} \rightarrow \texttt{Nat}$$

However, a general effect system may have judgements of the form:

$$\Gamma \vdash e :^{\phi} (\mathtt{Nat}^{\phi_1} \rightarrow \mathtt{Nat}^{\phi_2})$$

with the semantics that, upon evaluation, e produces the effect ϕ and reduces potentially to a value, which, when it will accept an argument with effect ϕ_1 that reduces potentially to a `Nat` value, it will return an expression that upon evaluation will produce an effect ϕ_2 and yield a `Nat` value. Contrast this to the first type which indicates the possibility of generating an effect ϕ either during evaluation of e or potential application of e to some argument. Notice that such an elaborate effect

system has to be tied to the operational semantics of choice (and here we assume call-by-name).

Generalizing this idea, we are considering the trade-offs of introducing a particular *monad* $\mathtt{T}^{\phi}\mathtt{::}\star \rightarrow \star$, indexed by effects. In the case where the effect of interest is non-termination, we let the type $\mathtt{T}^{\Uparrow}\,\sigma$ be isomorphic to σ, since we wish to write most of our code without contaminating types with the indexed monad constructor `T`. This elaborate analysis would allow programmers to write functions with types such as:

$$\forall f\,.\,\mathtt{T}^{f}\,\mathtt{Nat} \rightarrow \mathtt{T}^{f}\,\mathtt{Nat} \quad \text{or} \quad \mathtt{T}^{\Uparrow}(\mathtt{Nat} \rightarrow \mathtt{T}^{\Downarrow}\,\mathtt{Nat})$$

The first type would be the type of a potentially diverging computation, that, once evaluated, could accept a `Nat` with any effect, and return an expression that produced the same effect. The second type would be the type of a potentially diverging computation, that, once evaluated, could accept a (potentially diverging) `Nat` and return an always terminating `Nat` (i.e. it could not evaluate its argument).

Our non-standard choice of letting $\mathtt{T}^{\Uparrow}\,\sigma$ be isomorphic to σ is driven by practical considerations. If we were designing a language intended for theorem proving, we'd prefer to be total and pollute only a few non-pure terms with a *partiality monad*. However, in the context of a practical programming language we might wish to be unrestricted and non-pure, with only the types of total proof terms or dependent type indices living inside our indexed monad.

3.2 Type equivalence and pattern matching

Type equivalence lies in the core of the type checking engine of every dependently typed system. In our system it is intimately connected with pattern matching, since it must take into account the knowledge gained via pattern matching, as the examples from Section 2 suggest.

Program equivalence, and provable equivalence The notion of type and term equivalence that we use in PIE is full β-convertibility (also referred to as *intensional equality*) taking into account the definitions in the environment. It is an invariant, due to our effect analysis system (which includes termination checking), that β-reductions performed at compile time terminate.

However, β-convertibility is often not enough. For example, the type checker cannot establish the equivalence:

```
Eq (leq n n) True ≡ Eq True True
```

because `leq n n` is not convertible to `True`, although it is *provably equal to* `True`. For such examples we plan to experiment with using proof terms of equivalence as coercions, whenever automatic convertibility fails. Special care has to be taken to prevent such coercions from having runtime overhead.

Dependent pattern matching The information that we gain from pattern matching should be automatically used when checking the right hand side of pattern match clauses. This automation is provided in a restricted form in some systems. For example, Coq does not allow this knowledge gain to be used to refine the types of pre-existing variables in the environment. For example, the straightforward translation of the `lkup` function in the system Coq:

```
Fixpoint lkup (A:Set) (n:nat) (m:nat) (r:leq (S n) m = true)
```

```
  (l: List A m)  {struct n} :A :=
   match (n, l) with
    | (O, Cons A x l0) => x
    | (S n0, Cons m0 x l0) => lkup n0 r l0
    | (n0, Nil) => ...
  end.
```

fails to type check! The error message that Coq provides refers to the second branch of the pattern match.

```
The term "r" has type "le (S n) m = true" while it
is expected to have type "le (S n1) ?92 = true"
```

The reason is that the type of the proof object `r` has not been appropriately "refined", as happens in PIE. In Coq, the tactic language makes up for the weak typing rule for pattern matching. The most efficient way to write `lkup` is interactively—but the proof script bears little resemblance to a functional program.

PIE uses ideas originating in Coquand's pattern matching with dependent types [8], also found in recent implementations of GADTs in Haskell [17]. In particular, pattern matching a scrutinee against a pattern introduces equalities, which we can simplify using unification. In the case of higher-order dependently-typed constraints [9, 10] we cannot compute most general unifiers. However, it is possible to get a set of equations in solved form, in the form of a *pre-unifier* that will capture all the "easy" constraints. Such easy equations can then be automatically used during equivalence checking. If no such pre-unifier exists we have detected an inaccessible branch. A subtle point is that we have to make sure that the substitutions we create are well-typed in the sense that every variable in the domain has the same type as its image. In the general case of unification with dependent types (and especially in our case, where we "skip" hard unification problems) the process of splitting an equation may create ill-typed terms, as Elliot shows [9].

In the case where a pre-unifier is computed, but a residual set of equalities remains, the branch may or may not be dead code, but we cannot determine this. In such a case one can decide to simply reject the program. However, for such cases, PIE permits the binding of extra pattern variables, representing introduced equalities. In particular we may have pattern matches of the form:

```
... case x of
      C y1 ... ym crs -> ...
```

Assuming that `x` is of type $\mathtt{T}\ e_1 \cdots e_n$ and the constructor `C` applied to patterns `y1...ym` yields $\mathtt{T}\ u_1 \ldots u_n$, the variable `crs` binds to a term that represents the

equivalence[1] $\mathtt{x}^{\mathtt{T}\ e_1 \ldots e_n} \equiv (\mathtt{C}\ \mathtt{y1} \ldots \mathtt{ym})^{\mathtt{T}\ u_1 \ldots u_n}$, and can hence be used in the right-hand side of the pattern.

A significant difference between pattern matching with GADTs and pattern matching with dependent types is that the equalities introduced by the type of the scrutinee compared to the type of the pattern are not enough. For example, typechecking the `lkup` function from the introduction crucially relies on refining the argument `n` to `Z` or `S n0`, and not on refining its type. This is not hard to support, as the effect analysis comes to the rescue: Any term-level equality can still be used, as long as the effects of the participating terms are ε. We have noted earlier that λ-bound variables enter the environment with the assumption that they represent effect-free computations, and hence in the `lkup` example `n` can indeed be refined in the two cases to either `Z` or `S n0`.

4 FUTURE WORK

The PIE language is still in its early stages. We summarize here the areas that we plan to further investigatge.

Improving termination checking and general effect analysis Although our basic approach to termination setting is settled, there are many issues to be resolved: For example, there is no satisfactory treatment in the related work for mutually recursive datatypes or function definitions. Additionally combining stage variables with polymorphic types and user type annotations is something that needs to be investigated more. Finally we plan to include more general effects, such as references or I/O, to make PIE more appealing.

Incorporation of provable equality To gain the full power of dependent types, since type checking automations are resticted by nature, we plan to give the programmer the ability to construct and decompose (heterogeneous) equality proof terms. Our intention is to integrate this in a "functional" way to the language, avoiding the introduction of proof scripts as parts of programs. However we wish to keep the operational overhead of such functions on proof terms as low as possible, as the next item indicates.

Erasure of proof terms and indices Proof objects and functions on them (i.e. proofs) should impose no operational overhead. However, in a language where ordinary term expressions may index types, the distinction between terms that can be erased at runtime (static), and terms that should be evaluated (dynamic) may

[1]This is an example of heterogeneous equality, or McBride's "John Major" equivalence [14],

where the terms are annotated with their types, which in turn may be different.

be harder to enforce. To circumvent these difficulties, PIE will provide syntax that allows programmers to specify this distinction. An alternative approach that we consider is to separate proof terms using a special kind `Prop`, similarly to the system Coq, so that all expressions that have types that live in `Prop` may be erased at runtime, without affecting the operational behavior of the program. However, notice that the (static) indices of length-indexed environments, are ordinary `Nats` that live in $\star$, and hence a sub-kinding relation $\star \sqsubseteq \texttt{Prop}$ will be required in order to achieve erasure of such arguments. Term/type inference should be able to infer some of these static terms, but notice that we should not exclude term inference of dynamic terms^2. We are currently evaluating these possibilities for PIE.

Term and type inference The current implementation of PIE offers only a limited form of local type inference [18], which relies on user annotations. However, from the examples of Section 2 we observe that in many cases the indices of datatypes could be inferred and need not be passed explicitly to constructors such as `Cons` or functions. An argument of type `Env` n is enough to let the type checker reconstruct the actual index n. However, it may not always be clear which arguments should be implicit. Therefore, we propose the following distinction, similar to one found in the Twelf system [?]. In a typing annotation, all variables that are not explicitly bound are implicit and need not be supplied when the function is called.

Additionally, when we add parametric polymorphism to PIE, we will need to pass dependent sums and products as first-class values and instantiate polymorphic variables with such types. Type inference for these features is undecidable, so we envision a mixture of type checking and type inference. Moreover, the type checker may support some automatic coercions: For example a value of type: `List` n could be viewed as a value of the dependent product `{n::Nat, List n}`.

5 CONCLUSIONS

We have presented the initial considerations for the design and implementation of a new dependently-typed programming language, PIE. The design space is very broad, especially with respect to pattern matching and equivalence checking, so we believe that the experience from our implementation will be essential, both in identifying new paths to explore, and in evaluating our approach.

[2] In support for this opinion, observe that, for example, type classes in Haskell are dynamic terms (dictionaries) but still inferred by the type checker.

REFERENCES

[1] Lars Birkedal Aleksandar Nanevski, Greg Morrisett. Polymorphism and separation in hoare type theory. In *Proceedings of International Conference on Functional Programming (ICFP '06)*, Portland, Oregon, September 2006.

[2] Lennart Augustsson. Cayenne–a language with dependent types. In *Third ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 239–250, September 1998.

[3] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203, INRIA-Rocquencourt, CNRS and ENS Lyon, 1997.

[4] G. Barthe, B. Grégoire, and F. Pastawski. Type-based termination of recursive definitions in the calculus of inductive constructions. In *Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, November 2006. To appear.

[5] Richard S. Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[6] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Tallinn, Estonia, September 2005. To appear.

[7] James Cheney and Ralf Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.

[8] Thierry Coquand. Pattern matching with dependent types. In B. Nordstrm, K. Petersson, and G. Plotkin, editors, *Informal Proceedings Workshop on Types for Proofs and Programs, Bstad, Sweden, 8–12 June 1992*, pages 71–84. Dept. of Computing Science, Chalmers Univ. of Technology and Gteborg Univ., 1992.

[9] C. M. Elliott. Higher-order unification with dependent function types. In *RTA-89: Proceedings of the 3rd international conference on Rewriting Techniques and Applications*, pages 121–136, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[10] Gérard P. Huet. Unification in typed lambda calculus. In *Proceedings of the Symposium on Lambda-Calculus and Computer Science Theory*, pages 192–212, London,

UK, 1975. Springer-Verlag.

[11] Daniel R. Licata and Robert Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University Department of Computer Science, 2005.

[12] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[13] C McBride and J McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[14] Conor McBride. Elimination with a motive. In *TYPES '00: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 197–216, London, UK, 2002. Springer-Verlag.

[15] James Mckinna. Why dependent types matter. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–1, New York, NY, USA, 2006. ACM Press.

[16] Emir Pašalić, Jeremy Siek, and Walid Taha. Concoqtion: Mixing depedendent types and Hindley-Milner type inference. Draft available at http://www.metaocaml.org/concoqtion/, 2006.

[17] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Confer-*

ence on Functional Programming (ICFP), Portland, OR, USA, September 2006.

[18] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–265, San Diego, CA, 1998.

[19] Carsten Schürmann, Richard Fontana, and Yu Liao. Delphin: Functional programming with deductive systems. Available from http://cs-www.cs.yale.edu/homes/carsten/papers/delphin.pdf, 2002.

[20] Tim Sheard. Languages of the future. In *ACM Conference on Object Orientated Programming Systems, Languages and Applicatioons (OOPSLA'04)*, 2004.

[21] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Proc 4th International Workshop on Logical Frameworks and Meta-languages (LFM'04), Cork*, July 2004.

[22] Martin Sulzmann, Jeremy Wazny, and Peter Stuckey. A framework for extended algebraic data types. In *FLOPS 2006*, 2006.

[23] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[24] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. Boxy type inference for higher-rank types and impredicativity. In *International Conference on Functional Programming (ICFP)*, Portland, OR, USA, September 2006.

[25] Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages

268–279, New York, NY, USA, 2005. ACM Press.

[26] Hongwei Xi. Dependent types for practical programming via applied type system. Available from http://www.cs.bu.edu/ hwxi/ATS/ATS.html, September 2004.

[27] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235. ACM Press, 2003.

[28] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 214–227, San Antonio, Texas, January 1999.