An Essay on Dependent Types

Advanced Topics in Programming Languages

Pattern matching in languages with dependent types must be able to retain consistency between dependencies within different terms in a pattern. As these dependencies introduce information that is only useful at compile-time, we should also figure out how to eliminate any runtime agnostic computations in order to improve program efficiency.

In his PhD thesis *Towards a practical programming language based on dependent type theory*, Norell defines Agda, a dependently typed functional programming language that can perform pattern matching over inductive families. They aim to reduce the computation needed to match at runtime by marking deducible patterns as inaccessible. Tejiščák argues that inaccessibility is insufficient to reduce dependent types' overhead to acceptable levels. They define a dependently typed calculus with pattern matching alongside an algorithm for infering erasure, the property for a term to be unnecessary at runtime. Their evaluation shows how using erasure makes both the compiler and the resulting program run faster. Through an example, in this essay I show how their respective pattern matching algorithms reduce runtime overhead.

Dependent pattern matching

In a dependently typed calculus, we can form types from terms. Take the following Agda data type for vectors and their corresponding append function with all indices made explicit:

```
data Vec (A : Set) : (n : ℕ) → Set where
  []   : Vec A zero
  cons : (n : ℕ) → (x : A) → (xs : Vec A n) → Vec A (suc n)

app : {A : Set} -> (m n : ℕ) → Vec A m → Vec A n → Vec A (m + n)
app m  n []            ys = ys
app m' n (cons m x xs) ys = cons (m + n) x (app m n xs ys)
```

We have made the `Vec`s length indices `m` and `n` explicit. When matching on the first pattern, the value of `m` can be deduced from the first vector `[]`. This first pattern could therefore be written as `zero n [] ys` without loss of generality. Thus, we can see that when pattern matching on dependent types, patterns can be iteratively refined as we glean more information from the dependencies.

We can similarly resolve the second pattern. Matching on the first vector tells us that the first argument must be `suc m`, by the definition of `cons`. If we had instead done the pattern matching based on the first argument `suc m`, then we would have been able to deduce that the first vector's length index would have been `m`. This shows how dependent pattern matching can be non-linear since depending on the order in which we match the patterns we information in a different order. The non-linearity can be traced to the fact that we could have multiple instances of the same term (i.e. `m`) on the left-hand side of the pattern. Using the same algorithms as we would for a standard functional language would therefore not work as these can only have a single instance of each variable in a pattern's left-hand side.

Inaccessible Patterns

Norell introduces the notion of inaccessible patterns, i.e. those that arise from the instantiation of indices, like `zero` and `suc m` above. Inaccessible patterns only provide information useful for compile-time type checking and so they can be ignored at runtime. Moreover, we note that repeated pattern variables only arise within these inaccessible patterns. Thus, we have a well-formed linear pattern if

we only focus on the accessible patterns. We augment the definition of app to explicitly show the inaccessible patterns marked by a . in Agda:

```
app : {A : Set} -> (m n : ℕ) → Vec A m → Vec A n → Vec A (m + n)
app .zero    n []            ys             = ys
app .(suc m) n (cons m x xs) ys = cons (m + n) x (app m n xs ys)
```

The pattern matching algorithm

A dependent pattern matching algorithm needs to ensure that all patterns are well typed (i.e. that all patterns have the type specified in the function declaration), and to resolve all inaccessible patterns. Norell defines an algorithm that does just that. It takes a configuration of the form $\langle \bar{p}, \sigma : \Delta \to \Gamma \rangle$ where Γ are the function's argument types, σ is the mapping of pattern variables to terms and $\bar{p}$ is the user-defined patterns matching those in σ. The algorithm then iteratively refines this configuration until only variables are left in the pattern. To see how this is done we use the first clause of our running example, where $\Gamma = (mn : \mathbb{N})(xs : \mathsf{Vec}\ A\ m)(ys : \mathsf{Vec}\ A\ n)$:

$$\langle m\ n\ []\ ys,\ \ m; n; xs; ys : \Gamma \to \Gamma \rangle$$

To refine this configuration, we identify a constructor pattern to ensure that it can legally be used to instantiate a variable of the corresponding type. In this case we have no choice but to check [] which should have the type in our context Γ as $(xs : \mathsf{Vec}\ A\ m)$. The output of this Split operation is a new mapping that instantiates the variable alongside the necessary substitutions so that maintin its well-typedness. This, is where we can further instantiate our dependencies, producing the inaccessible patterns. For our clause above Split requires the following:

$$xs : \mathsf{Vec}\ A\ m \qquad \mathsf{Vec}_A : \mathbb{N} \to \mathsf{Set} \qquad []_A : \mathsf{Vec}\ A\ \mathsf{zero}$$

$$\mathrm{UNIFY}(m = \mathsf{zero} : \mathbb{N}) \Rightarrow [m = .\mathsf{zero}]$$

$$\delta = [m := .\mathsf{zero}]; n; [xs := []]; ys$$

$$\mathrm{SPLIT}(\bar{p}_1; []; \bar{p}_2, \Gamma) \Rightarrow \delta : (n : \mathbb{N})(ys : \mathsf{Vec}\ A\ n) \to \Gamma$$

This new pattern mapping δ is then matched against our pattern and we see that it indeed matches with $m\ n\ []\ ys$ albeit with the substitution $m := .\mathsf{zero}$. Here we see that we have identified m as an inaccessible pattern. The resulting configuration is:

$$\langle m\ n\ xs\ ys,\ \ .\mathsf{zero}; n; []; ys : (n : \mathbb{N})(ys : \mathsf{Vec}\ A\ n) \to \Gamma \rangle$$

Seeing as our pattern is now comprised entirely of variables, we are done. Perfoming this procedure on the second clause results in a similar result where the first argument is identified to be the inaccessible pattern $.\mathsf{suc}\ m$:

$$\langle m\ n\ xs\ ys,\ \ .\mathsf{suc}\ m'; n; \mathsf{cons}\ m'\ x\ xs'; ys : (m'\ n : \mathbb{N})(x : A)(xs' : \mathsf{Vec}\ A\ m')(ys : \mathsf{Vec}\ A\ n) \to \Gamma \rangle$$

So just how non-dependently typed programming languages discard type annotations after the first stages of compilation, Agda can discard the extra compile-time information afforded by the inaccessible patterns.

Type erasure

Tejiščák notices that naively implementing dependent types changes the asymptotic complexity of our programs. Sticking with the same example, we notice that in the second pattern of our append function, we are having to perform unary arithmetic to satisfy the length indices of the right hand side:

```
app .(suc m) n (cons m x xs) ys = cons (m + n) x (app m n xs ys)
```

As unary arithmetic is linear in complexity, this raises our append from being linear in the number of elements, to being quadratic. Thus, we see that using inaccessible patterns is insufficient. To preserve the complexity of our programs, Tejiščák proposes an algorithm to identify and erase from our pattern matching clauses the terms that only serve a purpose at compile-time.

They introduce erasure annotations which we attach to any name binding and any function or constructor application on throughout the entire program. I show here how they would attach for the append function:

$$\mathsf{app} : \{A :_1 \mathsf{Set}\} \to (m\ n :_{2,3} \mathbb{N}) \to (xs :_4 \mathsf{Vec}\ A\ m) \to (ys :_5 \mathsf{Vec}\ A\ n) \to \mathsf{Vec}\ A\ (m + n)$$
$$\mathsf{app}\ {}_6\ .0\ {}_7\ n\ {}_8\ []\ {}_9\ ys = ys$$
$$\mathsf{app}\ {}_{10}\ .(\mathsf{suc}\ {}_{11}\ m)\ {}_{12}\ n\ {}_{13}\ (\mathsf{cons}\ {}_{14}\ m\ {}_{15}\ x\ {}_{16}\ xs)\ {}_{17}\ ys = \ldots$$

These type annotations have been numbered with erasure variables. While the algorithm runs, these variables have their dependencies tracked and get evaluated such that they collapse to either E or R. These respectively correspond to a binding or application that can be erased, or one that should be retained at runtime.

The erasure algorithm

In their paper, Tejiščák define $\mathsf{TT}_\star$, a dependently typed calculus with the aforementioned erasure annotations. To transfrom user code that omits these annotations to code that is fully annotated, they describe an algorithm as a pipeline of code transformations.

We first transform the user code into the calculus, adding all unknown erasure annotations ($\bullet$):

$$\mathsf{app} : \{A :_\bullet \mathsf{Set}\} \to (m\ n :_{\bullet,\bullet} \mathbb{N}) \to (xs :_\bullet \mathsf{Vec}\ A\ m) \to (ys :_\bullet \mathsf{Vec}\ A\ n) \to \mathsf{Vec}\ A\ (m + n)$$
$$\mathsf{app}\ {}_\bullet\ .0\ {}_\bullet\ n\ {}_\bullet\ []\ {}_\bullet\ ys = ys$$
$$\mathsf{app}\ {}_\bullet\ .(\mathsf{suc}\ {}_\bullet\ m)\ {}_\bullet\ n\ {}_\bullet\ (\mathsf{cons}\ {}_\bullet\ m\ {}_\bullet\ x\ {}_\bullet\ xs)\ {}_\bullet\ ys$$
$$= (\mathsf{cons}\ {}_\bullet\ (m + n)\ {}_\bullet\ x\ {}_\bullet\ (\mathsf{app}\ {}_\bullet\ m\ {}_\bullet\ n\ {}_\bullet\ xs\ {}_\bullet\ ys))$$

Next, we replace all unknown erasure annotation with unique erasure variables:

$$\mathsf{app} : \{A :_1 \mathsf{Set}\} \to (m\ n :_{2,3} \mathbb{N}) \to (xs :_4 \mathsf{Vec}\ A\ m) \to (ys :_5 \mathsf{Vec}\ A\ n) \to \mathsf{Vec}\ A\ (m + n)$$
$$\mathsf{app}\ {}_6\ .0\ {}_7\ n\ {}_8\ []\ {}_9\ ys = ys$$
$$\mathsf{app}\ {}_{10}\ .(\mathsf{suc}\ {}_{11}\ m)\ {}_{12}\ n\ {}_{13}\ (\mathsf{cons}\ {}_{14}\ m\ {}_{15}\ x\ {}_{16}\ xs)\ {}_{17}\ ys$$
$$= (\mathsf{cons}\ {}_{18}\ (m + n)\ {}_{19}\ x\ {}_{20}\ (\mathsf{app}\ {}_{21}\ m\ {}_{22}\ n\ {}_{23}\ xs\ {}_{23}\ ys))$$

A set of constraint generating rules act on this program to substantiate the erasure annotations. These constraints are of the form $G \to r$ saying that if all annotations in G are retained then so must r be retained. For example, we have the erasure inference rule:

$$\frac{\Gamma \vdash F :_G (n :_t \sigma) \to \rho \mid \Delta \qquad \Gamma \vdash X :_{G \wedge s} \sigma \mid \Sigma}{\Gamma \vdash F\ {}_s\ X :_G \rho[n \mapsto X] \mid \Delta \cup \Sigma \cup t \leftrightarrow s}\ \text{APP}$$

Without explaining the specific details, applying this on the annotation $\mathsf{app}\ {}_{21}\ m$ produces the constraints $\{2 \to 21\} \cup \{21 \to 2\}$, where 2 is the corresponding type annotation at the function's head. The resulting set of constraints is then solved as a logic program containing only Horn clauses that can be solved in linear time, yielding the following:

$$\mathsf{app} : \{A :_R \mathsf{Set}\} \to (m\ n :_{E,E} \mathbb{N}) \to (xs :_R \mathsf{Vec}\ A\ m) \to (ys :_R \mathsf{Vec}\ A\ n) \to \mathsf{Vec}\ A\ (m + n)$$
$$\mathsf{app}\ {}_E\ .0\ {}_E\ n\ {}_R\ []\ {}_R\ ys = ys$$
$$\mathsf{app}\ {}_E\ .(\mathsf{suc}\ {}_E\ m)\ {}_R\ n\ {}_R\ (\mathsf{cons}\ {}_E\ m\ {}_R\ x\ {}_R\ xs)\ {}_R\ ys$$
$$= (\mathsf{cons}\ {}_E\ (m + n)\ {}_R\ x\ {}_R\ (\mathsf{app}\ {}_E\ m\ {}_E\ n\ {}_R\ xs\ {}_R\ ys))$$

Finally, type and erasure checks are performed to ensure the dependently typed program is consistent. Then, defining the erasure translation as $\langle\bullet\rangle$, the terms are transformed by recursively applying erasure rules which include $\langle F\ {}_E\ X\rangle = \langle F\rangle$ and $\langle F\ {}_R\ X\rangle = \langle F\rangle\ \langle X\rangle$. Our resulting program is now:

$$\mathsf{app} : \{A : \mathsf{Set}\} \to (xs : \mathsf{Vec}\ A) \to (ys : \mathsf{Vec}\ A) \to \mathsf{Vec}\ A$$
$$\mathsf{app}\ \ []\ ys = ys$$
$$\mathsf{app}\ \ (\mathsf{cons}\ x\ xs)\ ys = (\mathsf{cons}\ x\ (\mathsf{app}\ xs\ ys))$$

This is exactly the program that we would have written in a standard functional programming language. In particular, we note how the expensive unary addition of the output `cons` is gone. Thanks to erasure, we are able to use the strictness of a dependent type system that ensures our `Vecs` all have the appropriate length, without sacrificing performance. Whereas inaccessible patterns saved the runtime from checking *some* compile-time information, erasure allows us to eliminate it altogether.

Does erasure measure up?

The $\mathsf{TT}_\star$ calculus has had the benefit of time, building up on recent research discussing the erasabilty of dependent types. We have shown theoretically how it greatly reduces the computation required by programs, and Tejiščák further shows empirically through a series of benchmarks how their compiler produces code that is asymptotically faster. In most of these benchmarks, more than half of the erasure annotations are marked E showing how powerful this analysis is in reducing runtime work.

However, they do not compare their system against other established dependently typed languages such as Agda and Idris that do perform some irrelevance analysis to reduce code size. They do argue, however, that irrelevance in these languages is too loose, ignoring type indices that should be enforced. For example, our erasure analysis removes all numerical indices from the `app` function after it has been type checked. If we were to mark the index of `Vec (A : Set) : .(n : ℕ) → Set` as irrelevant in Agda, then we would lose the guarantee that the output of `app` is of the correct length. As erasure is done dependently on the entire program we are ensured the consistency imposed by the type dependencies. Therefore, we would expect that a language employing the $\mathsf{TT}_\star$ calculus should perform better than an Agda program that enforces the indices properly (i.e. doesn't mark them as irrelevant).

In conclusion, while powerful, the "dependent" in dependent types proves to be quite costly if not treated properly. Ideally, like simple types, we want dependent types to only be used at compile-time in order to not slow down the resulting program. Norell identifies that dependencies in the indices of dependent types can be ignored for pattern matching at runtime. However, in many cases, the resulting program still retains the computation of indices in a pattern match clause's output, potentially making it asymptotically more expensive. Tejiščák successfully combats this issue through their erasure annotations, allowing for compiled programs to preserve their asymptotic complexity.