Builtin Types viewed as Inductive Families

Guillaume Allais

January 5, 2023

Abstract

State of the art optimisation passes for dependently typed languages can help erase the redundant information typical of invariant-rich data structures and programs. These automated processes do not dramatically change the *structure* of the data, even though more efficient representations could be available.

Using Quantitative Type Theory, we demonstrate how to define an invariant-rich, typechecking time data structure packing an efficient runtime representation together with runtime irrelevant invariants. The compiler can then aggressively erase all such invariants during compilation.

Unlike other approaches, the complexity of the resulting representation is entirely predictable, we do not require both representations to have the same structure, and yet we are able to seamlessly program as if we were using the high-level structure.

1 Introduction

Dependently typed languages have empowered users to precisely describe their domain of discourse by using inductive families [Dyb94]. Programmers can bake crucial invariants directly into their definitions thus refining both their functions' inputs and outputs. The constrained inputs allow them to only consider the relevant cases during pattern matching, while the refined outputs guarantee that client code can safely rely on the invariants being maintained. This programming style is dubbed 'correct by construction'.

However, relying on inductive families can have a non-negligible runtime cost if the host language is compiling them naïvely. And even state of the art optimisation passes for dependently typed languages cannot make miracles: if the source code is not efficient, the executable will not be either.

A state of the art compiler will for instance successfully compile length-indexed lists to mere lists thus reducing the space complexity from quadratic to linear in the size of the list. But, confronted with a list of booleans whose length is statically known to be less than 64, it will fail to pack it into a single machine word thus spending linear space when constant would have sufficed.

In section 2, we will look at an optimisation example that highlights both the strengths and the limitations of the current state of the art when it comes to removing the runtime overheads potentially incurred by using inductive families.

In section 3 we will give a quick introduction to Quantitative Type Theory, the expressive language that grants programmers the ability to have both strong invariants and, reliably, a very efficient runtime representation.

In section 4 we will look at an inductive family that we use in a performance-critical way in the TypOS project [AAM+22] and whose compilation suffers from the limitations highlighted in section 2. Our current and unsatisfactory approach is to rely on the safe and convenient inductive family when experimenting in Agda and then replace it with an unsafe but vastly more efficient representation in our actual Haskell implementation.

Finally in section 5, we will study the actual implementation of our efficient and invariant-rich solution implemented in Idris 2. We will also demonstrate that we can recover almost all the conveniences of programming with inductive families thanks to smart constructors and views.

2 An Optimisation Example

The prototypical examples of the naïve compilation of inductive families being inefficient are probably the types of vectors (`Vect`) and finite numbers (`Fin`). Their interplay is demonstrated by the `lookup` function. Let us study this example and how successive optimisation passes can, in this instance, get rid of the overhead introduced by using indexed families over plain data.

A vector is a length-indexed list. The type `Vect` is parameterised by the type of values it stores and indexed over a natural number corresponding to its length. More concretely, its `Nil` constructor builds an empty vector of size `Z` (i.e. zero), and its `(::)` (pronounced 'cons') constructor combines a value of type `a` (the head) and a subvector of size `n` (the tail) to build a vector of size (`S n`) (i.e. successor of n).

```
data Vect : Nat -> Type -> Type where
  Nil : Vect Z a
  (::) : a -> Vect n a -> Vect (S n) a
```

The size `n` is not explicitly bound in the type of `(::)`. In Idris 2, this means that it is automatically generalised over in a prenex manner reminiscent of the handling of free type variables in languages in the ML family. This makes it an implicit argument of the constructor. Consequently, given that `Nat` is a type of *unary* natural numbers, a naïve runtime representation of a (`Vect n a`) would have a size quadratic in `n`. A smarter representation with perfect sharing would still represent quite an overhead as observed by Brady, McBride, and McKinna [BMM03].

A finite number is a number known to be strictly smaller than a given natural number. The type `Fin` is indexed by said bound. Its `Z` constructor models `0` and is bound by any non-zero bound, and its `S` constructor takes a number bound by `n` and returns its successor, bound by (`1 + n`). A naïve compilation would here also lead to a runtime representation suffering from a quadratic blowup.

```
data Fin : Nat -> Type where
  Z : Fin (S n)
  S : Fin n -> Fin (S n)
```

This leads us to the definition of the `lookup` function. Provided a vector of size `n` and a finite number `k` bound by this same `n`, we can define a *total* function looking up the value stored at position `k` in the vector. It is guaranteed to return a value. Note that we do not need to consider the case of the empty vector in the pattern matching clauses as all of the return types of the `Fin` constructors force the index to be non-zero and, because the vector and the finite number talk about the same `n`, having an empty vector would automatically imply having a value of type (`Fin 0`) which is self-evidently impossible.

```
lookup : Vect n a -> Fin n -> a
lookup (x :: _) Z = x
lookup (_ :: xs) (S k) = lookup xs k
```

Thanks to our indexed family, we have gained the ability to define a function that cannot possibly fail, as well as the ability to only talk about the pattern matching clauses that make sense. This seemed to be at the cost of efficiency but luckily for us there has already been extensive work on erasure to automatically detect redundant data [BMM03] or data that will not be used at runtime [Tej20].

2.1 Optimising `Vect`, `Fin`, and `lookup`

An analysis in the style of Brady, McBride, and McKinna's [BMM03] can solve the quadratic blowup highlighted above by observing that the natural number a vector is indexed by is entirely determined by the spine of the vector. In particular, the length of the tail does not need to be stored as part of the constructor: it can be reconstructed as the predecessor of the length of the overall vector. As a consequence, a vector can be adequately represented at runtime by a pair of a natural number and a list. Similarly a bounded number can be adequately represented by a pair of natural numbers. Putting all of this together and remembering that the vector and the finite number share the same `n`, `lookup` can be compiled to a function taking two natural numbers and a list. In Idris 2 we would write the optimised `lookup` as follows (we use the **`partial`** keyword because this transformed version is not total at that type).

```
partial
lookup : (n : Nat) -> List a -> Nat -> a
lookup (S n) (x :: _) Z = x
lookup (S n) (_ :: xs) (S k) = lookup n xs k
```

We can see in the second clause that the recursive call is performed on the tail of the list (formerly vector) and so the first argument to `lookup` corresponding to the vector's size is decreased by one. The invariant, despite not being explicit anymore, is maintained.

A Tejiščák-style analysis [Tej20] can additionally notice that the lookup function never makes use of the bound's value and drop it entirely. This leads to the lookup function on vectors being compiled to its partial-looking counterpart acting on lists.

```
partial
lookup : List a -> Nat -> a
lookup (x :: _) Z = x
lookup (_ :: xs) (S k) = lookup xs k
```

Even though this is in our opinion a pretty compelling example of erasing away the apparent complexity introduced by inductive families, this approach has two drawbacks.

Firstly, it relies on the fact that the compiler can and will automatically perform these optimisations. But nothing in the type system prevents users from inadvertently using a value they thought would get erased, thus preventing the Tejiščák-style optimisation from firing. In performance-critical settings, users may rather want to state their intent explicitly and be kept to their word by the compiler in exchange for predictable and guaranteed optimisations.

Secondly, this approach is intrinsically limited to transformations that preserve the type's overall structure: the runtime data structures are simpler but very similar still. We cannot expect much better than that. It is so far unrealistic to expect e.g. a change of representation to use a balanced binary tree instead of a list in order to get logarithmic lookups rather than linear ones.

2.2 No Magic Solution

Even if we are able to obtain a more compact representation of the inductive family at runtime through enough erasure, this does not guarantee runtime efficiency. As the Coq manual [CDT22] reminds its users, extraction does not magically optimise away a user-defined quadratic multiplication algorithm when extracting unary natural numbers to an efficient machine representation. In a pragmatic move, Coq, Agda, and Idris 2 all have ad-hoc rules to replace convenient but inefficiently implemented numeric functions with asymptotically faster counterparts in the target language.

However this approach is not scalable: if we may be willing to extend our trusted core to a high quality library for unbounded integers, we do not want to replace our code only proven correct thanks to complex invariants with a wildly different untrusted counterpart purely for efficiency reasons.

In this paper we use Quantitative Type Theory [McB16, Atk18] as implemented in Idris 2 [Bra21] to bridge the gap between an invariant-rich but inefficient representation based on an inductive family and an unsafe but efficient implementation using low-level primitives. Inductive families allow us to *view* [Wad87, MM04] the runtime relevant information encoded in the low-level and efficient representation as an information-rich compile time data structure. Moreover the quantity annotations guarantee that this additional information

will be erased away during compilation.

3 Some Key Features of Idris 2

Idris 2 implements Quantitative Type Theory, a Martin-Löf type theory enriched with a semiring of quantities classifying the ways in which values may be used. In a type, each binder is annotated with the quantity by which its argument must abide.

3.1 Quantities

A value may be *runtime irrelevant*, *linear*, or *unrestricted*.

Runtime irrelevant values (`0` quantity) cannot possibly influence control flow as they will be erased entirely during compilation. This forces the language to impose strong restrictions on pattern-matching over these values. Typical examples are types like the `a` parameter in (`List a`), or indices like the natural number `n` in (`Vect n a`). These are guaranteed to be erased at compile time. The advantage over a Tejiščák-style analysis is that users can state their intent that an argument ought to be runtime irrelevant and the language will insist that it needs to be convinced it indeed is.

Linear values (`1` quantity) have to be used exactly once. Typical examples include the `%World` token used by Idris 2 to implement the `IO` monad à la Haskell, or file handles that cannot be discarded without first explicitly closing the file. At runtime these values can be updated destructively. We will not use linearity in this paper.

Last, *unrestricted* values (denoted by no quantity annotation) can flow into any position, be duplicated or thrown away. They are the usual immutable values of functional programming.

The most basic of examples mobilising both the runtime irrelevance and unrestricted quantities is the identity function.

```
id : {0 a : Type} -> (x : a) -> a
id x = x
```

Its type starts with a binder using curly braces. This means it introduces an implicit variable that does not need to be filled in by the user at call sites and will be reconstructed by unification. The variable it introduces is named `a` and has type `Type`. It has the `0` quantity annotation which means that this argument is runtime irrelevant and so will be erased during compilation.

The second binder uses parentheses. It introduces an explicit variable whose name is `x` and whose type is the type `a` that was just bound. It has no quantity annotation which means it will be an unrestricted variable.

Finally the return type is the type `a` bound earlier. This is, as expected, a polymorphic function from `a` to `a`. It is implemented using a single clause that binds `x` on the left-hand side and immediately returns it on the right-hand side.

If we were to try to annotate the binder for `x` with a **`0`** quantity to make it runtime irrelevant then Idris 2 would rightfully reject the definition. The following **`failing`** block shows part of the error message complaining that `x` cannot be used at an unrestricted quantity on the right-hand side.

```
failing "x is not accessible in this context."
  id : {0 a : Type} -> (0 x : a) -> a
  id x = x
```

3.2 Proof Search

In Idris 2, Haskell-style ad-hoc polymorphism [WB89] is superseded by a more general proof search mechanism. Instead of having blessed notions of type classes, instances and constraints, the domain of any dependent function type can be marked as **`auto`**. This signals to the compiler that the corresponding argument will be an implicit argument and that it should not be reconstructed by unification alone but rather by proof search. The search algorithm will use the appropriate user-declared hints as well as the local variables in scope.

By default, a datatype's constructors are always added to the database of hints. And so the following declaration brings into scope both an indexed family `So` of proofs that a given boolean is `True`, and a unique constructor `Oh` that is automatically added as a hint.

```
data So : Bool -> Type where
  Oh : So True
```

As a consequence, we can for instance define a record type specifying what it means for `n` to be an even number by storing its `half` together with a proof that is both runtime irrelevant and filled in by proof search. Because (`2 * 3 == 6`) computes to `True`, Idris 2 is able to fill-in the missing proof in the definition of `even6` using the `Oh` hint.

```
record Even (n : Nat) where
  constructor MkEven
  half : Nat
  {auto 0 prf : So (2 * half == n)}

even6 : Even 6
even6 = MkEven { half = 3 }
```

We will use both `So` and the **`auto`** mechanism in section 5.3.

3.3 Application: `Vect`, as `List`

We can use the features of Quantitative Type Theory to give an implementation of `Vect` that is guaranteed to erase to a `List` at runtime independently of the optimisation passes implemented by the compiler. The advantage over the optimisation passes described in section 2 is that the user has control over the

runtime representation and does not need to rely on these optimisations being deployed by the compiler.

The core idea is to make the slogan 'a vector is a length-indexed list' a reality by defining a record packing together the `encoding` as a list and a proof its length is equal to the expected `Nat` index. This proof is marked as runtime irrelevant to ensure that the list is the only thing remaining after compilation.

```
record Vect (n : Nat) (a : Type) where
  constructor MkVect
  encoding : List a
  0 valid : length encoding === n
```

Smart constructors Now that we have defined vectors, we can recover the usual building blocks for vectors by defining smart constructors, that is to say functions `Nil` and `(::)` that act as replacements for the inductive family's data constructors.

```
  Nil : Vect Z a
  Nil = MkVect [] Refl
```

The smart constructor `Nil` returns an empty vector. It is, unsurprisingly, encoded as the empty list (`[]`). Because (`length []`) statically computes to `Z`, the proof that the encoding is valid can be discharged by reflexivity.

```
  (::) : a -> Vect n a -> Vect (S n) a
  x :: MkVect xs eq = MkVect (x :: xs) (cong S eq)
```

Using `(::)` we can combine a head and a tail of size `n` to obtain a vector of size (`S n`). The encoding is obtained by consing the head in front of the tail's encoding and the proof this is valid (`cong S eq`) uses the fact that propositional equality is a congruence and that (`length (x :: xs)`) computes to (`S (length xs)`).

View Now that we know how to build vectors, we demonstrate that we can also take them apart using a view.

A view for a type T, in the sense of Wadler [Wad87], and as refined by McBride and McKinna [MM04], is an inductive family V indexed by T together with a total function mapping every element t of T to a value of type $(V\,t)$. This simple gadget provides a powerful, user-extensible, generalisation of pattern-matching. Patterns are defined inductively as either a pattern variable, a forced term (i.e. an arbitrary expression that is determined by a constraint arising from another pattern), or a data constructor fully applied to subpatterns. In contrast, the return indices of an inductive family's constructors can be arbitrary expressions.

In the case that interests us, the view allows us to emulate 'matching' on which of the two smart constructors `Nil` or `(::)` was used to build the vector being taken apart.

```
data View : Vect n a -> Type where
  Nil : View Nil
  (::) : (x : a) -> (xs : Vect n a) -> View (x :: xs)
```

The inductive family `View` is indexed by a vector and has two constructors corresponding to the two smart constructors. We use Idris 2's overloading capabilities to give each of the `View`'s constructors the name of the smart constructor it corresponds to. By pattern-matching on a value of type (`View xs`), we will be able to break `xs` into its constitutive parts and either observe it is equal to `Nil` or recover its head and its tail.

```
view : (xs : Vect n a) -> View xs
view (MkVect [] Refl) = Nil
view (MkVect (x :: xs) Refl) = x :: MkVect xs Refl
```

The function `view` demonstrates that we can always tell which constructor was used by inspecting the `encoding` list. If it is empty, the vector was built using the `Nil` smart constructor. If it is not then we got our hands on the head and the tail of the encoding and (modulo some re-wrapping of the tail) they are effectively the head and the tail that were combined using the smart constructor.

3.3.1 Application: `map`

We can then use these constructs to implement the function `map` on vectors without ever having to explicitly manipulate the encoding. The maximally sugared version of `map` is as follows:

```
map : (a -> b) -> Vect n a -> Vect n b
map f xs@_ with (view xs)
  _ | [] = []
  _ | hd :: tl = f hd :: map f tl
```

On the left-hand side the view lets us seamlessly pattern-match on the input vector. Using the **`with`** keyword we have locally modified the function definition so that it takes an extra argument, here the result of the intermediate computation (`view xs`). Correspondingly, we have two clauses matching on this extra argument; the symbol **`|`** separates the original left-hand side (here elided using **`_`** because it is exactly the same as in the parent clause) from the additional pattern. This pattern can either have the shape `[]` or (`hd :: tl`) and, correspondingly, we learn that `xs` is either `[]` or (`hd :: tl`).

On the right-hand side the smart constructors let us build the output vector. Mapping a function over the empty vector yields the empty vector while mapping over a cons node yields a cons node whose head and tail have been appropriately modified.

This sugared version of `map` is equivalent to the following more explicit one:

```
map : (a -> b) -> Vect n a -> Vect n b
map f xs with (view xs)
  map f .([]) | [] = []
  map f .(hd :: tl) | hd :: tl = f hd :: map f tl
```

In the parent clause we have explicitly bound `xs` instead of merely introducing an alias for it by writing (`xs@_`) and so we will need to be explicit about the ways in which this pattern is refined in the two with-clauses.

In the with-clauses, we have explicitly repeated the refined version of the parent clause's left-hand side. In particular we have used dotted patterns to insist that `xs` is now entirely *forced* by the match on the result of (`view xs`).

We have seen that by matching on the result of the (`view xs`) call, we get to 'match' on `xs` as if `Vect` were an inductive type. This is the power of views.

3.3.2 Application: `lookup`

The type (`Fin n`) can similarly be represented by a single natural number and a runtime irrelevant proof that it is bound by `n`. We leave these definitions out, and invite the curious reader to either attempt to implement them for themselves or look at the accompanying code.

Bringing these definitions together, we can define a `lookup` function which is similar to the one defined in section 2.

```
lookup : Vect n a -> Fin n -> a
lookup xs@_ k@_ with (view xs) | (view k)
  _ | hd :: _ | Z = hd
  _ | _ :: tl | S k' = lookup tl k'
```

We are seemingly using `view` at two different types (`Vect` and `Fin` respectively) but both occurrences actually refer to separate functions: Idris 2 lets us overload functions and performs type-directed disambiguation.

For pedagogical purposes, this sugared version of `lookup` can also be expanded to a more explicit one that demonstrates the views' power.

```
lookup : Vect n a -> Fin n -> a
lookup xs k with (view xs) | (view k)
  lookup .(hd :: tl) .(Z) | hd :: tl | Z = hd
  lookup .(hd :: tl) .(S k') | hd :: tl | S k' = lookup tl k'
```

The main advantage of this definition is that, based on its type alone, we know that this function is guaranteed to be processing a list and a single natural number at runtime. This efficient runtime representation does not rely on the assumption that state of the art optimisation passes will be deployed.

We have seen some of Idris 2's powerful features and how they can be leveraged to empower users to control the runtime representation of the inductive families they manipulate. This simple example only allowed us to reproduce the performance that could already be achieved by compilers deploying state of the art optimisation passes. In the following sections, we are going to see how

we can use the same core ideas to compile an inductive family to a drastically different runtime representation while keeping good high-level ergonomics.

4 Thinnings, cooked two ways

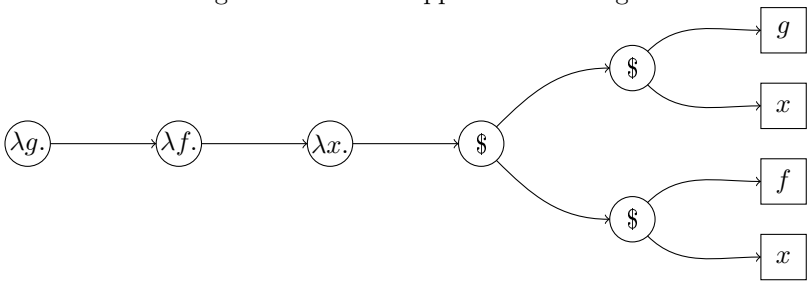
We experienced a major limitation of compilation of inductive families during our ongoing development of TypOS [AAM+22], a domain specific language to define concurrent typecheckers and elaborators. Core to this project is the definition of actors manipulating a generic notion of syntax with binding. Internally the terms of this syntax with binding are based on a co-de Bruijn representation (an encoding we will explain below) which relies heavily on thinnings. A thinning (also known as an Order Preserving Embedding [Cha09]) between a source and a target scope is an order preserving injection of the smaller scope into the larger one. They are usually represented using an inductive family. The omnipresence of thinnings in the co-de Bruijn representation makes their runtime representation a performance critical matter.

Let us first remind the reader of the structure of abstract syntax trees in a named, a de Bruijn, and a co-de Bruijn representation. We will then discuss two representations of thinnings: a safe and convenient one as an inductive family, and an unsafe but efficient encoding as a pair of arbitrary precision integers.

4.1 Named, de Bruijn, and co-de Bruijn syntaxes

In this section we will use the S combinator ($\lambda g.\lambda f.\lambda x.gx(fx)$) as a running example and represent terms using a syntax tree whose constructor nodes are circles and variable nodes are squares. To depict the S combinator we will only need λ-abstraction and application (rendered \$) nodes. A constructor's arguments become its children in the tree. The tree is laid out left-to-right and a constructor's arguments are displayed top-to-bottom.

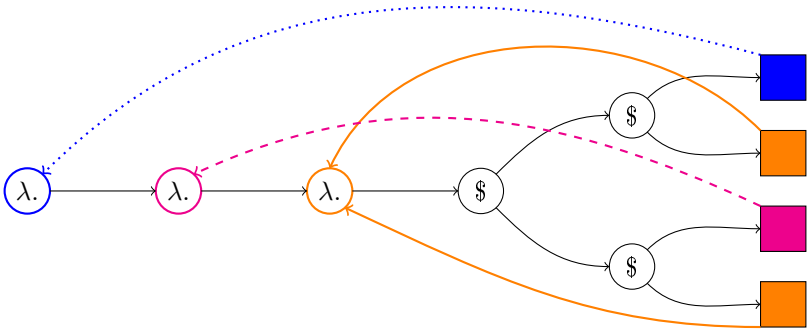
Named syntax The first representation is using explicit names. Each binder has an associated name and each variable node carries a name. A variable refers to the closest enclosing binder which happens to be using the same name.



To check whether two terms are structurally equivalent (*α-equivalence*) potentially requires renaming bound names. In order to have a simple and cheap α-equivalence check we can instead opt for a nameless representation.

De Bruijn syntax An abstract syntax tree based on de Bruijn indices [dB72] replaces names with natural numbers counting the number of binders separating a variable from its binding site. The S combinator is now written $(\lambda\,\lambda\,\lambda\,2\,0\,(1\,0))$.

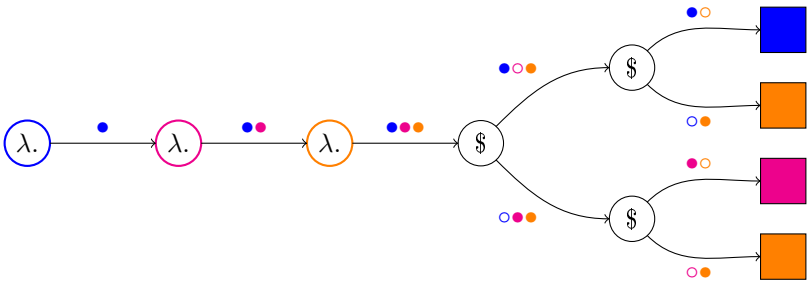
You can see in the following graphical depiction that λ-abstractions do not carry a name anymore and that variables are simply pointing to the binder that introduced them. We have left the squares empty but in practice the various coloured arrows would be represented by a natural number. For instance the dashed magenta one corresponds to 1 because you need to ignore one λ-abstraction (the orange one) on your way towards the root of the tree before you reach the corresponding magenta binder.



To check whether a subterm does not mention a given set of variables (a *thickening* test, the opposite of a *thinning* which extends the current scope with unused variables), you need to traverse the whole term. In order to have a simple cheap thickening test we can ensure that each subterms knows precisely what its *support* is and how it embeds in its parent's.

Co-de Bruijn syntax In a co-de Bruijn representation [McB18] each subterm selects exactly the variables that stay in scope for that term, and so a variable constructor ultimately refers to the only variable still in scope by the time it is reached. This representation ensures that we know precisely what the scope of a given term currently is.

In the following graphical rendering, we represent thinnings as lists of full (•) or empty (○) discs depending on whether the corresponding variable is either kept or discarded. For instance the thinning represented by ○•• throws the blue variable away, and keeps both the magenta and orange ones.



We can see that in such a representation, each node in the tree stores one thinning per subterm. This will not be tractable unless we have an efficient representation of thinnings.

4.2 The Performance Challenges of co-de Bruijn

Using the co-de Bruijn approach, a term in an arbitrary context is represented by the pairing of a term in co-de Bruijn syntax with a thinning from its support into the wider scope. Having such a precise handle on each term's support allows us to make operations such as thinning, substitution, unification, or common sub-expression elimination more efficient.

Thinning a term does not require us to traverse it anymore. Indeed, embedding a term in a wider context will not change its support and so we can simply compose the two thinnings while keeping the term the same.

Substitution can avoid traversing subterms that will not be changed. Indeed, it can now easily detect when the substitution's domain does not intersect with the subterm's support.

Unification requires performing thickening tests when we want to solve a metavariable declared in a given context with a terms seemingly living in a wider one. We once more do not need to traverse the term to perform this test, and can simply check whether the outer thinning can be thickened.

Common sub-expression elimination requires us to identify alpha-equivalent terms potentially living in different contexts. Using a de Bruijn representation, these can be syntactically different: a variable represented by the natural number v in Γ would be $(1+v)$ in Γ, σ but $(2+v)$ in Γ, τ, ν. A co-de Bruijn representation, by discarding all the variables not in the support, guarantees that we can once more use syntactic equality to detect alpha-equivalence. This encoding is used for instance (albeit unknowingly) by Maziarz, Ellis, Lawrence, Fitzgibbon, and Peyton-Jones in their 'Hashing modulo alpha-equivalence' work [MEL+21].

For all of these reasons we have, as we mentioned earlier, opted for a co-de Bruijn representation in the implementation of TypOS [AAM+22]. And so it is crucial for performance that we have a compact representation of thinnings.

4.2.1 Thinnings in TypOS

We first carefully worked out the trickier parts of the implementation in Agda before porting the resulting code to Haskell. This process highlighted a glaring gap between on the one hand the experiments done using a strongly typed inductive representation of thinnings and on the other hand their more efficient but unsafe encoding in Haskell.

Agda The Agda-based experiments use inductive families that make the key invariants explicit which helps tracking complex constraints and catches design flaws at typechecking time. The indices guarantee that we always transform the

thinnings appropriately when we add or remove bound variables. In Idris 2, the inductive family representation of thinnings would be written:

```
data Thinning : (sx, sy : SnocList a) -> Type where
  Done : Thinning [<] [<]
  Keep : Thinning sx sy -> (0 x : a) -> Thinning (sx :< x) (sy :< x)
  Drop : Thinning sx sy -> (0 x : a) -> Thinning sx (sy :< x)
```

The `Thinning` family is indexed by two scopes (represented as snoclists i.e. lists that are extended from the right, just like contexts in inference rules): `sx` the tighter scope and `sy` the wider one. The `Done` constructor corresponds to a thinning from the empty scope to itself (`[<]` is Idris 2 syntactic sugar for the empty snoclist), and `Keep` and `Drop` respectively extend a given thinning by keeping or dropping the most local variable (`:<` is the 'snoc' constructor, a sort of flipped 'cons'). The 'name' (`x` of type `a`) is marked with the quantity `0` to ensure it is erased at compile time (cf. section 3).

During compilation, Idris 2 would erase the families' indices as they are forced (in the sense of Brady, McBride, and McKinna [BMM03]), and drop the constructor arguments marked as runtime irrelevant. The resulting inductive type would be the following simple data type.

```
data Thinning = Done | Keep Thinning | Drop Thinning
```

At runtime this representation is therefore essentially a linked list of booleans (`Done` being `Nil`, and `Keep` and `Drop` respectively (`True ::`) and (`False ::`)).

Haskell The Haskell implementation uses this observation and picks a packed encoding of this list of booleans as a pair of integers. One integer represents the length `n` of the list, and the other integer's `n` least significant bits encode the list as a bit pattern where `1` is `Keep` and `0` is `Drop`.

Basic operations on thinnings are implemented by explicitly manipulating individual bits. It is not indexed and thus all the invariant tracking has to be done by hand. This has led to numerous and hard to diagnose bugs.

4.2.2 Thinnings in Idris 2

Idris 2 is a self-hosting language whose core datatype is currently based on a well-scoped de Bruijn representation. This precise indexing of terms by their scope helped entirely eliminate a whole class of bugs that plagued Idris 1's unification machinery.

If we were to switch to a co-de Bruijn representation for our core language we would want, and should be able, to have the best of both worlds: a safe *and* efficient representation!

Thankfully Idris 2 implements Quantitative Type Theory (QTT) which gives us a lot of control over what is to be runtime relevant and what is to be erased during compilation. This should allow us to insist on having a high-level interface that resembles an inductive family while ensuring that everything but

a pair of integers is erased at compile time. We will exploit the key features of QTT presented in section 3 to have our cake and eat it.

5 An Efficient Invariant-Rich Representation

We can combine both approaches highlighted in section 4.2 by defining a record parameterised by a source (`sx`) and target (`sy`) scopes corresponding to the two ends of the thinnings, just like we would for the inductive family. This record packs two numbers and a runtime irrelevant proof.

Firstly, we have a natural number called `bigEnd` corresponding to the size of the big end of the thinning (`sy`). We are happy to use a (unary) natural number here because we know that Idris 2 will compile it to an unbounded integer.

Secondly, we have an integer called `encoding` corresponding to the thinning represented as a bit vector stating, for each variable, whether it is kept or dropped. We only care about the integer's `bigEnd` least significant bits and assume the rest is set to 0.

Thirdly, we have a runtime irrelevant proof `invariant` that `encoding` is indeed a valid encoding of size `bigEnd` of a thinning from `sx` to `sy`. We will explore the definition of the relation `Invariant` later on in section 5.3.

```
record Th {a : Type} (sx, sy : SnocList a) where
  constructor MkTh
  bigEnd : Nat
  encoding : Integer
  0 invariant : Invariant bigEnd encoding sx sy
```

The first sign that this definition is adequate is our ability to construct any valid thinning. We demonstrate it is the case by introducing functions that act as smart constructor analogues for the inductive family's data constructors.

5.1 Smart Constructors for `Th`

The first and simplest one is `done`, a function that packs a pair of `0` (the size of the big end, and the empty encoding) together with a proof that it is an adequate encoding of the thinning from the empty scope to itself. In this instance, the proof is simply the `Done` constructor.

```
done : Th [<] [<]
done = MkTh { bigEnd = 0, encoding = 0, invariant = Done }
```

To implement both `keep` and `drop`, we are going to need to perform bit-level manipulations. These are made easy by Idris 2's `Bits` interface which provides us with functions to shift the bit patterns left or right (`shiftl`, `shiftr`), set or clear bits at specified positions (`setBit`, `clearBit`), take bitwise logical operations like disjunction (`.|.`) or conjunction (`.&.`), etc.

In both keep and drop, we need to extend the encoding with an additional bit. For this purpose we introduce the cons function which takes a bit b and an existing encoding bs and returns the new encoding $bs{\cdot}b$.

```
cons : Bool -> Integer -> Integer
cons b bs = let bs0 = bs `shiftL` 1 in
            if b then (bs0 `setBit` 0) else bs0
```

No matter what the value of the new bit is, we start by shifting the encoding to the left to make space for it; this gives us bs0 which contains the bit pattern $bs{\cdot}0$. If the bit is True then we need to additionally set the bit at position 0 to obtain $bs{\cdot}1$. Otherwise if the bit is False, we can readily return the $bs{\cdot}0$ encoding obtained by left shifting. The correctness of this function is backed by two lemma: testing the bit at index 0 after consing amounts to returning the cons'd bit, and shifting the cons'd encoding to the right takes us back to the unextended encoding.

```
testBit0Cons : (b : Bool) -> (bs : Integer) ->
               testBit (cons b bs) 0 === b

consShiftR : (b : Bool) -> (bs : Integer) ->
             (cons b bs) `shiftR` 1 === bs
```

The keep smart constructor demonstrates that from a thinning from sx to sy and a runtime irrelevant variable x we can compute a thinning from the extended source scope (sx :< x) to the target scope (sy :< x) where x was kept.

```
keep : Th sx sy -> (0 x : a) -> Th (sx :< x) (sy :< x)
keep th x = MkTh
 { bigEnd = S (th .bigEnd)
 , encoding = cons True (th .encoding)
 , invariant =
    let 0 b = eqToSo $ testBit0Cons True (th .encoding) in
    Keep (rewrite consShiftR True (th .encoding) in th.invariant) x
 }
```

The outer scope has grown by one variable and so we increment bigEnd. The encoding is obtained by cons-ing the boolean True to record the fact that this new variable is kept. Finally, we use the two lemmas shown above to convince Idris 2 the invariant has been maintained.

Similarly the drop function demonstrates that we can compute a thinning getting rid of the variable x freshly added to the target scope.

```
drop : Th sx sy -> (0 x : a) -> Th sx (sy :< x)
drop th x = MkTh
 { bigEnd = S (th .bigEnd)
 , encoding = cons False (th .encoding)
 , invariant =
   let 0 prf = testBit0Cons False (th .encoding)
       0 nb = eqToSo $ cong not prf in
    Drop (rewrite consShiftR False (th .encoding) in th .invariant) x
}
```

We once again increment the `bigEnd`, use `cons` to record that the variable is being discarded and use the lemmas ensuring its correctness to convince Idris 2 the invariant is maintained.

We can already deploy these smart constructors to implement functions producing thinnings. We use `which` as our example. It is a filter-like function that returns a dependent pair containing the elements that satisfy a boolean predicate together with a proof that there is a thinning embedding them back into the input snoclist.

```
which : (a -> Bool) -> (sy : SnocList a) ->
        (sx : SnocList a ** Th sx sy)
which p [<] = ([<] ** done)
which p (sy :< y) =
  let (sx ** th) = which p sy in
  if p y then (sx :< y ** keep th y)
         else (sx ** drop th y)
```

If the input snoclist is empty then the output shall also be, and `done` builds a thinning from `[<]` to itself. If it is not empty we can perform a recursive call on the tail of the snoclist and then depending on whether the predicates holds true of the head we can either `keep` or `drop` it.

We are now equipped with these smart constructors that allow us to seamlessly build thinnings. To recover the full expressive power of the inductive family, we also need to be able to take these thinnings apart. We are now going to tackle this issue.

5.2 Pattern Matching on `Th`

The `View` family is a sum type indexed by a thinning. It has one data constructor associated to each smart constructor and storing its arguments.

```
data View : Th sx sy -> Type where
  Done : View done
  Keep : (th : Th sx sy) -> (0 x : a) -> View (keep th x)
  Drop : (th : Th sx sy) -> (0 x : a) -> View (drop th x)
```

The accompanying `view` function witnesses the fact that any thinning arises as one of these three cases.

```
view : (th : Th sx sy) -> View th
```

We show the implementation of `view` in its entirety but leave out the technical auxiliary lemma it invokes. The interested reader can find them in the accompanying material. We will however inspect the code `view` compiles to after erasure in section 5.5 to confirm that these auxiliary definitions do not incur any additional runtime cost.

We first start by pattern matching on the `bigEnd` of the thinning. If it is `0` then we know the thinning has to be the empty thinning. Thanks to an inversion lemma called `isDone`, we can collect a lot of equality proofs: the encoding `bs` *has to* be `0`, the source and target scopes `sx` and `sy` *have to* be the empty snoclists, and the proof `prf` of the invariant has to be of a specific shape. Rewriting by these equalities changes the goal type enough for the typechecker to ultimately see that the thinning was constructed using the `done` smart constructor and so we can use the view's `Done` constructor.

```
view (MkTh 0 bs prf) =
  let 0 eqs = isDone prf in
  rewrite bsIsZero eqs in
  rewrite fstIndexIsLin eqs in
  rewrite sndIndexIsLin eqs in
  rewrite invariantIsDone eqs in
  Done
```

In case the thinning is non-empty, we need to inspect the 0-th bit of the encoding to know whether it keeps or discards its most local variable. This is done by calling the `choose` function which takes a boolean `b` and returns a value of type `(Either (So b) (So (not b))` i.e. we not only inspect the boolean but also record which value we got in a proof using the `So` family introduced in section 3.

```
view (MkTh (S i) bs prf) = case choose (testBit bs Z) of
```

If the bit is set then we know the variable is kept. And so we can invoke an inversion lemma that will once again provide us with a lot of equalities that we immediately deploy to reshape the goal's type. This ultimately lets us assemble a sub-thinning and use the view's `Keep` constructor.

```
  Left so =>
    let 0 eqs = isKeep prf so in
    rewrite fstIndexIsSnoc eqs in
    rewrite sndIndexIsSnoc eqs in
    rewrite invariantIsKeep eqs in
    rewrite isKeepInteger bs so in
    let th : Th eqs.fstIndexTail eqs.sndIndexTail
        th = MkTh i (bs `shiftR` 1) eqs.subInvariant in
    cast $ Keep th eqs.keptHead
```

If the bit is not set then we learn that the thinning was constructed using

drop. We can once again use an inversion lemma to rearrange the goal and finally invoke the view's Drop constructor.

```
Right soNot =>
  let 0 eqs = isDrop prf soNot in
  rewrite sndIndexIsSnoc eqs in
  rewrite invariantIsDrop eqs in
  rewrite isDropInteger bs soNot in
  let th : Th sx eqs.sndIndexTail
      th = MkTh i (bs `shiftR` 1) eqs.subInvariant in
  cast $ Drop th eqs.keptHead
```

We can readily use this function to implement pattern matching functions taking a thinning apart. We can for instance define kept, the function that counts the number of keep smart constructors used when manufacturing the input thinning and returns a proof that this is exactly the length of the source scope sx.

```
kept : Th sx sy -> (n : Nat ** length sx === n)
kept th = case view th of
  Done      => (0 ** Refl)
  Keep th x => let (n ** eq) = kept th in
               (S n ** cong S eq)
  Drop th x => kept th
```

We proceed by calling the view function on the input thinning which immediately tells us that we only have three cases to consider. The Done case is easily handled because the branch's refined types inform us that both sx and sy are the empty snoclist [<] whose length is evidently 0. In the Keep branch we learn that sx has the shape (_ :< x) and so we must return the successor of whatever the result of the recursive call gives us. Finally in the Drop case, sx is untouched and so a simple recursive call suffices. Note that the function is correctly detected as total because the target scope sy is indeed getting structurally smaller at every single recursive call. It is runtime irrelevant but it can still be successfully used as a termination measure by the compiler.

5.3 The Invariant Relation

We have shown the user-facing Th and have claimed that it is possible to define smart constructors done, keep, and drop, as well as a view function. This should become apparent once we show the actual definition of Invariant.

5.3.1 Definition of Invariant

The relation maintains the invariant between the record's fields bigEnd (a Nat) and encoding (an Integer) and the index scopes sx and sy. Its definition can favour ease-of-use of runtime efficiency because we statically know that all of

the Invariant proofs will be erased during compilation.

```
data Invariant : (i : Nat) -> (bs : Integer) ->
                 (sx, sy : SnocList a) -> Type where
  Done : Invariant Z 0 [<] [<]
  Keep : Invariant i (bs `shiftR` 1) sx sy -> (0 x : a) ->
         {auto 0 b  : So (testBit bs Z)} ->
         Invariant (S i) bs (sx :< x) (sy :< x)
  Drop : Invariant i (bs `shiftR` 1) sx sy -> (0 x : a) ->
         {auto 0 nb : So (not (testBit bs Z))} ->
         Invariant (S i) bs sx (sy :< x)
```

As always, the Done constructor is the simplest. It states that the thinning of size Z and encoded as the bit pattern 0 is the empty thinning.

The Keep constructor guarantees that the thinning of size (S i) and encoding bs represents an injection from (sx :< x) to (sy :< x) provided that the bit at position Z of bs is set, and that the rest of the bit pattern (obtained by a right shift on bs) is a valid thinning of size i from sx to sy.

The Drop constructor is structured the same way, except that it insists the bit at position Z should *not* be set.

We can readily use this relation to prove that some basic encoding are valid representations of useful thinnings.

5.3.2 Examples of Invariant proofs

For instance, we can always define a thinning from the empty scope to an arbitrary scope sy.

```
  none : (sy : SnocList a) -> Th [<] sy
  none sy = MkTh (length sy) 0 (none sy)
```

The encoding of this thinning is 0 because every variable is being discarded and its bigEnd is the length of the outer scope sy. The proof that this encoding is valid is provided by the none lemma proven below. We once again use Idris 2's overloading to give the same to functions that play similar roles but at different types.

```
none : (sy : SnocList a) -> Invariant (length sy) 0 [<] sy
none [<] = Done
none (sy :< y) = Drop (none sy) y
```

The proof proceeds by induction over the outer scope sy. If it is empty, we can simply use the constructor for the empty thinning. Otherwise we can invoke Drop on the induction hypothesis. This all typechecks because (testBit 0 Z) computes to False and so the nb proof can be constructed automatically by Idris 2's proof search (cf. section 3.2), and (0 `shiftR` 1) evaluates to 0 which means the induction hypothesis has exactly the right type.

The definition of the identity thinning is a bit more involved. For a scope of size n, we are going to need to generate a bit pattern consisting of n ones. We define it in two steps. First, `cofull` defines a bit pattern of k zeros followed by infinitely many ones by shifting k places to the left a bit pattern of ones only. Then, we obtain `full` by taking the complement of `cofull`.

```
cofull : Nat -> Integer
cofull n = oneBits `shiftL` n
```

```
full : Nat -> Integer
full n = complement (cofull n)
```

We can then define the identity thinning for a scope of size n by pairing (`full n`) as the `encoding` and `n` as the `bigEnd`.

```
ones : (sx : SnocList a) -> Th sx sx
ones sx = let n : Nat; n = length sx in MkTh n (full n) (ones sx)
```

The bulk of the work is once again in the eponymous lemma proving that this encoding is valid.

```
ones : (sx : SnocList a) ->
       let n = length sx in Invariant n (full n) sx sx
ones [<] = Done
ones (sx :< x) =
  let 0 nb = eqToSo (testBitFull (S (length sx)) Z) in
  Keep (rewrite shiftRFull (length sx) in ones sx) x
```

This proof proceeds once more by induction on the scope. If the scope is empty then once again the constructor for the empty thinning will do. In the non-empty case, we first appeal to an auxiliary lemma (not shown here) to construct a proof `nb` that the bit at position `Z` for a non-zero `full` integer is known to be `True`. We then need to use another lemma to cast the induction hypothesis which mentions (`full (length sx)`) so that it may be used in a position where we expect a proof talking about (`full (length (sx :< x)) `shiftR` 1`).

5.3.3 Properties of the `Invariant` relation

This relation has a lot of convenient properties.

First, it is proof irrelevant: any two proofs that the same `i`, `bs`, `sx`, and `sy` are related are provably equal. Consequently, equality on `Th` values amounts to equality of the `bigEnd` and `encoding` values. In particular it is cheap to test whether a given thinning is the empty or the identity thinning.

Second, it can be inverted [CT95] knowing only two bits: whether the natural number is empty and what the value of the bit at position `Z` of the encoding is. This is what allowed us to efficiently implement the `view` function by using these two checks and then inverting the `Invariant` proof to gain access to the proof that the remainder of the thinning's encoding is valid. We will see in section 5.5 that this leads to efficient runtime code for the view.

5.4 Choose Your Own Abstraction Level

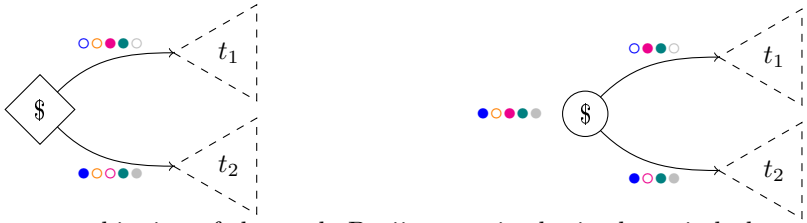
Access to both the high-level `View` and the internal `Invariant` relation means that programmers can pick the level of abstraction at which they want to work. They may need to explicitly manipulate bits to implement key operators that are used in performance-critical paths but can also stay at the highest level for more negligible operations, or when proving runtime irrelevant properties.

In the previous section we saw simple examples of these bit manipulations when defining `none` (using the constant 0 bit pattern) and `ones` using bit shifting and complement to form an initial segment of 1s followed by 0s.

Other natural examples include the *meet* and *join* of two thinnings sharing the same wider scope. The join can for instance be thought of either as a function defined by induction on the first thinning and case analysis on the second, emitting a `Keep` constructor whenever either of the inputs does. Or we can observe that the bit pattern in the join is exactly the disjunction of the inputs' respective bit patterns and prove a lemma about the `Invariant` relation instead. This can be visualised as follows. In each column, the meet is a • whenever either of the inputs is.

The join is of particular importance because it appears when we convert an 'opened' view of a term into its co-de Bruijn counterpart. As we mentioned earlier, co-de Bruijn terms in an arbitrary scope are represented by the pairing of a term indexed by its precise support with a thinning embedding this support back into the wider scope. When working with such a representation, it is convenient to have access to an 'opened' view where the outer thinning has been pushed inside therefore exposing the term's top-level constructor, ready for case-analysis.

The following diagram shows the correspondence between an 'opened' application node using the view (the diamond '$' node) with two subterms both living in the outer scope and its co-de Bruijn form (the circular '$' node) with an outer thinning selecting the term support.



The outer thinning of the co-de Bruijn term is obtained precisely by computing the join of the respective outer thinnings of the 'opened' application's function and argument.

These explicit bit manipulations will be preserved during compilation and thus deliver more efficient code.

5.5 Compiled Code

The following code block shows the JavaScript code that is produced when compiling the `view` function. We chose to use the JavaScript backend rather than e.g. the ChezScheme one because it produces fairly readable code. We have modified the backend to also write comments reminding the reader of the type of the function being defined and the data constructors the natural number tags correspond to. These changes are now available to all in Idris 2's current development version.

The only manual modifications we have performed are the inlining of a function corresponding to a **case** block, renaming variables and property names to make them human-readable, introducing the `$tail` definitions to make lines shorter, and slightly changing the layout.

```
/* Thin.Smart.view : (th : Th sx sy) -> View th */
function Thin_Smart_view($th) {
  switch($th.bigEnd) {
    case 0n: return {h: 0 /* Done */};
    default: {
      const $predBE = ($th.bigEnd-1n);
      const $test = choose(notEq(($th.encoding&1n), 0n));
      switch($test.tag) {
        case 0: /* Left */ {
          const $tail = $th.encoding>>1n;
          return { tag: 1 /* Keep */
                 , val: {bigEnd: $predBE, encoding: $tail}}; }
        case 1: /* Right */ {
          const $tail = $th.encoding>>1n;
          return { tag: 2 /* Drop */
                 , val: {bigEnd: $predBE, encoding: $tail}}; }
}}}}
```

Readers can see that the compilation process has erased all of the indices and the proofs showing that the invariant tying the efficient runtime representation to the high-level specification is maintained. A thinning is represented at runtime by a JavaScript object with two properties corresponding to `Th`'s runtime relevant fields: `bigEnd` and `encoding`. Both are storing a JavaScript `bigInt` (one corresponding to the `Nat`, the other to the `Integer`). For instance the thinning [01101] would be at runtime `{ bigEnd: 5n, encoding: 13n }`.

The view proceeds in two steps. First if the `bigEnd` is `0n` then we know the thinning is empty and can immediately return the `Done` constructor. Otherwise we know the thinning to be non-empty and so we can compute the big end of its tail (`$predBE`) by subtracting one to the non-zero `bigEnd`. We can then inspect the bit at position `0` to decide whether to return a `Keep` or a `Drop` constructor. This is performed by using a bit mask to 0-out all the other bits (`$th.bigEnd&1n`) and checking whether the result is zero. If it is not equal to 0 then we emit

`Keep` and compute the `$tail` of the thinning by shifting the original encoding to drop the 0th bit. Otherwise we emit `Drop` and compute the same tail.

By running `view` on this [01101] thinning, we would get back (`Keep` [0110]), that is to say `{ tag: 1, val: { bigEnd: 4n, encoding: 6n } }`.

Thanks to Idris 2's implementation of Quantitative Type Theory we have managed to manufacture a high level representation that can be manipulated like a classic inductive family using smart constructors and views without giving up an inch of control on its runtime representation.

The remaining issues such as the fact that we form the view's constructors only to immediately take them apart thus creating needless allocations can be tackled by reusing Wadler's analysis (section 12 of [Wad87]).

6 Conclusion

We have seen that inductive families provide programmers with ways to root out bugs by enforcing strong invariants. Unfortunately these families can get in the way of producing performant code despite existing optimisation passes erasing redundant or runtime irrelevant data. This tension has led us to take advantage of Quantitative Type Theory in order to design a library combining the best of both worlds: the strong invariants and ease of use of inductive families together with the runtime performance of explicit bit manipulations.

6.1 Related Work

For historical and ergonomic reasons, idiomatic code in Coq tends to center programs written in a subset of the language quite close to OCaml and then prove properties about these programs in the runtime irrelevant `Prop` fragment. This can lead to awkward encodings when the unrefined inputs force the user to consider cases which ought to be impossible. Common coping strategies involve relaxing the types to insert a modicum of partiality e.g. returning an option type or taking an additional input to be used as the default return value. This approach completely misses the point of type-driven development. We benefit a lot from having as much information as possible available during interactive editing. This information not only helps tremendously getting the definitions right by ensuring we always maintain vital invariants thus making invalid states unrepresentable, it also gives programmers access to type-driven tools and automation. Thankfully libraries such as Equations [Soz10, SM19] can help users write more dependently typed programs, by taking care of the complex encoding required in Coq. A view-based approach similar to ours but using `Prop` instead of the zero quantity ought to be possible. We expect that the views encoded this way in Coq will have an even worse computational behaviour given that Equations uses a sophisticated elaboration process to encode dependent pattern-matching into Gallina. However Coq does benefit from good automation support for unfolding lemmas, inversion principles, and rewriting

by equalities which may compensate for the awkwardness introduced by the encoding.

Prior work on erasure [Tej20] has the advantage of offering a fully automated analysis of the code. The main inconvenience is that users cannot state explicitly that a piece of data ought to be runtime irrelevant and so they may end up inadvertently using it which would prevent its erasure. Quantitative Type Theory allows us users to explicitly choose what is and is not runtime relevant, with the quantity checker keeping us true to our word. This should ensure that the resulting program has a much more predictable complexity.

A somewhat related idea was explored by Brady, McKinna, and Hammond in the context of circuit design [BMH07]. In their verification work they index an efficient representation (natural numbers as a list of bits) by its meaning as a unary natural number. All the operations are correct by construction as witnessed by the use of their unary counterparts acting as type-level specifications. In the end their algorithms still process the inductive family instead of working directly with binary numbers. This makes sense in their setting where they construct circuits and so are explicitly manipulating wires carrying bits. By contrast, in our motivating example we really want to get down to actual (unbounded) integers rather than linked lists of bits.

6.2 Limitations and Future Work

Overall we found this case study using Idris 2, a state of the art language based on Quantitative Type Theory, very encouraging. The language implementation is still experimental (see for instance appendix B for some of the bugs we found) but none of the issues are intrinsic limitations. We hope to be able to push this line of work further, tackling the following limitations and exploring more advanced use cases.

6.2.1 Limitations

Unfortunately it is only *propositionally* true that (`view (keep th x)`) computes to (`Keep th x`) (and similarly for `done`/`Done` and `drop`/`Drop`). This means that users may need to manually deploy these lemmas when proving the properties of functions defined by pattern matching on the result of calling the `view` function. This annoyance would disappear if we had the ability to extend Idris 2's reduction rules with user-proven equations as implemented in Agda and formally studied by Cockx, Tabareau, and Winterhalter [CTW21].

In this paper's case study, we were able to design the core `Invariant` relation making the invariants explicit in such a way that it would be provably proof irrelevant. This may not always be possible given the type theory currently implemented by Idris 2. Adding support for a proof-irrelevant sort of propositions (see e.g. Altenkirch, McBride, and Swierstra's work [AMS07]) could solve this issue once and for all.

The Idris 2 standard library thankfully gave us access to a polished pure interface to explicitly manipulate an integer's bits. However these built-in oper-

ations came with no built-in properties whatsoever. And so we had to postulate a (minimal) set of axioms (see appendix A) and prove a lot of useful corollaries ourselves. There is even less support for other low-level operations such as reading from a read-only array, or manipulating pointers.

We also found the use of runtime irrelevance (the `0` quantity) sometimes somewhat frustrating. Pattern-matching on a runtime irrelevant value in a runtime relevant context is currently only possible if it is manifest for the compiler that the value could only arise using one of the family's constructors. In non-trivial cases this is unfortunately only merely provable rather than self-evident. Consequently we are forced to jump through hoops to appease the quantity checker, and end up defining complex inversion lemmas to bypass these limitations. This could be solved by a mix of improvements to the typechecker and meta-programming using prior ideas on automating inversion [CT95, McB96, Mon10].

6.2.2 Future work

We are planning to explore more memory-mapped representations equipped with a high level interface.

We already have experimental results demonstrating that we can use a read-only array as a runtime representation of a binary search tree. Search can be implemented as a proven-correct high level decision procedure that is seemingly recursively exploring the "tree". At runtime however, this will effectively execute like a classic search by dichotomy over the array.

More generally, we expect that a lot of the work on programming on serialised data done in LoCal [VKR$^+$19] thanks to specific support from the compiler can be done as-is in a QTT-based programming language. Indeed, QTT's type system is powerful enough that tracking these invariants can be done purely in library code.

In the short term, we would like to design a small embedded domain specific language giving users the ability to more easily build and take apart products and sums efficiently represented in the style we presented here. Staging would help here to ensure that the use of the eDSL comes at no runtime cost. There are plans to add type-enforced staging to Idris 2, thus really making it the ideal host language for our project.

Our long term plan is to go beyond read-only data and look at imperative programs proven correct using separation logic and see how much of this after-the-facts reasoning can be brought back into the types to enable a high-level correct-by-construction programming style that behaves the same at runtime.

Acknowledgements We are grateful to Conor McBride for discussions pertaining to the fine details of the unsafe encoding used in TypOS, as well as James McKinna, Fredrik Nordvall Forsberg, Ohad Kammar, and Jacques Carette for providing helpful comments and suggestions on early versions of this paper.

References

[AAM+22] Guillaume Allais, Malin Altenmüller, Conor McBride, Georgi Nakov, Fredrik Nordvall Forsberg, and Craig Roy. TypOS: An operating system for typechecking actors. In *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, Nantes, France*, 2022.

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 57–68. ACM, 2007.

[Atk18] Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018.

[BMH07] Edwin C. Brady, James McKinna, and Kevin Hammond. Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types. In Marco T. Morazán, editor, *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007*, volume 8 of *Trends in Functional Programming*, pages 159–176. Intellect, 2007.

[BMM03] Edwin C. Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003.

[Bra21] Edwin C. Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[CA20] Jesper Cockx and Andreas Abel. Elaborating dependent (co)pattern matching: No pattern left behind. *J. Funct. Program.*, 30:e2, 2020.

[CDT22] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.15.2*, May 2022.

[Cha09] James Maitland Chapman. *Type checking and normalisation.* PhD thesis, University of Nottingham, July 2009.

[CT95] Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in coq. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 85–104. Springer, 1995.

[CTW21] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The taming of the rew: a type theory with computational assumptions. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[dB72] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[Dyb94] Peter Dybjer. Inductive families. *Formal Aspects Comput.*, 6(4):440–465, 1994.

[McB96] Conor McBride. Inverting inductively defined relations in LEGO. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop TYPES'96, Aussois, France, December 15-19, 1996, Selected Papers*, volume 1512 of *Lecture Notes in Computer Science*, pages 236–253. Springer, 1996.

[McB16] Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016.

[McB18] Conor McBride. Everybody's got to be somewhere. In Robert Atkey and Sam Lindley, editors, *Proceedings of the 7th Workshop on Mathematically Structured Functional Programming, MSFP@FSCD 2018, Oxford, UK, 8th July 2018*, volume 275 of *EPTCS*, pages 53–69, 2018.

[MEL+21] Krzysztof Maziarz, Tom Ellis, Alan Lawrence, Andrew W. Fitzgibbon, and Simon Peyton Jones. Hashing modulo alpha-equivalence. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 960–973. ACM, 2021.

[MM04] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[Mon10] Jean-François Monin. Proof Trick: Small Inversions. In Yves Bertot, editor, *Second Coq Workshop*, Edinburgh, United Kingdom, July 2010. Yves Bertot.

[SM19] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: high-level dependently-typed functional programming and proving in coq. *Proc. ACM Program. Lang.*, 3(ICFP):86:1–86:29, 2019.

[Soz10] Matthieu Sozeau. Equations: A dependent pattern-matching compiler. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2010.

[Tej20] Matúš Tejiščák. A dependently typed calculus with pattern matching and erasure inference. *Proc. ACM Program. Lang.*, 4(ICFP):91:1–91:29, 2020.

[VKR+19] Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. Local: a language for programs operating on serialized data. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 48–62. ACM, 2019.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 307–313. ACM Press, 1987.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989.

A Postulated lemmas for the `Bits` interface

It is often more convenient to reason about integers in terms of their bits. We define the notion of bitwise equality as the pointwise equality according to the `testBit`.

```
(~~~) : Integer -> Integer -> Type
bs ~~~ cs = (i : Nat) -> testBit bs i === testBit cs i
```

Our first postulate is a sort of extensionality principle stating that bitwise equality implies propositional equality.

```
extensionally : {bs, cs : Integer} -> bs ~~~ cs -> bs === cs
```

This gives us a powerful reasoning principle once combined with axioms explaining the behaviour of various primitives at the bit level. This is why almost all of the remaining axioms are expressed in terms of `testBit` calls.

A.1 Logical operations

Our first batch of axioms relates logical operations on integers to their boolean counterparts. This is essentially stating that these operations are bitwise.

```
testBitAnd : (bs, cs : Integer) -> (i : Nat) ->
  testBit (bs .&. cs) i === (testBit bs i && testBit cs i)

testBitOr : (bs, cs : Integer) -> (i : Nat) ->
  testBit (bs .|. cs) i === (testBit bs i || testBit cs i)


testBitComplement : (bs : Integer) -> (i : Nat) ->
  testBit (complement bs) i === not (testBit bs i)
```

Together with the extensionality principle mentioned above this already allows us to prove for instance that the binary operators are commutative and associative, that the de Morgan laws hold, or that conjunction distributes over disjunction.

A.2 Bit Shifting

The second set of axiom describes the action of left and right shifting on bit patterns.

A right shift of size `k` will drop the `k` least significant bits; consequently testing the bit `i` on the right-shifted integer amounts to testing the bit `(k + i)` on the original integer.

```
testBitShiftR : (bs : Integer) -> (k : Nat) ->
  (i : Nat) -> testBit (bs `shiftR` k) i === testBit bs (k + i)
```

A left shift will add `k` new least significant bits initialised at `0`; consequently testing a bit `i` on the left-shifted integer will either return `False` if `i` is strictly less than `k`, or the bit at position `(i - k)` in the original integer.

For simplicity we state these results without mentioning the 'strictly less than' relation, by considering on the one hand the effect of a non-zero left shift, and on the other the fact that a left-shift by 0 bits is the identity function.

```
testBit0ShiftL : (bs : Integer) -> (k : Nat) ->
  testBit (bs `shiftL` S k) Z === False

testBitSShiftL : (bs : Integer) -> (k : Nat) -> (i : Nat) ->
  testBit (bs `shiftL` S k) (S i) === testBit (bs `shiftL` k) i
```

```
shiftL0 : (bs : Integer) -> (bs `shiftL` 0) === bs
```

A.3 Bit testing

The last set of axioms specifies what happens when a bit is set.

Testing a bit other than the one that was set amounts to testing it on the original integer.

```
testSetBitOther : (bs : Integer) -> (i, j : Nat) -> Not (i === j) ->
  testBit (setBit bs i) j === testBit bs j
```

Finally, we have an axiom stating that the integer (`bit i`) (i.e. 2^i) is non-zero.

```
bitNonZero : (i : Nat) -> (bit i == 0) === False
```

B Current Limitations of Idris 2

This challenge, suggested by Jacques Carette, highlights some of the current limitations of Idris 2.

B.1 Problem statement

The goal is to use the `Vect` type defined in section 3.3 and define a view that un-does vector-append. This is a classic exercise in dependently-typed programming, the interesting question being whether we can implement the function just as seamlessly with our encoding.

Vector append can easily be defined by induction over the first vector.

```
(++) : Vect m a -> Vect n a -> Vect (m + n) a
xs@_ ++ ys with (view xs)
  _ | [] = ys
  _ | hd :: tl = hd :: (tl ++ ys)
```

If the first vector is empty we can readily return the second vector. If it is cons-headed, we can return the head and compute the tail by performing a recursive call.

Equipped with this definition, we can declare the view type which we call `SplitAt` by analogy with its weakly typed equivalent processing lists. It states that a vector `xs` of length `p` can be split at `m` if `p` happens to be `(m + n)` and `xs` happens to be `(pref ++ suff)` where `pref` and `suff`'s respective lengths are `m` and `n`.

```
data SplitAt : (m : Nat) -> (xs : Vect p a) -> Type where
  MkSplitAt : (pref : Vect m a) -> (suff : Vect n a) ->
              SplitAt m (pref ++ suff)
```

The challenge is to define the function proving that a vector of size (`m + n`) can be split at `m`.

B.2 Failing attempts

The proof will necessarily go by induction on `m`, followed by a case analysis on the input vector and a recursive call in the non-zero case.

Our first failing attempt successfully splits the natural number, calls the view on the vector `xs` to take it apart but then fails when performing the recursive call to `splitAt`.

```
failing "tl is not accessible in this context"

  splitAt : (m : Nat) -> (xs : Vect (m + n) a) -> SplitAt m xs
  splitAt 0 xs = MkSplitAt [] xs
  splitAt (S m) xs@_ with (view xs)
    _ | hd :: tl@_ with (splitAt m tl)
      _ | res = ?a
```

This reveals an issue in Idris 2's handling of the interplay between `@`-patterns and quantities: the compiler arbitrarily decided to make the alias `tl` runtime irrelevant only to then complain that `tl` is not accessible when we want to perform the recursive call (`splitAt m tl`)!

In order to work around this limitation, we decided to let go of `@`-patterns and write the fully explicit clause ourselves, using dotted patterns to mark the forced expressions.

```
failing "Can't match on ?postpone [no locals in scope] (User dotted)"

  splitAt : (m : Nat) -> (xs : Vect (m + n) a) -> SplitAt m xs
  splitAt 0 xs = MkSplitAt [] xs
  splitAt (S m) xs@_ with (view xs)
    _ | hd :: tl with (splitAt m tl)
      splitAt (S m) .(hd :: (pref ++ suff))
        | hd :: .(pref ++ suff)
        | MkSplitAt pref suff = ?a
```

The left-hand side now typechecks but the case tree builder fails with a perplexing error. This reveals a bug in Idris 2's implementation of elaboration of pattern-matching functions to case trees. Instead of ignoring dotted expressions when building the case tree (these expressions are forced and so the variables they mention will have necessarily been bound in another pattern), it attempts to use them to drive the case-splitting strategy. This is a well-studied problem and should be fixable by referring to Cockx and Abel's work [CA20].

B.3 Working Around Idris 2's Limitations

This leads us to our working solution. Somewhat paradoxically, working around these Idris 2 bugs led us to a more principled solution whereby the pattern-matching step needed to adjust the result returned by the recursive call is abstracted away in an auxiliary function whose type clarifies what is happening.

From an `m` split on `xs`, we can easily compute an `(S m)` split on `(x :: xs)` by cons-ing `x` on the prefix.

```
(::) : (x : a) -> SplitAt m xs -> SplitAt (S m) (x :: xs)
x :: MkSplitAt pref@(MkVect _ Refl) suff
  = MkSplitAt (x :: pref) suff
```

In this auxiliary function, `xs` is clearly runtime irrelevant and so the case-splitter will not attempt to inspect it, thus generating the correct case tree. We are forced to match further on `pref` (in particular by making the equality proof `Refl`) so that just enough computation happens at the type level for the typechecker to see that things do line up. A proof irrelevant type of propositional equality would have helped us here.

We can put all of these pieces together and finally get our `splitAt` view.

```
splitAt : (m : Nat) -> (xs : Vect (m + n) a) -> SplitAt m xs
splitAt 0 xs = MkSplitAt [] xs
splitAt (S m) xs@_ with (view xs)
  _ | hd :: tl = hd :: splitAt m tl
```

We do want to reiterate that these limitations are not intrinsic limitations of the approach, there are just flaws in the current experimental implementation of the Idris 2 language and can and should be remedied.