Dependent types for low-level programming

In this paper, we bridge the gap between functional programming languages and direct memory access by providing primitives to manipulate pointers, all while using the rest of the language to prove statements about the data structures that those pointers represent. We do this by extending Quantitative Type Theory (QTT).

ACM Reference Format:
. 2023. Dependent types for low-level programming. 1, 1 (June 2023), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Motivation

Dear reader,
imagine yourself writing this program:

```
reverseOn : List a -> List a -> List a
reverseOn acc [] = acc
reverseOn acc (x :: xs) = reverseOn (x :: acc) xs

reverse : List a -> List a
reverse = reverseOn []
```

Fig. 1. How to reverse a list in the Idris programming language

You wrote this function in tail-recursive form because you want it to be fast. You also want to write proofs about your programs so you write the following:

```
reverse_lemma : (xs, ys : _) -> reverse (reverseOn ys xs) = reverseOn xs ys
reverse_lemma [] ys = Refl
reverse_lemma (x :: xs) ys  =
  reverse_lemma xs (x :: ys)

reverse_reverse : (ls : List a) -> reverse (reverse ls) = ls
reverse_reverse [] = Refl
reverse_reverse (x :: xs) = reverse_lemma xs [x]
```

Fig. 2. A proof that reversing a list twice is the identity

Author's address:

This is great, you have a fast implementation of list-reverse and a proof that it behaves the way you'd expect. But something is not quite right. When you benchmark your implementation of list reverse you notice there are a lot of alloc and free operations.

```
main : IO ()
main = do
  let ls = [1,2,3,4]
  printLn $
    reverse $
    ...         -- n times
    reverse $
    ls
```

Number of reverses	Memory in bytes
1	164288
1000	1762944
10000	16160400

Fig. 3. An idris program that reverses a list n times in successtion. And a table of how much memory is consumed by the program for multiple values of n. The program was run using the Chez backend.

This is a bummer, because you know that, since the list is unique, and it is not shared between calls, the implementation of `reverse` ought to perform the reversal in-place without allocating memory. Instead, the increasing amount of memory consumed by the benchmark shows that every reversal allocates a new list in memory.

If you represent a list as a series of cells and pointers between them, the reverse operation can be achieved by only *mutating the pointers* and leaving the data untouched [1]. It should go from this memory layout:

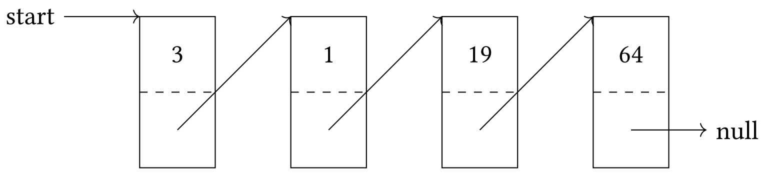

Fig. 4. Linked list in memory, going forward

To this one:

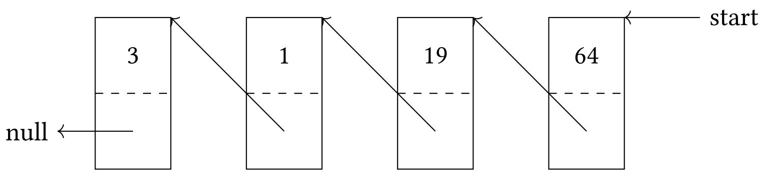

Fig. 5. Linked list in memory, going backward

You can write this program in C.

[1]In this example, we use the numbers 3,1,19,64 to represent data in memory. We don't ascribe any meaning to those numbers, but the fact that they remain in the same positions in the same order in the two examples illustrates the fact that we perform no change to the data itself, only the pointers.

```
Cell * in_place_reverse(Cell * list) {

    Cell * currentTail = NULL;
    Cell * currentNext = NULL;
    Cell * currentList = list;

    while (list) {
        currentList = list;
        currentNext = list->tail;
        list->tail = currentTail;
        currentTail = list;
        list = currentNext;
    }

    return currentList;
}
```

Unfortunately, the cost of writing this memory-efficient version in C, is that we have lost the ability to prove anything about our implementation. This paper is about bridging this gap, writing your high-level programs, data structures and proofs the way you expect from a functional programming language while also allowing to write efficient implementation of those programs.

By the end, it should be clear how the following type signature performs list-reverse in-place without allocating any extra memory:

```
rev_inplace : (0 ls : List a) -> (start, end : Word) ->
  (1 ev : RepList start ls end) -> (s' : Word × RepList s' (reverse ls) end)
```

1.2 Prior art

Some of the success of Functional Programming can be attributed to its ability to reason about programs at a very abstract level. Higher-order functions, pattern matching, inductive data types and immutability have helped the programmer to write software. So much so that commercial programming languages have adopted a number of those features themelves, despite not being advertised as functional programming languages (Java streams, C++ lambda, Javascript destructuring).

However, this high-level approach often comes at the cost of a disconnect with the bare hardware. Relying on a garbage collector obscures the relationship between data structures and their underlying memory layout. And removing the ability to freely control memory allocation renders the use of such programming languages unsuitable for a number of critical applications such as garbage collectors themselves, drivers, schedulers, and other programs that are close to the execution platform they run on.

A lot of work has been done over the years to bring functional programs closer to this realm. Deforestation [Wadler 1990], unboxed values [Jones et al. 1991], reference counting [Reinking et al. 2021], linear types [Bernardy et al. 2017], and more, are all significant advances toward providing a language whose strengths shine in all circumstances. But the feature we would like to use is direct memory manipulation to write algorithms which use cycles, trees with nodes pointing to their siblings, and pointer-based arithmetic.

As of today, the practice is to write those programs in C and either live with the lack of safety or prove statements about the program in another environment like in Iris [Jung et al. 18ed].

To regain safety and keep direct access to memory layout, one can use Rust. It ensures safety of programs through a type system akin to affine types, but shields its lowest-level memory manipulation through an *unsafe* fragment of the language. This itself is not without challenges [Qin et al. 2020]. Haskell gets quite close to the bare metal by using linear types, it also has some form of dependent types which allows it to write some proofs within its type system. However, at the time of writing, those features still are experimental, which makes them unsuitable for our goal of direct memory layout manipulation.

ATS [Xi and Zhu 2018] is the perfect fit for a *full stack* programming language that can manipulate bytes and proofs about those bytes. However, all those facilities are difficult to integrate together. In particular, there is no way to *directly* relate the implementation of a function written inductively with the implementation written with pointers. Where ATS gets very close to our goal is by allowing us to write a definition using its linear fragement, and then write proofs about it.

What we achieve in this paper is similar in spirit to what is achieved by [Allais 2023], where the data structure has the API one expects in a high-level Functional Programming language, and the implementation is closer to what one expects from a low-level bare-metal implementation. Our low-level implementation works at the pointer-level and is based on linear types rather than working at the byte-level and being based on erased values.

1.3 Structure of this paper

We start by presenting our extension to QTT: PtrQTT (QTT with pointers). Then we present the building-blocks that link existing functional programming langauges into PtrQTT. Finally we show how to use PtrQTT to implement traditional pointer-heavy operations such as in-place list reversal. Our contributions are as follows:

- We extend QTT to be able to read and write pointers.
- We provide an implementation in Idris demonstrating the capabilities of such system.

2 THE META THEORY

This section will recap the basics of Quantitative Type theory and then extend it with primitives in order to represent the programs we wish to write. Finally we conclude with how we relate this new theory to existing ones such as separation logic.

2.1 QTT Recap

Quantitative Type Theory (QTT) combines quantitative reasoning and Martin-Löf Type Theory [Martin-Löf [n. d.]].

QTT is parameterised over a semiring $(R : Type, \cdot : R \to R \to R, + : R \to R \to R, 0 : R, 1 : R)$ with the following laws 0 is the neutral element for $(+)$ and 1 the neutral for $(\cdot)$.

- $(R, +, 0)$ is a commutative monoid, $x + y = y + x$
- $(R, \cdot, 1)$ is a monoid
- $\forall x : R. x \cdot 0 = 0 \cdot x = 0$
- $\forall x, y : R. x + y = 0 \Rightarrow x = 0 \land y = 0$
- $\forall x, y : R. x \cdot y = 0 \Rightarrow x = 0 \lor y = 0$

For our purposes we use the semiring with values $(0, 1, \omega)$ where 0 and 1 are the neutral elements we expect and ω absorbs other values except 0 $\omega \cdot 0 = 0$, $\omega \cdot \omega = \omega$, $\omega \cdot 1 = \omega$. Semantically, 0 indicates an erased value, 1 a linear value and ω an unrestricted one. The crux of QTT is that type-formers live in the *erased* fragment of the language, while runtime values live in the rest of the language. While we refer to the seminal literature for more details ([Atkey 2018], [McBride 2016]) it is important to aknowledge that weakening is admissible:

$$\frac{\Gamma, \Gamma' \vdash \mathcal{J} \qquad 0\Gamma \vdash U}{\Gamma, x :^0 U, \Gamma' \vdash \mathcal{J}}\ \textsc{Weaken}$$

$\mathcal{J}$ indicates that the context is valid and U is a valid type. The rule states that given a valid context that we can split between Γ and Γ' we can construct a valid context $\Gamma, x :^0 U, \Gamma'$ where x is unused.

Weakening of contexts is crucial in how to relate this version of QTT with separation logic.

2.2 Low-Level QTT

We implement a type system that exhibits all the necessary tools by extending Atkey's QTT [Atkey 2018] with an additional type and two additional functions to manipulate values of this type:

$$\frac{}{\Gamma \vdash \mathrm{Word} : \mathrm{Type}}\ \textsc{Word} \qquad \frac{\Gamma \vdash x : \mathrm{Word}}{\Gamma \vdash \mathrm{Inc}\ x : \mathrm{Word}}\ \textsc{Inc}$$

$$\frac{0\Gamma \vdash a :^0 \mathrm{Word}, b :^0 \mathrm{Word}}{0\Gamma \vdash a \rightsquigarrow b : \mathrm{Type}}\ \textsc{Points-To}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : \mathrm{Word}, b :^0 \mathrm{Word}, ev :^1 a \rightsquigarrow b \\ 0\Gamma, x :^0 \mathrm{Word}, p :^0 a \rightsquigarrow x, prf :^0 b = x \vdash M\ x\ p\ prf :^0 \mathrm{Type}\end{array}}{\Gamma, x : \mathrm{Word}, p :^1 a \rightsquigarrow x, prf :^0 b = x \vdash M\ x\ p\ prf}\ \textsc{Read}$$

$$\frac{\Gamma \vdash a : \mathrm{Word}, b :^0 \mathrm{Word}, ev :^1 a \rightsquigarrow b, val : \mathrm{Word}}{\Gamma \vdash a \rightsquigarrow val}\ \textsc{Write}$$

The type $\rightsquigarrow$ is read as "points to" and so $a \rightsquigarrow b$ reads "a points to b" as to say "the pointer a points to the value b". You will notice that every single use of $\rightsquigarrow$ is in a linear position. This is paired with QTT's ability to assign quantities at binding site, by forbidding sub-usaging, we ensure that the only way to use a $\rightsquigarrow$ value is to bind it with quantity 1 when it is produced.

We make use of Σ to return multiple values together, because Σ can be bound with any multiplicity, we ensure that $a \rightsquigarrow x$ is bound with multiplicity 1.

We could very well have this program:

```
let ω v = write ... in ...
```

But without subusaging between unrestricted and linear variables it is impossible for "v" to be consumed with "read" or "write".

2.3 Relation with separation logic

Our introduction of the primitives $a \rightsquigarrow b$ immitates what one would expect from separation logic. Indeed our functions declare in what memory state they expect and return another memory state. The type signature of our programs layout the memory layout before the function runs and the return type informs how the memory layout has been changed.

This function requires a memory layout containing at least two pointers a and b pointing to x and y respectively, and then returns evidence that the memory layout has been altered such that a points to y and b points to x.

```
swap : (a, b : Word) -> (1 p1 : a ~> x) -> (1 p2 : b ~> y)
    -> LPair (a ~> y) (b ~> x)
swap a b p1 p2 = readCont _ _ p1 $ \x', xPrf, xEv =>
                 readCont _ _ p2 $ \y', yPrf, yEv =>
                     rewrite yPrf in write _ _ xEv y'
                   # rewrite xPrf in write _ _ yEv x'
```

Fig. 6. A function that swaps the content of two pointers in memory

The implementation (after removing rewrites) is fairly straightforward: read the two values from the given pointers a and b and overwrite their previous values. We will see in the next section what `readCont` is.

3 LOW-LEVEL QTT IN PRACTICE

In this section, we demonstrate the practicality of such system by implementing our example in Idris2 [Brady 2021] which implements QTT.

3.1 Bootstraping Postulates

To get us off the ground we are going to use `Word` as an alias for `Int64` and `Inc : Word -> Word` can just be the integer increment function.

```
Word : Type
Word = Int32

Inc : Word -> Word
Inc = (+ 1)
```

We need to write our $a \rightsquigarrow b$ introduction rule and elimination rule.

```
write : (location : Word) -> (0 value : Word) ->
        (1 ptr : location ~> value) -> (newValue : Word) ->
        (location ~> newValue)
```

We write our dependent eliminator for $\rightsquigarrow$ and we also write a non-dependent version with an easily inferrable motive.

```
readElim : (loc : Word) -> (0 val : Word) -> (1 ev : loc ~> val) ->
    (0 motive : (actual : Word) -> (val === actual) -> (loc ~> actual) -> Type) ->
    (1 f : (actual : Word) -> (0 prfEq : val === actual) ->
           (1 ev : loc ~> actual) -> motive actual prfEq ev) ->
    motive val Refl ev

readCont :
     (location : Word)
  -> (0 value : Word)
  -> (1 ptr : location ~> value)
  -> (1 f : (actual : Word)
         -> (0 samePrf : value === actual)
         -> (1 ptr : location ~> actual)
         -> x)
  -> x
readCont location value ptr = readElim  location value ptr (\_,_,_ => x)
```

With those primitives we have everything we need to write program such as swap 6.

3.2 List Representation

There are a lot of different list representations, we are going to focus on a singly forward linked-list. Typically we represent those like in figure 4. Using C we define a list cell as a struct with two pointers, one to a value, and one to another list cell, we call this the tail pointer. If the tail pointer is `NULL` we have reached the end of the list.

```
struct Cell {
    int * value;
    Cell * tail;
};
```

Fig. 7. A List cell in C.

```
data List a = Nil
            | Cons a (List a)
```

Fig. 8. An inductive defintiion of list.

What we are looking for is a way to describe the memory layout in figure 7, while keeping around all the fact that we know from list as an inductive defintion:

For this we need a constructor that looks a bit like this:

```
RepList : List a -> Type
```

It describes a family of types inhabited by *runtime representations* of its index.

A function `rev` that reverses a list would take an existing memory layout that represents a list and return a new one with the added guarantee that the semantic list attached to is has been reversed in the way defined by `reverse` from figure 1.

We also want to specify that the list representation is concrete at runtime but the semantic list is erased from the runtime. This way, we guarantee that the only memory allocation we will see is the one from our efficient memory implementation. We borrow notation from [McBride 2016] and write:

```
rev : (0 ls : List a) -> RepList ls -> RepList (reverse ls)
```

The `RepList` data structure packs the pointer layout in memory that stores the list. This way we can read this type signature as "given a list, and a memory state that holds this list, returns a memory state that holds the reverse list".

To fully represent our list as pointers we also need a starting pointer and an ending pointer. They will tell us how to get a hold on the head of the list, and when we have reached the end of the list, respectively. While we are defining `RepList` inductively, we will see in section 3.3 how this is not an issue for runtime performance.

With that said here is the full definition of `RepList`:

```
data RepList : (start : Word) -> (content : List Word) -> (end : Word) -> Type where
  Empty : RepList x [] x
  Cell : (1 notEnd : So (start /= end))
      -> (1 ev1 : start ~> value)
      -> (1 ev2 : Inc start ~> next)
      -> (1 evt : RepList next tail end)
      -> RepList start (value :: tail) end
```

With this definition we are able to represent and implement list reverse with in-place mutation of pointers by leveraging the read and write operations

```
rev_helper :
      (current : Word) ->
      (newStart : Word) ->
      (end : Word) ->
      (1 acc : RepList newStart cs end) ->
      (1 left : RepList current ys end) ->
      DPairUL Word (\s' =>
              RepList s' (reverseOn cs ys) end)
rev_helper current newStart end acc left =
  case matchView current end left of
       LRight IsEmptyRep => newStart !# acc
       LLeft (IsCellRep {prf=notEnd} {val} {toTail=tailPtr} {tail=tailEv}) =>
            readCont (Inc current) _ tailPtr $ \tVal, tPrf, tEv =>
              let c' = Cell notEnd val (write _ _ tEv newStart) acc
              in rev_helper tVal current end c'
                   (rewrite sym tPrf in tailEv)

rev : (start, end : Word) ->
      RepList start content end -@
      DPairUL Word (\start' => RepList start' (reverse content) end)
rev start end listEv = rev_helper start end end Empty listEv
```

By which we start with a proof that the end pointer is also a valid empty List. The `revOn` function then moves the forward pointers back and replace the current pointer by the next. The evidence that the accumulator builds the reverse list is tracked at the type while the pointers drop in and out of the stack without allocating any memory on the heap.

This program makes use of a number of constructions we have not mentionned yet, we tackle them now.

3.2.1 the `matchView` *function.*

André: todo

`matchView`, a function that analyses the given pointer, and returns a *view* of the list. It is important that we do not match on the proof of `RepList`.

3.2.2 Singleton Evidence. The `matchView` function returns a choice of singletons. One of them is `IsEmptyRep` which only value is evidence that the last pointer is an end pointer and therefore the list is empty. Similarly `IsCellRep`'s only inhabitant is evidence that the list is a cons cell.

```
data EmptyRep : RepList start content end -> Type where
  IsEmptyRep : EmptyRep {content = []} Empty

data CellRep : RepList start content end -> Type where
  IsCellRep : {1 prf, val, toTail, tail : _} ->
              CellRep (Cell prf val toTail tail)
```

We return those singleton values in an `Either` type which ensures its left and right values are linear, it's constructors are therefore `LLeft` and `LRight`.

3.2.3 Returning Pointers.

André: todo

`DPairUL`, a type constructor for sigma types with quantities. The convention used here is that the first projection is unrestructed but the second projection is linear. We use this to return a pair of a pointer (unrestricted) along with it's accompanying linear proof.

3.3 A note on performance

Intuitively, one might ask themselves why we even bother with this if we need to carry around and pattern-match on a proof of memory layout, rather than only carry the data. Does that not entirely defeats the purpose of low-level programming?

A detailed look into the `RepList` data structure reveals its secrets: it is made of mostly nothing. `Empty` contains no data, it is a tagged unit value. `Cell` contains `So` which is also a unit, and the *points to* predicates are uninhabited. The last argument is another `RepList` which is itself made of mostly nothing.

Given this knowledge, one could concieve of a compiler that identifies the structure of this data and optimises it away, since this entire data structure is nothing but smoke and units. Its sole purpose is to provide compile-time evidence that the memory layout represents the data for which we wrote proofs. This is something that ATS identifies explicitly and calls a dataview, a linear prop (what ref can I use for Prop?) which has no runtime representation but needs to be linearly consumed.

In the current state of the art, the `RepList` data structure could be identified as a `Nat` which itself could be represented as an `Int`. This would allow the program to be equivalent to a list-as-memory-layout along with a number which represents its length.

A closer look at `matchView` tells us that, in order to extract the proofs of pointer ownership, we need to match on an `Either` type. But the types returned by the either are themselves prop-like singletons, which suggests our `Either` type is nothing but a `Bool` in disguise.

4 CONCLUSION AND FUTURE WORK

We were able to directly tie our low-level program with our inductive definition of the same program. But our approach could still benefit from many improvements. For example, we only talk about lists that contain `Words` values, but we would like to support lists that contain pointers to richer data structures. Another thing we would like to explore is the generation of representable data like the `RepList` data definition for any other strictly postive data type.

Finally, given the relationship with separation logic, it would be interesting to reproduce some of their results using term inference provided by modern compilers.

REFERENCES

Guillaume Allais. 2023. Builtin Types Viewed as Inductive Families. https://doi.org/10.48550/arXiv.2301.02194 arXiv:2301.02194 [cs] Comment: 32 pages, accepted for publication at ESOP'23.

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '18*. ACM Press, Oxford, United Kingdom, 56–65. https://doi.org/10.1145/3209108.3209189

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. (Oct. 2017). https://doi.org/10.1145/3158093

Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. *arXiv:2104.00480 [cs]* (April 2021). arXiv:2104.00480 [cs] http://arxiv.org/abs/2104.00480 Comment: To appear in proceedings of ECOOP 2021.

SL Peyton Jones, J. Launchbury, and Simon Peyton Jones. 1991. Unboxed Values as First Class Citizens. In *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*. 636–666. https://www.microsoft.com/en-us/research/publication/unboxed-values-as-first-class-citizens/

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018/ed. Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programming* 28 (2018/ed), e20. https://doi.org/10.1017/S0956796818000151

Per Martin-Löf. [n. d.]. *An Intuitionistic Theory of Types.*

Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Vol. 9600. Springer International Publishing, Cham, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. https://doi.org/10.1145/3385412.3386036

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage Free Reference Counting with Reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 96–111. https://doi.org/10.1145/3453483.3454032

Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science* 73, 2 (June 1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

Hongwei Xi and Dengping Zhu. 2018. To Memory Safety through Proofs. https://doi.org/10.48550/arXiv.1810.12190 arXiv:1810.12190 [cs]