NumLin: Linear Types for Linear Algebra

Dhruv C. Makwana
Unaffiliated
https://dhruvmakwana.com
dcm41@cam.ac.uk

Neelakantan R. Krishnaswami
Department of Computer Science and Technology, University of Cambridge, United Kingdom
https://www.cl.cam.ac.uk/~nk480/
nk480@cl.cam.ac.uk

Abstract
We present NumLin, a functional programming language whose type system is designed to enforce the safe usage of the APIs of low-level linear algebra libraries (such as BLAS/LAPACK). We do so through a brief description of its key features and several illustrative examples. We show that NumLin’s type system is sound and that its implementation improves upon naïve implementations of linear algebra programs, almost towards C-levels of performance. By doing so, we demonstrate (a) that linear types are well-suited to expressing the APIs of low-level linear algebra libraries accurately and concisely and (b) that, despite the complexity of prior work on it, fractional permissions can actually be implemented using simple, well-known techniques and be used practically in real programs.

2012 ACM Subject Classification Theory of computation → Program specifications
Keywords and phrases numerical, linear, algebra, types, permissions, OCaml

Introduction
Programmers writing numerical software often find themselves caught on the horns of a dilemma. The foundational, low-level linear algebra libraries such as BLAS and LAPACK offer programmers very precise control over the memory lifetime and usage of vector and matrix values. However, this power comes paired with the responsibility to manually manage the memory associated with each array object, and in addition to bringing in the familiar difficulties of reasoning about lifetimes, aliasing and sharing that plague low-level systems programming; this also moves the APIs away from the linear-algebraic, mathematical style of thinking that numerical programmers want to use.

As a result, programmers often turn to higher-level languages such as Matlab, R and NumPy, which offer very high-level array abstractions that can be viewed as ordinary mathematical values. This makes programming safer, as well as making prototyping and verification much easier, since it lets programmers write programs which bear a closer resemblance to the formulas that the mathematicians and statisticians designing these algorithms prefer to work with, and ensures that program bugs will reflect incorrectly-computed values rather than heap corruption.
The intention is that these languages can use libraries BLAS and LAPACK, without having to expose programmers to explicit memory management. However, this benefit comes at a price: because user programs do not worry about aliasing, the language implementations cannot in general exploit the underlying features of the low-level libraries that let them explicitly manage and reuse memory. As a result, programs written in high-level statistical languages can be much less memory-efficient than programs that make full use of the powers the low-level APIs offer.

So in practice, programmers face a trade-off: they can eschew safety and exploit the full power of the underlying linear algebra libraries, or they can obtain safety at the price of unneeded copies and worse memory efficiency. In this work, we show that this trade-off is not a fundamental one.

**NumLin** is a functional programming language whose type system is designed to enforce the safe usage of the APIs of low-level linear algebra libraries (such as BLAS/LAPACK). It does so by combining linear types, fractional permissions, runtime errors and recursion into a small, easily understandable, yet expressive set of core constructs.

**NumLin** allows a novice to understand and work with complicated linear algebra library APIs, as well as point out subtle aliasing bugs and reduce memory usage in existing programs. In fact, we were able to use **NumLin** to find linearity and aliasing bugs in a linear algebra algorithm that was generated by another program specifically designed to translate matrix expressions into an efficient sequence of calls to linear algebra routines. We were also able to reduce the number of temporaries used by the same algorithm, using **NumLin**’s type system to guide us.

**NumLin**’s implementation supports several syntactic conveniences as well as a usable integration with real OCaml libraries.

### 1.1 Contributions

Our contribution is the idea applying of linear types with fractional permissions to enforce the correct usage (as opposed to implementation) of linear algebra libraries. We explain the idea in detail and provide evidence for its efficacy. Prior type systems for fractional permissions [11, 9, 8] are quite complex. This is because these type systems typically encode a sophisticated analysis to automatically infer how fractional permissions should be split and rejoined.

In contrast, in **NumLin**, we made sharing and merging explicit. As a result, we were able to drastically simplify the type system. Therefore, our formal system is very close to standard presentations of linear logic, and the implementation complexity is no worse than that for parametric polymorphism.

In this paper

- we describe **NumLin**, a linearly typed language for linear algebra programs
- we illustrate that **NumLin**’s design and features are well-suited to its intended domain with progressively sophisticated examples
- we prove **NumLin**’s soundness, using a step-indexed logical relation
- we describe a very simple, unification based type-inference algorithm for polymorphic fractional permissions (similar to ones used for parametric polymorphism), demonstrating an alternative approach to dataflow analysis [9]
- we describe an implementation that is compatible with and usable from existing code
- we show an example of how using **NumLin** helped highlight linearity and aliasing bugs, and reduce the memory usage of a generated linear algebra program
we show that using NumLin, we can achieve parity with C for linear algebra routines, whilst having much better static guarantees about the linearity and aliasing behaviour of our programs.

2 NumLin Overview and Examples

2.1 Type System and Other Features

The core type theory of NumLin is a nearly off-the-shelf linear type theory [3], supporting familiar features such as linear function spaces \( A \rightarrow B \) and tensor products \( A \otimes B \). We adopt linearity – the restriction that each program variable be used exactly once – since it allows us to express purely functional APIs for numerical library routines that mutate arrays and matrices [20]. Due to linearity, values cannot alias and are only used once, which means that linearly-typed updates result in no observable mutation.

As a result, programmers can reason about NumLin expressions as if they were ordinary mathematical expressions – as indeed they are! We are merely adopting a stricter type discipline than usual to make managing memory safe.

2.1.1 Intuitionism: ! and Many

However, linearity by itself is not sufficient to produce an expressive enough programming language. For values such as booleans, integers, floating-point numbers as well as pure functions, we need to be able to use them intuitionistically, that is, more than once or not at all. For this reason, we have the ! constructor at the type level and its corresponding Many constructor and let Many <id> = .. in .. eliminator at the term level. Because we want to restrict how a programmer can alias pointers and prevent a programmer from ignoring them (a memory leak), NumLin enforces simple syntactic restrictions on which values can be wrapped up in a Many constructor (details in Section 3).

2.1.2 Fractional Permissions

There are also valid cases in which we would want to alias pointers to a matrix. The most common is exemplified by the BLAS routine gemm, which (rather tersely) stands for GEneric Matrix Multiplication. A simplified definition of gemm(\( \alpha \), A, B, \( \beta \), C) is \( C := \alpha AB + \beta C \). In this case, A and B may alias each other but neither may alias C, because it is being written to. Related to mutating arrays and matrices is freeing them. Here, we would also wish to restrict aliasing so that we do not free one alias and then attempt to use another. Although linearity on its own suffices to prevent use-after-free errors when values are not aliased (a freed value is out of scope for the rest of the expression), we still need another simple, yet powerful concept to provide us with the extra expressivity of aliasing without losing any of the benefits of linearity.

Fractional permissions provide exactly this. Concretely, types of (pointers to) arrays and matrices are parameterised by a fraction. A fraction is either 1 (\( 2^0 \)) or exactly half of another fraction (\( 2^{-k} \), for natural k). The former represents complete ownership of that value: the programmer may mutate or free that value as they choose; the latter represents read-only access or a borrow: the programmer may read from the value but not write to or free it. Creating an array/matrix gives you ownership of it, so too does having one (with a fractional permission of \( 2^0 \)) passed in as an argument.

In NumLin, we can produce two aliases of a single array/matrix, by sharing it. If the original alias had a fractional permission of \( 2^{-k} \) then the two new aliases of it will have a fractional permission of \( 2^{-(k+1)} \) each. Thanks to linearity, the original array/matrix with a
fractional permission of $2^{-k}$ will be out of scope after the sharing. When an array/matrix is shared as such, we can prevent the programmer from freeing or mutating it by making the types of free and set (for mutation) require a whole ($2^0$) permission.

If we have two aliases to the same matrix with identical fractional permissions ($2^{-(k+1)}$), we can recombine or unshare them back into a single one, with a larger $2^{-k}$ permission. As before, thanks to linearity, the original two aliases will be out of scope after unsharing.

### 2.1.3 Runtime Errors

Aside from out-of-bounds indexing, matrix unsharing is one of only two operations that can fail at runtime (the other being dimension checks, such as for gemm). The check being performed is a simple sanity check that the two aliasing pointers passed to unshare point to the same array/matrix. Section 5 contains an overview of how we could remove the need for this by tracking pointer identities statically by augmenting the type system further.

### 2.1.4 Recursion

The final feature of NumLin which makes it sufficiently expressive is recursion (and of course, conditional branches to ensure termination). Conditional branches are implemented by ensuring that both branches use the same set of linear values. A function can be recursive if it captures no linear values from its environment. Like with Many, this is enforced via simple syntactic restrictions on the definition of recursive functions.

### 2.2 Syntax

NumLin’s concrete syntax is inspired by that of OCaml. It desugars (Figure 2) into a smaller, core type and expression grammar (Figure 1).

As described in Section 2.1.2, fractional permissions $f$ are either variables $\prime fc$, z ($2^0$) or $fs$ ($2^{-(f+1)}$).

Types are either simple (unit, bool, int, elt), indexed by fractional permission ($f$ arr, $f$ mat) or compound (! constructor for intuitionism, universally-quantifying over a fractional-permission $fc$ in t for fraction-polymorphic types, pairs, linear functions).

Expressions are standard with the exception of $\prime fc$. t introduction (fractional permission abstraction) and elimination (fractional permission specialisation) forms.

The ! annotation on variables is a syntactic convenience for declaring the variable to used intuitionistically; why it desugars the way it does is explained in Section 3.1.

Array/matrix indexing and duplication (non-destructive and destructive) also have special syntax to lessen the syntactic overhead of re-binding identifiers. Furthermore, to aid readability, there is support for using BLAS methods via conventional-looking matrix expressions.

In particular, the syntax let $y$ <- new $(m, n)$ [| alpha * x1 * x2 |] is syntactic sugar for first creating a new $m \times n$ matrix (let $y$ = matrix $m$ $n$) and then storing the result of the multiplication in it (let ((x1, x2), $y$) = .. in ..).

Note that, the pattern let $y$ <- [| $x^T$ * $x$ + beta * $y$ |] translates to (syrk true 1. $x$ beta $y$), which uses $x$ once only.
\begin{align*}
f & ::= \ 'fc | z | f \ s \\
t & ::= \ \text{unit} | \ \text{bool} | \ \text{int} | \ \text{elt} | \ f \ \text{arr} | \ f \ \text{mat} | \ !t | \ 'fc, \ t | t \otimes t' | t \rightarrow t' \\
e & ::= \ p \ (\text{primitives}) | \ x \ (\text{variable}) | \ \text{let} \ x = e \ \text{in} \ e' | () | \ \text{let} \ () = e \ \text{in} \ e' | \ \text{true} | \ \text{false} \\
\text{if} \ e \ \text{then} \ e_1 \ \text{else} \ e_2 & | \ k \ (\text{integer}) | \ l \ (\text{heap location}) | \ el \ (\text{array element}) \\
\text{Many} \ v & | \ \text{let Many} \ x = e \ \text{in} \ e' | \ \text{fun} \ 'fc \rightarrow e \ (\text{frac. perm. abstraction}) \\
e \ [f] & (\text{frac. perm. specialisation}) | \ (e, e') | \ \text{let} \ (a, b) = e \ \text{in} \ e' \\
\text{fun} \ x : t \rightarrow e & | e \ e' | \ \text{fix} \ (g, x : t, e : t')
\end{align*}

\textbf{Figure 1} Core fraction $f$, type $t$ and expression $e$ grammar of NumLIN. Values $v$ are a subset of the expressions, their full definition and a list of all primitives $p$ is in the Appendix.

\begin{align*}
x[e] & \Rightarrow \ \text{get} \ _x \ (e) \quad \text{(similarly for matrices)} \\
x[e_1] := e_2 & \Rightarrow \ \text{set} \ x \ (e_1) \ (e_2) \quad \text{(similarly for matrices)} \\
\text{pat} & ::= () | x | !x | \text{Many} \ \text{pat} | \ (\text{pat}, \ \text{pat}) \\
\text{let} \ !x = e_1 \ \text{in} \ e_2 & \Rightarrow \ \text{let Many} \ x = e_1 \ \text{in} \ e_2 \\
\text{let Many} \ (\text{pat}_a) = e_1 \ \text{in} \ e_2 & \Rightarrow \ \text{let Many} \ x = x \ \text{in} \ e_2 \\
\text{let} \ ((\text{pat}_a, \ \text{pat}_b)) = e_1 \ \text{in} \ e_2 & \Rightarrow \ \text{let} \ (a, b) = a \_b \ \text{in} \ \text{let} \ (\text{pat}_a) = a \ \text{in} \\
\text{let} \ (\text{pat}_a) = b \ \text{in} \ e_2 & \Rightarrow \ \text{let} \ (\text{pat}_a) = b \ \text{in} \ e_2 \\
\text{fun} \ ((\text{pat}_a) : t) \rightarrow e & \Rightarrow \ \text{fun} \ (x : t) \ \rightarrow \ \text{let} \ (\text{pat}_a) = x \ \text{in} \ e \\
\text{arg} & ::= ((\text{pat}) : t) | \ '(x) \ (\text{fractional permission variable}) \\
\text{fun} \ (\text{arg}_{1..n}) \rightarrow e & \Rightarrow \ \text{fun} \ (\text{arg}_1) \rightarrow .. \ \text{fun} \ (\text{arg}_n) \rightarrow e \\
\text{let} \ f \ (\text{arg}_{1..n}) = e_1 \ \text{in} \ e_2 & \Rightarrow \ \text{let} \ f = \text{fun} \ (\text{arg}_{1..n}) \rightarrow e_1 \ \text{in} \ e_2 \\
\text{fixpoint} & \equiv \ \text{fix} \ (f, x : t, \text{fun} \ (\text{arg}_{0..n}) \rightarrow e_1 : t') \\
\text{let rec} \ f \ (x : t) \ (\text{arg}_{0..n}) : t' = e_1 \ \text{in} \ e_2 & \Rightarrow \ \text{let} \ f = \text{fixpoint} \ \text{in} \ e_2 \\
\text{let rec} \ f \ (x : t) \ (\text{arg}_{0..n}) : t' = e_1 \ \text{in} \ e_2 & \Rightarrow \ \text{let Many} \ f = \text{Many} \ \text{fixpoint} \ \text{in} \ e_2
\end{align*}

\textbf{Figure 2} Desugaring NumLIN.
let \( v \leftarrow x[e] \) in \( e' \) ⇒ let \((x,v)\) = \( x[e] \) in \( e' \) (similarly for matrices)

let \( x_2 \leftarrow \text{new } [1][x_1] \) in \( e \) ⇒ let \((x_1,x_2)\) = \( \text{copyM } _- x_1 \) in \( e \)

let \( x_2 \leftarrow [1][x_1] \) in \( e \) ⇒ let \((x_1,x_2)\) = \( \text{copyM } _{-} x_1 \) \( x_2 \) in \( e \)

\[
M ::= X \mid X^T \mid \text{sym}(X)
\]

let \( Y \leftarrow \text{new } (n,k) [1][\alpha M_1 M_2] \) in \( e \) ⇒

let \( Y = \text{matrix } n \) \( k \) \( \text{in } Y \leftarrow [1][\alpha M_1 M_2 + 0 Y] \) in \( e \)

let \( Y \leftarrow [1][\alpha X^T X + \beta Y] \) in \( e \) ⇒

let \((X,Y)\) = \( \text{syrk } \text{false } \alpha \_ X \_ \beta Y \) in \( e \)

let \( Y \leftarrow [1][\alpha \text{sym}(X_1) X_2 + \beta Y] \) in \( e \) ⇒

let \((X_1,X_2),Y\) = \( \text{symm } \text{false } \alpha \_ X_1 \_ X_2 \_ \beta Y \) in \( e \)

let \( Y \leftarrow [1][\alpha X_2 \text{sym}(X_1) + \beta Y] \) in \( e \) ⇒

let \((X_1,X_2),Y\) = \( \text{symm } \text{true } \alpha \_ X_1 \_ X_2 \_ \beta Y \) in \( e \)

let \( Y \leftarrow [1][\alpha X_1^T X_2^T + \beta Y] \) in \( e \) ⇒

let \((X_1,X_2),Y\) = \( \text{gemm } \alpha \_ (X_1,\text{false}) \_ (X_2,\text{true}) \_ \beta Y \) in \( e \)

- Figure 3: Purely syntactic pattern-matching translations of matrix expressions.

### 2.3 Examples

#### 2.3.1 Factorial

Although a factorial function (Figure 4) may seem like an aggressively pedestrian first example, in a linearly typed language such as \textsc{NumLin} it represents the culmination of many features.

To simplify the design and implementation of \textsc{NumLin}'s type system, recursive functions must have full type annotations (non-recursive functions need only their argument types annotated). The body of the factorial function is a closed expression (with respect to the function’s arguments), so it type-checks (since it does not capture any linear values from its environment).

The only argument is \( !x : !\text{int} \). As explained before (Section 2.2), this declares \( x \) to be used intuitionistically.

The condition for an if may or may not use linear values (here, with \( x < 0 \mid \mid x = 0 \), it does not). Any linear values used by the condition would not be in scope in either branch of the if-expression. Both branches use \( x \) differently: one ignores it completely and the other uses it twice.

All numeric and boolean literals are implicitly wrapped in a \texttt{Many} and all primitives involving them return a \( !\text{int}, !\text{bool} \) or \( !\text{elt} \) (types of elements of arrays/matrices, typically 64-bit floating-point numbers). The short-circuiting \( \mid \mid \) behaves in exactly the same way as a boolean-valued if-expression.
let rec factorial (x : int) : int =  
  if x < 0 || x = 0 then  
    1  
  else  
    x * factorial (x - 1) in  
  factorial ;;

Figure 4 Factorial function in NumLin.

let rec sum_array (i : int) (n : int) (x0 : elt) (x) (row : x arr) : x arr * elt =  
  if i = n then  
    (row, x0)  
  else  
    let (row, x1) = row[i] in  
    sum_array (i + 1) n (x0 +. x1) x row in  
  sum_array ;;

Figure 5 Summing over an array in NumLin.

2.3.2 Summing over an Array

Now we can add fractional permissions to the mix: Figure 5 shows a simple, tail-recursive implementation of summing all the elements in an array. There are many new features; first among them is !x0 : !elt, the type of array/matrix elements (64-bit floating point).

Second is ('x) (row : 'x arr) which is an array with a universally-quantified fractional permission. In particular, this means the body of the function cannot mutate or free the input array, only read from it. If the programmer did try to mutate or free row, then they would get a helpful error message (Figure 6).

Alongside taking a row : 'x arr, the function also returns an array with exactly the same fractional permission as the row (which can only be row). This is necessary because of linearity: for the caller, the original array passed in as an argument would be out of scope for the rest of the expression, so it needs to be returned and then rebound to be used for the rest of the function.

An example of this consuming and re-binding is in let (row, x1) = row[i]. Indexing is implemented as a primitive get : 'x. 'x arr ⇒ lint ⇒ 'x arr ⊗ !elt. Although fractional permissions can be passed around explicitly (as done in the recursive call), they can also be automatically inferred at call sites: row[i] ⇒ get _ row i takes advantage of this convenience.

2.3.3 One-dimensional Convolution

Figure 7 extends the set of features demonstrated by the previous examples by mutating one of the input arrays. A one-dimensional convolution involves two arrays: a read-only kernel (array of weights) and an input vector. It modifies the input vector in-place by replacing each write[i] with a weighted (as per the values in the kernel) sum of it and its neighbours; intuitively, sliding a dot-product with the kernel across the vector.
let row = row[i] := x1 in (* or *) let () = free row in

(* Could not show equality: *)
(* z arr *)
(* with *)
(* 'z arr *)
(* *)

(* Var 'z is universally quantified *)
(* Are you trying to write to/free/unshare an array you don't own? *)
(* In examples/sum_array.lt, at line: 7 and column: 19 *)

Figure 6 Attempting to write to or free a read only array in NumLin.

let rec simp_oned_conv (!i : !int) (!n : !int) (!x0 : !elt)
    (write : z arr) ('x) (weights : 'x arr)
    : 'x arr * z arr =
    if n = i then (weights, write) else
    let !w0 <- weights[0] in
    let !w1 <- weights[1] in
    let !w2 <- weights[2] in
    let !x1 <- write[i] in
    let !x2 <- write[i + 1] in
    let written = write[i] := w0 *. x0 +. (w1 *. x1 +. w2 *. x2) in
    simp_oned_conv (i + 1) n x1 written _ weights
    simp_oned_conv ;;

Figure 7 Simplified one-dimensional convolution.

What’s implemented in Figure 7 is a simplified version of this idea, so as to not distract from the features of NumLin. The simplifications are:

- the kernel has a length 3, so only the value of write[i-1] (prior to modification in the previous iteration) needs to be carried forward using x0
- write is assumed to have length n+1
- i’s initial value is assumed to be 1
- x0’s initial value is assumed to be write[0]
- the first and last values of write are ignored.

Mutating an array is implemented similarly to indexing one – a primitive set : z arr → !int → 'elt → z arr. It consumes the original array and returns a new array with the updated value.

Since write: z arr (where z stands for k = 0, representing a fractional permission of \(2^{-k} = 2^{-0} = 1\)), we may mutate it, but since we only need to read from weights, its fractional permission index can be universally-quantified. In the recursive call, we see _ being used explicitly to tell the compiler to infer the correct fractional permission based on the given arguments.

2.3.4 Digression: Types of Primitives

The most pertinent aspect of NumLin is the types of its primitives (Figure 8). While the types of operations such as get and set might be borderline obvious, the types of BLAS/LAPACK routines become an incredibly useful, automated check for using the API correctly.

We determine the types for these routines by consulting their documentation. Each routine has a record of the expected aliasing behaviour and whether or not it modifies or
symm : !bool → elt → 'x 'x mat → 'y 'y mat → !elt → z mat →
  ('x mat ⊗ 'y mat) ⊗ z mat

gemm : !elt → 'x 'x mat ⊗ !bool → 'y 'y mat ⊗ !bool → !elt →
  z mat → ('x mat ⊗ 'y mat) ⊗ z mat

gesv : z mat → z mat → z mat ⊗ z mat

gesv : z mat → z mat → z mat ⊗ z mat

gesv : z mat → z mat → z mat ⊗ z mat

Figure 8 Types of some NumLin primitives.

```
let !square ('x) (x : 'x mat) =
  let (x, (m, n)) = sizeM _ x in
  let (x1, x2) = shareM _ x in
  let answer <- new (m, n) [| x1 * x2 |] in
  let x = unshareM _ x1 x2 in
  (x, answer) in
square ;;
```

Figure 9 Squaring a matrix.

consumes its argument in any way. We use that to derive the types in Figure 8. Since most of these low-level routines are very careful not to do any allocation themselves, it is generally very easy to give each a NumLin type – every argument that can modify/consume its argument needs a full permission, and all others can be fraction-polymorphic. Taking Fortran as an example, it has a notion of in, out and inout parameters. The latter two would need full z permissions; the first would be fraction-polymorphic.

2.3.5 Squaring a Matrix

Figure 9 shows how a linearly-typed matrix squaring function may be written in NumLin. It is a non-recursive function declaration (the return type is inferred). Since we would like to be able to use a function like square more than once, it is marked with a ! annotation (which also ensures it captures no linear values from the surrounding environment).

To square a matrix, first, we extract the dimensions of the argument x. Then, because we need to use x twice (so that we can multiply it by itself) but linearity only allows one use, we use shareM : 'x 'x mat → 'x s mat ⊗ 'x s mat to split the permission 'x (which represents 2^{-x}) into two halves ('x s, which represents 2^{-(x+1)}).

Even if x had type z mat, sharing it now enforces the assumption of all BLAS/LAPACK routines that any matrix which is written to (which, in NumLin, is always of type z mat) does not alias any other matrix in scope. So if we did try to use one of the aliases in mutating way, the expression would not type check, and we would get an error similar to the one in Figure 6.

By using some simple pattern-matching and syntactic sugar (Figure 3), we can:

- write normal-looking, apparently non-linear code
- use matrix expressions directly and have a call to an efficient call to a BLAS/LAPACK routine inserted with appropriate re-bindings
let !lin_reg ('x) (x : 'x mat) =
    ('y) (y : 'y mat) =
let (x, (!n, !m)) = sizeM _ x in
let xy <- new (m, 1) [\( x^T \cdot y \)] in
let x_T_x <- new (m, m) [\( x^T \cdot x \)] in
let (to_del, answer) = posv x_T_x xy in
let () = freeM to_del in
((x, y), answer)
lin_reg ;;

Figure 10 Linear regression (OLS): \( \hat{\beta} = (X^TX)^{-1}X^Ty \).

retain the safety of linear types with fractional permissions by having the compiler
statically enforce the aliasing and read/write rules implicitly assumed by BLAS/LAPACK
routines.

2.3.6 Linear Regression

In Figure 10, we wish to compute \( \hat{\beta} = (X^TX)^{-1}X^Ty \). To do that, first, we extract the
dimensions of matrix \( x \). Then, we say we would like \( xy \) to be a new matrix, of dimension
\( m \times 1 \), which contains the result of \( X^Ty \) (using syntactic sugar for matrix and gemm calls
similar to that used in Figure 9, with a \( ^T \) annotation on \( x \) to set \( x \)'s “transpose indices”-flag
to true).

Note that \( x \) can appear twice in the pattern without any calls to share because the
pattern is matched to a BLAS call to syrk true 1. x 0. x_T_x, which uses \( x \) once only.

After computing \( x_T_x \), we need to invert it and then multiply it by \( xy \). The BLAS
routine posv: z mat \( \rightarrow z \) mat \( \rightarrow z \) mat \( \odot z \) mat does exactly that: assuming the first
argument is symmetric, posv mutates its second argument to contain the desired value. Its
first argument is also mutated to contain the (upper triangular) Cholesky decomposition
factor of the original matrix. Since we do not need that matrix (or its memory) again, we
free it. If we forgot to, we would get a Variable to_del not used error. Lastly, we return
the answer alongside the untouched input matrices \((x, y)\).

2.3.7 L1-norm Minimisation on Manifolds

L1-norm minimisation is often used in optimisation problems, as a regularisation term for
reducing the influence of outliers. Although the below formulation [12] is intended to be
used with sparse computations, NumLin’s current implementation only implements dense
ones. However, it still serves as a useful example of explaining NumLin’s features.

Primitives like transpose: 'x. 'x mat \( \rightarrow 'x \) mat \( \odot 'x \) mat and eye: !int \( \rightarrow 'z \) mat
allocate new matrices; transpose returns the transpose of a given matrix and eye \( k \) evaluates
to a \( k \times k \) identity matrix.

We also see our first example of re-using memory for different matrices: like with to_del
and posv in the previous example, we do not need the value stored in tmp_n_n after the call
to gesv (a primitive similar to posv but for a non-symmetric first argument). However, we
can re-use its memory much later to store answer with let answer <- [\( 0. \cdot tmp_n_n + q_inv_u \cdot inv_u_T \)]]. Again, thanks to linearity, the identifiers q and tmp_n_n are out
of scope by the time answer is bound. Although during execution, all three refer to the same
piece of memory, logically they represent different values throughout the computation.
let l1_norm_min (q : z mat) (u : z mat) = 
  let (u, (!n, !k)) = sizeM _ u in
  let (u, u_T) = transpose _ u in
  let (tmp_n_n , q_inv_u ) = gesv q u in
  let i = eye k in
  let to_inv <- [| i + u_T * q_inv_u |] in
  let (tmp_k_k, inv_u_T ) = gesv to_inv u_T in
  let () = freeM tmp_k_k in
  let answer <- [| 0. * tmp_n_n + q_inv_u * inv_u_T |] in
  let () = freeM q_inv_u in
  let () = freeM inv_u_T in
  answer in
l1_norm_min ;;

Figure 11 L1-norm minimisation on manifolds: \( Q^{-1}U(I + U^TQ^{-1}U)^{-1}U^T \).

2.3.8 Kalman Filter

A Kalman Filter [16] is an algorithm for combining prior knowledge of a state, a statistical
model and measurements from (noisy) sensors to produce an estimate a more reliable
estimated of the current state. It has various applications (navigation, signal-processing,
econometrics) and is relevant here because it is usually presented as a series of complex
matrix equations.

Figure 12 shows a NumLin implementation of a Kalman filter (equations in Figure 13). A
few new features and techniques are used in this implementation:

- sym annotations in matrix expressions: when this is used, a call to symm (the equivalent
  of gemm but for symmetric matrices so that only half the operations are performed) is
  inserted
- copyM_to is used to re-use memory by overwriting the contents of its second argument
  to that of its first (erroring if dimensions do not match)
- posvFlip is like posv except for solving \( XA = B \)
- a lot of memory re-use; the following sets of identifiers alias each other:
  - \( r_1, r_2 \) and \( k\_by\_k \)
  - \( data_1 \) and \( data_2 \)
  - \( mu \) and \( new\_mu \)
  - \( sigma\_hT \) and \( x \).

The NumLin implementation is much longer than the mathematical equations for two
reasons. First, the NumLin implementation is a let-normalised form of the Kalman equations:
since there a large number of unary/binary (and occasionally ternary) sub-expressions in
the equations, naming each one line at a time makes the implementation much longer.
Second, NumLin has the additional task of handling explicit allocations, aliasing and frees
of matrices. However, it is exactly this which makes it possible (and often, easy) to spot
additional opportunities for memory re-use. Furthermore, a programmer can explore those
opportunities easily because NumLin’s type system statically enforces correct memory
management and the aliasing assumptions of BLAS/LAPACK routines.

3 Formal System

3.1 Core Type Theory

The full typing rules are in the Appendix, but the key ideas are as follow.

- A typing judgement consists of \( \Theta; \Delta; \Gamma \vdash e : t \).
let 'kalman
  ('s) (sigma : 's mat) (* n,n *)
  ('h) (h : 'h mat) (* k,n *)
  (mu : z mat) (* n,1 *)
  (r_1 : z mat) (* k,k *)
  (data_1 : z mat) (* k,1 *) =
  let (h, (k, n)) = sizeM _ h in
  (* could use ![sym(sigma) * h^T] but would need a (n,k) temporary hT = transpose _ h *)
  let sigma_hT <- new (n, k) ![sigma * h^T] in
  let r_2 <- ![r_1 + h * sigma_hT] in
  let new_mu <- ![mu + x * data_2] in
  let sigma2 <- new ![sigma] in
  let new_sigma <- ![sigma2 - x_h * sym(sigma)] in
  kalman

Figure 12 Kalman filter: see Figure 13 for the equations this code implements and the Appendix for an equivalent Cblas/Lapacke implementation.

\[
\begin{align*}
\mu' &= \mu + \Sigma H^T (R + H\Sigma H^T)^{-1} (H\mu - \text{data}) \\
\Sigma' &= \Sigma (I - H^T (R + H\Sigma H^T)^{-1} H) 
\end{align*}
\]

Figure 13 Kalman filter equations (credit: matthewrocklin.com).

- \(\Theta\) is the environment that tracks which fractional permission variables in scope. Fractional permissions (the \texttt{Perm} judgement) and types (the \texttt{Type} judgement) are well-formed if all of their free fractional variables are in \(\Theta\).
- \(\Delta\) is the environment storing non-linearly or intuitionistically typed variables.
- \(\Gamma\) is the environment storing linearly typed variables.

Note that rules for typing (), booleans, integers and elements are typed with respect to an empty linear environment: this means no linear values are needed to produce a value of those types.

\[ \Theta; \Delta; \cdot \vdash () : \texttt{unit} \quad \text{TY\_UNIT\_INTRO} \]

Conversely, whenever two or more subexpressions need to be typed, they must consume a disjoint set of linear values (pairs, let-expressions). In the case of if-expressions, both branches must consume the same set of linear values (disjoint to the ones used to evaluate the condition).

\[
\begin{align*}
\Theta; \Delta; \Gamma \vdash e : \!\texttt{bool} \\
\Theta; \Delta; \Gamma' \vdash e_1 : t' \\
\Theta; \Delta; \Gamma' \vdash e_2 : t \\
\Theta; \Delta; \Gamma' \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : t 
\end{align*}
\]

\[ \text{TY\_BOOL\_ELIM} \]

The Many introduction and elimination rules are very important. Producing !-type values may only be done if the expression inside is a syntactic value which is not a location. This
allows all safely duplicable resources, including functions which capture non-linear resources
from their environments, but prevents producing aliases of (pointers to) arrays and matrices.
This is exactly the same as value-restriction from the world of parametric polymorphism;
without it, the expression \(\text{let Many } x = \text{Many (array 5)} \in \text{let } () = \text{free } x \in x[0]\) would
\typecheck but error at runtime.

Consuming a variable that refers to a \(!\)-type value \(\text{moves it from the linear environment}\ \Gamma \ \text{and into the intuitionistic environment} \ \Delta\).

Using this, we can explain how the \(!\) annotation on variables – first introduced in the
factorial example in 2.3.1 – works. That is, we can explain why the meaning of
\(\text{let Many } x = e \in \text{let Many } x = \text{Many (Many } x \text{)} \in e'\)\(^1\) The reader is invited to quickly convince themselves that the
following meta-rule is provable using \(\text{Ty_Bang_Intro} \ \text{(twice)}, \ \text{Ty_Bang_Elim} \ \text{(twice)}\) and weakening the intuitionistic environment \(\Delta\) (once).

\[
\begin{align*}
\Theta; \Delta; \Gamma & \vdash e : !t \\
\Theta; \Delta; \Gamma & \vdash x : !t; \Gamma' \vdash e' : t' \\
\Theta; \Delta; \Gamma, \Gamma' & \vdash \text{let Many } x = e \in e' : t' \\
\end{align*}
\]

\(\text{Meta_Ty_Let_Bang}\)

Rules \(\text{Ty_GEN}\) and \(\text{Ty_SPC}\) are for fractional permission generalisation and specialisation
respectively. They allow the definition and use of functions that are polymorphic in the
fractional permission index of their results and one or more of their arguments.

\[
\begin{align*}
\Theta; \Delta; \Gamma & \vdash e : t \\
\Theta; \Delta; \Gamma & \vdash \text{fun } f : t \rightarrow e : f : t
\end{align*}
\]

\(\text{Ty_GEN}\)

\[
\begin{align*}
\Theta & \vdash f \text{ Perm} \\
\Theta; \Delta; \Gamma & \vdash e : f : t \\
\Theta; \Delta; \Gamma & \vdash e[f/x] : t[f/x]
\end{align*}
\]

\(\text{Ty_SPC}\)

Rule \(\text{Ty_FIX}\) shows how recursive functions are typed. Even though recursive functions
are fully annotated, type checking them is interesting for two reasons: to type check the
body of the fixpoint, the type of the recursive function is in the \textit{intuitionistic} environment \(\Delta\)
(without this, you would not be able to write a base case) whilst the argument and its type
are the \textit{only things in the linear environment} \(\Gamma\). The latter means that recursive functions
can be type checked in an empty environment (thus be wrapped in \texttt{Many} and used zero or
multiple times).

\[
\begin{align*}
\Theta; \Delta; \Gamma & \vdash \text{fix } (g, x : t, e : t') : t \rightarrow t' \\
\Theta; \Delta; & \vdash g : t \rightarrow t' ; x : t \vdash e : t'
\end{align*}
\]

\(\text{Ty_FIX}\)

Lastly, types of almost all \texttt{NumLin} primitives, as embedded in OCaml’s type system, are
shown in the Appendix, with some similar ones (like those for binary arithmetic operators)

\footnote{Why we have this at all is for the sake of ergonomics when using binary arithmetic operations (e.g.
of type \texttt{int -> int -> int}): a programmer should be able to write \texttt{let x = 5 + 5 in x - x}, which,
although non-linear in \(x\), is morally right because integers and operations on them rarely need to be
linear. Though it should be possible to handle this using a LNL-style presentation of linear types \[5\]
(using adjoint modalities to distinguish between intrinsically linear and intrinsically intuitionistic types)
that is a pretty big digression from the stated goals of this paper.}
omitted for brevity. The main difference between the OCaml type of a primitive like `gemm` and its NumLin counterpart (Figure 8) is the inclusion of explicit universal-quantification of fractional permission variables in the latter.

### 3.2 Dynamic Semantics

The full, small-step transition relation is in the Appendix, but the key ideas are as follow.

Heaps $\sigma$ are multisets containing triples of an abstract location $l$, a fractional permission $f$ and sized matrices $m_{n,k}$. The notation $l \mapsto f m_{k_1,k_2}$ should be read as “location $l$ represents $f$ ownership over matrix $m$ (of size $k_1 \times k_2$)”. Each heap-and-expression either steps to another heap-and-expression or a runtime error $\text{err}$. In the full grammar definition we see a definition of values and contexts in the language.

We draw the reader’s attention to the definitions relating to fractional permissions. Specifically, unlike a lambda, the body of a `fun `fc → v` must be a syntactic value. The context `fun `fc → [−]` means expressions can be reduced inside a fractional permission generalisation. This is to emphasize that fractions are merely compile-time constructs and do not affect runtime behaviour. Correct usage of fractions is enforced by the type system, so programs do not get stuck. Fractional permissions are specialised using substitution over both the heap and an expression (Op_Frac_Per).

\[
\frac{\langle \sigma, \text{(fun f}c \rightarrow v)\rangle[f]}{\langle \sigma[f/\text{fc}], v[f/\text{fc]}]\rangle} \quad \text{Op_Frac_Per}
\]

Like with the static semantics, the interesting rules in the dynamic semantics are those relating to primitives. Creating a matrix (matrix $k_1 k_2$) successfully (Op_Matrix) requires non-negative dimensions and returns a (fresh) location of a matrix of those dimensions, extending the heap to reflect that $l$ represents a complete ownership over the new matrix.

\[
\frac{0 \leq k_1, k_2 \quad l \text{ fresh}}{\langle \sigma, \text{matrix }k_1 k_2\rangle \rightarrow \langle \sigma + \{ l \mapsto \text{M}_{k_1, k_2}\}, l \rangle} \quad \text{Op_Matrix}
\]

Dually, Op_Free, requires a location represent complete ownership before removing it and the matrix it points to from the heap.

\[
\frac{\langle \sigma + \{ l \mapsto \text{M}_{k_1, k_2}\}, \text{free }l \rangle \rightarrow \langle \sigma, () \rangle}{\langle \sigma, \text{free }l \rangle} \quad \text{Op_Free}
\]

Choosing a multiset representation as opposed to a set allows for two convenient invariants: multiplicity of a triple $l \mapsto f m_{k_1, k_2}$ in the heap corresponds to the number of aliases of $l$ in the expression with type $f$ mat and the sum of all the fractions for $l$ will always be 1 (for a closed, well-typed expression). With this in mind, the rules Op_Share and Op_Unshare_EQ are fairly natural.

\[
\frac{\langle \sigma + \{ l \mapsto f \text{M}_{k_1, k_2}\}, \text{share}[f/l] \rangle \rightarrow \langle \sigma + \{ l \mapsto \frac{1}{2f} \text{M}_{k_1, k_2}\} + \{ l \mapsto \frac{1}{2f} \text{M}_{k_1, k_2}\}.(l,l)\rangle}{\langle \sigma, \text{share}[f/l] \rangle} \quad \text{Op_Share}
\]

\[
\frac{\sigma' \equiv \sigma + \{ l \mapsto \frac{1}{2f} \text{M}_{k_1, k_2}\} + \{ l \mapsto \frac{1}{2f} \text{M}_{k_1, k_2}\}}{\langle \sigma', \text{unshare}[f/l] \rangle \rightarrow \langle \sigma + \{ l \mapsto f \text{M}_{k_1, k_2}\}, l \rangle} \quad \text{Op_Unshare_EQ}
\]

Combining all of these features, we see that Op_Gemm_Match requires that the location being updated ($l_3$) has complete ownership of over matrix $m_3$ and can thus change what value it stores to $m_1 m_2 + m_3$. In particular, this places no restriction on $l_2$ and $l_3$: they
could be shared aliases of the same matrix. Transition rules for other primitives (omitted) follow the same structure: \( \rightarrow_1 \) for any locations that are written to and \( \rightarrow_f \) for anything else.

\[
\begin{align*}
\sigma' &\equiv \sigma + \{ l_1 \mapsto f_{c_1} m_{1k_1, l_2} \} + \{ l_2 \mapsto f_{c_2} m_{2k_2, l_3} \} \\
\sigma_1 &\equiv \sigma' + \{ l_1 \mapsto m_{3k_1, l_5} \} \\
\sigma_2 &\equiv \sigma' + \{ l_1 \mapsto (m_1 m_2 + m_3)_{l_1, l_6} \}
\end{align*}
\]

\[\langle \sigma_1, \text{gemm}[f_{c_1} \ l_1 \ f_{c_2} \ l_2 \ l_3] \rangle \rightarrow \langle \sigma_2, ((l_1, l_2), l_3) \rangle. \quad \text{Op\_Gemm\_Match}
\]

### 3.3 Logical Relation

First, we define an interpretation of heaps with fractional permissions in the style of Bornat et. al [10] (interpreting the multiset as a partial map from locations to the sum of all its associated fractions and a matrix) as well as the n-fold iteration of \( \rightarrow \).

\[\mathcal{H}[\sigma] = \star (l, f, m) \in \sigma[l \mapsto f \ m]\]

where

\[(\varsigma_1 \ast \varsigma_2)(l) \equiv
\begin{cases}
\varsigma_1(l) & \text{if } l \in \text{dom}(\varsigma_1) \land l \notin \text{dom}(\varsigma_2) \\
\varsigma_2(l) & \text{if } l \in \text{dom}(\varsigma_2) \land l \notin \text{dom}(\varsigma_1) \\
(f_1 + f_2, m) & \text{if } (f_1, m) = \varsigma_1(l) \land (f_2, m) = \varsigma_2(l) \land f_1 + f_2 \leq 1 \\
\text{undefined} & \text{otherwise}
\end{cases}\]

We then define a step-indexed logical relation in the style of Ahmed et. al [2]. \((\varsigma, v) \in \mathcal{V}_k[t]\) means it takes a heap with exactly \(\varsigma\) resources to produce a value \(v\) of type \(t\) in at most \(k\) steps. So, something like a unit or \(a!\) need no resources, whereas a \(f\) mat needs exactly \(f\) ownership of a some matrix and a pair needs a \(\ast\) combination of the heaps required for each component.

\[
\begin{align*}
\mathcal{V}_k[\text{unit}] &= \{ (\emptyset, \ast) \} \\
\mathcal{V}_k[\text{mat}] &= \{ \{(l \mapsto \_\_ \_), l\} \} \\
\mathcal{V}_k[t] &= \{ (\emptyset, \text{Many } v) \mid (\emptyset, v) \in \mathcal{V}_k[t] \} \\
\mathcal{V}_k[t_1 \otimes t_2] &= \{ (\varsigma_1 \ast \varsigma_2, (v_1, v_2)) \mid (\varsigma_1, v_1) \in \mathcal{V}_k[t_1] \land (\varsigma_2, v_2) \in \mathcal{V}_k[t_2] \} \\
\end{align*}
\]

The definition of \(\mathcal{V}_k[\text{fc. } t]\) says a value and heap must be the same regardless of what fraction is substituted into both; the \(k - 1\) is to take into account fraction specialisation takes one step (Op\_SpC).

\[
\mathcal{V}_k[\text{fc. } t] = \{ (\varsigma, \text{fun } f/c \rightarrow v) \mid \forall f. (\varsigma[f/fc], v[f/fc]) \in \mathcal{V}_{k-1}[t[f/fc]] \}
\]

To understand the definition of \(\mathcal{V}_k[t' \rightarrow t]\), we must first look at \(\mathcal{C}_k[t]\), the computational interpretation of types. Intuitively, it is a combination of a frame rule on heaps (no interference), type-preservation and termination (in \(j < k\) steps) to either an error or a heap-and-expression. For the case of termination to a heap-and-expression, there is a further condition: if the expression is a value syntactically then it is also one semantically.

\[
\mathcal{C}_k[t] = \{ (\varsigma_s, e_s) \mid \forall j < k, \sigma_r, \varsigma_s \ast \varsigma_r \text{ defined } \Rightarrow \sigma_s + \sigma_r, e_s \rightarrow^j \text{err } \lor \exists f, e_f, \langle \sigma_s + \sigma_r, e_s \rangle \rightarrow^j \langle \sigma_f + \sigma_r, e_f \rangle \land (e_f \text{ is a value } \Rightarrow (\varsigma_f \ast \varsigma_r, e_f) \in \mathcal{V}_{k-j}[t]) \}
\]
In this light, $V_k [ t' \rightarrow t ]$ simply says that $v$ is a function and that evaluating the application of it to any argument (of the correct type, requiring its own set of resources, bounded by $k$ steps) satisfies all the aforementioned properties.

$$V_k [ t' \rightarrow t ] = \{ (\varsigma, v) \mid (v \equiv \text{fun } x : t' \rightarrow e \lor v \equiv \text{fix}(g, x : t', e : t)) \land \forall j \leq k, (\varsigma_{v'}, v') \in V_j [ t' ] \cdot \varsigma_v \star \varsigma_{v'} \text{ defined } \Rightarrow (\varsigma_v \star \varsigma_{v'}, v v') \in C_j [ t ] \}$$

The interpretation of typing environments $\Delta$ and $\Gamma$ are with respect to an arbitrary substitution of fractional permissions $\theta$. Note that only the interpretation of $\Gamma$ involves a (potentially) non-empty heap.

$$I_k [ \Delta, x : t ] \theta = \{ \delta[x \mapsto v_x] \mid \delta \in I_k [ \Delta ] \theta \land (\theta, v_x) \in V_k [ \theta(t) ] \}$$

$$L_k [ \Gamma, x : t ] \theta = \{ (\varsigma \star \varsigma_x, \gamma[x \mapsto v_x]) \mid (\varsigma, \gamma) \in L_k [ \Gamma ] \theta \land (\varsigma_x, v_x) \in V_k [ \theta(t) ] \}$$

And so the final semantic interpretation of a typing judgement simply quantifies over all possible fractional permission substitutions $\theta$, linear value substitutions $\gamma$, intuitionistic value substitutions $\delta$ and heaps $\varsigma$. Note that, $\varsigma \equiv H[ \theta[\varsigma] ]$.

$$k [ \Theta; \Delta; \Gamma \vdash e : t ] = \forall \theta, \delta, \gamma, \varsigma. \Theta = \text{dom}(\theta) \land (\varsigma, \gamma) \in L_k [ \Gamma ] \theta \land \delta \in I_k [ \Delta ] \theta \Rightarrow (\varsigma, \theta(\delta(\gamma(e)))) \in C_k [ \theta(t) ]$$

3.4 Soundness Theorem

**Theorem 1.** *(The Fundamental Lemma of Logical Relations)*

$$\forall \Theta, \Delta, \Gamma, e, t. \Theta; \Delta; \Gamma \vdash e : t \Rightarrow \forall k. k [ \Theta; \Delta; \Gamma \vdash e : t ]$$

3.4.1 Explanation

If an expression $e$ is syntactically type-checked (against a type $t$), then

- for an arbitrary number of steps $k$,
- under any substitution of
  - free fractional permission variables $\theta$,
  - linear variables with a suitable heap $\gamma$ and $\varsigma$
  - intuitionistic variables $\delta$,
- the aforementioned suitable heap and expression $(\varsigma, \theta(\delta(\gamma(e))))$
- are in the computational interpretation $C_k [ \theta(t) ]$ of the type $t$.

The *computational interpretation* is as defined before (Section 3.3); it identifies executions that do no un- or ill-defined behaviours (e.g. adding a boolean and an integer). Since our operational semantics explicitly models deallocation, we now know no well-typed program will ever try to access deallocated memory, establishing the correctness of our memory management checking.

3.4.2 Proof Sketch

To prove the above theorem, we need several lemmas; the interesting ones are: the moral equivalent of the frame rule, monotonicity for the step-index, splitting up environments corresponds to splitting up heaps and heap-and-expressions take the same steps of evaluation under any substitution of their free fractional permissions; all of these (and more) are stated formally and proved in the Appendix.
The proof proceeds by induction on the typing judgement. The case for \( \text{Ty}_\text{Fix} \) is the reason we quantify over the step-index \( k \) in the conclusion of the soundness theorem. It allows us to then induct over the step-index and assume exactly the thing we need to prove at a smaller index.

The case for \( \text{Ty}_\text{Gen} \) follows a similar pattern, but has the extra complication of reducing an expression with an arbitrary fractional permission variable in it, and then instantiating it at the last moment to conclude, which is where one of the lemmas (heap-and-expressions take the same steps of evaluation under any substitution of their free fractional permissions) is used.

The rest of the cases are either very simple base cases (variables, unit, boolean, integer or element literals) or follow very similar patterns; for these, only \( \text{Ty}_\text{Let} \) is presented in full and other similar cases simply highlight exactly what would be different. The general idea is to split up the linear substitution and heap along the same split of \( \Gamma / \Gamma' \), then (by induction) use \( C_k[[-]] \) and one “half” of the linear substitution and heap to conclude the “first” sub-expression either takes \( j < k \) steps to \text{err} or another heap-and-expression.

In the first case, you use \( \text{Op}_\text{Context}\_\text{Err} \) to conclude the whole let-expression does the same. Similarly we use \( \text{Op}_\text{Context} \) \( j \) times in the second case. However, a small book-keeping wrinkle needs to be taken care of in the case that the heap-and-expression turns into a value in \( i \leq j \) steps: \( \text{Op}_\text{Context} \) is not functorial for the n-fold iteration of \( \rightarrow \). Basically, the following is not quite true:

\[
(\sigma, e) \rightarrow^j (\sigma', e') \quad \text{Op}_\text{Context}
\]

because after the \( i \) steps, we need to invoke \( \text{Op}_\text{Let}_\text{Var} \) to proceed evaluation for any remaining \( j - i \) steps. After that, it suffices to use the induction hypothesis on the second sub-expression to finish the proof. To do so, we need to construct a valid linear substitution and heap (one in \( L_k[\Gamma', x : t | \theta] \)). We take the other “half” of the linear substitution and heap (from the initial split at the start) and extend it with \( [x \mapsto v] \), (where \( x \) is the variable bound in the let-expression and \( v \) is the value we assume the first sub-expression evaluated to in \( i \) steps).

### 4 Implementation

#### 4.1 Implementation Strategy

\textsc{NumLin} transpiles to OCaml and its implementation follows the structure of a typical domain-specific language (DSL) compiler. Although \textsc{NumLin}'s current implementation is not as an embedded DSL, its general design is simple enough to adapt to being so and also to target other languages.

Alongside the transpiler, a “Read-Check-Translate” loop, benchmarking program and a test suite are included in the artifacts accompanying this paper.

1. ** Parsing. ** A generated, LR(1) parser parses a text file into a syntax tree. In general, this part will vary for different languages and can also be dealt with using combinators or syntax-extensions (the EDSL approach) if the host language offers such support.

2. ** Desugaring. ** The syntax tree is then desugared into a smaller, more concise, abstract syntax tree. This allows for the type checker to be simpler to specify and easier to implement.
3. **Matrix Expressions** are also desugared into the abstract syntax tree through pattern-matching.

4. **Type checking.** The abstract syntax tree is explicitly typed, with some inference to make writing typical programs more convenient.

5. **Code Generation.** The abstract syntax tree is translated into OCaml, with a few “optimisations” to produce more readable code. This process is type-preserving: NumLin’s type system is embedded into OCaml’s (Figure 14), so the OCaml type checker acts as a sanity check on the generated code.

A very pleasant way to use NumLin is to have the build system generate code at compile-time and then have the generated code be used by other modules like normal OCaml functions. This makes it possible and even easy to use NumLin alongside existing OCaml libraries; in fact, this is exactly how the benchmarking program and test-suite use code written in NumLin.

### 4.1.1 Desugaring, Matrix Expressions and Type Checking

As seen earlier (Figure 2), desugaring is conventional. Matrix expressions are translated into BLAS/LAPACK calls via purely syntactic pattern-matching (also seen earlier in Figure 3).

### 4.1.2 Type checking

Type checking is mostly standard for a linearly typed language, with the exception of fractional permission inference. By restricting fractions to be non-positive integer powers of two, we only need to keep track of the logarithm of the fractions used. Explicit sharing and unsharing removes the need for performing dataflow analysis. As a result, all fractional arithmetic can be solved with unification, and in doing so, fractions become directly usable in NumLin’s type-system as opposed to a convenient theoretical tool.

Because all functions must have their argument types explicitly annotated, inferring the correct fraction at a call-site is simply a matter of unification. We believe full-inference of fractional permissions is similarly just matter of unification (thanks to an experimental implementation of just this feature), even though the formal system we present here is for an explicitly-typed language.

There are a few differences between the type system as presented in 3.2 and how we implemented it: the environment changes as a result of type checking an expression (the standard transformation to avoid a non-deterministic split of the environment for checking pairs); variables are marked as used rather than removed for better error messages; variables are tagged as linear or intuitionistic in one environment as opposed to being stored in two separate ones (this allows scoping/variable look-up to be handled uniformly).

### 4.1.3 Code Generation

Code generation is a straightforward mapping from NumLin’s core constructs to high-level OCaml ones. We embed NumLin’s type- and term- constructors into OCaml as a sanity check on the output (Figure 14).

This is also useful when using NumLin from within OCaml; for example, we can use existing tools to inspect the type of the function we are using (Figure 15). It is worth reiterating that only the type- and term- constructors are translated into OCaml, NumLin’s precise control over linearity and aliasing are not brought over.

We actually use this fact to our advantage to clean up the output OCaml by removing what would otherwise be redundant re-bindings (Figure 16). Combined with a code-formatter,
the resulting code is not obviously correct and exactly what an expert would intend to write by hand, but now with the guarantees and safety of NumLIN behind it. A small example is shown in Figure 17, a larger one in the Appendix.

4.2 Performance Metrics

We think that using NumLIN has two primary benefits: safety and performance. We discuss safety in 5.1, where we describe how we used NumLIN to find linearity and aliasing bugs in a linear algebra algorithm that was generated by another program.

4.2.1 Setup

For performance, we measured the execution times of four equivalent implementations of a Kalman filter: in C (using Cblas), NumLIN (using Owl’s low-level Cblas bindings), OCaml (using Owl’s intended, safe/copying-by-default interface), and Python (using NumPy, with the interpreter started and functions interpreted). We measured execution time in micro-
let Many x = x in
let Many x = Many (Many x) in $\Rightarrow$ <exp>

(* fixp = fix (f, x:t, <exp> : t') *)
(*1#*) let Many f = Many fixp in <body> $\Rightarrow$ let rec f x = <exp> in <body>
(*2#*) let f = fixp in <body>
(*1#*) let Many x = <exp> in
(*-#*) let Many x = Many (Many x) in <body> $\Rightarrow$ let x = <exp> in <body>
(*2#*) let Many x = Many <exp> in <body>
(*3#*) (fun x : t -> <body>) <exp>

Figure 16 Removing redundant re-bindings during translation to OCaml.

let rec sum_array i n x0 row =
  if Prim.extract @@ Prim.eqI i n then (row, x0)
  else
    let row, x1 = Prim.get row i in
    sum_array (Prim.addI i (Many 1)) n (Prim.addE x0 x1) row in
  sum_array

Figure 17 Recursive OCaml function for a summing over an array, generated (at compile time) from the code in Figure 5, passed through ocamlformat for presentation.

seconds, against an exponentially (powers of 5) increasing scaling factor for matrix size parameters $n = 5$ and $k = 3$.

For large scaling factors ($n = 5^4, 5^5$), we triggered a full garbage-collection before measuring the execution time of a single call of a function. However, due to the limitations of the micro-benchmarking library we used, for smaller scaling factors ($n = 5^1, 5^2, 5^3$), we measured the execution time of multiple calls to a function in a loop, thus including potential garbage-collection effects.

We also measured the execution times of L1-norm minimisation and the “linear-regression” $((X^TX)^{-1}X^T y)$ similarly, but without a C implementation.

4.2.2 Hypothesis

We expected the C implementation to be faster than the NUMLIN one because the latter has the additional (but relatively low) overhead of dimension checks and crossing the OCaml/C FFI for each call to a CBLAS routine, even though the calls and their order are exactly the same. We expected the OCaml and Python implementations to be slower because they allocate more temporaries (so possibly less cache-friendly) and carry out more floating-point operations – the CBLAS and NUMLIN implementations use ternary kernels (coalescing steps), a Cholesky decomposition (of a symmetric matrix, which is more efficient than the LU decomposition used for inverting a matrix in OWL and NUMPy) and symm (symmetric matrix multiplication, halving the number of floating-point multiplications required).

4.2.3 Results

The results in Figures 18 are as we expected: C is the fastest, followed by NUMLIN, with OCaml and Python last. Differences in timings are quite pronounced at small matrix sizes, but are still significant at larger ones. Specifically for the Kalman filter, for $n = 625$, CBLAS
took 112 ± 35 ms, NumLin took 105 ± 25 ms, Owl took 124 ± 38 ms and NumPy took 112 ± 12 ms; for $n = 3125$, CBLAS took 10.8 ± 0.7 s, NumLin took 12.0 ± 1.2 s, Owl took 13.3 ± 0.2 s and NumPy took 12.7 ± 0.6 s.

Worth highlighting here is the other major advantage of using NumLin is reduced memory usage. Whilst the Owl and NumPy use 11 temporary matrices for the Kalman filter, (excluding the 2 matrices which store the results), using $n + n^2 + 4nk + 3k^2 + 2k \approx 4n^2$ (for $k = 3n/5$) words of memory, CBLAS and NumLin use only 2 temporary matrices (excluding the one matrix which stores one of the results), using only $n^2 + nk \leq 2n^2$ words of memory.

4.2.4 Analysis

As matrix sizes increase, assuming sufficient memory, the difference in the number of floating-point operations ($O(n^3)$) dominates execution times. However for small matrix sizes, since $n$ is small and the measurements were over multiple calls to a function in a loop, the large number of temporaries show the adverse effect of not re-using memory at even quite small matrix sizes: creating pressure on the garbage collector.

5 Discussion and Related Work

5.1 Finding Bugs in SymPy’s Output

Prior to this project, we had little experience with linear algebra libraries or the problem of matrix expression compilation. As such, we based our initial NumLin implementation of a Kalman filter using BLAS and LAPACK, on a popular GitHub gist of a Fortran implementation, one that was automatically generated from SymPy’s matrix expression compiler [18].

Once we translated the implementation from Fortran to NumLin, we attempted to compile it and found that (to our surprise) it did not type-check. This was because the original implementation contained incorrect aliasing, unused variables and unnecessary temporaries, and did not adhere to Fortran’s read/write permissions (with respect to intent annotations in, out and inout) all of which were now highlighted by NumLin’s type system.

The original implementation used 6 temporaries, one of which was immediately spotted as never being used due to linearity. It also contained two variables which were marked as intent(in) but would have been written over by calls to “gemm”, spotted by the fractional permissions feature. Furthermore, it used a matrix twice in a call to “symm”, once with a read permission but once with a write permission. Fortran assumes that any parameter being written to is not aliased and so this call was not only incorrect, but illegal according to the standard, both aspects of which were captured by linearity and fractional permissions.

Lastly, it contained another unnecessary temporary, however one that was not obvious without linear types. To spot it, we first performed live-range splitting (checked by linearity) by hoisting calls to freeM and then annotated the freed matrices with their dimensions. After doing so and spotting two disjoint live-ranges of the same size, we replaced a call to freeM followed by allocating call to copy with one, in-place call to copyM_to. We believe the ability to boldly refactor code which manages memory is good evidence of the usefulness of linearity as a tool for programming.
Small matrices and timings \( n \leq 5^3 \) were micro-benchmarked with the Core_bench library. Larger ones used Unix’s `getrusage` functionality, sandwiched between calls to \texttt{Gc.full_major} for the OCaml implementations.
5.2 Related Work

5.2.1 Linear types for implementing linear algebra routines

Using linear types for BLAS routines is a particularly good domain fit (given the implicit restrictions on aliasing arguments), and as a result the idea of using substructural types to express array computations is not a particularly new one [19, 14, 7]. However, many of these designs have been focused on building languages to implement the kernel linear algebra functions, and as a result, they tend to add additional limitations on the language design. Both Futhark [14] and Single Assignment C [19] omit higher-order functions to facilitate compilation to GPUs. The work of [7] forbids term-level recursion, in order to ensure that all higher-order computations can be statically normalized away and thereby maximize opportunities for array fusion.

5.2.2 Our contribution: linear types for enforcing correct usage of linear algebra routines

In contrast, our approach is to begin with the assumption that we can take existing efficient BLAS-like libraries, and then enforce their correct usage using a linear type discipline with fractional permissions.

5.2.3 Traditionally complex approaches to sharing

Our approach is similar to the one taken in linear algebra libraries for Rust – these libraries typically take advantage of the distinction that Rust’s type system offers between mutable views/references to arrays. The work of [21] and [15] suggest that Rust’s borrow-checker can be expressed in simpler terms using fractional permissions, though to our knowledge the programmer-visible lifetime analysis in Rust has never been formalized.

Working explicitly with fractional permissions has two main benefits. First, our type system demonstrates that type systems for fractional permissions can be dramatically simpler than existing state-of-the-art approaches, including both industrial languages like Rust, as well as academic (such as those developed by [9]). Bierhoff et al’s type system, much like Rust’s, builds a complex dataflow analysis into the typing rules to infer when variables can be shared or not. This allows for more natural-looking user programs, but can create the impression that using fractional permissions requires a heavy theoretical and engineering effort going well beyond that needed for supporting basic linear types.

5.2.4 Our contribution: a simpler approach to sharing

Instead, our approach, of requiring sharing to be made explicit, lets us demonstrate that the existing unification machinery already in place for ordinary ML-style type inference can be reused to support fractions. Basically, we can view sharing a value as dividing a fraction by two, and after taking logarithms all fractions are Peano numbers, whose equality can be established with ordinary unification.

5.2.5 Implications

This fact is important because there are major upcoming implementations of linear types such as Linear Haskell [6], which do not have built-in support for fractional permissions. Instead, Linear Haskell takes a slightly different definition of linearity, one based on arrows as opposed to kinds: for $f : a \to b$, if $fu$ is used exactly once then $u$ is used exactly once. Whilst
this has the advantage of being backwards-compatible, it also means that the type system
has no built-in support for the concurrent reader, exclusive writer pattern that fractional
permissions enable.

However, since our type system shows unification is “all one needs” for fractions, it should
be possible to encode NumLin’s approach to fractional permissions in Linear Haskell by
adding a GADT-style natural number index to array types tracking the fraction, which
should enable supporting high-performance BLAS bindings in Linear Haskell. Actually
implementing this is something we leave for future work, as there remains one issue which
we do not see a good encoding for. Namely, only having support for linear functions makes
it a bit inconvenient to manipulate linear values directly – programs end up taking on a
CPS-like structure. This seems to remain an advantage of a direct implementation of linear
types over the Linear Haskell style.

5.3 Simplicity and Further Work

We are pleasantly surprised at how simple the overall design and implementation of NumLin
is, given its expressive power and usability. So simple in fact, that fractions, a convenient
theoretical abstraction until this point, could be implemented by restricting division and
multiplication to be by 2 only [11], thus turning any required arithmetic into unification.

Indeed, the focus on getting a working prototype early on (so that we could test it with
real BLAS/LAPACK routines as soon as possible) meant that we only added features to
the type system when it was clear that they were absolutely necessary: these features were
!-types and value-restriction for the Many constructor.

Going forwards, one may wish to eliminate even more runtime errors from NumLin, by
extending its type system. For example, we could have used existential types to statically
track pointer identities [2], or parametric polymorphism.

We could also attempt to catch mismatched dimensions at compile time as well. While
this could be done with generative phantom types [1], using dependent types may offer more
flexibility in partitioning regions [17] or statically enforcing dimensions related constraints of
the arguments at compile-time. ATS [13] is an example of a language which combines linear
types with a sophisticated proof layer. But although it provides BLAS bindings, it does not
aim to provide aliasing restrictions as demonstrated in this paper.

Taking this idea one step even further, since matrix dimensions are typically fixed at
runtime, we could stage NumLin programs and compile matrix expressions using more soph-
isticated algorithms [4]. However, it is worth noting that without care, such algorithms [18],
usually based on graph-based, ad-hoc dataflow analysis, can produce erroneous output which
would not get past a linear type system with fractions.

We also think that this concept (and the general design of its implementation) need not
be limited to linear algebra: we could conceivably “backport” this idea to other contexts that
need linearity (concurrency, single-use continuations, zero-copy buffer, streaming I/O) or
combine it with dependent types to achieve even more expressive power to split up a single
block of memory into multiple regions in an arbitrary manner [17].
References