NumLin: Linear Types for Linear Algebra

Dhruv C. Makwana
Unaffiliated `dhruvmakwana.com`
dcm41@cam.ac.uk

Neelakantan R. Krishnaswami
Department of Computer Science and Technology, University of Cambridge, United Kingdom
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

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.9

Related Version A full version of the paper is available at `https://github.com/dc-mak/NumLin/blob/master/write-up/paper.pdf`.

Supplement Material `https://github.com/dc-mak/NumLin`

Acknowledgements We would like to thank Stephen Dolan for his advice and support with the implementation and evaluation of NumLin. We would also like to thank the (anonymous) reviewers for their feedback and suggestions.

1 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. Thefore, 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 $A \multimap B$ 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(`α`, A, B, `β`, 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 $'fc$, `z` (2^0) or f `s` ($2^{-(f+1)}$).

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 $'fc.\ t$ 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.

$$
\begin{array}{lll}
f & ::= & 'fc \mid \mathtt{z} \mid f\ \mathtt{s} \\
\\
t & ::= & \textbf{unit} \mid \textbf{bool} \mid \textbf{int} \mid \textbf{elt} \mid f\ \textbf{arr} \mid f\ \textbf{mat} \mid !t \mid 'fc.\ t \mid t \otimes t' \mid t \multimap t' \\
\\
e & ::= & p\ (\text{primitives}) \mid x\ (\text{variable}) \mid \textbf{let}\ x = e\ \textbf{in}\ e' \mid () \mid \textbf{let}\ () = e\ \textbf{in}\ e' \mid \textbf{true} \mid \textbf{false} \\
 & & \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid k\ (\text{integer}) \mid l\ (\text{heap location}) \mid el\ (\text{array element}) \\
 & & \textbf{Many}\ v \mid \textbf{let Many}\ x = e\ \textbf{in}\ e' \mid \textbf{fun}\ 'fc \rightarrow e\ (\text{frac. perm. abstraction}) \\
 & & e\ [f]\ (\text{frac. perm. specialisation}) \mid (e, e') \mid \textbf{let}\ (a, b) = e\ \textbf{in}\ e' \\
 & & \textbf{fun}\ x : t \rightarrow e \mid e\ e' \mid \textbf{fix}\,(g, x : t, e : t')
\end{array}
$$

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 Appendix E.

$$
\begin{array}{rcll}
x[e] & \Rightarrow & \textbf{get}\ \_\ x\ (e) & (\text{similarly for matrices}) \\
x[e_1] := e_2 & \Rightarrow & \textbf{set}\ x\ (e_1)\ (e_2) & (\text{similarly for matrices}) \\
\\
pat & ::= & () \mid x \mid !x \mid \textbf{Many}\ pat \mid (pat, pat) & \\
\textbf{let}\ !x = e_1\ \textbf{in}\ e_2 & \Rightarrow & \textbf{let Many}\ x = e_1\ \textbf{in} & \\
 & & \textbf{let Many}\ x = \textbf{Many}\ (\textbf{Many}\ x)\ \textbf{in}\ e_2 & \\
\textbf{let Many}\ \langle pat_x \rangle\ = e_1\ \textbf{in}\ e_2 & \Rightarrow & \textbf{let Many}\ x = x\ \textbf{in} & \\
 & & \textbf{let}\ \langle pat_x \rangle\ = x\ \textbf{in}\ e_2 & \\
\textbf{let}\ (\langle pat_a \rangle, \langle pat_b \rangle)\ = e_1\ \textbf{in}\ e_2 & \Rightarrow & \textbf{let}\ (a, b)\ = a\_b\ \textbf{in let}\ \langle pat_a \rangle\ = a\ \textbf{in} & \\
 & & \textbf{let}\ \langle pat_b \rangle\ = b\ \textbf{in}\ e_2 & \\
\textbf{fun}\ (\langle pat_x \rangle : t) \rightarrow e & \Rightarrow & \textbf{fun}\ (x : t) \rightarrow \textbf{let}\ \langle pat_x \rangle = x\ \textbf{in}\ e & \\
\\
arg & ::= & (\langle pat \rangle : t)\ \mid\ ('x)\ (\text{fractional permission variable}) & \\
\textbf{fun}\ \langle arg_{1..n} \rangle \rightarrow e & \Rightarrow & \textbf{fun}\ \langle arg_1 \rangle \rightarrow ..\ \textbf{fun}\ \langle arg_n \rangle \rightarrow e & \\
\textbf{let}\ f\ \langle arg_{1..n} \rangle = e_1\ \textbf{in}\ e_2 & \Rightarrow & \textbf{let}\ f = \textbf{fun}\ \langle arg_{1..n} \rangle \rightarrow e_1\ \textbf{in}\ e_2 & \\
\textbf{let}\ !f\ \langle arg_{1..n} \rangle = e_1\ \textbf{in}\ e_2 & \Rightarrow & \textbf{let Many}\ f = \textbf{Many}\ (\textbf{fun}\ \langle arg_{1..n} \rangle \rightarrow e_1)\ \textbf{in}\ e_2 & \\
\text{fixpoint} & \equiv & \textbf{fix}\ (f, x : t, \textbf{fun}\ \langle arg_{0..n} \rangle \rightarrow e_1 : t') & \\
\textbf{let rec}\ f\ (x : t)\ \langle arg_{0..n} \rangle : t' = e_1\ \textbf{in}\ e_2 & \Rightarrow & \textbf{let}\ f = \text{fixpoint}\ \textbf{in}\ e_2 & \\
\textbf{let rec}\ !f\ (x : t)\ \langle arg_{0..n} \rangle : t' = e_1\ \textbf{in}\ e_2 & \Rightarrow & \textbf{let Many}\ f = \textbf{Many}\ \text{fixpoint}\ \textbf{in}\ e_2 &
\end{array}
$$

Figure 2 Desugaring NumLin.

$$\begin{array}{rcll}
\textbf{let } v \leftarrow x[e] \textbf{ in } e' & \Rightarrow & \textbf{let } (x, !v) \; = x[e] \textbf{ in } e' & \text{(similarly for matrices)} \\
\textbf{let } x_2 \leftarrow \textbf{new } [|\; x_1 \;|] \textbf{ in } e & \Rightarrow & \textbf{let } (x_1, x_2) \; = \textbf{copyM } _ \; x_1 \textbf{ in } e & \\
\textbf{let } x_2 \leftarrow [|\; x_1 \;|] \textbf{ in } e & \Rightarrow & \textbf{let } (x_1, x_2) \; = \textbf{copyM_to } _ \; x_1 \; x_2 \textbf{ in } e &
\end{array}$$

$$M ::= X \;\mid\; X^T \;\mid\; \mathrm{sym}(X)$$

$$\begin{array}{l}
\textbf{let } Y \leftarrow \textbf{new } (n, k) \; [|\; \alpha M_1 M_2 \;|] \textbf{ in } e \Rightarrow \\
\quad \textbf{let } Y \; = \textbf{matrix } n \; k \textbf{ in let } Y \leftarrow [|\; \alpha M_1 M_2 + 0Y \;|] \textbf{ in } e \\
\textbf{let } Y \leftarrow [|\; \alpha X X^T + \beta Y \;|] \textbf{ in } e \Rightarrow \\
\quad \textbf{let } (X, Y) \; = \textbf{syrk false } \alpha \; _ \; X \; \beta \; Y \textbf{ in } e \\
\textbf{let } Y \leftarrow [|\; \alpha X^T X + \beta Y \;|] \textbf{ in } e \Rightarrow \\
\quad \textbf{let } (X, Y) \; = \textbf{syrk true } \alpha \; _ \; X \; \beta \; Y \textbf{ in } e \\
\textbf{let } Y \leftarrow \; [|\; \alpha \, \mathrm{sym}(X_1) \, X_2 + \beta Y \;|] \textbf{ in } e \Rightarrow \\
\quad \textbf{let } ((X_1, X_2), Y) \; = \textbf{symm false } \alpha \; _ \; X_1 _ \; X_2 \; \beta \; Y \textbf{ in } e \\
\textbf{let } Y \leftarrow \; [|\; \alpha \, X_2 \, \mathrm{sym}(X_1) + \beta Y \;|] \textbf{ in } e \Rightarrow \\
\quad \textbf{let } ((X_1, X_2), Y) \; = \textbf{symm true } \alpha \; _ \; X_1 _ \; X_2 \; \beta \; Y \textbf{ in } e \\
\textbf{let } Y \leftarrow [|\; \alpha X_1^{T?} X_2^{T?} + \beta Y \;|] \textbf{ in } e \Rightarrow \\
\quad \textbf{let } ((X_1, X_2), Y) \; = \; \textbf{gemm } \alpha \; _ \; (X_1, {}^{\textbf{true}}_{\textbf{false}}) \; _ \; (X_2, {}^{\textbf{true}}_{\textbf{false}}) \; \beta \; Y \textbf{ in } e
\end{array}$$

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 NumLin it represents the culmination of many features.

To simplify the design and implementation of 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 : !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 || 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 `Many` and all primitives involving them return a `!int`, `!bool` or `!elt` (types of elements of arrays/matrices, typically 64-bit floating-point numbers). The short-circuiting `||` 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 $\mathbf{get} : \ 'x.\ 'x\ \mathbf{arr} \multimap \mathbf{!int} \multimap 'x\ \mathbf{arr} \otimes \mathbf{!elt}$. 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.

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`

```
let row = row[i] := x1 in (* or *) let () = free row in
(* Could not show equality:                                        *)
(*      z arr                                                      *)
(* with                                                            *)
(*      'x arr                                                     *)
(*                                                                 *)
(* Var 'x 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 in
  simp_oned_conv ;;
```

Figure 7 *Simplified* one-dimensional convolution.

- `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 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

$$
\begin{array}{rcl}
\textbf{symm} & : & !\textbf{bool} \multimap !\textbf{elt} \multimap {}'x.\ {}'x\ \textbf{mat} \multimap {}'y.\ {}'y\ \textbf{mat} \multimap !\textbf{elt} \multimap \textbf{z mat} \multimap \\
 & & ({}'x\ \textbf{mat} \otimes {}'y\ \textbf{mat}) \otimes \textbf{z mat} \\
\textbf{gemm} & : & !\textbf{elt} \multimap {}'x.\ {}'x\ \textbf{mat} \otimes !\textbf{bool} \multimap {}'y.\ {}'y\ \textbf{mat} \otimes !\textbf{bool} \multimap !\textbf{elt} \multimap \\
 & & \textbf{z mat} \multimap ({}'x\ \textbf{mat} \otimes {}'y\ \textbf{mat}) \otimes \textbf{z mat} \\
\textbf{gesv} & : & \textbf{z mat} \multimap \textbf{z mat} \multimap \textbf{z mat} \otimes \textbf{z mat} \\
\textbf{posv} & : & \textbf{z mat} \multimap \textbf{z mat} \multimap \textbf{z mat} \otimes \textbf{z mat} \\
\textbf{potrs} & : & {}'x.\ {}'x\ \textbf{mat} \multimap \textbf{z mat} \multimap {}'x\ \textbf{mat} \otimes \textbf{z mat} \\
\textbf{syrk} & : & !\textbf{bool} \multimap !\textbf{elt} \multimap {}'x.\ {}'x\ \textbf{mat} \multimap !\textbf{elt} \multimap \textbf{z mat} \multimap {}'x\ \textbf{mat} \otimes \textbf{z mat}
\end{array}
$$

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

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 $\textbf{shareM} : {}'x.\ {}'x\ \textbf{mat} \multimap {}'x\ \texttt{s}\ \textbf{mat} \otimes {}'x\ \texttt{s}\ \textbf{mat}$ 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
- 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

```
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 * y |] in
  let x_T_x <- new (m, m) [| x^T * x |] in
  let (to_del, answer) = posv x_T_x xy in
  let () = freeM to_del in
  ((x, y), answer) in
lin_reg ;;
```

Figure 10 Linear regression (OLS): $\hat{\beta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}$

routines.

2.3.6 Linear Regression

In Figure 10, we wish to compute $\hat{\beta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}$. 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 $\mathbf{X^T y}$ (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 $\textbf{posv} : \ \textbf{z mat} \multimap \textbf{z mat} \multimap \textbf{z mat} \otimes \textbf{z mat}$ 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 $\textbf{transpose} : \ {}'x.\, {}'x\ \textbf{mat} \multimap {}'x\ \textbf{mat} \otimes \textbf{z mat}$ and $\textbf{eye} : \ !\textbf{int} \multimap \textbf{z mat}$ 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. * tmp_n_n + q_inv_u * 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: $\mathbf{Q^{-1}U(I + U^T Q^{-1} U)^{-1} U^T}$

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 Appendix A.1, but the key ideas are as follow.

```
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) * hT |] but would
     need a (n,k) temporary hT = tranpose _ h *)
  let sigma_hT <- new (n, k) [| sigma * h^T |] in
  let r_2 <- [| r_1 + h * sigma_hT |] in
  let (k_by_k, x) = posvFlip r_2 sigma_hT in
  let data_2 <- [| h * mu - data_1 |] in
  let new_mu <- [| mu + x * data_2 |] in
  let x_h <- new (n,n) [| x * h |] in
  let () = freeM (* n,k *) x in
  let sigma2 <- new [| sigma |] in
  let new_sigma <- [| sigma2 - x_h * sym(sigma) |] in
  let () = freeM (* n,n *) x_h in
  ((sigma, h), (new_sigma, (new_mu, (k_by_k, data_2)))) in
kalman ;;
```

Figure 12 Kalman filter: see Figure 13 for the equations this code implements and Figure 20 for an equivalent CBLAS/LAPACKE implementation.

$$\mu' = \mu + \Sigma H^T (R + H\Sigma H^T)^{-1}(H\mu - \text{data})$$
$$\Sigma' = \Sigma(I - H^T(R + H\Sigma H^T)^{-1} H\Sigma)$$

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

- A typing judgement consists of $\Theta; \Delta; \Gamma \vdash e : t$.
- Θ is the environment that tracks which fractional permission variables in scope. Fractional permissions (the Perm judgement) and types (the Type judgement) are *well-formed* if all of their free fractional variables are in Θ.
- Δ is the environment storing non-linearly or *inuitionistically* typed variables.
- Γ 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.

$$\frac{}{\Theta; \Delta; \cdot \vdash () : \mathbf{unit}} \quad \textsc{Ty\_Unit\_Intro}$$

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).

$$\frac{\begin{array}{l}\Theta; \Delta; \Gamma \vdash e : !\mathbf{bool} \\ \Theta; \Delta; \Gamma' \vdash e_1 : t' \\ \Theta; \Delta; \Gamma' \vdash e_2 : t'\end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : t} \quad \textsc{Ty\_Bool\_Elim}$$

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 **let Many** $x =$ **Many** (**array** 5) **in let** $() =$ **free** x **in** $x[0]$ would type-check but error at runtime.

$$\frac{\Theta; \Delta; \cdot \vdash v : t \qquad v \neq l}{\Theta; \Delta; \cdot \vdash \textbf{Many}\, v : !t} \quad \textsc{Ty\_Bang\_Intro}$$

Consuming a variable that refers to a !-type value *moves it* from the linear environment Γ and *into* the intuitionistic environment Δ.

$$\frac{\Theta; \Delta; \Gamma \vdash e : !t \qquad \Theta; \Delta, x : t; \Gamma' \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash \textbf{let Many}\, x = e\, \textbf{in}\, e' : t'} \quad \textsc{Ty\_Bang\_Elim}$$

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 **let** $!x = e$ **in** e' can be expressed using only the rules presented thus far, as **let Many** $x = e$ **in let Many** $x =$ **Many** (**Many** x) **in** e'.[1] The reader is invited to quickly convince themselves that the following meta-rule is provable using Ty_Bang_Intro (twice), Ty_Bang_Elim (twice) and weakening the intuitionistic environment Δ (once).

$$\frac{\Theta; \Delta; \Gamma \vdash e : !t \qquad \Theta; \Delta, x : !t; \Gamma' \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash \textbf{let}\, !x = e\, \textbf{in}\, e' : t'} \quad \textsc{Meta\_Ty\_Let\_Bang}$$

Rules Ty_Gen and 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.

$$\frac{\Theta, fc; \Delta; \Gamma \vdash e : t}{\Theta; \Delta; \Gamma \vdash \textbf{fun}\, 'fc \to e : \; 'fc.t} \quad \textsc{Ty\_Gen} \qquad \frac{\Theta \vdash f\ \mathsf{Perm} \qquad \Theta; \Delta; \Gamma \vdash e : \; 'fc.t}{\Theta; \Delta; \Gamma \vdash e[f] : t[f/fc]} \quad \textsc{Ty\_Spc}$$

Rule 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 recusive function is in the *intuitionistic* environment Δ (without this, you would not be able to write a base case) whilst the argument and its type are the *only things in the linear environment* Γ. The latter means that recursive functions can be type checked in an empty environment (thus be wrapped in `Many` and used zero or multiple times).

[1] *Why* we have this at all is for the sake of ergonomics when using binary arithmetic operations (e.g. of type $!\textbf{int} \multimap !\textbf{int} \multimap !\textbf{int}$): a programmer should be able to write `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.

$$\frac{\Theta; \Delta, g : t \multimap t'; \cdot, x : t \vdash e : t'}{\Theta; \Delta; \cdot \vdash \mathbf{fix}\,(g, x : t, e : t') : t \multimap t'} \quad \textsc{Ty\_Fix}$$

Lastly, types of almost all NumLin primitives, as embedded in OCaml's type system, are shown in Appendix F, with some similar ones (like those for binary arithmetic operators) 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 Appendix A.2, but the key ideas are as follow.

Heaps σ 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 **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** $'fc \to v$ must be a syntactic value. The context **fun** $'fc \to [-]$ 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_Perm).

$$\frac{}{\langle \sigma, (\mathbf{fun}\,'fc \to v)[f] \rangle \to \langle \sigma[f/fc], v[f/fc] \rangle} \quad \textsc{Op\_Frac\_Perm}$$

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.

$$\frac{\begin{array}{l} 0 \leq k_1, k_2 \\ l \text{ fresh} \end{array}}{\langle \sigma, \mathbf{matrix}\, k_1\ k_2 \rangle \to \langle \sigma + \{l \mapsto_1 M_{k_1,k_2}\}, l \rangle} \quad \textsc{Op\_Matrix}$$

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

$$\frac{}{\langle \sigma + \{l \mapsto_1 m_{k_1,k_2}\}, \mathbf{free}\, l \rangle \to \langle \sigma, () \rangle} \quad \textsc{Op\_Free}$$

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.

$$\frac{}{\langle \sigma + \{l \mapsto_f m_{k_1,k_2}\}, \mathbf{share}[f]\, l \rangle \to \langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\}, (l, l) \rangle} \quad \textsc{Op\_Share}$$

$$\frac{\sigma' \equiv \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\}}{\langle \sigma', \mathbf{unshare}[f]\, l\ l \rangle \to \langle \sigma + \{l \mapsto_f m_{k_1,k_2}\}, l \rangle} \quad \textsc{Op\_Unshare\_Eq}$$

Combining all of these features, we see that $\textsc{Op\_Gemm\_Match}$ 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 **share**d aliases of the same matrix. Transition rules for other primitives (omitted) follow the same structure: $\mapsto_1$ for any locations that are written to and $\mapsto_{fc}$ for anything else.

$$\frac{\begin{array}{l}\sigma' \equiv \sigma + \{l_1 \mapsto_{fc_1} m_{1\,k_1,k_2}\} + \{l_2 \mapsto_{fc_2} m_{2\,k_2,k_3}\} \\ \sigma_1 \equiv \sigma' + \{l_3 \mapsto_1 m_{3\,k_1,k_3}\} \\ \sigma_2 \equiv \sigma' + \{l_3 \mapsto_1 (m_1\, m_2 + m_3)_{k_1,k_3}\}\end{array}}{\langle \sigma_1, \mathbf{gemm}[fc_1]\; l_1 [fc_2]\; l_2\; l_3\rangle \to \langle \sigma_2, ((l_1, l_2), l_3)\rangle}\ \textsc{Op\_Gemm\_Match}$$

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 $\to$.

$$\mathcal{H}[\![\sigma]\!] = \bigstar_{(l,f,m)\in\sigma}[l \mapsto_f m]$$

where

$$(\varsigma_1 \star \varsigma_2)(l) \equiv \begin{cases} \varsigma_1(l) & \text{if } l \in \mathrm{dom}(\varsigma_1) \land l \notin \mathrm{dom}(\varsigma_2) \\ \varsigma_2(l) & \text{if } l \in \mathrm{dom}(\varsigma_2) \land l \notin \mathrm{dom}(\varsigma_1) \\ (f_1 + f_2, m) & \text{if } (f_1, m) = \varsigma_1(l) \land (f_2, m) = \varsigma_2(l) \land f_1 + f_2 \le 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

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 ς resources to produce a value v of type t in at most k steps. So, something like a **unit** or a $!t$ need no resources, whereas a f **mat** needs exactly f ownership of a some matrix and a pair needs a $\star$ combination of the heaps required for each component.

$$\begin{aligned} \mathcal{V}_k[\![\mathbf{unit}]\!] &= \{(\emptyset, *)\} \\ \mathcal{V}_k[\![f\ \mathbf{mat}]\!] &= \{(\{l \mapsto_{2^{-f}} \_\}, l)\} \\ \mathcal{V}_k[\![!t]\!] &= \{(\emptyset, \mathbf{Many}\ v) \mid (\emptyset, v) \in \mathcal{V}_k[\![t]\!]\} \\ \mathcal{V}_k[\![t_1 \otimes t_2]\!] &= \{(\varsigma_1 \star \varsigma_2, (v_1, v_2)) \mid (\varsigma_1, v_1) \in \mathcal{V}_k[\![t_1]\!] \land (\varsigma_2, v_2) \in \mathcal{V}_k[\![t_2]\!]\} \end{aligned}$$

The definition of $\mathcal{V}_k[\![\prime fc.\ t]\!]$ 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 ($\textsc{Op\_Spc}$).

$$\mathcal{V}_k[\![\prime fc.\ t]\!] = \{(\varsigma, \mathbf{fun}\ \prime fc \to v) \mid \forall f.\ (\varsigma[f/fc], v[f/fc]) \in \mathcal{V}_{k-1}[\![t[f/fc]]\!]\}$$

To understand the definition of $\mathcal{V}_k[\![t' \multimap t]\!]$, 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.

$$\begin{aligned} \mathcal{C}_k[\![t]\!] = \{(\varsigma_s, e_s) \mid \forall j < k, \sigma_r.\ \varsigma_s \star \varsigma_r \text{ defined } &\Rightarrow \langle \sigma_s + \sigma_r, e_s\rangle \to^j \mathbf{err}\ \lor \exists \sigma_f, e_f. \\ &\langle \sigma_s + \sigma_r, e_s\rangle \to^j \langle \sigma_f + \sigma_r, e_f\rangle \land (e_f \text{ is a value } \Rightarrow (\varsigma_f \star \varsigma_r, e_f) \in \mathcal{V}_{k-j}[\![t]\!])\} \end{aligned}$$

In this light, $\mathcal{V}_k[\![t' \multimap t]\!]$ 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.

$$\begin{aligned}\mathcal{V}_k[\![t' \multimap t]\!] = \{(\varsigma_v, v) \mid (v \equiv \textbf{fun}\ x : t' \rightarrow e \vee v \equiv \textbf{fix}(g, x : t', e : t)) \wedge \\ \forall j \leq k, (\varsigma_{v'}, v') \in \mathcal{V}_j[\![t']\!].\ \varsigma_v \star \varsigma_v' \text{ defined } \Rightarrow (\varsigma_v \star \varsigma_v', v\, v') \in \mathcal{C}_j[\![t]\!]\}\end{aligned}$$

The interpretation of typing environments Δ and Γ are with respect to an arbitrary substitution of fractional permissions θ. Note that only the interpretation of Γ involves a (potentially) non-empty heap.

$$\mathcal{I}_k[\![\Delta, x : t]\!]\theta = \{\delta[x \mapsto v_x] \mid \delta \in \mathcal{I}_k[\![\Delta]\!]\theta \wedge (\emptyset, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]\}$$
$$\mathcal{L}_k[\![\Gamma, x : t]\!]\theta = \{(\varsigma \star \varsigma_x, \gamma[x \mapsto v_x]) \mid (\varsigma, \gamma) \in \mathcal{L}_k[\![\Gamma]\!]\theta \wedge (\varsigma_x, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]\}$$

And so the final semantic interpretation of a typing judgement simply quantifies over all possible fractional permission substitutions θ, linear value substitutions γ, intuitionistic value substitutions δ and heaps σ. Note that, $\varsigma \equiv \mathcal{H}[\![\theta(\sigma)]\!]$.

$$\begin{aligned}{}_k[\![\Theta; \Delta; \Gamma \vdash e : t]\!] = \forall \theta, \delta, \gamma, \sigma.\ \Theta = \text{dom}(\theta) \wedge (\varsigma, \gamma) \in \mathcal{L}_k[\![\Gamma]\!]\theta \wedge \delta \in \mathcal{I}_k[\![\Delta]\!]\theta \Rightarrow \\ (\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\![\theta(t)]\!]\end{aligned}$$

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 θ,
  - linear variables with a suitable heap (γ, ς) and
  - intuitionistic variables δ,
- the aforementioned suitable heap and expression $(\varsigma, \theta(\delta(\gamma(e))))$
- are in the computational interpretation $\mathcal{C}_k[\![\theta(t)]\!]$ 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 (C.1), monotonicity for the step-index (C.5), splitting up environments corresponds to splitting up heaps (C.7) and heap-and-expressions take the same steps of evaluation under any substitution of their free fractional permissions (C.8).

The proof proceeds by induction on the typing judgement. The case for Ty_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 Ty_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 C.8 (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 Ty_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 Γ/Γ', then (by induction) use $\mathcal{C}_k[\![-]\!]$ and one 'half' of the linear substitution and heap to conclude the 'first' sub-expression either takes $j < k$ steps to **err** or another heap-and-expression.

In the first case, you use Op_Context_Err to conclude the whole let-expression does the same. Similarly we use Op_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: Op_Context is not functorial for the n-fold iteration of $\rightarrow$. Basically, the following is not quite true:

$$\frac{\langle \sigma, e \rangle \rightarrow^j \langle \sigma', e' \rangle}{\langle \sigma, C[e] \rangle \rightarrow^j \langle \sigma', C[e'] \rangle}\ \textsc{Op\_Context}$$

because after the i steps, we need to invoke Op_Let_Var to proceed evalution 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 $\mathcal{L}_k[\![\Gamma', x : t]\!]\theta$). We take the other 'half' of the linear substitution and heap (from the inital 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

NumLin transpiles to OCaml and its implementation follows the structure of a typical domain-specific language (DSL) compiler. Although NumLin's current implementation is not as an embedded DSL, its the 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 Figure 19.

$$
\begin{array}{rl}
f ::= & \\
\mid & \text{'}fc \\
\mid & \mathbf{z} \\
\mid & f\ \mathbf{s} \\
t ::= & \\
\mid & \mathbf{unit} \\
\mid & \mathbf{bool} \\
\mid & \mathbf{int} \\
\mid & \mathbf{elt} \\
\mid & f\ \mathbf{arr} \\
\mid & f\ \mathbf{mat} \\
\mid & !\ t \\
\mid & \text{'}fc.\ t \\
\mid & t \otimes t' \\
\mid & t \multimap t'
\end{array}
$$

```
module Arr =
  Owl.Dense.Ndarray.D

type z = Z
type 'a s = Succ

type 'a arr =
  A of Arr.arr
  [@@unboxed]

type 'a mat =
  M of Arr.arr
  [@@unboxed]

type 'a bang =
  Many of 'a
  [@@unboxed]
```

$$
\begin{aligned}
[\![\text{'}fc]\!] &= \texttt{'fc} \\
[\![\mathbf{z}]\!] &= \texttt{z} \\
[\![f\ \mathbf{s}]\!] &= [\![f]\!]\ \texttt{s} \\
[\![\mathbf{unit}]\!] &= \texttt{unit} \\
[\![\mathbf{bool}]\!] &= \texttt{bool} \\
[\![\mathbf{int}]\!] &= \texttt{int} \\
[\![\mathbf{elt}]\!] &= \texttt{float} \\
[\![f\ \mathbf{arr}]\!] &= [\![f]\!]\ \texttt{arr} \\
[\![f\ \mathbf{mat}]\!] &= [\![f]\!]\ \texttt{mat} \\
[\![!\ t]\!] &= [\![t]\!]\ \texttt{bang} \\
[\![\text{'}fc.\ t]\!] &= [\![t]\!] \\
[\![t \otimes t']\!] &= [\![t]\!] * [\![t']\!] \\
[\![t \multimap t']\!] &= [\![t]\!] \to [\![t']\!]
\end{aligned}
$$

Figure 14 NumLin's type grammar (left) and its embedding into OCaml (right).

```
  1 let lt4la_kalman ~sigma ~h ~mu ~r ~data =
  0     Examples.Kalman.it (M sigma) (M h) (M mu) (M r) (M data)
 NORMAL  test/examples_test.ml
'a mat ->
'b mat ->
'c mat ->
z mat ->
z mat -> ('a mat * ('b mat * ('c mat * (z mat * z mat)))) * (z mat * z mat)
 :merlin-type-history:
  0   let fact = Examples.Factorial.it in
 NORMAL  test/examples_test.ml
int bang -> int bang
```

Figure 15 Using NumLin functions from OCaml.

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-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

```
        let Many x = x in
        let Many x = Many (Many x) in <exp>   ⇒   <exp>

      (* fixp = fix (f, x:t, <exp> : t') *)
      (*1*) let Many f = Many fixp in <body>   ⇒   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>
   (*2*) let Many x = Many <exp> in <body>     ⇒   let x = <exp> in <body>
   (*3*) (fun x : t -> <body>) <exp>
```

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

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

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.

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" ($(\mathbf{X^T X})^{-1}\mathbf{X^T y}$) similary, 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 $112 \pm 35\,ms$, NumLin took $105 \pm 25\,ms$, Owl took $124 \pm 38\,ms$ and NumPy took $112 \pm 12\,ms$; for $n = 3125$, CBlas took $10.8 \pm 0.7\,s$, NumLin took $12.0 \pm 1.2\,s$, Owl took $13.3 \pm 0.2\,s$ and NumPy took $12.7 \pm 0.6\,s$.

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.

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 $f : a \multimap b$, if $f u$ 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.

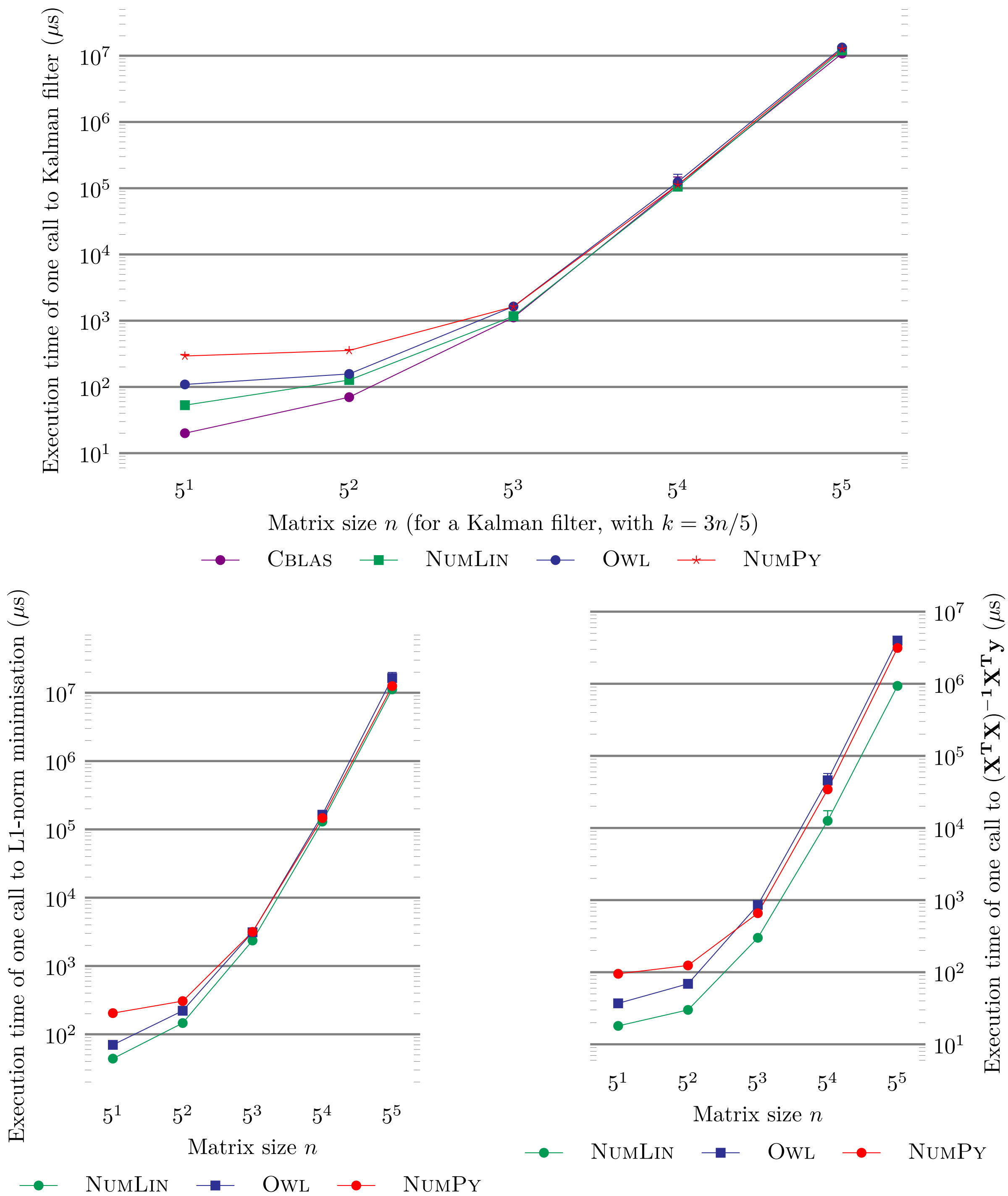
Indeed, the focus on getting a working prototype early on (so that we could test it with real BLAS/LAPACK routines as soon as possible) meant that we only added features to the type system when it was clear that they were absolutely necessary: these features were !-types and value-restriction for the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



■ **Figure 18** Comparison of execution times (error bars are present but quite small). 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 `Gc.full_major` for the OCaml implementations.

References

1 Akinori Abe and Eijiro Sumii. A simple and practical linear algebra library interface with static size checking. *arXiv preprint arXiv:1512.01898*, 2015.

2 Amal Ahmed, Matthew Fluet, and Greg Morrisett. Lˆ 3: a linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, 2007.

3 Andrew Barber and Gordon Plotkin. *Dual intuitionistic linear logic*. University of Edinburgh, Department of Computer Science, Laboratory for . . . , 1996.

4 Henrik Barthels, Marcin Copik, and Paolo Bientinesi. The generalized matrix chain algorithm. *arXiv preprint arXiv:1804.04021*, 2018.

5 P Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic*, pages 121–135. Springer, 1995.

6 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Procedings of the ACM on Programming Languages*, 2(POPL):5, 2017.

7 Jean-Philippe Bernardy, Vıctor López Juan, and Josef Svenningsson. Composable efficient array computations using linear types. *Unpublished Draft*, 2016.

8 Kevin Bierhoff and Jonathan Aldrich. Plural: checking protocol compliance under aliasing. In *Companion of the 30th international conference on Software engineering*, pages 971–972. ACM, 2008.

9 Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. Fraction polymorphic permission inference.

10 Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *ACM SIGPLAN Notices*, volume 40, pages 259–270. ACM, 2005.

11 John Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.

12 Alex Bronstein, Yoni Choukroun, Ron Kimmel, and Matan Sela. Consistent discretization and minimization of the l1 norm on manifolds. In *3D Vision (3DV), 2016 Fourth International Conference on*, pages 435–440. IEEE, 2016.

13 Sa Cui, Kevin Donnelly, and Hongwei Xi. Ats: A language that combines programming with theorem proving. In *International Workshop on Frontiers of Combining Systems*, pages 310–320. Springer, 2005.

14 Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. *ACM SIGPLAN Notices*, 52(6):556–571, 2017.

15 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018. URL: `https://doi.org/10.1145/3158154`, `doi:10.1145/3158154`.

16 Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

17 Conor McBride. Code mesh london 2016, keynote: Spacemonads. `https://www.youtube.com/watch?v=QojLQY5H0RI`. Accessed: 08/05/2018.

18 Matthew Rocklin. *Mathematically informed linear algebra codes through term rewriting*. PhD thesis, 2013.

19 Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of functional programming*, 13(6):1005–1059, 2003.

20 Philip Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, April 1990. North-Holland.

21 Aaron Weiss, Daniel Patterson, and Amal Ahmed. Rust distilled: An expressive tower of languages. *arXiv preprint arXiv:1806.02693*, 2018.

A NumLin Specification

A.1 Static Semantics

$\boxed{\Theta; \Delta; \Gamma \vdash e : t}$ Typing rules for expressions

$$\frac{}{\Theta; \Delta; \cdot, x : t \vdash x : t} \textsc{Ty\_Var\_Lin}$$

$$\frac{x : t \in \Delta}{\Theta; \Delta; \cdot \vdash x : t} \textsc{Ty\_Var}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : t \quad \Theta; \Delta; \Gamma', x : t \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let}\, x = e\, \mathbf{in}\, e' : t'} \textsc{Ty\_Let}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash () : \mathbf{unit}} \textsc{Ty\_Unit\_Intro}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : \mathbf{unit} \quad \Theta; \Delta; \Gamma' \vdash e' : t}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let}\, () = e\, \mathbf{in}\, e' : t} \textsc{Ty\_Unit\_Elim}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash \mathbf{true} : \mathbf{bool}} \textsc{Ty\_Bool\_True}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash \mathbf{false} : \mathbf{bool}} \textsc{Ty\_Bool\_False}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : !\mathbf{bool} \quad \Theta; \Delta; \Gamma' \vdash e_1 : t' \quad \Theta; \Delta; \Gamma' \vdash e_2 : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{if}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 : t} \textsc{Ty\_Bool\_Elim}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash k : \mathbf{int}} \textsc{Ty\_Int\_Intro}$$

$$\frac{}{\Theta; \Delta; \cdot \vdash el : \mathbf{elt}} \textsc{Ty\_Elt\_Intro}$$

$$\frac{\Theta; \Delta; \cdot \vdash v : t \quad v \neq l}{\Theta; \Delta; \cdot \vdash \mathbf{Many}\, v : !t} \textsc{Ty\_Bang\_Intro}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : !t \quad \Theta; \Delta, x : t; \Gamma' \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{let\, Many}\, x = e\, \mathbf{in}\, e' : t'} \textsc{Ty\_Bang\_Elim}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : t \quad \Theta; \Delta; \Gamma' \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash (e, e') : t \otimes t'} \textsc{Ty\_Pair\_Intro}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e_{12} : t_1 \otimes t_2 \quad \Theta; \Delta; \Gamma', a : t_1, b : t_2 \vdash e : t}{\Theta; \Delta; \Gamma, \Gamma' \vdash \textbf{let}\,(a, b) = e_{12}\,\textbf{in}\,e : t} \quad \textsc{Ty\_Pair\_Elim}$$

$$\frac{\Theta \vdash t'\ \mathsf{Type} \quad \Theta; \Delta; \Gamma, x : t' \vdash e : t}{\Theta; \Delta; \Gamma \vdash \textbf{fun}\,x : t' \to e : t' \multimap t} \quad \textsc{Ty\_Lambda}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e : t' \multimap t \quad \Theta; \Delta; \Gamma' \vdash e' : t'}{\Theta; \Delta; \Gamma, \Gamma' \vdash e\ e' : t} \quad \textsc{Ty\_App}$$

$$\frac{\Theta, fc; \Delta; \Gamma \vdash e : t}{\Theta; \Delta; \Gamma \vdash \textbf{fun}\,'fc \to e :\ 'fc.t} \quad \textsc{Ty\_Gen}$$

$$\frac{\Theta \vdash f\ \mathsf{Perm} \quad \Theta; \Delta; \Gamma \vdash e :\ 'fc.t}{\Theta; \Delta; \Gamma \vdash e[f] : t[f/fc]} \quad \textsc{Ty\_Spc}$$

$$\frac{\Theta; \Delta, g : t \multimap t'; \cdot, x : t \vdash e : t'}{\Theta; \Delta; \cdot \vdash \textbf{fix}\,(g, x : t, e : t') : t \multimap t'} \quad \textsc{Ty\_Fix}$$

A.2 Dynamic Semantics

$\boxed{\langle \sigma, e \rangle \to \mathit{Config}}$ Operational semantics

$$\frac{}{\langle \sigma, \textbf{let}\,() = ()\,\textbf{in}\,e \rangle \to \langle \sigma, e \rangle} \quad \textsc{Op\_Let\_Unit}$$

$$\frac{}{\langle \sigma, \textbf{let}\,x = v\,\textbf{in}\,e \rangle \to \langle \sigma, e[v/x] \rangle} \quad \textsc{Op\_Let\_Var}$$

$$\frac{}{\langle \sigma, \textbf{if}\,(\textbf{Many}\,\textbf{true})\,\textbf{then}\,e_1\,\textbf{else}\,e_2 \rangle \to \langle \sigma, e_1 \rangle} \quad \textsc{Op\_If\_True}$$

$$\frac{}{\langle \sigma, \textbf{if}\,(\textbf{Many}\,\textbf{false})\,\textbf{then}\,e_1\,\textbf{else}\,e_2 \rangle \to \langle \sigma, e_2 \rangle} \quad \textsc{Op\_If\_False}$$

$$\frac{}{\langle \sigma, \textbf{let}\,\textbf{Many}\,x = \textbf{Many}\,v\,\textbf{in}\,e \rangle \to \langle \sigma, e[v/x] \rangle} \quad \textsc{Op\_Let\_Many}$$

$$\frac{}{\langle \sigma, \textbf{let}\,(a, b) = (v_1, v_2)\,\textbf{in}\,e \rangle \to \langle \sigma, e[v_1/a][v_2/b] \rangle} \quad \textsc{Op\_Let\_Pair}$$

$$\frac{}{\langle \sigma, (\textbf{fun}\,'fc \to v)[f] \rangle \to \langle \sigma[f/fc], v[f/fc] \rangle} \quad \textsc{Op\_Frac\_Perm}$$

$$\frac{}{\langle \sigma, \textbf{fix}\,(g, x : t, e : t')\ v \rangle \to \langle \sigma, e[v/x][\textbf{fix}\,(g, x : t, e : t')/g] \rangle} \quad \textsc{Op\_App\_Fix}$$

$$\frac{}{\langle \sigma, (\textbf{fun}\,x : t \to e)\ v \rangle \to \langle \sigma, e[v/x] \rangle} \quad \textsc{Op\_App\_Lambda}$$

$$\frac{\langle \sigma, e\rangle \to \langle \sigma', e'\rangle}{\langle \sigma, C[e]\rangle \to \langle \sigma', C[e']\rangle}\ \textsc{Op\_Context}$$

$$\frac{\langle \sigma, e\rangle \to \mathbf{err}}{\langle \sigma, C[e]\rangle \to \mathbf{err}}\ \textsc{Op\_Context\_Err}$$

$$\frac{0 \le k_1, k_2 \qquad l \text{ fresh}}{\langle \sigma, \mathbf{matrix}\, k_1\, k_2\rangle \to \langle \sigma + \{l \mapsto_1 M_{k_1,k_2}\}, l\rangle}\ \textsc{Op\_Matrix}$$

$$\frac{k_1 < 0 \text{ or } k_2 < 0}{\langle \sigma, \mathbf{matrix}\, k_1\, k_2\rangle \to \mathbf{err}}\ \textsc{Op\_Matrix\_Neg}$$

$$\frac{}{\langle \sigma + \{l \mapsto_1 m_{k_1,k_2}\}, \mathbf{free}\, l\rangle \to \langle \sigma, ()\rangle}\ \textsc{Op\_Free}$$

$$\frac{}{\langle \sigma + \{l \mapsto_f m_{k_1,k_2}\}, \mathbf{share}[f]\, l\rangle \to \langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\}, (l, l)\rangle}\ \textsc{Op\_Share}$$

$$\frac{\sigma' \equiv \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\}}{\langle \sigma', \mathbf{unshare}[f]\, l\, l\rangle \to \langle \sigma + \{l \mapsto_f m_{k_1,k_2}\}, l\rangle}\ \textsc{Op\_Unshare\_Eq}$$

$$\frac{l \ne l'}{\langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\} + \{l' \mapsto_{\frac{1}{2}f} m'_{k_1,k_2}\}, \mathbf{unshare}[f]\, l\, l'\rangle \to \mathbf{err}}\ \textsc{Op\_Unshare\_Neq}$$

$$\frac{\begin{array}{l}\sigma' \equiv \sigma + \{l_1 \mapsto_{fc_1} m1_{k_1,k_2}\} + \{l_2 \mapsto_{fc_2} m2_{k_2,k_3}\}\\ \sigma_1 \equiv \sigma' + \{l_3 \mapsto_1 m3_{k_1,k_3}\}\\ \sigma_2 \equiv \sigma' + \{l_3 \mapsto_1 (m_1\, m_2 + m_3)_{k_1,k_3}\}\end{array}}{\langle \sigma_1, \mathbf{gemm}[fc_1]\, l_1 [fc_2]\, l_2\, l_3\rangle \to \langle \sigma_2, ((l_1, l_2), l_3)\rangle}\ \textsc{Op\_Gemm\_Match}$$

$$\frac{\begin{array}{l}k_2 \ne k_2'\\ \sigma' \equiv \sigma + \{l_1 \mapsto_{fc_1} m1_{k_1,k_2}\} + \{l_2 \mapsto_{fc_2} m2_{k_2',k_3}\}\end{array}}{\langle \sigma' + \{l_3 \mapsto_1 m1_{k_1,k_3}\}, \mathbf{gemm}[fc_1]\, l_1 [fc_2]\, l_2\, l_3\rangle \to \mathbf{err}}\ \textsc{Op\_Gemm\_Mismatch}$$

B Interpretation

B.1 Definitions

Operationally, $\mathit{Heap} \sqsubseteq \mathit{Loc} \times \mathit{Permission} \times \mathit{Matrix}$ (a multiset), denoted with a σ.
Define its *interpretation* to be $\mathit{Loc} \rightharpoonup \mathit{Permission} \times \mathit{Matrix}$ with $\star : \mathit{Heap} \times \mathit{Heap} \rightharpoonup \mathit{Heap}$ as follows:

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

Commutativity and associativity of $\star$ follows from that of $+$.
$\varsigma_1 \star \varsigma_2$ is *defined* if it is for all $l \in \mathrm{dom}(\varsigma_1) \cup \mathrm{dom}(\varsigma_2)$.
Define $\mathcal{H}[\![\sigma]\!] = \bigstar_{(l,f,m)\in\sigma}[l \mapsto_f m]$ and **implicitly denote** $\varsigma \equiv \mathcal{H}[\![\theta(\sigma)]\!]$.

The $n-$fold iteration for the $\to$ (functional) relation, is also a (functional) relation:

$$\forall n.\ \mathbf{err} \to^n \mathbf{err} \qquad \langle\sigma, v\rangle \to^n \langle\sigma, v\rangle \qquad \langle\sigma, e\rangle \to^0 \langle\sigma, e\rangle \qquad \langle\sigma, e\rangle \to^{n+1} ((\langle\sigma, e\rangle \to) \to^n)$$

Hence, all bounded iterations end in either an **err**, a heap-and-expression or a heap-and-value.

B.2 Interpretation

$$\mathcal{V}_k[\![\mathbf{unit}]\!] = \{(\emptyset, *)\}$$

$$\mathcal{V}_k[\![\mathbf{bool}]\!] = \{(\emptyset, true), (\emptyset, false)\}$$

$$\mathcal{V}_k[\![\mathbf{int}]\!] = \{(\emptyset, n) \mid 2^{-63} \leq n \leq 2^{63} - 1\}$$

$$\mathcal{V}_k[\![\mathbf{elt}]\!] = \{(\emptyset, f) \mid f \text{ a IEEE Float64 }\}$$

$$\mathcal{V}_k[\![f\,\mathbf{mat}]\!] = \{(\{l \mapsto_{2^{-f}} \_\}, l)\}$$

$$\mathcal{V}_k[\![!t]\!] = \{(\emptyset, \mathbf{Many}\, v) \mid (\emptyset, v) \in \mathcal{V}_k[\![t]\!]\}$$

$$\mathcal{V}_k[\![{'}fc.\ t]\!] = \{(\varsigma, \mathbf{fun}\,{'}fc \rightarrow\ v) \mid \forall f.\ (\varsigma[f/fc], v[f/fc]) \in \mathcal{V}_{k-1}[\![t[f/fc]]\!]\}$$

$$\mathcal{V}_k[\![t_1 \otimes t_2]\!] = \{(\varsigma_1 \star \varsigma_2, (v_1, v_2)) \mid (\varsigma_1, v_1) \in \mathcal{V}_k[\![t_1]\!] \wedge (\varsigma_2, v_2) \in \mathcal{V}_k[\![t_2]\!]\}$$

$$\begin{aligned}\mathcal{V}_k[\![t' \multimap t]\!] = \{(\varsigma_v, v) \mid\ & (v \equiv \mathbf{fun}\, x : t' \rightarrow e \vee v \equiv \mathbf{fix}(g, x : t', e : t)) \wedge \\ & \forall j \leq k, (\varsigma_{v'}, v') \in \mathcal{V}_j[\![t']\!].\ \varsigma_v \star \varsigma_v' \text{ defined } \Rightarrow (\varsigma_v \star \varsigma_v', v\, v') \in \mathcal{C}_j[\![t]\!]\}\end{aligned}$$

$$\begin{aligned}\mathcal{C}_k[\![t]\!] = \{(\varsigma_s, e_s) \mid\ & \forall\, j < k, \sigma_r.\ \varsigma_s \star \varsigma_r \text{ defined } \Rightarrow \langle \sigma_s + \sigma_r, e_s\rangle \rightarrow^j \mathbf{err}\ \vee \exists \sigma_f, e_f. \\ & \langle \sigma_s + \sigma_r, e_s\rangle \rightarrow^j \langle \sigma_f + \sigma_r, e_f\rangle \wedge (e_f \text{ is a value } \Rightarrow (\varsigma_f \star \varsigma_r, e_f) \in \mathcal{V}_{k-j}[\![t]\!])\}\end{aligned}$$

$$\mathcal{I}_k[\![\cdot]\!]\theta = \{[]\}$$

$$\mathcal{I}_k[\![\Delta, x : t]\!]\theta = \{\delta[x \mapsto v_x] \mid \delta \in \mathcal{I}_k[\![\Delta]\!]\theta \wedge (\emptyset, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]\}$$

$$\mathcal{L}_k[\![\cdot]\!]\theta = \{(\emptyset, [])\}$$

$$\mathcal{L}_k[\![\Gamma, x : t]\!]\theta = \{(\varsigma \star \varsigma_x, \gamma[x \mapsto v_x]) \mid (\varsigma, \gamma) \in \mathcal{L}_k[\![\Gamma]\!]\theta \wedge (\varsigma_x, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]\}$$

$$\begin{aligned}\mathcal{H}[\![\sigma]\!] &= \bigstar_{(l,f,m)\in\sigma}[l \mapsto_f m] \\ \varsigma &\equiv \mathcal{H}[\![\theta(\sigma)]\!]\end{aligned}$$

$$\begin{aligned}{}_k[\![\Theta; \Delta; \Gamma \vdash e : t]\!] = \forall \theta, \delta, \gamma, \sigma.\ & \Theta = \mathrm{dom}(\theta) \wedge (\varsigma, \gamma) \in \mathcal{L}_k[\![\Gamma]\!]\theta \wedge \delta \in \mathcal{I}_k[\![\Delta]\!]\theta \Rightarrow \\ & (\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\![\theta(t)]\!]\end{aligned}$$

C Lemmas

C.1 Moral equivalent of frame-rule

$\forall \sigma_s, \sigma_r, e.\ \varsigma_s \star \varsigma_r$ **defined** $\Rightarrow \forall n.\ \langle \sigma_s, e \rangle \to^n = \langle \sigma_s + \sigma_r, e \rangle \to^n$

SUFFICES: By induction on n, consider only the cases $\langle \sigma_s, e \rangle \to \langle \sigma_f, e_f \rangle$ where $\sigma_s \neq \sigma_f$.

PROOF SKETCH: Only OP_{FREE, MATRIX, SHARE, UNSHARE_EQ, GEMM_MATCH} change the heap: the rest are either parametric in the heap or step to an **err**.

PROVE: $\langle \sigma_s + \sigma_r, e \rangle \to \langle \sigma_f + \sigma_r, e_f \rangle$.

$\langle 1 \rangle 1$. CASE: OP_FREE, $\sigma_s \equiv \sigma' + \{l \mapsto_1 m\}$, $\sigma_f = \sigma'$.
PROOF: Instantiate OP_FREE with $(\sigma' + \sigma_r) + \{l \mapsto_1 m\}$,
valid because $l \notin \mathrm{dom}(\varsigma_r)$ by $\varsigma' \star [l \mapsto_1 m] \star \varsigma_r$ defined (assumption).

$\langle 1 \rangle 2$. CASE: OP_MATRIX
PROOF: Rule has no requirements on σ_s so will also work with $\sigma_s + \sigma_r$.

$\langle 1 \rangle 3$. CASE: OP_SHARE, $\sigma_s \equiv \sigma' + \{l \mapsto_f m\}$, $\sigma_f = \sigma' + \{l \mapsto_{\frac{1}{2} \cdot f} m\} + \{l \mapsto_{\frac{1}{2} \cdot f} m\}$.
PROOF: Union-ing σ_r does not remove $l \mapsto_f m$, so that can be split out of $\sigma_s + \sigma_r$ as before.

$\langle 1 \rangle 4$. CASE: OP_UNSHARE_EQ, $\sigma_s \equiv \sigma' + \{l \mapsto_{\frac{1}{2} \cdot f} m\} + \{l \mapsto_{\frac{1}{2} \cdot f} m\}$, $\sigma_f = \sigma' + \{l \mapsto_f m\}$.

$\langle 2 \rangle 1$. Union-ing σ_r does not remove $l \mapsto_{\frac{1}{2} \cdot f} m$, so that can still be split out of $\sigma_s + \sigma_r$.

$\langle 2 \rangle 2$. There may also be other valid splits introduced by σ_r.

$\langle 2 \rangle 3$. However, by assumption of $\varsigma_s \star \varsigma_r$ defined, any splitting of $\sigma_s + \sigma_r$ will satisfy $f \leq 1$.

$\langle 1 \rangle 5$. CASE: OP_GEMM_MATCH

$\langle 2 \rangle 1$. By assumption of $\varsigma_s \star \varsigma_r$ defined, either l_1 (or l_2, or both) are not in σ_r, or they are and the matrix values they point to are the same.

$\langle 2 \rangle 2$. The permissions (of l_1 and/or l_2) may differ, but OP_GEMM_MATCH universally quantifies over them and leaves them unchanged, so they are irrelevant.

$\langle 2 \rangle 3$. Only the pointed to matrix value at l_3 changes.

$\langle 2 \rangle 4$. SUFFICES: $l_3 \notin \pi_1[\sigma_r]$.

$\langle 2 \rangle 5$. By assumption of $\varsigma_s \star \varsigma_r$ defined, $l_3 \notin \mathrm{dom}(\varsigma_r)$.

$\langle 2 \rangle 6$. Hence $l_3 \notin \pi_1[\sigma_r]$.

C.2 Semantically, values are expressions $\forall k, t.\ \mathcal{V}_k[\![t]\!] \subseteq \mathcal{C}_k[\![t]\!]$

Follows from definition of $\mathcal{C}_k[\![t]\!]$, $\rightarrow^j$ ($\forall n.\ \langle\sigma, v\rangle \rightarrow^n \langle\sigma, v\rangle$) for arbitrary $j \leq k$ and C.1.

C.3 Values remain values under all substitutions $\forall\theta, \delta, \gamma, v.\ \theta(\delta(\gamma(v)))$ is a value.

θ is irrelevant because it only maps fractional permission variables to fractional permissions. By construction, δ and γ only map variables to values, and values are closed under substitution.

C.4 Stepping reduces the step-index $\forall k, \sigma, \sigma', e, e', t.\ (\varsigma', e') \in \mathcal{C}_k[\![t]\!] \wedge \langle\sigma, e\rangle \rightarrow \langle\sigma', e'\rangle \Rightarrow (\varsigma, e) \in \mathcal{C}_{k+1}[\![t]\!]$

In the lemma, and for the rest of its proof, $\varsigma = \mathcal{H}[\![\sigma]\!]$.

Assume: arbitrary $j < k+1$, and σ_r such that $\varsigma \star \varsigma_r$ defined.

$\langle 1\rangle 1$. Case: $j = 0$. Clearly $\sigma_f = \sigma_s + \sigma_r$ and $e' = e$.
Remains to show that if e is a value then $(\varsigma_s \star \varsigma_r, e) \in \mathcal{V}_k[\![t]\!]$.
This is true vacuously, because by assumption, e is not a value.

$\langle 1\rangle 2$. Case: $j \geq 1$. We have $\langle\sigma, e\rangle \rightarrow^j = \langle\sigma', e'\rangle \rightarrow^{j-1}$.
Instantiate $(\varsigma', e') \in \mathcal{C}_k[\![t]\!]$, with $j - 1 < k$ and σ_r to conclude the required conditions.

C.5 Monotonicity for step-index $j \leq k \Rightarrow \_\ _k[\![\cdot]\!] \subseteq \_\ _j[\![\cdot]\!]$

For the rest of this proof, $\varsigma = \mathcal{H}[\![\sigma]\!]$.
Lemma C.4 is the inductive step for this lemma for the $\mathcal{C}[\![]\!]$ case.
Need to prove for $\mathcal{V}[\![]\!]$, by induction on t and then index.

Suffices: Consider only $t \multimap t'$ case, rest use k directly on structure of type.
Assume: Arbitrary $j \leq k$ and $(\varsigma_{v'}, v') \in \mathcal{V}_k[\![t \multimap t']\!]$.
Prove: $(\varsigma_{v'}, v') \in \mathcal{V}_j[\![t \multimap t']\!]$.

$\langle 1\rangle 1$. v' is of the correct syntactic form (lambda or fixpoint) by assumption.

$\langle 1\rangle 2$. Assume: arbitrary $j' \leq j$ and $(\varsigma_v, v) \in \mathcal{V}_{j'}[\![t]\!]$ such that $\varsigma_{v'} \star \varsigma_v$ is defined.

$\langle 1\rangle 3$. Suffices: to show $(\varsigma_{v'} \star \varsigma_v, v'v) \in \mathcal{C}_{j'}[\![t']\!]$.

$\langle 1\rangle 4$. This is true by instantiating $(\varsigma_{v'}, v') \in \mathcal{V}_k[\![t \multimap t']\!]$ with $j' \leq k$ and $(\varsigma_v, v) \in \mathcal{V}_{j'}[\![t]\!]$.

C.6 Domains match $\forall\Delta, \Gamma, t, k, \theta, \delta, \gamma.\ \delta \in \mathcal{I}_k[\![\Delta]\!]\theta \wedge \gamma \in \pi_2[\mathcal{L}_k[\![\Gamma]\!]\theta] \Rightarrow \mathrm{dom}(\Delta) = \mathrm{dom}(\delta)$ and $\mathrm{dom}(\Gamma) = \mathrm{dom}(\gamma)$

Proof: By induction on Δ and Γ.

C.7 Splitting up linear environments corresponds to splitting up heaps

$\forall k, \Gamma, \Gamma', \theta, \sigma_+, \gamma_+.\ (\varsigma_+, \gamma) \in \mathcal{L}_k[\![\Gamma, \Gamma']\!]\theta \wedge \Gamma, \Gamma'$ **disjoint** $\Rightarrow$ $\exists \sigma, \gamma, \sigma', \gamma'.\ \sigma_+ = \sigma + \sigma' \wedge \gamma, \gamma'$ **disjoint** $\wedge \gamma_+ = \gamma \cup \gamma' \wedge (\varsigma, \gamma) \in \mathcal{L}_k[\![\Gamma]\!] \wedge (\varsigma', \gamma') \in \mathcal{L}_k[\![\Gamma']\!]$

PROOF: By induction on Γ'.

C.8 Fractional permission substitutions preserve progress

$\forall e, \sigma, e', \sigma', \theta.\ \langle \sigma, e \rangle \to \langle \sigma', e' \rangle \Rightarrow \langle \theta(\sigma), \theta(e) \rangle \to \langle \theta(\sigma'), \theta(e') \rangle$

PROOF: By induction on $\to$.

$\langle 1 \rangle 1$. ASSUME: Arbitrary $e, \sigma, e', \sigma', \theta$ such that $\langle \sigma, e \rangle \to \langle \sigma', e' \rangle$.

$\langle 1 \rangle 2$. SUFFICES: To consider only the following rules which mention fractional permission variables: OP_FRAC_PERM, OP_SHARE, OP_UNSHARE_(N)EQ and OP_GEMM_(MIS)MATCH.

$\langle 1 \rangle 3$. CASE: OP_FRAC_PERM.
Because substitution avoids capture,
$\langle \theta(\sigma), \theta((\mathbf{fun}\,'\mathit{fc} \to v)\,[f]) \rangle \to \langle \theta(\sigma'\,[f/\mathit{fc}]), \theta(v\,[f/\mathit{fc}]) \rangle$.

$\langle 1 \rangle 4$. The rest of the cases are parametric in their use of fractional permission variables and so will take the same step ater any substitution.

$\langle 1 \rangle 5$. COROLLARY: If $\langle \sigma\,[f_1/\mathit{fc}], e\,[f_1/\mathit{fc}] \rangle \to^n \langle \sigma_2, e_2' \rangle$ and $\langle \sigma\,[f_2/\mathit{fc}], e\,[f_2/\mathit{fc}] \rangle \to^n \langle \sigma_2, e_2' \rangle$, then $\exists\,\sigma, e'.\ \sigma_1 = \sigma\,[f_1/\mathit{fc}] \wedge \sigma_2 = \sigma\,[f_2/\mathit{fc}] \wedge e_1' = e'\,[f_1/\mathit{fc}] \wedge e_2' = e'\,[f_2/\mathit{fc}]$.

D Soundness

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

D.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 θ,
  - linear variables with a suitable heap (γ, ς) and
  - intuitionistic variables δ,
- the aforementioned suitable heap and expression $(\varsigma, \theta(\delta(\gamma(e))))$
- are in the computational interpretation $\mathcal{C}_k[\![\theta(t)]\!]$ 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.

D.2 Proof

PROOF SKETCH: Induction over the typing judgements.

ASSUME: 1. Arbitrary $\Theta, \Delta, \Gamma, e, t$ such that $\Theta; \Delta; \Gamma \vdash e : t$.
2. Arbitrary $k, \theta, \delta, \gamma, \sigma$ such that:
a. $\Theta = \mathrm{dom}(\theta)$
b. $\delta \in \mathcal{I}_k[\![\Delta]\!]\theta$.
c. $(\varsigma, \gamma) \in \mathcal{L}_k[\![\Gamma]\!]\theta$
3. W.l.o.g., all variables are distinct, hence Θ, $\mathrm{dom}(\Delta)$ and $\mathrm{dom}(\Gamma)$ are disjoint so order of θ, δ and γ (as substitutions defined recursively over expressions) is irrelevant.

PROVE: $(\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\![\theta(t)]\!]$.
ASSUME: Arbitrary $j < k$ and σ_r, such that $\varsigma \star \varsigma_r$ defined.
SUFFICES: $\langle \sigma + \sigma_r, e \rangle \to^j \mathbf{err} \ \vee \exists \sigma_f, e_f.\ \langle \sigma + \sigma_r, e \rangle \to^j \langle \sigma_f + \sigma_r, e_f \rangle$
$\wedge$ (e_f is a value $\Rightarrow (\varsigma_f \star \varsigma_r, e_f) \in \mathcal{V}_{k-j}[\![t]\!]$).
SUFFICES: By C.1, to show $\langle \sigma, e \rangle \to^j \mathbf{err} \ \vee \exists \sigma_f, e_f.\ \langle \sigma, e \rangle \to^j \langle \sigma_f, e_f \rangle$
$\wedge$ (e_f is a value $\Rightarrow (\varsigma_f, e_f) \in \mathcal{V}_{k-j}[\![t]\!]$)

⟨1⟩1. CASE: TY_LET.

⟨2⟩1. By induction,
1. $\forall k.\ {}_k[\![\Theta; \Delta; \Gamma \vdash e : t]\!]$
2. $\forall k.\ {}_k[\![\Theta; \Delta; \Gamma', x : t \vdash e' : t']\!]$.

⟨2⟩2. By 2c, 3 and C.7, we know there exists the following (for all k):
1. $(\varsigma_e, \gamma_e) \in \mathcal{L}_k[\![\Gamma]\!]$
2. $\gamma = \gamma_e \cup \gamma_{e'}$
3. $\sigma = \sigma_e + \sigma_{e'}$.

⟨2⟩3. So, using $k, \theta, \delta, \gamma_e, \sigma_e$, we have $(\varsigma_e, \theta(\delta(\gamma_e(e)))) \in \mathcal{C}_k[\![\theta(t)]\!]$.

⟨2⟩4. By ⟨2⟩2 ($\gamma = \gamma_e \cup \gamma_{e'}$), have $(\varsigma_e, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\![\theta(t)]\!]$.

⟨2⟩5. By definition of $\mathcal{C}_k[\![\cdot]\!]$ and ⟨2⟩2, we instantiate with j and $\sigma_r = \sigma_{e'}$ to conclude that
$\langle \theta(\sigma), \theta(\delta(\gamma(e))) \rangle$ either takes j steps to **err** or another heap-and-expression $\langle \sigma_f, e_f \rangle$.

⟨2⟩6. CASE: j steps to **err**
By OP_CONTEXT_ERR, the whole expression reduces to **err** in $j < k$ steps.

⟨2⟩7. CASE: j steps to another heap-and-expression.
If it is not a value, then OP_CONTEXT runs j times and we are done.

⟨2⟩8. If it is, then $\exists i \leq j.\ (\varsigma_f, v_1) \in \mathcal{V}_{k-i}[\![\theta(t_1)]\!] \subseteq \mathcal{V}_{k-j}[\![\theta(t_1)]\!]$ by C.3 and C.5.
So, OP_CONTEXT runs i times, and then we have the following.
SUFFICES: $(\varsigma_f \star \varsigma_{e'}, \mathbf{let}\, x = v\, \mathbf{in}\, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i}[\![\theta(t')]\!]$ by C.4 i times.
SUFFICES: $(\varsigma_f \star \varsigma_{e'}, \theta(\delta(\gamma(e')))[v/x]) \in \mathcal{C}_{k-i-1}[\![\theta(t')]\!]$ by C.4.

⟨2⟩9. By C.5, $(\varsigma_{e'}, \gamma_{e'}[x \mapsto v]) \in \mathcal{L}_k[\![\Gamma', x : t]\!]\theta \subseteq \mathcal{L}_{k-i-1}[\![\Gamma', x : t]\!]\theta$.

⟨2⟩10. Instantiate 2 of step ⟨2⟩1 with $k - i - 1, \theta, \delta, \gamma_{e'}[x \mapsto v], \sigma_{e'}$ to conclude $(\varsigma_{e'}, \theta(\delta(\gamma_{e'}[x \mapsto v](e')))) \in \mathcal{C}_{k-i-1}[\![\theta(t')]\!]$.

⟨2⟩11. By 3, we have $\theta(\delta(\gamma(e')))[v/x] = \theta(\delta(\gamma_{e'}[x \mapsto v](e')))$ and by C.1 we conclude $(\varsigma_f \star \varsigma_{e'}, \theta(\delta(\gamma(e')))[v/x]) \in \mathcal{C}_{k-i-1}[\![\theta(t')]\!]$

⟨1⟩2. Case: Ty_Pair_Elim.
Proof sketch: Similar to Ty_Let, but with the following key differences.

⟨2⟩1. When $(\varsigma_f, v) \in \mathcal{V}_{k-i}[\![\theta(t_1) \otimes \theta(t_2)]\!]$, we have $v = (v_1, v_2)$.

⟨2⟩2. Suffices: $(\varsigma_{e'}, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i-1}[\![\theta(t')]\!]$ by C.4 $i + 1$ times.

⟨2⟩3. By C.5, $(\varsigma_{e'}, \gamma_{e'}[a \mapsto v_1, b \mapsto v_2]) \in \mathcal{L}_k[\![\Gamma', a : t_1, b : t_2]\!]\theta \subseteq \mathcal{L}_{k-i-1}[\![\Gamma', a : t_1, b : t_2]\!]\theta$.

⟨2⟩4. Instantiate ${}_{k-i-1}[\![\Theta; \Delta; \Gamma', a : t_1, b : t_2 \vdash e' : t']\!]$ with $\theta, \delta, \gamma_{e'}[a \mapsto v_1, b \mapsto v_2], \sigma_{e'}$.

⟨2⟩5. By 3 (for $\gamma = \gamma_e \cup \gamma_{e'}$ and a, b), conclude $(\varsigma_{e'}, \theta(\delta(\gamma(e'[v_1/a][v_2/b])))) \in \mathcal{C}_{k-i-1}[\![\theta(t')]\!]$.

⟨1⟩3. Case: Ty_Bang_Elim.
Proof sketch: Similar to Ty_Let, but with the following key differences.

⟨2⟩1. When $(\varsigma_f, v) \in \mathcal{V}_{k-i}[\![\theta(!t)]\!]$, since $\mathcal{V}_{k-i}[\![\theta(!t)]\!] = \mathcal{V}_{k-i}[\![!\theta(t)]\!]$, we have $\varsigma_f = \emptyset$ and $v = \textbf{Many}\, v'$ for some $(\emptyset, v') \in \mathcal{V}_{k-i}[\![\theta(t)]\!]$.

⟨2⟩2. Suffices: $(\varsigma_{e'}, \textbf{let Many}\, x \ = \ \textbf{Many}\, v' \,\textbf{in}\, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i}[\![\theta(t)]\!]$.

⟨2⟩3. Suffices: $(\varsigma_{e'}, \theta(\delta(\gamma(e')))[v/x]) \in \mathcal{C}_{k-i-1}[\![\theta(t)]\!]$ by C.4 $i + 1$ times.

⟨2⟩4. Instantiate ${}_{k-i-1}[\![\Theta; \Delta, x : t, \Gamma' \vdash e' : t']\!]$ with $\theta, \delta_{e'} = \delta[x \mapsto v'], \gamma_{e'}, \sigma_{e'}$.

⟨2⟩5. By 3, $(\varsigma_{e'}, \theta(\delta(\gamma(e')))[v/x]) \in \mathcal{C}_{k-i-1}[\![\theta(t)]\!]$.

⟨1⟩4. Case: Ty_Unit_Elim.
Proof sketch: Similar to Ty_Let, but with the following key differences.

⟨2⟩1. When $(\varsigma_f, v) \in \mathcal{V}_{k-i}[\![\textbf{unit}]\!]$, we have $\varsigma_f = \emptyset$ and $v = ()$.

⟨2⟩2. Suffices: $(\varsigma_{e'}, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i-1}[\![\theta(t')]\!]$ by C.4 $i + 1$ times.

⟨2⟩3. By C.5, $(\varsigma_{e'}, \gamma_{e'}) \in \mathcal{L}_k[\![\Gamma']\!]\theta \subseteq \mathcal{L}_{k-i-1}[\![\Gamma']\!]\theta$.

⟨2⟩4. Instantiate ${}_{k-i-1}[\![\Theta; \Delta; \Gamma' \vdash e' : t']\!]$ with $\theta, \delta, \gamma_{e'}, \sigma_{e'}$.

⟨2⟩5. By 3 $(\varsigma_{e'}, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-i-1}[\![\theta(t')]\!]$.

⟨1⟩5. Case: Ty_Bool_Elim.
Proof sketch: Similar to Ty_Unit_Elim but with Op_If_{True,False}, $\varsigma_f = \emptyset$ and $v = \textbf{Many true}$ or $v = \textbf{Many false}$.

$\langle 1\rangle 6$. CASE: TY_BANG_INTRO.

$\langle 2\rangle 1$. We have, $e = v$ for some value $v \neq l$, $\Gamma = \emptyset$ and so
$\forall k.\ {}_k[\![\Theta; \Delta; \cdot \vdash v : t]\!]$ by induction.

$\langle 2\rangle 2$. SUFFICES: $(\emptyset, \mathbf{Many}\, \theta(\delta(v))) \in \mathcal{C}_k[\![!\theta(t)]\!]$ by 2c ($\varsigma = \emptyset, \gamma = []$).

$\langle 2\rangle 3$. Instantiate ${}_k[\![\Theta; \Delta; \cdot \vdash v : t]\!]$ with $\theta, \delta, \gamma = [], \sigma = \emptyset$ to obtain $(\emptyset, \theta(\delta(v))) \in \mathcal{C}_k[\![\theta(t)]\!]$.

$\langle 2\rangle 4$. Instantiate $(\emptyset, \theta(\delta(v))) \in \mathcal{C}_k[\![\theta(t)]\!]$ with $j = 0$, $\sigma_r = \emptyset$ and C.3 ($\theta(\delta(v))$ is a value), to conclude $(\emptyset, \theta(\delta(v))) \in \mathcal{V}_k[\![\theta(t)]\!]$.

$\langle 2\rangle 5$. By definition of $\mathcal{V}_k[\![!\theta(t)]\!]$, C.3 and C.2 we have $(\emptyset, \mathbf{Many}\, \theta(\delta(v))) \in \mathcal{C}_k[\![!\theta(t)]\!]$.

$\langle 1\rangle 7$. CASE: TY_PAIR_INTRO.

$\langle 2\rangle 1$. By 2c, 3 and C.7, we know there exists the following (for all k):
1. $(\varsigma_1, \gamma_1) \in \mathcal{L}_k[\![\Gamma_1]\!]$
2. $(\varsigma_2, \gamma_2) \in \mathcal{L}_k[\![\Gamma_2]\!]$
3. $\gamma = \gamma_1 \cup \gamma_2$
4. $\sigma = \sigma_1 + \sigma_2$.

$\langle 2\rangle 2$. By induction,
1. $\forall k.\ {}_k[\![\Theta; \Delta; \Gamma_1 \vdash e_1 : t_1]\!]$
2. $\forall k.\ {}_k[\![\Theta; \Delta; \Gamma_2 \vdash e_2 : t_2]\!]$.

$\langle 2\rangle 3$. Instantiate the first with $k, \theta, \delta, \gamma_1, \sigma_1$.

$\langle 2\rangle 4$. By that and $\langle 2\rangle 1$, $(\varsigma_1, \theta(\delta(\gamma_1(e_1)))) = (\varsigma_1, \theta(\delta(\gamma(e_1)))) \in \mathcal{C}_k[\![\theta(t)]\!]$.

$\langle 2\rangle 5$. So, $\langle \theta(\sigma_1 + \sigma_2), \theta(\delta(\gamma_1(e_1)))\rangle$ either takes j steps to **err** or a heap-and-expression $\langle \sigma_{1f}, e_{1f}\rangle$.

$\langle 2\rangle 6$. CASE: j steps to **err**
By OP_CONTEXT_ERR, the whole expression reduces to **err** in $j < k$ steps.

$\langle 2\rangle 7$. CASE: j steps to another heap-and-expression.
If it is not a value, then OP_CONTEXT runs j times and we are done.

$\langle 2\rangle 8$. If it is, then $\exists i_1 \leq j.\ (\varsigma_{1f}, v_1) \in \mathcal{V}_{k-i_1}[\![\theta(t_1)]\!] \subseteq \mathcal{V}_{k-j}[\![\theta(t_1)]\!]$ by C.3 and C.5.
So, OP_CONTEXT runs i_1 times, and then we have the following.
SUFFICES: By C.4, $(\varsigma_{1f} \star \varsigma_2,\ (v_1, e_2)\,) \in \mathcal{C}_{k-i_1}[\![\theta(t_1 \otimes t_2)]\!]$.

$\langle 2\rangle 9$. Instantiate the second IH with $k, \theta, \delta, \gamma_2, \sigma_2$.

$\langle 2\rangle 10$. So, $\langle \theta(\sigma_{1f} + \sigma_2), \theta(\delta(\gamma_2(e_2)))\rangle$ either takes j steps to **err** or a heap-and-expression $\langle \sigma_{2f}, e_{2f}\rangle$.

$\langle 2\rangle 11$. CASE: j steps to **err**
By OP_CONTEXT_ERR, the whole expression reduces to **err** in $j < k$ steps.

$\langle 2\rangle 12.$ CASE: j steps to another heap-and-expression.
If it is not a value, then OP_CONTEXT runs j times and we are done.

$\langle 2\rangle 13.$ If it is, then $\exists i_2 \leq j.\ (\varsigma_{2f}, v_2) \in \mathcal{V}_{k-i_2}[\![\theta(t_2)]\!] \subseteq \mathcal{V}_{k-j}[\![\theta(t_2)]\!]$ by C.3 and C.5.
So, OP_CONTEXT runs i_2 times, and then we have the following.
SUFFICES: By C.4, $(\varsigma_{1f} \star \varsigma_{2f},\ (v_1, v_2)\,) \in \mathcal{V}_{k-i_1-i_2}[\![\theta(t_1) \otimes \theta(t_2)]\!]$.

$\langle 2\rangle 14.$ By C.5 and $k - i_1 - i_2 \leq k - i_1, k - i_2$, have
$(\varsigma_{1f}, v_1) \in \mathcal{V}_{k-i_1}[\![\theta(t_1)]\!] \subseteq \mathcal{V}_{k-i_1-i_2}[\![\theta(t_1)]\!]$ and
$(\varsigma_{2f}, v_2) \in \mathcal{V}_{k-i_2}[\![\theta(t_2)]\!] \subseteq \mathcal{V}_{k-i_1-i_2}[\![\theta(t_2)]\!]$ as needed.

$\langle 1\rangle 8.$ CASE: TY_LAMBDA.
SUFFICES: By C.2, to show $(\,\varsigma, \theta(\delta(\gamma(\mathbf{fun}\, x : t \to e)))\,) \in \mathcal{V}_k[\![\theta(t \multimap t')]\!]$.
ASSUME: Arbitrary $j \leq k$, $(\varsigma_v, v) \in \mathcal{V}_j[\![\theta(t)]\!]$ such that $\varsigma \star \varsigma_v$ is defined.
SUFFICES: $(\varsigma \star \varsigma_v, \theta(\delta(\gamma(\mathbf{fun}\, x : t \to e)))\, v) \in \mathcal{C}_j[\![\theta(t')]\!]$.
SUFFICES: $(\varsigma \star \varsigma_v, \theta(\delta(\gamma(e)))[v/x]) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$ by C.4.

$\langle 2\rangle 1.$ By induction, $\forall k.\ {}_k[\![\Theta; \Delta; \Gamma, x : t \vdash e]\!]$.

$\langle 2\rangle 2.$ Instantiate it $j - 1, \theta, \delta, \gamma[x \mapsto v], \sigma + \sigma_v$.

$\langle 2\rangle 3.$ Hence, $(\,\varsigma \star \varsigma_v, \theta(\delta(\gamma[x \mapsto v](e)))\,) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$.

$\langle 2\rangle 4.$ By 3, $\theta(\delta(\gamma[x \mapsto v](e))) = \theta(\delta(\gamma(e)))[v/x]$, we are done.

$\langle 1\rangle 9.$ CASE: TY_APP.

$\langle 2\rangle 1.$ By 2c, 3 and C.7, we know there exists the following (for all k):
1. $(\varsigma_e, \gamma_e) \in \mathcal{L}_k[\![\Gamma_e]\!]$
2. $(\varsigma_{e'}, \gamma_{e'}) \in \mathcal{L}_k[\![\Gamma_{e'}]\!]$
3. $\gamma = \gamma_e \cup \gamma_{e'}$
4. $\sigma = \sigma_e + \sigma_{e'}$.

$\langle 2\rangle 2.$ By induction,
1. $\forall k.\ {}_k[\![\Theta; \Delta; \Gamma \vdash e : t' \multimap t]\!]$
2. $\forall k.\ {}_k[\![\Theta; \Delta; \Gamma' \vdash e' : t']\!]$.

$\langle 2\rangle 3.$ Instantiate the first with $k, \theta, \delta, \gamma_e, \sigma_e$ to conclude $(\varsigma_e, \theta(\delta(\gamma_e(e)))) \in \mathcal{C}_k[\![\theta(t') \multimap \theta(t)]\!]$.

$\langle 2\rangle 4.$ Instantiate *this* with j and $\sigma_{e'}$ and use $\langle 2\rangle 1$ to conclude $\langle \theta(\sigma_e + \sigma_{e'}), \theta(\delta(\gamma(e)))\rangle$ either takes j steps to **err** or a heap-and-expression $\langle \sigma_f + \sigma_{e'}, e_f\rangle$.

$\langle 2\rangle 5.$ CASE: j steps to **err**
By OP_CONTEXT_ERR, the whole expression reduces to **err** in $j < k$ steps.

$\langle 2\rangle 6.$ CASE: j steps to another heap-and-expression.
If it is not a value, then OP_CONTEXT runs j times and we are done.

$\langle 2\rangle 7.$ If it is, then $\exists i_e \leq j.\ (\varsigma_f, e_f) \in \mathcal{V}_{k-i_e}[\![\theta(t') \multimap \theta(t)]\!] \subseteq \mathcal{V}_{k-j}[\![\ldots]\!]$ by C.3 and C.5.
So, OP_CONTEXT runs i_e times, and then we have the following.
SUFFICES: By C.4 i_e times, $(\varsigma_f \star \varsigma_{e'},\ e_f\, e'\,) \in \mathcal{C}_{k-i_e}[\![\theta(t')]\!]$.

$\langle 2 \rangle 8$. By C.5, $(\varsigma_{e'}, \gamma_{e'} \in \mathcal{L}_k[\![\Gamma']\!]\theta \subseteq \mathcal{L}_{k-i_e}[\![\Gamma']\!]\theta$.

$\langle 2 \rangle 9$. So, instantiate the second IH with $k - i_e, \theta, \delta, \gamma_{e'}, \sigma_{e'}$ to conclude $(\varsigma_{e'}, \theta(\delta(\gamma_{e'}(e')))) \in \mathcal{C}_{k-i_e}[\![\theta(t')]\!]$.

$\langle 2 \rangle 10$. Instantiate *this* with $j - i_e$ and σ_f to conclude $\langle \theta(\sigma_f + \sigma_{e'}), \theta(\delta(\gamma_{e'}(e'))) \rangle$ either takes $j - i_e$ steps to **err** or $\langle \sigma_f + \sigma'_f, e'_f \rangle$.

$\langle 2 \rangle 11$. Case: $j - i_e$ steps to **err**
By Op_Context_Err, the whole expression reduces to **err** in $j - i_e < k - i_e$ steps.

$\langle 2 \rangle 12$. Case: $j - i_e$ steps to another heap-and-expression.
If it is not a value, then Op_Context runs $j - i_e$ times and we are done.

$\langle 2 \rangle 13$. If it is, then $\exists i_{e'} \leq j - i_e.\ (\varsigma'_f, v_{e'}) \in \mathcal{V}_{k-i_e-i'_e}[\![\theta(t')]\!]$ by C.3.
So, Op_Context runs $i_{e'}$ times, and then we have the following.
Suffices: By C.4 $i_{e'}$ times, $(\varsigma_f \star \varsigma'_f,\ e_f\, e'_f\,) \in \mathcal{C}_{k-i_e-i_{e'}}[\![\theta(t')]\!]$.

$\langle 2 \rangle 14$. Instantiate $(\varsigma_f, e_f) \in \mathcal{V}_{k-i_e}[\![\theta(t') \multimap \theta(t)]\!]$ with $k - i_e - i_{e'} \leq k - i_e$ and $(\varsigma_{v'}, v_{e'}) \in \mathcal{V}_{k-i_e-i_{e'}}[\![\theta(t')]\!]$, to conclude $(\varsigma_f \star \varsigma'_f, e_f\, e'_f) \in \mathcal{C}_{k-i_e-i_{e'}}[\![\theta(t)]\!]$ as needed.

$\langle 1 \rangle 10$. Case: Ty_Gen.

$\langle 2 \rangle 1$. By induction, $\forall k.\ {}_k[\![\Theta, fc; \Delta; \Gamma \vdash e : t]\!]$.

$\langle 2 \rangle 2$. Let: f be arbitrary; $\theta' \equiv \theta[fc \mapsto f]$.
Instantiate induction hypothesis with $k - 1, \theta', \delta, \gamma, \sigma$,
to conclude $(\varsigma, \theta'(\gamma(\delta(e)))) \in \mathcal{C}_{k-1}[\![\theta'(t)]\!]$ (for all f, by C.8).

$\langle 2 \rangle 3$. Instantiate *this* with j and $\emptyset$ to conclude $\langle \theta'(\sigma), \theta'(\gamma(\delta(e))) \rangle$ either takes j steps to **err** or a heap-and-expression $\langle \sigma', e' \rangle$ (for all f, by C.8).

$\langle 2 \rangle 4$. Case: j steps to **err**.
By Op_Context_Err, whole expression reduces to **err** in $j < k - 1$ steps (for $f = fc$).

$\langle 2 \rangle 5$. Case: j steps to another heap-and-expression.
If it is not a value, then for $f = fc$, Op_Context runs j times and we are done.

$\langle 2 \rangle 6$. If it is, then $\exists i_e \leq j.\ (\varsigma', e') \in \mathcal{V}_{k-1-i_e}[\![\theta'(t)]\!] \subseteq \mathcal{V}_{k-1-j}[\![\ldots]\!]$ by C.3 and C.5 (for all f, by C.8).

$\langle 2 \rangle 7$. So, Op_Context runs i_e times, and then we have the following.
Suffices: By C.4 i_e times, $(\varsigma', \mathbf{fun}\, {'fc} \to e') \in \mathcal{V}_{k-i_e}[\![\theta({'fc}.\ t)]\!]$ (for $f = fc$).

$\langle 2 \rangle 8$. Assume: Arbitrary f'.
Suffices: $(\varsigma', e'[f'/fc]) \in \mathcal{V}_{k-1-i_e}[\![\theta(t)[f'/fc]]\!]$ (for $f = fc$).

$\langle 2 \rangle 9$. This is true by instantiating $\langle 2 \rangle 6$ with $f = f'$.

⟨1⟩11. CASE: TY_SPC.

⟨2⟩1. By induction, $\forall k.\ {}_k[\![\Theta; \Delta; \Gamma \vdash e : {}'\!fc.\ t]\!]$.

⟨2⟩2. Instantiate with $k, \theta, \delta, \gamma, \sigma$ to conclude $(\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\![\theta({}'\!fc.\ t]\!])$.

⟨2⟩3. Instantiate *this* with j and $\emptyset$ and to conclude $\langle\theta(\sigma), \theta(\delta(\gamma(e)))\rangle$ either takes j steps to **err** or a heap-and-expression $\langle\sigma_f, e_f\rangle$.

⟨2⟩4. CASE: j steps to **err**.
By OP_CONTEXT_ERR, the whole expression reduces to **err** in $j < k$ steps.

⟨2⟩5. CASE: j steps to another heap-and-expression.
If it is not a value, then OP_CONTEXT runs j times and we are done.

⟨2⟩6. If it is, then $\exists i_e \leq j.\ (\varsigma_f, e_f) \in \mathcal{V}_{k-i_e}[\![\theta({}'\!fc.t)]\!] \subseteq \mathcal{V}_{k-j}[\![\ldots]\!]$ by C.3 and C.5.
So $e_f \equiv \mathbf{fun}\,{}'\!fc \to v$ for some v.

⟨2⟩7. So, OP_CONTEXT runs i_e times, and then we have the following.
SUFFICES: By C.4 i_e times, $(\varsigma_f, (\mathbf{fun}\,{}'\!fc \to v)\,[f]) \in \mathcal{C}_{k-i_e}[\![\theta(t[f/fc])]\!]$.
SUFFICES: By C.4 once more, $(\varsigma_f, v[f/fc]) \in \mathcal{C}_{k-i_e-1}[\![\theta(t[f/fc])]\!]$.

⟨2⟩8. This is true by instantiating ⟨2⟩6 with f and C.2.

⟨1⟩12. CASE: TY_FIX.
SUFFICES: $(\emptyset, \theta(\delta(\mathbf{fix}(g, x : t, e : t')))) \in \mathcal{V}_k[\![\theta(t \multimap t')]\!]$, by C.2 ($\sigma = \{\}$, $\gamma = []$).
ASSUME: Arbitrary $j \leq k$, $(\varsigma_v, v) \in \mathcal{V}_j[\![\theta(t)]\!]$ ($\varsigma = \emptyset$, so $\varsigma \star \varsigma_v$ is defined).
LET: $\tilde{e} \equiv \theta(\delta(e)))$.
SUFFICES: $(\varsigma_v, \mathbf{fix}(g, x : t, \tilde{e} : t')\ v) \in \mathcal{C}_j[\![\theta(t')]\!]$.
SUFFICES: $(\varsigma_v, \tilde{e}\,[v/x]\,[\mathbf{fix}(g, x : t, \tilde{e} : t')/g]) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$ by C.4.

⟨2⟩1. By induction, $\forall k.\ {}_k[\![\Theta; \Delta, g : t \multimap t'; x : t \vdash e : t']\!]$.

⟨2⟩2. Instantiate this with $j-1, \delta[g \mapsto \mathbf{fix}(g, x : t, \tilde{e} : t')], \gamma = [x \mapsto v], \sigma_v$.

⟨2⟩3. We have $(\emptyset, \mathbf{fix}(g, x : t, \tilde{e} : t')) \in \mathcal{V}_{j-1}[\![\theta(t \multimap t')]\!]$.

⟨3⟩1. Again by induction (over k), $(\emptyset, \mathbf{fix}(g, x : t, \tilde{e} : t')) \in \mathcal{C}_{j-1}[\![\theta(t \multimap t')]\!]$.

⟨3⟩2. Instantiate *this* with $j = 0$ and $\emptyset$ and we are done.

⟨2⟩4. We have $(\varsigma_v, v) \in \mathcal{V}_{j-1}[\![\theta(t)]\!]$ by assumption and C.5.

⟨2⟩5. So we conclude $(\varsigma_v, \theta(\delta'(\gamma(e)))) \in \mathcal{C}_{j-1}[\![\theta(t')]\!]$ as required.

⟨1⟩13. CASE: TY_VAR_LIN.
PROVE: $(\varsigma, \theta(\delta(\gamma(x)))) \in \mathcal{C}_k[\![\theta(t)]\!]$.

⟨2⟩1. $\Gamma = \{x : t\}$ by assumption of TY_VAR_LIN.

⟨2⟩2. SUFFICES: $(\varsigma, \gamma(x)) \in \mathcal{C}_k[\![\theta(t)]\!]$ by 3 (θ and δ irrelevant).

⟨2⟩3. By 2c, there exist $(\varsigma_x, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]$, such that $\varsigma = \varsigma_x$ and $\gamma = [x \mapsto v_x]$.

⟨2⟩4. Hence, $(\varsigma_x, v_x) \in \mathcal{C}_k[\![\theta(t)]\!]$, by C.2.

⟨1⟩14. CASE: TY_VAR.
PROVE: $(\varsigma, \theta(\delta(\gamma(x)))) \in \mathcal{C}_k[\![\theta(t)]\!]$.

⟨2⟩1. $x : t \in \Delta$ and $\Gamma = \emptyset$ by assumption of TY_VAR.

⟨2⟩2. SUFFICES: $(\emptyset, \delta(x)) \in \mathcal{C}_k[\![\theta(t)]\!]$ by 3.

⟨2⟩3. By 2b, there exists v_x such that $(\emptyset, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]$ (θ irrelevant and γ empty).

⟨2⟩4. Hence, $(\emptyset, v_x) \in \mathcal{C}_k[\![\theta(t)]\!]$, by C.2.

⟨1⟩15. CASE: TY_UNIT_INTRO.
True by C.2 and definition of $\mathcal{V}_k[\![\mathbf{unit}]\!]$.

⟨1⟩16. CASE: TY_BOOL_TRUE, TY_BOOL_FALSE, TY_INT_INTRO, TY_ELT_INTRO.
Similar to TY_UNIT_INTRO.

D.3 Well-formed types

$\boxed{\Theta \vdash f\ \mathsf{Perm}}$ Well-formed fractional permissions

$$\frac{fc \in \Theta}{\Theta \vdash fc\ \mathsf{Perm}}\ \text{WF\_PERM\_VAR}$$

$$\frac{}{\Theta \vdash 1\ \mathsf{Perm}}\ \text{WF\_PERM\_ZERO}$$

$$\frac{\Theta \vdash f\ \mathsf{Perm}}{\Theta \vdash \frac{1}{2}f\ \mathsf{Perm}}\ \text{WF\_PERM\_SUCC}$$

$\boxed{\Theta \vdash t\ \mathsf{Type}}$ Well-formed types

$$\frac{}{\Theta \vdash \mathbf{unit}\ \mathsf{Type}}\ \text{WF\_TYPE\_UNIT}$$

$$\frac{}{\Theta \vdash \mathbf{bool}\ \mathsf{Type}}\ \text{WF\_TYPE\_BOOL}$$

$$\frac{}{\Theta \vdash \mathbf{int}\ \mathsf{Type}}\ \text{WF\_TYPE\_INT}$$

$$\frac{}{\Theta \vdash \mathbf{elt}\ \mathsf{Type}}\ \text{WF\_TYPE\_ELT}$$

$$\frac{\Theta \vdash f\ \mathsf{Perm}}{\Theta \vdash f\ \mathbf{arr}\ \mathsf{Type}}\ \text{WF\_TYPE\_ARRAY}$$

$$\frac{\Theta \vdash t\ \mathsf{Type}}{\Theta \vdash !t\ \mathsf{Type}}\ \text{WF\_TYPE\_BANG}$$

$$\frac{\Theta, fc \vdash t\ \mathsf{Type}}{\Theta \vdash {}'fc.t\ \mathsf{Type}}\ \textsc{WF\_Type\_Gen}$$

$$\frac{\Theta \vdash t\ \mathsf{Type} \quad \Theta \vdash t'\ \mathsf{Type}}{\Theta \vdash t \otimes t'\ \mathsf{Type}}\ \textsc{WF\_Type\_Pair}$$

$$\frac{\Theta \vdash t\ \mathsf{Type} \quad \Theta \vdash t'\ \mathsf{Type}}{\Theta \vdash t \multimap t'\ \mathsf{Type}}\ \textsc{WF\_Type\_Lolly}$$

E NumLin Grammar

m	$::=$			matrix expressions
	$\mid$	M		matrix variables
	$\mid$	$m + m'$		matrix addition
	$\mid$	$m\,m'$		matrix multiplication
	$\mid$	(m)	S	
f	$::=$			fractional permission
	$\mid$	fc		variable
	$\mid$	1		whole permission
	$\mid$	$\frac{1}{2}f$		
t	$::=$			linear type
	$\mid$	**unit**		unit
	$\mid$	**bool**		boolean (true/false)
	$\mid$	**int**		63-bit integers
	$\mid$	**elt**		array element
	$\mid$	f **arr**		arrays
	$\mid$	f **mat**		matrices
	$\mid$	$!t$		multiple-use type
	$\mid$	${}'fc.t$	bind fc in t	frac. perm. generalisation
	$\mid$	$t \otimes t'$		pair
	$\mid$	$t \multimap t'$		linear function
	$\mid$	(t)	S	parentheses
p	$::=$			primitive
	$\mid$	**not**		boolean negation
	$\mid$	$(+)$		integer addition
	$\mid$	$(-)$		integer subtraction
	$\mid$	$(*)$		integer multiplication
	$\mid$	$(/)$		integer division
	$\mid$	$(=)$		integer equality
	$\mid$	$(<)$		integer less-than
	$\mid$	$(+.)$		element addition

	|	$(-.)$	element subtraction
	|	$(*.)$	element multiplication
	|	$(/.)$	element division
	|	$(=.)$	element equality
	|	$(<.)$	element less-than
	|	**set**	array index assignment
	|	**get**	array indexing
	|	**share**	share array
	|	**unshare**	unshare array
	|	**free**	free arrary
	|	**array**	Owl: make array
	|	**copy**	Owl: copy array
	|	**sin**	Owl: map sine over array
	|	**hypot**	Owl: $x_i := \sqrt{x_i^2 + y_i^2}$
	|	**asum**	BLAS: $\sum_i \|x_i\|$
	|	**axpy**	BLAS: $x := \alpha x + y$
	|	**dot**	BLAS: $x \cdot y$
	|	**rotmg**	BLAS: see its docs
	|	**scal**	BLAS: $x := \alpha x$
	|	**amax**	BLAS: $\mathrm{argmax}\, i : x_i$
	|	**setM**	matrix index assignment
	|	**getM**	matrix indexing
	|	**shareM**	share matrix
	|	**unshareM**	unshare matrix
	|	**freeM**	free matrix
	|	**matrix**	Owl: make matrix
	|	**copyM**	Owl: copy matrix
	|	**copyM_to**	Owl: copy matrix onto another
	|	**sizeM**	dimension of matrix
	|	**trnsp**	transpose matrix
	|	**gemm**	BLAS: $C := \alpha A^{T?} B^{T?} + \beta C$
	|	**symm**	BLAS: $C := \alpha AB + \beta C$
	|	**posv**	BLAS: Cholesky decomp. and solve
	|	**potrs**	BLAS: solve with given Cholesky
	|	**syrk**	BLAS: $C := \alpha A^{T?} A^{T?} + \beta C$
v	::=		values
	|	p	primitives
	|	x	variable
	|	$()$	unit introduction
	|	**true**	true
	|	**false**	false
	|	k	integer
	|	l	heap location

	|	el		array element
	|	$\mathbf{Many}\ v$		!-introduction
	|	$\mathbf{fun}\ 'fc \to v$		frac. perm. abstraction
	|	(v, v')		pair introduction
	|	$\mathbf{fun}\ x : t \to e$	bind x in e	abstraction
	|	$\mathbf{fix}\ (g, x : t, e : t')$	bind $g \cup x$ in e	fixpoint
	|	(v)	S	parentheses
e	::=			expression
	|	p		primitives
	|	x		variable
	|	$\mathbf{let}\ x = e\ \mathbf{in}\ e'$	bind x in e'	let binding
	|	$()$		unit introduction
	|	$\mathbf{let}\ () = e\ \mathbf{in}\ e'$		unit elimination
	|	$\mathbf{true}$		true
	|	$\mathbf{false}$		false
	|	$\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$		if
	|	k		integer
	|	l		heap location
	|	el		array element
	|	$\mathbf{Many}\ e$		!-introduction
	|	$\mathbf{let}\ \mathbf{Many}\ x = e\ \mathbf{in}\ e'$		!-elimination
	|	$\mathbf{fun}\ 'fc \to e$		frac. perm. abstraction
	|	$e[f]$		frac. perm. specialisation
	|	(e, e')		pair introduction
	|	$\mathbf{let}\ (a, b) = e\ \mathbf{in}\ e'$	bind $a \cup b$ in e'	pair elimination
	|	$\mathbf{fun}\ x : t \to e$	bind x in e	abstraction
	|	$e\ e'$		application
	|	$\mathbf{fix}\ (g, x : t, e : t')$	bind $g \cup x$ in e	fixpoint
	|	(e)	S	parentheses
C	::=			evaluation contexts
	|	$\mathbf{let}\ x = [-]\ \mathbf{in}\ e$	bind x in e	let binding
	|	$\mathbf{let}\ () = [-]\ \mathbf{in}\ e$		unit elimination
	|	$\mathbf{if}\ [-]\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$		if
	|	$\mathbf{Many}\ [-]$		!-introduction
	|	$\mathbf{let}\ \mathbf{Many}\ x = [-]\ \mathbf{in}\ e$		!-elimination
	|	$\mathbf{fun}\ 'fc \to [-]$		frac. perm. abstraction
	|	$[-][f]$		frac. perm. specialisation
	|	$([-], e)$		pair introduction
	|	$(v, [-])$		pair introduction
	|	$\mathbf{let}\ (a, b) = [-]\ \mathbf{in}\ e$	bind $a \cup b$ in e	pair elimination
	|	$[-]e$		application
	|	$v[-]$		application

Θ	$::=$		fractional permission environment
	$\mid$	$\cdot$	
	$\mid$	Θ, fc	
Γ	$::=$		linear types environment
	$\mid$	$\cdot$	
	$\mid$	$\Gamma, x : t$	
	$\mid$	Γ, Γ'	
Δ	$::=$		intuitionistic types environment
	$\mid$	$\cdot$	
	$\mid$	$\Delta, x : t$	
σ	$::=$		heap (multiset of triples)
	$\mid$	$\{\}$	empty heap
	$\mid$	$\sigma + \{l \mapsto_f m_{k_1,k_2}\}$	location l points to matrix m
Config	$::=$		result of small step
	$\mid$	$\langle \sigma, e \rangle$	heap and expression
	$\mid$	**err**	error

F Primitives

```
module Arr = Owl.Dense.Ndarray.D
type z = Z
type 'a s = Succ
type 'a arr = A of Arr.arr [@@unboxed]
type 'a mat = M of Arr.arr [@@unboxed]
type 'a bang = Many of 'a [@@unboxed]
module Prim :
sig
  val extract : 'a bang -> 'a
  (** Boolean *)
  val not_ : bool bang -> bool bang
  (** Arithmetic, many omitted for brevity *)
  val addI : int bang -> int bang -> int bang
  val eqI : int bang -> int bang -> bool bang
  (** Arrays *)
  val set : z arr -> int bang -> float bang -> z arr
  val get : 'a arr -> int bang -> 'a arr * float bang
  val share : 'a arr -> 'a s arr * 'a s arr
  val unshare : 'a s arr -> 'a s arr -> 'a arr
  val free : z arr -> unit
  (** Owl *)
  val array : int bang -> z arr
  val copy : 'a arr -> 'a arr * z arr
  val sin : z arr -> z arr
  val hypot : z arr -> 'a arr -> 'a arr * z arr
  (** Level 1 BLAS *)
  val asum : 'a arr -> 'a arr * float bang
  val axpy : float bang -> 'a arr -> z arr -> 'a arr * z arr
  val dot : 'a arr -> 'b arr -> ('a arr * 'b arr) * float bang
  val scal : float bang -> z arr -> z arr
  val amax : 'a arr -> 'a arr * int bang
  (* Matrix, some omitted for brevity *)
  val matrix : int bang -> int bang -> z mat
  val eye : int bang -> z mat
  val copy_mat : 'a mat -> 'a mat * z mat
  val copy_mat_to : 'a mat -> z mat -> 'a mat * z mat
  val size_mat : 'a mat -> 'a mat * (int bang * int bang)
  val transpose : 'a mat -> 'a mat * z mat
  (* Level 3 BLAS/LAPACK *)
  val gemm : float bang -> ('a mat * bool bang) -> ('b mat * bool bang) ->
             float bang -> z mat -> ('a mat * 'b mat) * z mat
  val symm : bool bang -> float bang -> 'a mat -> 'b mat -> float bang ->
             z mat -> ('a mat * 'b mat) * z mat
  val gesv : z mat -> z mat -> z mat * z mat
  val posv : z mat -> z mat -> z mat * z mat
  val potrs : 'a mat -> z mat -> 'a mat * z mat
  val syrk : bool bang -> float bang -> 'a mat -> float bang -> z mat ->
            'a mat * z mat
end
```

G Kalman Filters from NumLin and C

```
let kalman sigma h mu r_1 data_1 =
  let h, _p_k_n_p_ = Prim.size_mat h in
  let k, n = _p_k_n_p_ in
  let sigma_hT = Prim.matrix n k in
  let (sigma, h), sigma_hT =
    Prim.gemm (Many 1.) (sigma, Many false) (h, Many true) (Many 0.) sigma_hT in
  let (h, sigma_hT), r_2 =
    Prim.gemm (Many 1.) (h, Many false) (sigma_hT, Many false) (Many 1.) r_1 in
  let k_by_k, x = Prim.posv_flip r_2 sigma_hT in
  let (h, mu), data_2 =
    Prim.gemm (Many 1.) (h, Many false) (mu, Many false) (Many (-1.)) data_1 in
  let (x, data_2), new_mu =
    Prim.gemm (Many 1.) (x, Many false) (data_2, Many false) (Many 1.) mu in
  let x_h = Prim.matrix n n in
  let (x, h), x_h =
    Prim.gemm (Many 1.) (x, Many false) (h, Many false) (Many 0.) x_h in
  let () = Prim.free_mat x in
  let sigma, sigma2 = Prim.copy_mat sigma in
  let (sigma, x_h), new_sigma =
    Prim.symm (Many true) (Many (-1.)) sigma x_h (Many 1.) sigma2 in
  let () = Prim.free_mat x_h in
  ((sigma, h), (new_sigma, (new_mu, (k_by_k, data_2))))
```

Figure 19 OCaml code for a Kalman filter, generated (at *compile time*) from the code in Figure 12, passed through `ocamlformat` for presentation.

```
static void kalman( const int n,                  const int k,
                    const double *sigma, /* n,n */ const double *h,   /* k,n */
                    const double *mu,    /* n,1 */ double *r,         /* k,k */
                    double *data,        /* k,1 */ double **ret_sigma /* n,n */ ) {
    double* n_by_k = (double *) malloc(n * k * sizeof(double));
    cblas_dgemm(RowMajor, NoTrans, Trans, n, k, n, 1., sigma, n, h, n, 0., n_by_k, k);
    cblas_dgemm(RowMajor, NoTrans, NoTrans, k, k, n, 1., h, n, n_by_k, k, 1., r, k);
    LAPACKE_dposv(LAPACK_COL_MAJOR, 'U', k, n, r, k, n_by_k, k);
    cblas_dgemm(RowMajor, NoTrans, NoTrans, k, 1, n, 1., h, n, mu, 1, -1., data, 1);
    cblas_dgemm(RowMajor, NoTrans, NoTrans, n, 1, k, 1., n_by_k, k, data, 1, 1., mu, 1);
    double* n_by_n = (double *) malloc(n * n * sizeof(double));
    cblas_dgemm(RowMajor, NoTrans, NoTrans, n, n, k, 1., n_by_k, k, h, n, 0., n_by_n, n);
    free(n_by_k);
    double* n_by_n2 = (double *) malloc(n * n * sizeof(double));
    cblas_dcopy(n*n, sigma, 1, n_by_n2, 1);
    cblas_dsymm(RowMajor, Right, Upper, n, n, -1., sigma, n, n_by_n, n, 1., n_by_n2, n);
    free(n_by_n);
    *ret_sigma = n_by_n2; }
```

Figure 20 CBLAS/LAPACKE implementation of a Kalman filter. I used C instead of Fortran because it is what Owl uses under the hood and OCaml FFI support for C is better and easier to use than that for Fortran. A distinct 'measure_kalman' function that sandwiches a call to this function with `getrusage` is omitted for brevity.