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 designed to express the APIs of low-level linear algebra libraries (such as BLAS/LAPACK) safely and explicitly, 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. Lastly, we contrast it to other recent developments in linear types and show that using linear types and fractional permissions to express the APIs of low-level linear algebra libraries is a simple and effective idea.

2012 ACM Subject Classification Theory of computation → Program specifications
Keywords and phrases numerical, linear, algebra, types, permissions, OCaml
Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23
Supplement Material www.github.com/dc-mak/lt4la

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 tradeoff: they can eschew safety and exploit the full power of the underlying linear algebra libraries, or they can obtain safety at the price of...
NumLin: Linear Types for Linear Algebra

unneeded copies and worse memory efficiency. In this work, show that this tradeoff is not a fundamental one.

NumLin is a functional programming language designed to express the APIs of low-level linear algebra libraries (such as BLAS/LAPACK) safely and explicitly. 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

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 [5]
- we describe an implementation that is both 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 Overview

The core type theory of NumLin is a nearly off-the-shelf linear type theory, supporting familiar features such as linear function spaces $A \rightarrow B$ and tensor products $A \otimes B$. We adopt linearity – the restriction that each program variable be used at most once – since it allows us to express purely functional APIs for numerical library routines that mutate arrays and matrices [17]. 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 $\text{gemm}(\alpha, A, B, \beta, C)$ is $C := \alpha AB + \beta C$.

In this case, $A$ and $B$ may alias each other but neither may alias $C$, because it is being written to. Related to mutating arrays and matrices is freeing them. Here, we would also wish to restrict aliasing so that we do not free one alias and then attempt to use another. Although linearity on its own suffices to prevent use-after-free errors when values are not aliased (a freed value is out of scope for the rest of the expression), we still need another simple, yet powerful concept to provide us with the extra expressivity of aliasing without losing any of the benefits of linearity.

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

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

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

2.1.3 Runtime Errors

Aside from out-of-bounds indexing, matrix unsharing is one of only two operations that can fail at runtime (the other being dimension checks, such as for $\text{gemm}$). 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 Examples

2.2.1 Factorial

Although a factorial function (Figure 1) 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). Its body 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. The ! annotation on x is a syntactic convenience for declaring the value to used intuitionistically, its full and precise meaning is described in Section 4.1.1.

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.

2.2.2 Summing over an Array

Now we can add fractional permissions to the mix: Figure 2 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 3).

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.
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)
  in
sum_array

Figure 2 Summing over an array in NumLin.

let row = row[i] := x1 in (* or *) let () = free row in
(* Could not show equality: *)
(* s arr *)
(* with *)
(* var 'z arr *)
(* var 'z is universally quantified *)
(* Are you trying to write to/free/unshare an array you don't own? *)
(* In examples/sum_array.lt, at line: 7 and column: 19 *)

Figure 3 Attempting to write to or free a read only array in NumLin.

for the rest of the expression, so it needs to be returned and then rebound to be used for the
rest of the function.

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

2.2.3 One-dimensional Convolution

Figure 4 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 4 is a simplified version of this idea, so as to not distract
from the features of NumLin. The simplifications are:
  - the kernel has a length 3, so only the value of write[i-1] (prior to modification in the
    previous iteration) needs to be carried forward using x0
  - write is assumed to have length n+1
  - i’s initial value is assumed to be 1
  - x0’s initial value is assumed to be write[0]
  - the first and last values of write are ignored.

Mutating an array is implemented similarly to indexing one: a primitive set: z arr
--o !int --o !elt --o z arr. It consumes the original array and returns a new array
with the updated value. let written = write[i] := <exp> is just syntactic sugar for let
written = set write i <exp>.
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

Figure 4 Simplified one-dimensional convolution.

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 5 Linear regression (OLS): \( \hat{\beta} = (X^T X)^{-1} X^T y \)

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.

2.2.4 Squaring a Matrix

The most pertinent aspect of NumLin is the types of its primitives. 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.

Figure 5 shows how a linearly-typed matrix squaring function may be written in NumLin. It is a non-recursive function declaration (the return type is inferred). Since we would like to be able to use a function like square more than once, it is marked with a ! annotation (which also ensures it captures no linear values from the surrounding environment).

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

Even if x had type z mat, sharing it now enforces the assumption of all BLAS/LAPACK routines that any matrix which is written to (which, in NumLin, is always of type z mat) does not alias any other matrix in scope. So if we did try to use one of the aliases in mutating way, the expression would not type check, and we would get an error similar to the one in Figure 3.
let !lin_reg ('x) (x : 'x mat) =
    let (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 6 Linear regression (OLS): \( \hat{\beta} = (X^T X)^{-1} X^T y \)

The line let answer <- new (m,n) [| x1 * x2 |] is syntactic sugar for first creating
a new \( m \times n \) matrix (let answer = matrix m n) and then storing the result of the mul-
tiplication in it (let ((x1, x2), answer) = gemm 1. _ (x1, false) _ (x2, false) 0.
answer). false means the matrix should not be accessed with indices transposed.
By using some simple pattern-matching and syntactic sugar, 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
  routines.

2.2.5 Linear Regression

In Figure 6, we wish to compute \( \hat{\beta} = (X^T X)^{-1} X^T y \). 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 \( X^T y \) (using syntactic sugar for matrix and gemm calls
similar to that used in Figure 5, with a \( ^T \) annotation on x to set x’s ‘transpose indices’-flag
to true).
However, the line let x_T_x <- new (m,m) [| x^T * x |], works for a slightly different
reason: that pattern is matched to a BLAS call to (syrk true 1. x 0. x_T_x), which
only uses x once. Hence x can appear twice in the pattern without any calls to share.
After computing x_T_x, we need to invert it and then multiply it by xy. The BLAS
routine posv: z mat --o z mat --o z mat * 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).

2.2.6 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 formation[8] 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.
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

Figure 7 L1-norm minimisation on manifolds: $Q^{-1}(I + U^TQ^{-1})^{-1}U^T$

Figure 7 shows even more pattern-matching. Patterns of the form `let <id> <- [| beta * c + alpha * a * b |]` are also desugared to `gemm` calls. Primitives like `transpose`: 'x. 'x mat --o 'x mat + z mat and `eye`: 'int --o 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_5_5` 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_5_5 + q_inv_u * inv_u_T |]`. Again, thanks to linearity, the identifiers `q` and `tmp_5_5` 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.

### 2.2.7 Kalman Filter

A Kalman Filter[12] is a 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 8 shows a NumLin implementation of a Kalman filter (equations in Figure 9). 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)
- `let new_r <- new [| r_2 |]` creates a copy of `r_2`
- `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`
let \( \text{kalman} \) =

\[
\begin{align*}
\text{let}& (h, (|k, |n)) = \text{sizeM }_\text{h} \\
\text{let}& \sigma_h^T \leftarrow \text{new } (n, k) \\{ | \sigma \ast h^T | \} \text{ in} \\
\text{let}& r_2 \leftarrow \{ | r_1 + h \ast \sigma_h^T | \} \text{ in} \\
\text{let}& (k_{by\_k}, x) = \text{posvFlip } r_2 \sigma_h^T \text{ in} \\
\text{let}& \text{new}_\text{mu} \leftarrow \{ | \mu + x \ast \text{data}_1 | \} \text{ in} \\
\text{let}& x_h \leftarrow \text{new } (n, n) \\{ | x + h | \} \text{ in} \\
\text{let}& () = \text{freeM } (| n, k *) \text{ in} \\
\text{let}& \text{new}_\text{sigma} \leftarrow \{ | \sigma_2 - x_h \ast \text{sym}(\sigma) | \} \text{ in} \\
\text{let}& () = \text{freeM } (| n, n *) \text{ in} \\
\text{let}& (\text{new}_\text{mu}, (\text{new}_\text{sigma}, (\text{new}_\text{mu}, (k_{by\_k}, \text{data}_2)))) \text{ in} \\
\text{kalman} \\
\end{align*}
\]

Figure 8: Kalman filter: see Figure 9 for the equations this code implements and Figure 17 for an equivalent CBLAS/LAPACKE implementation.

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

Figure 9: Kalman filter equations (credit: matthewrocklin.com).

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.

A typing judgement consists of \( \Theta; \Delta; \Gamma \vdash e : t \).

\( \Theta \) 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 \( \Theta \).

\( \Delta \) is the environment storing non-linearly or intuitionistically typed variables.
\( \Gamma \) is the environment storing linearly typed variables.

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

\[
\Theta; \Delta; \Gamma \vdash () : \text{unit}
\]

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).

\[
\Theta; \Delta; \Gamma \vdash e : \text{!bool}
\]

\[
\Theta; \Delta; \Gamma; e \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : t
\]

The \texttt{Many} introduction and elimination rules are very important. Producing \texttt{!}-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.

\[
\Theta; \Delta; \Gamma \vdash v : t
\]

\[
v \neq l
\]

\[
\Theta; \Delta; \Gamma \vdash \text{Many } v : \text{!}!
\]

Consuming a \texttt{!}-type value moves it from the linear environment \( \Gamma \) and into the intuitionistic environment \( \Delta \). This is exactly why \texttt{let} \texttt{!}x = e1 in e2 desugars to \texttt{let} \texttt{Many} x = e1 \texttt{in} \texttt{let} \texttt{Many} x = \texttt{Many} (\texttt{Many} x) \texttt{in} e2.

\[
\Theta; \Delta; \Gamma \vdash e : \text{!t}
\]

\[
\Theta; \Delta; \Gamma; x \vdash e' : t'
\]

\[
\Theta; \Delta; \Gamma; \Gamma' \vdash \text{let} \texttt{Many} x = e \texttt{in} e' : t'
\]

Rules \texttt{Ty_Gen} and \texttt{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.

\[
\Theta, fc; \Delta; \Gamma \vdash e : t
\]

\[
\Theta; \Delta; \Gamma \vdash \text{fun} fc \rightarrow e : \forall fc.t
\]

\[
\Theta \vdash f \text{Perm}
\]

\[
\Theta; \Delta; \Gamma \vdash e : \forall fc.t
\]

\[
\Theta; \Delta; \Gamma \vdash e[f/fc] : t[f/fc]
\]

Rule \texttt{Ty_Fix} shows how recursive functions are typed. Even though recursive functions are fully annotated, type checking them is interesting for two reasons: to type check the body of the fixpoint, the type of the recursive function is in the intuitionistic environment \( \Delta \) (without this, you would not be able to write a base case) whilst the argument and its type
The full, small-step transition relation is in Appendix A.2, but the key ideas are as follow.

\[ \Theta; \Delta, y : t \vdash t' \vdash e : t' \quad \Theta; \Delta ; \vdash \text{fix} (g, x : t, e : t') : t \rightarrow t' \quad \text{TY\_Fix} \]

Lastly, types of almost all NumLin primitives, as embedded in OCaml's type system, are shown in Appendix G, with some similar ones (like those for binary arithmetic operators) omitted for brevity. The main difference between the OCaml type of a primitive like \texttt{gemm} and its NumLIN counterpart is the inclusion of explicit '∀'s. So, \texttt{float bang \rightarrow \{ 'a mat * bool bang \} \rightarrow \{ 'b mat * bool bang \} \rightarrow float bang \rightarrow z mat \rightarrow \{ 'a mat * 'b mat \} * z mat} will correspond to

\[ \text{left} \rightarrow \forall x, x \text{ mat} \otimes \text{bool} \rightarrow \forall y, y \text{ mat} \otimes \text{bool} \rightarrow \text{left} \rightarrow z \text{ mat} \rightarrow (x \text{ mat} \otimes y \text{ mat}) \otimes z \text{ mat} \]

### 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 \texttt{fun fc \rightarrow [\_]} must be a syntactic value. The context \texttt{fun fc \rightarrow [\_]} 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 (\texttt{OP\_FRAC\_PERM}).

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

Like with the static semantics, the interesting rules in the dynamic semantics are those relating to primitives. Creating a matrix (\texttt{matrix} \( k_1 \ k_2 \)) successfully (\texttt{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.

\[ 0 \leq k_1, k_2 \]
\[ l \text{ fresh} \]
\[ \langle \sigma, \text{matrix} k_1 k_2 \rangle \rightarrow \langle \sigma + \{ l \mapsto_1 M_{k_1,k_2} \}, l \rangle \quad \text{OP\_MATRIX} \]

Dually, \texttt{OP\_FREE}, requires a location represent complete ownership before removing it and the matrix it points to from the heap.

\[ \langle \sigma + \{ l \mapsto_1 m_{k_1,k_2} \}, \text{free} l \} \rightarrow \langle \sigma, () \rangle \quad \text{OP\_FREE} \]

Choosing a multiset representation as opposed to a set allows for two convenient invariants: multiplicity of a triple \( l \mapsto f m_{k_1,k_2} \) in the heap corresponds to the number of aliases of l in the
expression with type \( f \mathbf{mat} \) 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 \texttt{OP\_SHARE} and \texttt{OP\_UNSHARE\_EQ} are fairly natural.

\[
\sigma + \{ l \mapsto_{f} m_{k, k_2} \}, \text{share}[f] l \rightarrow \sigma + \{ l \mapsto_{f} m_{k, k_2} \} + \{ l \mapsto_{g} m_{k, k_2} \}, (l, l) \quad \text{OP\_SHARE}
\]

\[
\sigma + \{ l \mapsto_{f} m_{k, k_2} \} + \{ l \mapsto_{g} m_{k, k_2} \}, \text{unshare}[f] ll \rightarrow \sigma + \{ l \mapsto_{f} m_{k, k_2} \}, l \quad \text{OP\_UNSHARE\_EQ}
\]

Combining all of these features, we see that \texttt{OP\_GEMM\_MATCH} requires that the location being updated \( (l_3) \) has complete ownership of the matrix \( m_3 \) and can thus change what value it stores to \( m_1m_2m_3 \). In particular, this places no restriction on \( l_2 \) and \( l_3 \); they could be shared aliases of the same matrix. Transition rules for other primitives (omitted) follow the same structure: \( \mapsto_1 \) for any locations that are written to and \( \mapsto_2 \) for anything else.

\[
\sigma' \equiv \sigma + \{ l_1 \mapsto_{f_1} m_{l_1, k_2} \} + \{ l_2 \mapsto_{f_2} m_{l_2, k_3} \} \\
\sigma_1 \equiv \sigma' + \{ l_1 \mapsto_{m_1} m_{l_1} \} \\
\sigma_2 \equiv \sigma' + \{ l_1 \mapsto_{m_3} m_{l_1} \}
\]

\[
\langle \sigma_1, \text{gemm}[f_1], l_1[f_2], l_2 \rangle \rightarrow \langle \sigma_2, (l_1, l_2), l_3 \rangle \quad \text{OP\_GEMM\_MATCH}
\]

### 3.3 Logical Relation

First, we define an interpretation of heaps with fractional permissions in the style of Bornat et. al [6] (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 \( \mapsto \).

\[
\mathcal{H}[\sigma] = \star_{\{ l, f, m \in \sigma \mid l \mapsto_{f} m \}}
\]

where

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

We then define a step-indexed logical relation in the style of Morrisett et. al [14]. \((\varsigma, v) \in \mathcal{V}_k[l] \) means it takes a heap with exactly \( \varsigma \) resources to produce a value \( v \) of type \( l \) in at most \( k \) steps. So, something like a \texttt{unit} or a \texttt{!} needs no resources, whereas a \texttt{f \mathbf{mat}} needs exactly \( f \) ownership of a some matrix and a pair needs a \( \star \) combination of the heaps required for each component.

\[
\mathcal{V}_k[\text{unit}] = \{ (\emptyset, *) \}
\]

\[
\mathcal{V}_k[f \mathbf{mat}] = \{ \{ l \mapsto_{2-f} \_ \} \}
\]

\[
\mathcal{V}_k[l] = \{ (\emptyset, \text{Many} v) \mid (\emptyset, v) \in \mathcal{V}_k[l] \}
\]

\[
\mathcal{V}_k[l_1 \otimes l_2] = \{ (\varsigma_1 \star \varsigma_2, (v_1, v_2)) \mid (\varsigma_1, v_1) \in \mathcal{V}_k[l_1] \wedge (\varsigma_2, v_2) \in \mathcal{V}_k[l_2] \}
\]

The definition of \( \mathcal{V}_k[f \mathbf{c} \mathbf{t}, l] \) 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 (\texttt{OP\_SPC}).

\[
\mathcal{V}_k[f \mathbf{c} \mathbf{t}, l] = \{ (\varsigma, \text{fun } fc \mapsto v) \mid \forall f. (\varsigma{[fc/f]}, v{[fc/f]}) \in \mathcal{V}_{k-1}[l{[fc/f]}] \}
\]
To understand the definition of $V_k[t' \rightarrow t]$, we must first look at $C_k[t]$, 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, with the further condition that if the expression is a syntactic value then it is also one semantically.

$$C_k[t] = \{ (\varsigma, e_s) \mid \forall j < k, \sigma_r, \varsigma_s \in \varsigma_r \text{ defined } \Rightarrow \langle \sigma_s + \sigma_r, e_s \rangle \rightarrow^j \text{ err } \lor \exists \sigma_f, e_f, \langle \sigma_s + \sigma_r, e_s \rangle \rightarrow^j \langle \sigma_f + \sigma_r, e_f \rangle \land (e_f \text{ is a value } \Rightarrow (\varsigma_f \star \varsigma_r, e_f) \in V_{k-j}[t]) \}$$

In this light, $V_k[t' \rightarrow t]$ simply says that $v$ is a function and that evaluating the application of it to any argument (of the correct type, requiring its own set of resources, bounded by $k$ steps) satisfies all the aforementioned properties.

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

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

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

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

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

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

### 3.4 Soundness Theorem

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

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

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 \( \Gamma / \Gamma' \), then (by induction) use \( C_k \) and one `half` of the linear substitution and heap to conclude the `first` sub-expression either takes \( j < k \) steps to \texttt{err} or another heap-and-expression.

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

\[
(\sigma, e) \rightarrow^i (\sigma', e') \\
(\sigma, C[e]) \rightarrow^j (\sigma', C'[e']) \quad \texttt{Op_Context}
\]

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

4. Implementation

4.1 Implementation Strategy

\texttt{NumLin} transpiles to OCaml and its implementation follows the structure of a typical domain-specific language (DSL) compiler. Although \texttt{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: \texttt{NumLin}'s type system is embedded into OCaml’s (Figure 11), so the OCaml type checker acts as a sanity check on the generated code.

A very pleasant way to use \texttt{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 \texttt{NumLin} alongside existing OCaml libraries; in fact, this is exactly how the benchmarking program and test-suite use code written in \texttt{NumLin}. 
4.1.1 Desugaring, Matrix Expressions and Type Checking

Desugaring is conventional, outlined in Appendix F. Matrix expressions are translated into BLAS/LAPACK calls via purely syntactic pattern-matching, outlined in Figure 10.

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

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 11).

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 12). 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 13). 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 14, a larger one in Figure 16.

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
let \(v \leftarrow x[e] \text{ in } e\) \(\Rightarrow\) let \((x,!)v = x[e] \text{ in } e\) (similarly for matrices)

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

let \(x_2 \leftarrow [\![ x_1 ]\!] \text{ in } e\) \(\Rightarrow\) let \((x_1, x_2) = \text{copyM } \text{to } _{-} x_1 \text{ } x_2 \text{ in } e\)

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

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

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

let \(Y \leftarrow [\![ \alpha XX^T + \beta Y ]\!] \text{ in } e\) \(\Rightarrow\)

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

let \(Y \leftarrow [\![ \alpha X^T X + \beta Y ]\!] \text{ in } e\) \(\Rightarrow\)

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

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

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

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

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

let \(Y \leftarrow [\![ \alpha X_1^T X_2^T + \beta Y ]\!] \text{ in } e\) \(\Rightarrow\)

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

\[ \text{Figure 10} \] Purely syntactic pattern-matching translations of matrix expressions.

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

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

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

### 4.2.2 Hypothesis

We expected the C implementation to be faster than the NumLIN one because the latter has the additional (but relatively low) overhead of dimension checks and crossing the OCaml/C FFI for each call to a CBLAS routine, even though the calls and their order are exactly the same. We expected the OCaml and Python implementations to be slower because they allocate more temporaries (so possibly less cache-friendly) and carry out more floating-point operations – the CBLAS and NumLIN implementations use ternary kernels (coalescing steps), a Cholesky decomposition (of a symmetric matrix, which is more efficient than the LU
\[
\begin{align*}
\text{f ::=} & \quad \text{module Arr = } \text{Owl.Dense.Ndarray.D} \\
| \space fc & \quad [fc] = 'fc \\
| \space Z & \quad [Z] = z \\
| \space S \space f & \quad [S \space f] = [f] \space s \\
t ::= & \quad \text{type } 'a \ space a = \text{Succ} \\
| \space unit & \quad [\text{unit}] = \text{unit} \\
| \space bool & \quad [\text{bool}] = \text{bool} \\
| \space int & \quad [\text{int}] = \text{int} \\
| \space elt & \quad [\text{elt}] = \text{float} \\
| \space f \space arr & \quad [\text{farr}] = [f] \space \text{arr} \\
| \space f \space mat & \quad [\text{fmat}] = [f] \space \text{mat} \\
| \space ! t & \quad [\text{bang}] = [t] \space \text{bang} \\
| \space t \otimes t' & \quad [t \otimes t'] = [t] \otimes [t'] \\
| \space t \multimap t' & \quad [t \multimap t'] = [t] \multimap [t'] 
\end{align*}
\]

**Figure 11** NumLIN’s type grammar (left) and its embedding into OCaml (right).

```
let it44a_kalman ~sigma ~h ~mu ~r ~data =
 0 examples.kalman.ml (M sigma) (M h) (M mu) (M r) (M r) (M data)

.mltype-history:
:t

let fact = examples.factorial.ml in

.mltype-history:
:mltype-history:

int bang -> int bang
```

**Figure 12** Using NumLIN functions from OCaml.

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 15 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
let Many x = x in
let Many x = Many (Many x) in \exp \Rightarrow \exp

(* fixp = fix (f, x:t, \exp : t) *)
(*1*) let Many f = Many fixp in \body
(*2*) let f = fixp in \body
(*3*) (fun x : t \rightarrow \body) \exp

**Figure 13** 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 14** Recursive OCaml function for a summing over an array, generated (at compile time) from the code in Figure 2, passed through ocamlformat for presentation.

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 [15].

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
but would have been written over by calls to ‘gemm’, spotted by the fractional-capabilities 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-capabilities.

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 \texttt{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 \texttt{freeM} followed by allocating call to \texttt{copy} with one, in-place call to \texttt{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

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[16, 10, 4]. 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 [10] and Single Assignment C [16] omit higher-order functions to facilitate compilation to GPUs. The work of [4] 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.

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.

This 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 [18] and [11] suggest that Rust’s borrow-checker can be expressed in simpler terms using \textit{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 [5]). 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.

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.

This fact is important because there are major upcoming implementations of linear types such as Linear Haskell [3], which do not have built-in support for fractional permissions. Instead, Linear Haskell takes a slightly different definition of linearity, one based on \textit{arrows} as opposed to \textit{kinds}: for $f : a \rightarrow b$, if $fu$ is used exactly once then $u$ is used exactly once. Whilst
this has the advantage of being backwards-compatible, it also means that the type system
has no built-in support for the concurrent reader, exclusive writer pattern that fractional
permissions enable.

However, since our type system shows demonstrates 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 [7], 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[14], 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[13] or statically enforcing dimensions related constraints of
the arguments at compile-time. ATS[9] 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[2]. However, it is worth noting that without care, such algorithms[15],
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[13].
Matrix size $n$ (for a Kalman filter, with $k = 3n/5$)

- CBLAS
- NUMLIN
- OWL
- NUMPY

Figure 15 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

A. NumLin Specification

A.1 Static Semantics

Typing rules for expressions

\[ \Theta; \Delta; \Gamma \vdash e : t \]  
TY_VAR_LIN

\[ \Theta; \Delta; \Gamma \vdash x : t \]  
TY_VAR

\[ \Theta; \Delta; \Gamma; x : t \vdash e : t' \]  
TY_LET

\[ \Theta; \Delta; \vdash () : \text{unit} \]  
TY_UNIT_INTRO

\[ \Theta; \Delta; \vdash e : \text{unit} \]  
TY_UNIT_ELIM

\[ \Theta; \Delta; \vdash \text{true} : \text{bool} \]  
TY_BOOL_TRUE

\[ \Theta; \Delta; \vdash \text{false} : \text{bool} \]  
TY_BOOL_FALSE

\[ \Theta; \Delta; \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : t \]  
TY_BOOL_ELIM

\[ \Theta; \Delta; \vdash k : \text{int} \]  
TY_INT_INTRO

\[ \Theta; \Delta; \vdash e : \text{elt} \]  
TY_ELT_INTRO

\[ \Theta; \Delta; \vdash v : t \]  
TY_BANG_INTRO

\[ \Theta; \Delta; \vdash \text{Many } v : !t \]  
TY_BANG_ELIM

\[ \Theta; \Delta; \Gamma; x : t; \Gamma' \vdash e : !t' \]  
TY_BANG_ELIM

\[ \Theta; \Delta; \Gamma \vdash \text{let Many } x = e \text{ in } e' : t' \]  
TY_BANG_ELIM

\[ \Theta; \Delta; \Gamma \vdash e : t \]  
TY_PAIR_INTRO

\[ \Theta; \Delta; \Gamma' \vdash e' : t' \]  
TY_PAIR_INTRO
A.2 Dynamic Semantics

\[ \langle \sigma, e \rangle \rightarrow \text{Config} \]  
Operational semantics

\[ \langle \sigma, \text{let } () = () \text{ in } e \rangle \rightarrow \langle \sigma, e \rangle \]  
\text{OP\_LET\_UNIT}

\[ \langle \sigma, \text{let } x = v \text{ in } e \rangle \rightarrow \langle \sigma, e[x/v] \rangle \]  
\text{OP\_LET\_VAR}

\[ \langle \sigma, \text{if } (\text{Many true}) \text{ then } e_1 \text{ else } e_2 \rangle \rightarrow \langle \sigma, e_1 \rangle \]  
\text{OP\_IF\_TRUE}

\[ \langle \sigma, \text{if } (\text{Many false}) \text{ then } e_1 \text{ else } e_2 \rangle \rightarrow \langle \sigma, e_2 \rangle \]  
\text{OP\_IF\_FALSE}

\[ \langle \sigma, \text{let Many } x = \text{Many } v \text{ in } e \rangle \rightarrow \langle \sigma, e[x/v] \rangle \]  
\text{OP\_LET\_MANY}

\[ \langle \sigma, \text{let } (a, b) = (v_1, v_2) \text{ in } e \rangle \rightarrow \langle \sigma, e[a/v_1][b/v_2] \rangle \]  
\text{OP\_LET\_PAIR}

\[ \langle \sigma, (~\text{fun } fc \rightarrow v)[f] \rangle \rightarrow \langle \sigma[f/c][v/f], v[f/c/f] \rangle \]  
\text{OP\_FRAC\_PERM}

\[ \langle \sigma, \text{fix } (g, x : t, e : t') v \rangle \rightarrow \langle \sigma, e[x/v][g/\text{fix } (g, x : t, e : t')] \rangle \]  
\text{OP\_APP\_FIX}

\[ \langle \sigma, (~\text{fun } x : t \rightarrow e) v \rangle \rightarrow \langle \sigma, e[x/v] \rangle \]  
\text{OP\_APP\_LAMBDA}
\[\langle \sigma, \epsilon \rangle \rightarrow \langle \sigma', \epsilon' \rangle\] \hspace{1cm} \text{Op\_CONTEXT}

\[\langle \sigma, C[\epsilon] \rangle \rightarrow \langle \sigma', C[\epsilon'] \rangle\] \hspace{1cm} \text{Op\_CONTEXT\_ERR}

\[0 \leq k_1, k_2 \quad \text{l fresh}\] \hspace{1cm} \text{Op\_MATRIX}

\[\langle \sigma, \text{matrix} k_1 k_2 \rangle \rightarrow \langle \sigma + \{l \mapsto l \mapsto M_{k_1, k_2} \}, l \rangle\] \hspace{1cm} \text{Op\_FREE}

\[\langle \sigma + \{ l \mapsto l \mapsto m_{k_1, k_2} \}, \text{free} l \rangle \rightarrow \langle \sigma, () \rangle\] \hspace{1cm} \text{Op\_SHARE}

\[\langle \sigma + \{ l \mapsto f m_{k_1, k_2} \}, \text{share}[f] l \rangle \rightarrow \langle \sigma + \{ l \mapsto \frac{1}{2} f m_{k_1, k_2} \} + \{ l \mapsto \frac{1}{2} f m_{k_1, k_2} \}, (l, l) \rangle\] \hspace{1cm} \text{Op\_UNSHARE\_EQ}

\[\langle \sigma + \{ l \mapsto \frac{1}{2} f m_{k_1, k_2} \} + \{ l \mapsto \frac{1}{2} f m_{k_1, k_2} \}, \text{unshare}[f] l l' \rangle \rightarrow \langle \sigma + \{ l \mapsto f m_{k_1, k_2} \}, l \rangle\] \hspace{1cm} \text{Op\_UNSHARE\_NEQ}

\[\langle \sigma' \equiv \sigma + \{ l_1 \mapsto \frac{1}{2} f m_{1, l_3} \} \} + \{ l_2 \mapsto \frac{1}{2} f m_{2, l_3} \}, l \rangle\] \hspace{1cm} \text{Op\_GEMM\_MATCH}

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

\[\langle \sigma' \equiv \sigma + \{ l_1 \mapsto \frac{1}{2} f m_{l_2, k_3} \} + \{ l_2 \mapsto \frac{1}{2} f m_{l_1, k_3} \}, \text{gemm}[f_{c_1}] l_1 [f_{c_2}] l_2 [l_3] \rangle \rightarrow \text{err}\] \hspace{1cm} \text{Op\_GEMM\_MISMATCH}

\[k_2 \neq k_3\]
B Interpretation

B.1 Definitions

Operationally, $\text{Heap} \subseteq \text{Loc} \times \text{Permission} \times \text{Matrix}$ (a multiset), denoted with a $\sigma$.

Define its interpretation to be $\text{Loc} \rightarrow \text{Permission} \times \text{Matrix}$ with $\star : \text{Heap} \times \text{Heap} \rightarrow \text{Heap}$ as follows:

\[
(\rho_1 \star \rho_2)(l) \equiv \begin{cases} 
\rho_1(l) & \text{if } l \in \text{dom}(\rho_1) \land l \notin \text{dom}(\rho_2) \\
\rho_2(l) & \text{if } l \in \text{dom}(\rho_2) \land l \notin \text{dom}(\rho_1) \\
(f_1 + f_2, m) & \text{if } (f_1, m) = \rho_1(l) \land (f_2, m) = \rho_2(l) \land f_1 + f_2 \leq 1 \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

Commutativity and associativity of $\star$ follows from that of $+$.

$\rho_1 \star \rho_2$ is defined if it is for all $l \in \text{dom}(\rho_1) \cup \text{dom}(\rho_2)$.

Define $\mathcal{H}[\sigma] = \star_{(l,f,m)\in\sigma} l \mapsto f, m$ and implicitly denote $\varsigma \equiv \mathcal{H}[\theta(\sigma)]$.

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

\[
\forall n. \text{err} \rightarrow^n \text{err} \quad \langle \sigma, v \rangle \rightarrow^n \langle \sigma, v \rangle \quad \langle \sigma, e \rangle \rightarrow^0 \langle \sigma, e \rangle \quad \langle \sigma, e \rangle \rightarrow^{n+1} ((\langle \sigma, e \rangle \rightarrow) \rightarrow^n)
\]

Hence, all bounded iterations end in either an err, a heap-and-expression or a heap-and-value.
B.2 Interpretation

\[ \mathcal{V}_k[\text{unit}] = \{(\emptyset, \ast)\} \]

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

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

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

\[ \mathcal{V}_k[\text{mat}] = \{(l \mapsto_{2-f} \_), l\} \]

\[ \mathcal{V}_k[l] = \{(\emptyset, \text{Many } v) \mid (\emptyset, v) \in \mathcal{V}_k[l]\} \]

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

\[ \mathcal{V}_k[l_1 \otimes l_2] = \{(s_1 \otimes s_2, (v_1, v_2)) \mid (s_1, v_1) \in \mathcal{V}_k[l_1] \wedge (s_2, v_2) \in \mathcal{V}_k[l_2]\} \]

\[ \mathcal{V}_k[l' \to l] = \{(s_v, v) \mid (v \equiv \text{fun } x : t' \to e \lor v \equiv \text{fix}(g, x : t', e : t)) \wedge \forall j \leq k, (s_{v'}, v') \in \mathcal{V}_j[l'], s_v \ast s_{v'} \text{ defined } \Rightarrow (s_v \ast s_{v'}, v v') \in \mathcal{C}_j[l]\} \]

\[ \mathcal{C}_k[l] = \{\{(s_v, e_s) \mid \forall j < k, \sigma_{v, e_s} \text{ defined } \Rightarrow (\sigma_{v + \sigma_v, e_s}) \rightarrow^j \text{ err } \lor \exists \sigma_f, e_f. (\sigma_{v + \sigma_v, e_s}) \rightarrow^j (\sigma_f + \sigma_v, e_f) \wedge (e_f \text{ is a value } \Rightarrow (\sigma_f \ast \sigma_v, e_f) \in \mathcal{V}_{k-j}[l]\}\} \]

\[ \mathcal{I}_k[l] = \{[]\} \]

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

\[ \mathcal{L}_k[l] = \{(\emptyset, [])\} \]

\[ \mathcal{L}_k[l] = \{(s \ast s_x, \gamma [x \mapsto v_x]) \mid (s, \gamma) \in \mathcal{L}_k[l] \wedge (s_x, v_x) \in \mathcal{V}_k[\theta(t)]\} \]

\[ \mathcal{H}[\sigma] \equiv \bigstar_{(i, f, m) \in \sigma} [l \mapsto_f m] \]

\[ k \equiv \mathcal{H}[\theta(\sigma)] \]

\[ k[\Theta; \Delta; \Gamma \vdash e : t] = \forall \theta, \delta, \gamma, \sigma. \Theta = \text{dom}(\theta) \wedge (s, \gamma) \in \mathcal{L}_k[l] \theta \wedge \delta \in \mathcal{I}_k[\Delta] \theta \Rightarrow (\sigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\theta(t)] \]
C

Lemmas

C.1 \( \forall \sigma_s, \sigma_r, e. \, \varsigma_s \times \varsigma_r \text{ defined } \Rightarrow \forall n. \, \langle \sigma_s, e \rangle \rightarrow^n = \langle \sigma_s + \sigma_r, e \rangle \rightarrow^n \)

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

PROOF SKETCH: Only \( \text{Op}_{\{\text{Free, Matrix, Share, Unshare\_Eq, Gemm\_Match}\}} \) change the heap: the rest are either parametric in the heap or step to an \text{err}.

PROVE: \( \langle \sigma_s + \sigma_r, e \rangle \rightarrow \langle \sigma_f + \sigma_r, e_f \rangle \).

(1) CASE: \( \text{Op\_Free} \), \( \sigma_s \equiv \sigma' + \{ l \mapsto m \}, \sigma_f = \sigma' \).

Proof: Instantiate \( \text{Op\_Free} \) with \( \{ \sigma' + \sigma_r \} + \{ l \mapsto m \} \),
valid because \( l \not\in \text{dom}(\varsigma_r) \) by \( \varsigma' \times [ l \mapsto m ] \times \varsigma_r \) defined (assumption).

(1) CASE: \( \text{Op\_Matrix} \)

Proof: Rule has no requirements on \( \sigma_s \) so will also work with \( \sigma_s + \sigma_r \).

(1) CASE: \( \text{Op\_Share} \), \( \sigma_s \equiv \sigma' + \{ l \mapsto m \}, \sigma_f = \sigma' + \{ l \mapsto m \} + \{ l \mapsto m \} \).

Proof: Union-ing \( \sigma_r \) does not remove \( l \mapsto m \), so that can be split out of \( \sigma_s + \sigma_r \) as before.

(1) CASE: \( \text{Op\_Unshare\_Eq} \), \( \sigma_s \equiv \sigma' + \{ l \mapsto m \} + \{ l \mapsto m \}, \sigma_f = \sigma' + \{ l \mapsto m \} \).

(2) UNION-ing \( \sigma_r \) does not remove \( l \mapsto m \), so that can still be split out of \( \sigma_s + \sigma_r \).

(2) However, by assumption of \( \varsigma_s \times \varsigma_r \) defined, any splitting of \( \sigma_s + \sigma_r \) will satisfy \( f \leq 1 \).

(1) CASE: \( \text{Op\_Gemm\_Match} \)

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

(2) 2. The permissions (of \( l_1 \) and/or \( l_2 \)) may differ, but \( \text{Op\_Gemm\_Match} \) universally quantifies over them and leaves them unchanged, so they are irrelevant.

(2) 3. Only the pointed to matrix value at \( l_3 \) changes.

(2) 4. SUFFICES: \( l_3 \not\in \pi_1[\sigma_r] \).

(2) 5. By assumption of \( \varsigma_s \times \varsigma_r \) defined, \( l_3 \not\in \text{dom}(\varsigma_r) \).

(2) 6. Hence \( l_3 \not\in \pi_1[\sigma_r] \).

C.2 \( \forall k, t. \, V_k[t] \subseteq C_k[t] \)

Follows from definition of \( C_k[t] \), \( \rightarrow^j \) (\( \forall n. \, \langle \sigma, v \rangle \rightarrow^n \langle \sigma, v \rangle \)) for arbitrary \( j \leq k \) and C.1.
\textbf{C.3} \quad \forall \theta, \delta, \gamma, v. \quad \theta(\delta(\gamma(v))) \text{ is a value.}

\textit{\textit{Assume:}} arbitrary \ j < k + 1, \ \text{and} \ \sigma_r \ \text{such that} \ \varsigma \ast \varsigma_r \ \text{defined.}

\textbf{C.4} \quad \forall k, \sigma, \sigma', e, e', t. \quad (\varsigma', e') \in C_k[t] \land \langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle \Rightarrow (\varsigma, e) \in C_{k+1}[t]

\textit{\textit{Assume:}} arbitrary \ j \ < \ k, \ \text{and} \ \sigma_r \ \text{such that} \ \varsigma \ast \varsigma_r \ \text{defined.}

\textbf{C.5} \quad j \leq k \Rightarrow \ast \downarrow j[t] \quad \text{by construction,} \ \delta \text{ and} \ \gamma \ \text{only map} \ \text{variables to values, and values are closed under substitution.}

\textbf{C.6} \quad \forall \Delta, \Gamma, t, k, \delta, \gamma. \quad \delta \in \mathcal{L}_k[\Delta] \land \gamma \in \pi_2[\mathcal{L}_k[\Gamma] \theta] \Rightarrow \text{dom}(\Delta) = \text{dom}(\delta) \quad \text{and} \quad \text{dom}(\Gamma) = \text{dom}(\gamma)

\textbf{C.7} \quad \forall k, \Gamma, \Gamma', \theta, \sigma_+, \gamma_+. \quad (\varsigma_+, \gamma) \in \mathcal{L}_k[\Gamma, \Gamma'] \theta \land \Gamma, \Gamma' \text{ disjoint} \quad \Rightarrow \quad \exists \sigma, \gamma, \sigma', \gamma'. \quad \sigma_+ = \sigma + \sigma' \land \gamma, \gamma' \text{ disjoint} \land \gamma_+ = \gamma \cup \gamma' \land (\varsigma, \gamma) \in \mathcal{L}_k[\Gamma] \land (\varsigma', \gamma') \in \mathcal{L}_k[\Gamma']

\text{\textit{\textit{Proof:}}} \text{ by induction on} \ \Delta \ \text{and} \ \Gamma.
C.8 \( \forall e, \sigma, e', \sigma', \theta. \langle \sigma, e \rangle \to \langle \sigma', e' \rangle \Rightarrow \langle \theta(\sigma), \theta(e) \rangle \to \langle \theta(\sigma'), \theta(e') \rangle \)

PROOF: By induction on \( \to \).

(1)1. Assume: Arbitrary \( e, \sigma, e', \sigma', \theta \) such that \( \langle \sigma, e \rangle \to \langle \sigma', e' \rangle \).

(1)2. Suffices: To consider only the following rules which mention fractional permission variables.

\begin{align*}
\text{OP\_FRAC\_PERM, OP\_SHARE, OP\_UNSHARE\_\(N\)EQ and OP\_GEMM\_\(M\)\(I\)\(S\)\(M\)\(A\)CH.}
\end{align*}

(1)3. Case: OP\_FRAC\_PERM.

Because substitution avoids capture,

\( \langle \theta(\sigma), \theta(\text{fun} f c \to v [f]) \rangle \to \langle \theta(\sigma' [f c/f]), \theta(v [f c/f]) \rangle \).

(1)4. The rest of the cases are parametric in their use of fractional permission variables and so will take the same step after any substitution.

(1)5. Corollary: If \( \langle \sigma [f c/f_1], e [f c/f_1] \rangle \to^n \langle \sigma_2, e_2' \rangle \) and \( \langle \sigma [f c/f_2], e [f c/f_2] \rangle \to^n \langle \sigma_2, e_2' \rangle \),

then \( \exists \sigma, e'. \sigma_1 = \sigma [f c/f_1] \land \sigma_2 = \sigma [f c/f_2] \land e_1 = e' [f c/f_1] \land e_2' = e' [f c/f_2] \).

D Soundness

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

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. \( \Theta = \text{dom}(\theta) \)
   b. \( \delta \in L_k[\Delta] \theta. \)
   c. \( (\varsigma, \gamma) \in L_k[\Gamma] \theta. \)

3. W.l.o.g., all variables are distinct, hence \( \theta, \delta \) and \( \gamma \) (as substitutions defined recursively over expressions) is irrelevant.

PROVE: \( (\varsigma, \theta(\delta(\gamma(\epsilon)))) \in C_k[\theta(t)]. \)

ASSUME: Arbitrary \( j < k \) and \( \sigma_r \), such that \( \zeta \ast \varsigma \) defined.

SUFFICES: \( \langle \sigma + \sigma_r, e \rangle \to^{j} \text{err} \lor \exists \sigma_f, e_f. \langle \sigma + \sigma_r, e \rangle \to^{j} \langle \sigma_f + \sigma_r, e_f \rangle \land (e_f) \) is a value \( \Rightarrow (\varsigma_j \ast \varsigma_r, e_f) \in V_{k-j}[t]. \)

SUFFICES: By C.1, to show \( \langle \sigma, e \rangle \to^{j} \text{err} \lor \exists \sigma_f, e_f. \langle \sigma, e \rangle \to^{j} \langle \sigma_f, e_f \rangle \land (e_f) \) is a value \( \Rightarrow (\varsigma_f, e_f) \in V_{k-j}[t]. \)

(1)1. Case: Ty\_LET.

(2)1. By induction,
   1. \( \forall k. \kappa[\Theta; \Delta; \Gamma \vdash e : t] \)
   2. \( \forall k. \kappa[\Theta; \Delta; \Gamma'; x : t \vdash e' : t'] \).
(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 \((\varsigma_e, \theta(\delta(\gamma_e(e)))) \in \mathcal{C}_k[\theta(t)]\).

(2.5) By definition of \(\mathcal{C}_k[\ ]\) and (2.2), we instantiate with \(j\) and \(\sigma_e = \sigma_e'\) to conclude that
\(\theta(\sigma), \theta(\delta(\gamma(e)))\) either takes \(j\) steps to \textbf{err} or another heap-and-expression \(\langle \sigma_f, e \rangle\).

(2.6) Case: \(j\) steps to \textbf{err}
By Op_Context_Err, the whole expression reduces to \textbf{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 \(\exists i \leq j\), \((\varsigma_e, v_1) \in \mathcal{V}_{k-i}, \theta(t_1) \subseteq \mathcal{V}_{k-j}, \theta(t_1)\) by C.3 and C.5.
So, Op_Context runs \(i\) times, and then we have the following.
SUFFICES: \((\varsigma_e * \varsigma_e, \text{let } x = v \text{ in } \theta(\delta(\gamma(e'))) \in \mathcal{C}_{k-i}[\theta(t')]\) by C.4 i times.
SUFFICES: \((\varsigma_e * \varsigma_e, \theta(\delta(\gamma(e')))[x/v]) \in \mathcal{C}_{k-i-1}[\theta(t')]\) by C.4.

(2.9) By C.5, \((\varsigma_e, \gamma_e'[x \mapsto v]) \in \mathcal{L}_k[\Gamma', x : t] \subseteq \mathcal{L}_{k-1}[\Gamma', x : t] \theta\).

(2.10) Instantiate 2 of step (2.1) with \(k - i - 1, \theta, \delta, \gamma_e'[x \mapsto v], \sigma_e'\) to conclude
\((\varsigma_e, \theta(\delta(\gamma_e'[x \mapsto v](e')))) \in \mathcal{C}_{k-1}[\theta(t')]\).

(2.11) By 3, we have \(\theta(\delta(\gamma_e(e')))[x/v] = \theta(\delta(\gamma_e[x \mapsto v](e')))\) and
by C.1 we conclude \((\varsigma_f * \varsigma_e', \theta(\delta(\gamma_e))[x/v]) \in \mathcal{C}_{k-1}[\theta(t')]\)

(1.2) Case: TY_PAIR_ELIM
Proof sketch: Similar to TY_LET, but with the following key differences.

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

(2.2) SUFFICES: \((\varsigma_e, \theta(\delta(\gamma(e')))) \in \mathcal{C}_{k-1}[\theta(t')]\) by C.4 i + 1 times.

(2.3) By C.5, \((\varsigma_e, \gamma_e'[a \mapsto v_1, b \mapsto v_2]) \in \mathcal{L}_k[\Gamma', a : t_1, b : t_2] \theta \subseteq \mathcal{L}_{k-1}[\Gamma', a : t_1, b : t_2] \theta\).

(2.4) Instantiate \(k-1-1[\Theta; \Delta; \Gamma'; a : t_1, b : t_2 \vdash e' : t']\) 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 \((\varsigma_e, \theta(\delta(\gamma_e[a/v_1][b/v_2])))) \in \mathcal{C}_{k-1}[\theta(t')]\).

(1.3) Case: TY_BANG_ELIM
Proof sketch: Similar to TY_LET, but with the following key differences.

(2.1) When \((\varsigma_f, v) \in \mathcal{V}_{k-1}[\theta(t)]\), since \(\mathcal{V}_{k-1}[\theta(t)] = \mathcal{V}_{k-i}[\theta(t)]\),
we have \(\varsigma_f = \emptyset\) and \(v = \text{Many } v'\) for some \((\emptyset, v') \in \mathcal{V}_{k-i}[\theta(t)]\).
(2.2) SUFFICES: \( \langle \omega', \text{let Many } x = \text{Many } v' \text{ in } \theta(\delta(\gamma'(e')))) \in C_{k-i}[[\theta(t)]] \).

(2.3) SUFFICES: \( \langle \omega', \theta(\delta(\gamma'(e'))) [[x/v]] \in C_{k-i-1}[[\theta(t)]] \) by C.4 \( i + 1 \) times.

(2.4) Instantiate \( k_{i-1} \Theta; \Delta, x : t, \Gamma' \vdash e' : t' \) with \( \theta, \delta, e' = \delta[x \mapsto v'], \gamma', \sigma' \).

(2.5) By 3, \( \langle \omega', \theta(\delta(\gamma'(e'))) [[x/v]] \in C_{k-i-1}[[\theta(t)]] \).

(1.4) CASE: TY_UNIT_ELIM.

PROOF SKETCH: Similar to TY.LET, but with the following key differences.

(2.1) When \( \langle \varsigma, v \rangle \in V_{k-i}[\text{unit}] \), we have \( \varsigma_f = \emptyset \) and \( v = () \).

(2.2) SUFFICES: \( \langle \omega', \theta(\delta(\gamma'(e'))) \in C_{k-i-1}[[\theta(t)]] \) by C.4 \( i + 1 \) times.

(2.3) By C.5, \( \langle \omega', \gamma' \rangle \in L_k[\Gamma'] \subseteq L_{k-i-1}[\Gamma'] \).

(2.4) Instantiate \( k_{i-1} \Theta; \Delta; \Gamma' \vdash e' : t' \) with \( \theta, \delta, \gamma', \sigma' \).

(2.5) By 3 \( \langle \omega', \theta(\delta(\gamma'(e'))) \in C_{k-i-1}[[\theta(t)]] \).

(1.5) CASE: TY_BOOL_ELIM.

PROOF SKETCH: Similar to TY_UNIT_ELIM but with \( \text{Op}_\text{If}_{\text{True}, \text{False}} \), \( \varsigma_f = \emptyset \)

and \( v = \text{Many true} \) or \( v = \text{Many false} \).

(1.6) CASE: TY_BANG_INTRO.

(2.1) 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.

(2.2) SUFFICES: \( \langle \emptyset, \text{Many } \theta(\delta(\gamma'(v))) \in C_k[[\theta(t)]] \) by 2c \( (\varsigma = \emptyset, \gamma = []) \).

(2.3) Instantiate \( k[[\Theta; \Delta; \cdot \vdash v : t]] \) with \( \theta, \delta, \gamma = [], \sigma = \emptyset \) to obtain \( \langle \emptyset, \theta(\delta(\gamma(v))) \in C_k[[\theta(t)]] \).

(2.4) Instantiate \( \langle \emptyset, \theta(\delta(v)) \rangle \in C_k[[\theta(t)]] \) with \( j = 0, \sigma_r = \emptyset \) and C.3 \( \theta(\delta(v)) \) is a value),

to conclude \( \langle \emptyset, \theta(\delta(v)) \rangle \in V_k[[\theta(t)]] \).

(2.5) By definition of \( V_k[[\theta(t)]] \), C.3 and C.2 we have \( \langle \emptyset, \text{Many } \theta(\delta(v)) \rangle \in C_k[[\theta(t)]] \).

(1.7) CASE: TY_PAIR_INTRO.

(2.1) By 2c, 3 and C.7, we know there exists the following (for all \( k \)):

1. \( \langle s_1, \gamma_1 \rangle \in L_k[\Gamma_1] \)
2. \( \langle s_2, \gamma_2 \rangle \in L_k[\Gamma_2] \)
3. \( \gamma = \gamma_1 \cup \gamma_2 \)
4. \( \sigma = \sigma_1 + \sigma_2 \).

(2.2) By induction,

1. \( \forall k. k[[\Theta; \Delta; \Gamma_1 \vdash e_1 : t_1]] \)
2. \( \forall k. k[[\Theta; \Delta; \Gamma_2 \vdash e_2 : t_2]] \).

(2.3) Instantiate the first with \( k, \theta, \delta, \gamma_1, \sigma_1 \).
(2.4) By that and (2.1), 
\((\varsigma_1, \theta(\delta(\gamma_1(e_1)))) = (\varsigma_1, \theta(\delta(\gamma(e_1)))) \in \mathcal{C}_k[\theta(t)]\). 

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

(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 \(\exists i_1 \leq j. (\varsigma_{1f}, v_1) \in \mathcal{V}_{k-i_1}[\theta(t_1)] \subseteq \mathcal{V}_{k-j}[\theta(t_1)]\) by C.3 and C.5. 
So, Op_Context runs \(i_1\) times, and then we have the following. 
SUFFICES: By C.4, \((\varsigma_{1f} \times \varsigma_{2f}, (v_1, e_2)) \in \mathcal{C}_{k-i_1}[\theta(t_1) \otimes \theta(t_2)]\). 

(2.9) Instantiate the second IH with \(k, \theta, \delta, \gamma_2, \sigma_2\). 

(2.10) So, \((\theta(\sigma_1 + \sigma_2), \theta(\delta(\gamma_2(e_2))))\) either takes \(j\) steps to err or a heap-and-expression 
\((\sigma_{2f}, e_{2f})\). 

(2.11) Case: \(j\) steps to err 
By Op_Context_Err, the whole expression reduces to err in \(j < k\) steps. 

(2.12) Case: \(j\) steps to another heap-and-expression. 
If it is not a value, then Op_Context runs \(j\) times and we are done. 

(2.13) 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, \((\varsigma_{1f} \times \varsigma_{2f}, (v_1, v_2)) \in \mathcal{V}_{k-i_1-i_2}[\theta(t_1) \otimes \theta(t_2)]\). 

(2.14) 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. 

(1.8) Case: Ty_Lambda. 
SUFFICES: By C.2, to show \((\varsigma, \theta(\delta(\gamma(\text{fun} \ x : t \rightarrow e)))) \in \mathcal{V}_k[\theta(t \rightarrow t')]\). 
ASSUME: Arbitrary \(j \leq k, (\varsigma_v, v) \in \mathcal{V}_j[\theta(t)]\) such that \(\varsigma \times \varsigma_v\) is defined. 
SUFFICES: \((\varsigma \times \varsigma_v, \theta(\delta(\gamma(\text{fun} \ x : t \rightarrow e)))) \in \mathcal{C}_j[\theta(t')]\). 
SUFFICES: \((\varsigma \times \varsigma_v, \theta(\delta(\gamma(e))))[x/v]) \in \mathcal{C}_{j-1}[\theta(t')]\) by C.4. 

(2.1) By induction, \(\forall k. k[\Theta; \Delta_i \Gamma, x : t \vdash e]\). 

(2.2) Instantiate it \(j - 1, \theta, \delta, \gamma[x \mapsto v], \sigma + \sigma_v\). 

(2.3) Hence, \((\varsigma \times \varsigma_v, \theta(\delta(\gamma[x \mapsto v](e)))) \in \mathcal{C}_{j-1}[\theta(t')]\). 

(2.4) By 3, \(\theta(\delta(\gamma[x \mapsto v](e))) = \theta(\delta(\gamma(e)))[x/v]\), we are done. 

(1.9) Case: Ty_App. 

(2.1) 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)2. By induction,
1. \( \forall k \cdot k[\Theta; \Delta; \Gamma \vdash e : t' \rightarrow t] \)
2. \( \forall k \cdot k[\Theta; \Delta; \Gamma' \vdash e' : t'] \).

(2)3. Instantiate the first with \( k, \theta, \delta, \gamma_e, \sigma_e \) to conclude \( (\varsigma_e, \theta(\delta(\gamma_e))) \in C_k[\theta(t') \rightarrow \theta(t)] \).

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

(2)5. CASE: \( j \) steps to \texttt{err}
By \texttt{Op\_Context\_Err}, the whole expression reduces to \texttt{err} in \( j < k \) steps.

(2)6. CASE: \( j \) steps to another heap-and-expression.
If it is not a value, then \texttt{Op\_Context} runs \( j \) times and we are done.

(2)7. If it is, then \( \exists i_e \leq j \cdot (\varsigma_f, e_j) \in V_{k-i_e}[\theta(t') \rightarrow \theta(t)] \subseteq V_{k-j} \ldots \) by C.3 and C.5.
So, \texttt{Op\_Context} runs \( i_e \) times, and then we have the following.
SUFFICES: By C.4 \( i_e \) times, \( (\varsigma_f \ast \varsigma_{e'}, e_f e') \in C_{k-i_e}[\theta(t')] \).

(2)8. By C.5, \( (\varsigma_e, \gamma_e) \in C_k[\Gamma'] \theta \subseteq C_{k-i_e}[\Gamma] \theta \).

(2)9. So, instantiate the second IH with \( k - i_e, \theta, \delta, \gamma_e', \sigma_e' \) to conclude
\( (\varsigma_e', \theta(\delta(\gamma_e'(e')))) \in C_{k-i_e}[\theta(t')] \).

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

(2)11. CASE: \( j - i_e \) steps to \texttt{err}
By \texttt{Op\_Context\_Err}, the whole expression reduces to \texttt{err} in \( j - i_e < k - i_e \)
steps.

(2)12. CASE: \( j - i_e \) steps to another heap-and-expression.
If it is not a value, then \texttt{Op\_Context} runs \( j - i_e \) times and we are done.

(2)13. If it is, then \( \exists i_{e'} \leq j - i_e \cdot (\varsigma_f', e_{e'}e) \in V_{k-i_e-i_{e'}}[\theta(t')] \) by C.3.
So, \texttt{Op\_Context} runs \( i_{e'} \) times, and then we have the following.
SUFFICES: By C.4 \( i_{e'} \) times, \( (\varsigma_f \ast \varsigma_{e'}, e_f e_{e'}) \in C_{k-i_e-i_{e'}}[\theta(t')] \).

(2)14. Instantiate \( (\varsigma_f, e_f) \in V_{k-i_e}[\theta(t') \rightarrow \theta(t)] \) with \( k - i_e - i_{e'} \leq k - i_e \) and
\( (\varsigma_{e'}, e_{e'}) \in V_{k-i_e-i_{e'}}[\theta(t')] \), to conclude \( (\varsigma_f \ast \varsigma_{e'}, e_f e_{e'}) \in C_{k-i_e-i_{e'}}[\theta(t)] \) as needed.

(1)10. CASE: \texttt{Ty\_Gen}.

(2)1. By induction, \( \forall k \cdot k[\Theta, fc; \Delta; \Gamma \vdash e : t] \).

(2)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 \( (\theta(\sigma), \theta'(\gamma(\delta(e)))) \)
either takes \( j \) steps to \text{err} or a heap-and-expression \( \langle \sigma', e' \rangle \) (for all \( f \), by C.8).

\( \langle 2 \rangle.4. \) \text{CASE:} \( j \) steps to \text{err}.
By \text{OP_CONTEXT_Err}, whole expression reduces to \text{err} in \( j < k - 1 \) steps (for \( f = \text{fc} \)).
\( \langle 2 \rangle.5. \) \text{CASE:} \( j \) steps to another heap-and-expression.
If it is not a value, then for \( f = \text{fc} \), \text{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, \text{OP_CONTEXT} runs \( i_e \) times, and then we have the following.
SUFFICES: By C.4 \( i_e \) times, \( (\varsigma', \text{fun}fc \rightarrow e') \in \mathcal{V}_{k-i_e}[\theta(\forall fc. t)] \) (for \( f = \text{fc} \)).
\( \langle 2 \rangle.8. \) \text{ASSUME:} Arbitrary \( f' \).
SUFFICES: \( (\varsigma', e'[fc/f']) \in \mathcal{V}_{k-1-i_e}[\theta(t)[fc/f']] \) (for \( f = \text{fc} \)).
\( \langle 2 \rangle.9. \) This is true by instantiating \( \langle 2 \rangle.6 \) with \( f = f' \).

(1)11. \text{CASE: TY_SPC}.
\( \langle 2 \rangle.1. \) By induction, \( \forall k. k[\Theta; \Delta; \Gamma \vdash e : \forall fc. t] \).
\( \langle 2 \rangle.2. \) Instantiate with \( k, \theta, \delta, \gamma, \sigma \) to conclude \( (\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\theta(\forall fc. t)] \).
\( \langle 2 \rangle.3. \) Instantiate this with \( j \) and \( \emptyset \) and to conclude \( (\theta(\sigma), \theta(\delta(\gamma(e)))) \)
either takes \( j \) steps to \text{err} or a heap-and-expression \( \langle \sigma_f, e_f \rangle \).
\( \langle 2 \rangle.4. \) \text{CASE:} \( j \) steps to \text{err}.
By \text{OP_CONTEXT_Err}, the whole expression reduces to \text{err} in \( j < k \) steps.
\( \langle 2 \rangle.5. \) \text{CASE:} \( j \) steps to another heap-and-expression.
If it is not a value, then \text{OP_CONTEXT} runs \( j \) times and we are done.
\( \langle 2 \rangle.6. \) If it is, then \( \exists i_e \leq j. (\varsigma_f, e_f) \in \mathcal{V}_{k-i_e}[\theta(\forall fc. t)] \subseteq \mathcal{V}_{k-j}[\ldots] \) by C.3 and C.5.
So \( e_f \equiv \text{fun}fc \rightarrow v \) for some \( v \).
\( \langle 2 \rangle.7. \) So, \text{OP_CONTEXT} runs \( i_e \) times, and then we have the following.
SUFFICES: By C.4 \( i_e \) times, \( (\varsigma_f, \text{fun}fc \rightarrow v)[f] \) \in \( \mathcal{C}_{k-i_e}[\theta(t)[fc/f]] \).
SUFFICES: By C.4 once more, \( (\varsigma_f, v[fc/f]) \in \mathcal{C}_{k-i_e-1}[\theta(t)[fc/f]] \).
\( \langle 2 \rangle.8. \) This is true by instantiating \( \langle 2 \rangle.6 \) with \( f \) and C.2.

(1)12. \text{CASE: TY_FIX}.
SUFFICES: \( (\emptyset, \theta(\delta(\text{fix}(g, x : t, e : t')))) \in \mathcal{V}_{k}[\theta(t \rightarrow t')] \), by C.2 (\( \sigma = \{ \}, \gamma = [] \).
ASSUME: Arbitrary \( j \leq k, (\varsigma_v, v) \in \mathcal{V}_j[\theta(t)] \) (\( \varsigma = \emptyset \), so \( \varsigma \ast \varsigma_v \) is defined).
Let $\tilde{e} \equiv \theta(\delta(e))$.

Suffices: $(\varsigma, \mathbf{fix}(g, x : t, \tilde{e} : t') v) \in C_j[\theta(t')]$.

Suffices: $(\varsigma, \tilde{e}[x/v] [g/\mathbf{fix}(g, x : t, \tilde{e} : t')]) \in C_{j-1}[\theta(t')]$ by C.4.

(2.1) By induction, $\forall k. k[\Theta; \Delta, g : t \leadsto t'; x : t \vdash e : t']$.

(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 $(\emptyset, \mathbf{fix}(g, x : t, \tilde{e} : t')) \in V_{j-1}[\theta(t \leadsto t')]$.

(3.1) Again by induction (over $k$), $(\emptyset, \mathbf{fix}(g, x : t, \tilde{e} : t')) \in C_{j-1}[\theta(t \leadsto t')]$.

(3.2) Instantiate this with $j = 0$ and $\emptyset$ and we are done.

(2.4) We have $(\varsigma, v) \in V_{j-1}[\theta(t)]$ by assumption and C.5.

(2.5) So we conclude $(\varsigma, \theta(\delta'(\gamma(e)))) \in C_{j-1}[\theta(t')]$ as required.

(1.13) Case: Ty__Var_Lin.

Prove: $(\varsigma, \theta(\delta(\gamma(x)))) \in C_k[\theta(t)]$.

(2.1) $\Gamma = \{ x : t \}$ by assumption of Ty__Var_Lin.

(2.2) Suffices: $(\varsigma, \gamma(x)) \in C_k[\theta(t)]$ by 3 (and $\delta$ irrelevant).

(2.3) By 2c, there exist $(\varsigma_x, v_x) \in V_k[\theta(t)]$, such that $\varsigma = \varsigma_x$ and $\gamma = [x \mapsto v_x]$.

(2.4) Hence, $(\varsigma_x, v_x) \in C_k[\theta(t)]$, by C.2.

(1.14) Case: Ty__Var.

Prove: $(\varsigma, \theta(\delta(\gamma(x)))) \in C_k[\theta(t)]$.

(2.1) $x : t \in \Delta$ and $\Gamma = \emptyset$ by assumption of Ty__Var.

(2.2) Suffices: $(\emptyset, \delta(x)) \in C_k[\theta(t)]$ by 3.

(2.3) By 2b, there exists $v_x$ such that $(\emptyset, v_x) \in V_k[\theta(t)]$ (and $\delta$ irrelevant and $\gamma$ empty).

(2.4) Hence, $(\emptyset, v_x) \in C_k[\theta(t)]$, by C.2.

(1.15) Case: Ty__Unit_Intro.

True by C.2 and definition of $V_k[\mathbf{unit}]$.

(1.16) Case: Ty__Bool__True, Ty__Bool__False, Ty__Int__Intro, Ty__Elt__Intro.

Similar to Ty__Unit__Intro.

D.1 Well-formed types

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

$fc \in \Theta \vdash fc \text{ Perm}$ WF_PERM_VAR
$\Theta \vdash 1 \text{Perm}$ \hspace{1cm} $\text{WF\_PERM\_ZERO}$

$\Theta \vdash f \text{Perm}$ \hspace{1cm} $\text{WF\_PERM\_SUCC}$

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

$\Theta \vdash t \text{Type}$ \hspace{1cm} Well-formed types

$\Theta \vdash \text{unit} \text{Type}$ \hspace{1cm} $\text{WF\_TYPE\_UNIT}$

$\Theta \vdash \text{bool} \text{Type}$ \hspace{1cm} $\text{WF\_TYPE\_BOOL}$

$\Theta \vdash \text{int} \text{Type}$ \hspace{1cm} $\text{WF\_TYPE\_INT}$

$\Theta \vdash \text{elt} \text{Type}$ \hspace{1cm} $\text{WF\_TYPE\_ELT}$

$\Theta \vdash f \text{Perm}$ \hspace{1cm} $\Theta \vdash f \text{arr} \text{Type}$ \hspace{1cm} $\text{WF\_TYPE\_ARRAY}$

$\Theta \vdash t \text{Type}$ \hspace{1cm} $\Theta \vdash !t \text{Type}$ \hspace{1cm} $\text{WF\_TYPE\_BANG}$

$\Theta, fc \vdash t \text{Type}$ \hspace{1cm} $\Theta \vdash \forall fc.t \text{Type}$ \hspace{1cm} $\text{WF\_TYPE\_GEN}$

$\Theta \vdash t \text{Type}$ \hspace{1cm} $\Theta \vdash t' \text{Type}$ \hspace{1cm} $\Theta \vdash t \otimes t' \text{Type}$ \hspace{1cm} $\text{WF\_TYPE\_PAIR}$

$\Theta \vdash t \text{Type}$ \hspace{1cm} $\Theta \vdash t' \text{Type}$ \hspace{1cm} $\Theta \vdash t \rightarrow t' \text{Type}$ \hspace{1cm} $\text{WF\_TYPE\_LOLLY}$

**E** NumLin Grammar

\[
m ::= M \hspace{1cm} \text{matrix expressions}
| \ M \hspace{1cm} \text{matrix variables}
| \ m + m' \hspace{1cm} \text{matrix addition}
| \ m m' \hspace{1cm} \text{matrix multiplication}
| \ (m) \ S
\]

\[
f ::= fc \hspace{1cm} \text{fractional permission}
| \ 1 \hspace{1cm} \text{whole permission}
| \ \frac{1}{2} f
\]
$t ::= \begin{array}{l}
\text{unit} \quad \text{linear type} \\
\text{bool} \quad \text{unit}
\text{int} \quad \text{boolean (true/false)}
\text{elt} \quad \text{63-bit integers}
\text{farr} \quad \text{array element}
\text{fmat} \quad \text{arrays}
\forall fc. t \quad \text{matrices}
\neg t \quad \text{multiple-use type}
\forall fc. \quad \text{frac. perm. generalisation}
\text{t} \otimes t' \quad \text{pair}
\text{t} \rightarrow t' \quad \text{linear function}
\text{(t)} \quad \text{S} \quad \text{parentheses}
\end{array}$

$p ::= \begin{array}{l}
\text{not} \quad \text{boolean negation}
(+) \quad \text{integer addition}
(\text{\textminus}) \quad \text{integer subtraction}
(\ast) \quad \text{integer multiplication}
(/) \quad \text{integer division}
(=) \quad \text{integer equality}
(<) \quad \text{integer less-than}
(+.) \quad \text{element addition}
(-.) \quad \text{element subtraction}
(\ast.) \quad \text{element multiplication}
(/.) \quad \text{element division}
(=.) \quad \text{element equality}
(<.) \quad \text{element less-than}
\text{set} \quad \text{array index assignment}
\text{get} \quad \text{array indexing}
\text{share} \quad \text{share array}
\text{unshare} \quad \text{unshare array}
\text{free} \quad \text{free array}
\text{array} \quad \text{Owl: make array}
\text{copy} \quad \text{Owl: copy array}
\text{sin} \quad \text{Owl: map sine over array}
\text{hypot} \quad \text{Owl: } x_i := \sqrt{x_i^2 + y_i^2}
\text{asum} \quad \text{BLAS: } \sum |x_i|
\text{axpy} \quad \text{BLAS: } x := \alpha x + y
\text{dot} \quad \text{BLAS: } x \cdot y
\text{rotmg} \quad \text{BLAS: see its docs}
\text{scal} \quad \text{BLAS: } x := \alpha x
\text{amax} \quad \text{BLAS: } \text{argmax } i : x_i
\text{setM} \quad \text{matrix index assignment}
\text{getM} \quad \text{matrix indexing}
\end{array}$
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: $C := \alpha A^T A^T + \beta C$

$e ::= \begin{align*}
\ & p \quad \text{primitives} \\
\ & x \quad \text{variable} \\
\ & () \quad \text{unit introduction} \\
\ & \text{true} \quad \text{true} \\
\ & \text{false} \quad \text{false} \\
\ & k \quad \text{integer} \\
\ & l \quad \text{heap location} \\
\ & el \quad \text{array element} \\
\ & \text{Many } e \quad \!-\text{introduction} \\
\ & \text{fun } f : c \rightarrow e \quad \text{frac. perm. abstraction} \\
\ & (v, v') \quad \text{pair introduction} \\
\ & \text{fun } (g, x : t, e : t') \quad \text{bind } x \text{ in } e \quad \text{abstraction} \\
\ & (v) \quad S \quad \text{parentheses} \\
\ \end{align*}$

$v ::= \begin{align*}
\ & p \quad \text{primitives} \\
\ & x \quad \text{variable} \\
\ & () \quad \text{unit introduction} \\
\ & \text{true} \quad \text{true} \\
\ & \text{false} \quad \text{false} \\
\ & k \quad \text{integer} \\
\ & l \quad \text{heap location} \\
\ & el \quad \text{array element} \\
\ & \text{Many } v \quad \!-\text{introduction} \\
\ & \text{fun } f : c \rightarrow e \quad \text{frac. perm. abstraction} \\
\ & (v, v') \quad \text{pair introduction} \\
\ & \text{fun } x : t \rightarrow e \quad \text{bind } x \text{ in } e \quad \text{abstraction} \\
\ & \text{fix } (g, x : t, e : t') \quad \text{bind } g \cup x \text{ in } e \quad \text{fixpoint} \\
\ & (v) \quad S \quad \text{parentheses} \\
\ \end{align*}$
23:40  NumLin: Linear Types for Linear Algebra

\[
\begin{align*}
C & ::= \\
& \text{evaluation contexts} \\
& \text{let } x = [-] \text{ in } e \quad \text{let binding} \\
& \text{let } () = [-] \text{ in } e \quad \text{unit elimination} \\
& \text{if } [-] \text{ then } e_1 \text{ else } e_2 \quad \text{if} \\
& \text{Many } [-] \quad \text{!-introduction} \\
& \text{let Many } x = [-] \text{ in } e \quad \text{!-elimination} \\
& \text{fun } fc \rightarrow [-] \quad \text{frac. perm. abstraction} \\
& [-][f] \quad \text{frac. perm. specialisation} \\
& (v, [-]) \quad \text{pair introduction} \\
& \text{let } (a, b) = [-] \text{ in } e \quad \text{pair elimination} \\
& [-]e \quad \text{application} \\
& v[-] \quad \text{application} \\
\end{align*}
\]

\[
\begin{align*}
\Theta & ::= \\
& \text{fractional permission environment} \\
& \cdot \\
& \Theta, fc \\
\end{align*}
\]

\[
\begin{align*}
\Gamma & ::= \\
& \text{linear types environment} \\
& \cdot \\
& \Gamma, x : t \\
& \Gamma, \Gamma' \\
\end{align*}
\]

\[
\begin{align*}
\Delta & ::= \\
& \text{intuitionistic types environment} \\
& \cdot \\
& \Delta, x : t \\
\end{align*}
\]

\[
\begin{align*}
\sigma & ::= \\
& \text{heap (multiset of triples)} \\
& \{\} \quad \text{empty heap} \\
& \sigma + \{l \mapsto m_{i_1, i_2}\} \quad \text{location } l \text{ points to matrix } m \\
\end{align*}
\]

\[
\begin{align*}
\text{Config} & ::= \\
& \text{result of small step} \\
& (\sigma, e) \quad \text{heap and expression} \\
& \text{err} \quad \text{error}
\end{align*}
\]
Desugaring NumLin

\[ x[e] \Rightarrow \text{get } x(e) \quad \text{(similarly for matrices)} \]
\[ x[e_1] := e_2 \Rightarrow \text{set } x(e_1)(e_2) \quad \text{(similarly for matrices)} \]

\[
\begin{align*}
\text{pat} &::= () \mid x \mid !x \mid \text{Many } \text{pat} \mid \langle \text{pat}, \text{pat} \rangle \\
\text{let } !x = e_1 \text{ in } e_2 &\Rightarrow \text{let Many } x = e_1 \text{ in } e_2 \\
\text{let Many}\langle \text{pat}_x \rangle = e_1 \text{ in } e_2 &\Rightarrow \text{let Many } x = \text{Many } (\text{Many } x) \text{ in } e_2 \\
\text{let } \langle \text{pat}_a, \text{pat}_b \rangle = e_1 \text{ in } e_2 &\Rightarrow \text{let } \langle a, b \rangle = a \cdot b \text{ in } \text{let } \langle \text{pat}_a \rangle = a \text{ in } e_2 \\
\text{fun } \langle \text{pat}_x \rangle : t \to e &\Rightarrow \text{fun } (x : t) \to \text{let } \langle \text{pat}_x \rangle = x \text{ in } e \\
\end{align*}
\]

\[
\begin{align*}
\text{arg} &::= \langle \text{pat} \rangle : t \mid \text{'x } \text{(fractional permission variable)} \\
\text{fun } \langle \text{arg}_1..n \rangle \to e &\Rightarrow \text{fun } \langle \text{arg}_1 \rangle \to .. \text{fun } \langle \text{arg}_n \rangle \to e \\
\text{let } f \langle \text{arg}_1..n \rangle = e_1 \text{ in } e_2 &\Rightarrow \text{let } f = \text{fun } \langle \text{arg}_1..n \rangle \to e_1 \text{ in } e_2 \\
\text{let !f } \langle \text{arg}_1..n \rangle = e_1 \text{ in } e_2 &\Rightarrow \text{let Many } f = \text{Many } (\text{fun } \langle \text{arg}_1..n \rangle \to e_1) \text{ in } e_2 \\
\text{fixpoint} &\equiv \text{fix } (f, x : t, \text{fun } \langle \text{arg}_1..n \rangle \to e_1 : t') \\
\text{let rec } f (x : t) \langle \text{arg}_1..n \rangle : t' = e_1 \text{ in } e_2 &\Rightarrow \text{let } f = \text{fixpoint } \text{in } e_2 \\
\text{let rec } !f (x : t) \langle \text{arg}_1..n \rangle : t' = e_1 \text{ in } e_2 &\Rightarrow \text{let Many } f = \text{Many } \text{fixpoint } \text{in } e_2
\end{align*}
\]
module Arr = Owl.Dense.Ndarray.D

type z = Z

type 'a s = Succ

type 'a arr = A of Arr.arr[@@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 *
      val share : 'a s arr -> 'a s arr *
      val unshare : 'a s arr -> 'a s arr *
      val free : z arr -> unit
        (** Owl *)
      val array : int bang -> z arr
      val copy : 'a arr -> 'a arr *
      val sin : z arr -> z arr
      val hypot : z arr -> 'a arr -> 'a arr *
        (** Level 1 BLAS *)
      val asum : 'a arr -> 'a arr *
      val axpy : float bang -> 'a arr -> z arr -> 'a arr *
      val dot : 'a arr -> 'b arr -> ('a arr * 'b arr) *
      val scal : float bang -> z arr -> z arr
      val amax : 'a arr -> 'a arr *
        (** Matrix, some omitted for brevity *)
      val matrix : int bang -> int bang -> z mat
      val eye : int bang -> int bang -> z mat
      val copy_mat : 'a mat -> 'a mat *
      val copy_mat_to : 'a mat -> z mat -> 'a mat *
      val size_mat : 'a mat -> 'a mat *
      val transpose : 'a mat -> 'a 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) *
      val symm : bool bang -> float bang -> 'a mat -> float bang -> float bang -> 'a mat *
      val gesv : z mat -> z mat -> 'b mat -> float bang -> z mat -> ('a mat * 'b mat) *
      val posv : z mat -> z mat -> 'b mat -> float bang -> z mat -> 'a mat *
      val potrs : 'a mat -> z mat -> 'a mat *
      val syrk : bool bang -> float bang -> 'a mat -> float bang -> z mat -> 'a mat *
    end
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) (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 16 OCaml code for a Kalman filter, generated (at compile time) from the code in Figure 8, 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 17 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.