

Formally Verified Cost of the Parallel Prefix Sum Algorithm

Andrew Zhou

August 2025

1 Introduction

Interactive theorem provers such as Agda are well-suited to verify behavioral properties of programs - for example, proving that merge sort and insertion sort both produce a sorted permutation of the original list [9]. Thus, mechanical verification of the behavior of programs is relatively straightforward. Ideally, we would like to extend this verification of behavior to also be able to mechanically verify a program's resource usage, or cost. Just as mechanical verification of behavior allows one to identify bugs in a program's behavior, mechanical verification of cost would allow one to identify bugs in a program's supposed runtime. However, verifying cost internally is difficult, essentially because one cannot conventionally have a function `cost : bool → nat` that computes the cost of its input. Furthermore, even if a program was annotated with its cost, this annotation cannot be allowed to interfere with the behavior of the program.

1.1 The Cost-Aware Logical Framework

The language of cost-aware logical frameworks (**Calf**) is a dependent type theory based on Levy's call-by-push-value that allows us to mechanically verify both the intensional (cost) and extensional (behavior) aspects of programs [8]. The primary feature of **Calf** that allows this is the *phase distinction*.

1.1.1 Call-by-push-value

In the framework of Levy's call-by-push-value [7], types are classified into two categories:

1. *Value*, or positive, types. This includes types representing pure data such as `5 : nat`, as well as suspensions, denoted as $U(X)$.
2. *Computation*, or negative, types, which are allowed to generate effects. This includes the type of return values `ret(a) : F(A)`, as well as the function type $A \rightarrow B$. Note that all functions in call-by-push-value are computations.

Calf is based on call-by-push-value in order to accomodate cost as an effect. In particular, in **Calf** programs are instrumented with the step-counting effect `stepc e` which annotates expression `e` with cost `c`. Because we treat cost as an effect, we obtain a theory for composition of cost through the composition of these effects. Furthermore, we can state that insertion sort and merge sort are not equivalent as functions, because they have different costs and thus produce different effects. However, under the *phase distinction*, we are still able to prove extensional equivalence.

1.1.2 The Phase Distinction

As previously stated, cost annotations cannot be allowed to interfere with the behavior of the program. **Calf** accomplishes this separation between cost and behavior through the *phase distinction*. The phase distinction is represented by a proposition \bigcirc , also referred to as the open modality. When \bigcirc is inhabited/true, then the step counting function, **step**, is equivalent to the identity function. This allows us to verify that two programs with the same behavior and different runtimes are extensionally equivalent, even though they are not equivalent functions. Going back to our sorting example, suppose we have functions **mergeSort** and **insertionSort** which implement merge sort and insertion sort, respectively. We can prove that $\bigcirc(\text{mergeSort} \equiv \text{insertionSort})$, because we have that \bigcirc is inhabited. However, note that when \bigcirc is false, this statement does not hold, as **mergeSort** and **insertionSort** have different costs.

1.1.3 Decalf, inequational reasoning, and monotonicity

Decalf[\[5\]](#) extends **Calf** to operate on inequational reasoning, instead of strict equality. In particular, this is useful when we are unable to prove a strict upper bound, and instead must prove a weak upper bound. When we combine the inequational reasoning features of **Decalf** with the parallel cost monoid, we obtain further features to reason about parallel cost via monotonicity. In particular, because the ordering \leq is a preorder, all functions are monotone with respect to \leq , and we are able to reason about function application in a similar manner to congruence relations. For example, if we have that $x \leq y$, then by monotonicity we have $f\ x \leq f\ y$. Formally, we write $\leq_{\text{mono}}: \{A, B, a, b\} (f : \text{val}(A) \rightarrow \text{val}(B)) \rightarrow a \leq b \rightarrow f(a) \leq f(b)$.

Monotonicity gives us a few powerful tools in reasoning about cost. Firstly, if we have $c \leq c'$, then we are able to say $\text{step}^c(e) \leq \text{step}^{c'}(e)$. Note we are also able to use monotonicity for simplifying arithmetic operations, such as $+$. This is particularly useful in simplifying costs that are being added together. Furthermore, because we have reflexivity of the preorder relation \leq , we are able to take e' as itself. Mathematically, this means we can trivially say $e \leq e$, while logically, we are able to say that “ e costs whatever it costs.”

1.2 The Parallel Prefix Sum Algorithm

One particularly interesting algorithm, especially with regards to parallelism, is the parallel prefix sum algorithm. Specifically, when the prefix sum is generalized to operate on any binary operation, the resulting higher-ordered function is known as **scan** [\[2\]](#). Practically, the parallel nature of **scan** lends itself well to distributed and high performance computing applications. For example, **scan** can be implemented in CUDA in order to make use of a GPU’s data parallelism[\[6\]](#).

1.2.1 Sequences

Key to the implementation of **scan** is the **sequence** abstract data type.

Definition 1.1 (Sequences). A sequence is an abstract data type defined by a mapping from $\{0, 1, \dots, n\}$ to α , where α is some arbitrary type. Intuitionally, sequences are somewhat similar to arrays and vectors, in that they are all enumerated collections of elements.[\[1\]](#)

What makes sequences important to the implementation of **scan** is that sequences, where the underlying data type is array segments, are particularly amenable to parallelism. For example, a primitive sequence operation is **tabulate** : $\text{nat} \rightarrow (\text{nat} \rightarrow 'a) \rightarrow 'a \text{ seq}$, where **tabulate** $f\ n$ returns the sequence $\langle f(0), \dots, f(n-1) \rangle$. To construct the sequence produced by **tabulate** $f\ n$, we

allocate a fresh array of n elements, and evaluate f at each index in the array. Note that as each call to f is independent, these calls to f can be evaluated in parallel.

1.2.2 Scan Behavior

Definition 1.2 (Prefix Sum). The *prefix sum* of a sequence S with length n is a tuple (S', sum) , where each index i in S' is the sum of the length i prefix of S , and sum is the overall sequence sum of S . `scan` essentially generalizes the prefix sum to operate on any binary function.

Specifically, `scan` : $(\text{'a} * \text{'a} \rightarrow \text{'a}) \rightarrow \text{'a} \rightarrow \text{'a seq} \rightarrow (\text{'a seq} * \text{'a})$ takes in three arguments: a function f , an initial element e , and an initial sequence S , and returns the computed prefix sums on f for that sequence. Notably, the parallel version of `scan` imposes several requirements on `scan`. In particular, f and e are required to form a monoid on 'a . This is because the parallel implementation requires that f is associative and that e is an identity element for f .

`scan` is specified to return the prefix sums of the sequence for the given function f . For example, suppose we have the expression `scan (op +) 0 [4, 2]`. Then this should evaluate to $([\text{sum (op +) 0 []}, \text{sum (op +) 0 [4]}, \text{sum (op +) 0 [4, 2]}], \text{sum (op +) 0 [4, 2]})$, where `sum` is some function which computes the sequence sum according to f . In turn, this should evaluate to $([0, 4], 6)$.

1.2.3 Parallelism of Scan, and Analysis in Calf

What makes `scan` particularly interesting from an algorithmic perspective is that there is significant opportunity for asymptotic speedup from parallelism. In particular, there are three common implementations of `scan`, which all have different *work* and *span*.

Definition 1.3 (Work and Span). *Work* refers to the sequential cost, where we assume access to only one processor, and all operations must be done sequentially on that processor. *Span* refers to the parallel cost, where we assume access to an infinite number of parallel processors under the fork-join model of parallelism.

1.2.4 The Parallel Cost Monoid

When **Calf** is instantiated with the *parallel cost monoid*, our step-counting annotations take the form $\text{step}^{(w,s)}$, where w represents the work and s represents the span. This allows us to formally reason about both work and span within the framework of **Calf**.

Suppose we have the expression $\text{step}^{(w_1, s_1)}(e_1); \text{step}^{(w_2, s_2)}(e_2)$ where the two expressions $\text{step}^{(w_1, s_1)}(e_1)$ and $\text{step}^{(w_2, s_2)}(e_2)$ are evaluated sequentially. Then this is equivalent to $\text{step}^*((w_1, s_1) \oplus (w_2, s_2))$, where \oplus represents sequential cost composition and we define $\text{step}^*(w, s) := \text{step}^{(w,s)}(\text{ret triv})$ to simplify the evaluation of multiple steps. Therefore this is in turn equivalent to $\text{step}^*(w_1 + w_2, s_1 + s_2)$.

However, under the parallel cost monoid, we also have access to the operation $\parallel : F(A) \times F(B) \rightarrow F(A \times B)$, which represents parallel composition. Now, suppose that we have the expression $\text{step}^{(w_1, s_1)}(e) \parallel \text{step}^{(w_2, s_2)}(e)$ where the two expressions $\text{step}^{(w_1, s_1)}(e)$ and $\text{step}^{(w_2, s_2)}(e)$ are evaluated in parallel. Then this is equivalent to $\text{step}^*((w_1, s_1) \otimes (w_2, s_2))$, where \otimes represents parallel cost composition. We define \otimes as $(w_1, s_1) \otimes (w_2, s_2) := (w_1 + w_2, s_1 \sqcup s_2)$ (where \sqcup is the maximum operation). While the left projection, representing work, remains the same as in sequential cost composition, the right projection, representing span, is replaced by the maximum operation. Thus this is ultimately equivalent to $\text{step}^*(w_1 + w_2, s_1 \sqcup s_2)$.

Note that we are also able to use monotonicity to aid our reasoning about parallel programs. In particular, suppose we have $e_1 \parallel e_2$. Then, by monotonicity, if we have that $e_1 \leq e'_1$ and $e_2 \leq e'_2$,

we are able to say that $e_1 \parallel e_2 \leq e'_1 \parallel e'_2$. Furthermore, because of reflexivity, we can take e'_1 to be e_1 itself, or vice versa, to only simplify one side of the parallel operation at a time.

1.2.5 Implementations of scan

As previously stated, there are three common implementations of **scan**, each with different asymptotic work and span. The three implementations are **bruteforce scan**, **divide-and-conquer scan**, and **contraction scan**.

First, there is **bruteforce scan**, where each element of the result is computed sequentially in order. As this is a purely sequential algorithm, this is $O(n)$ work and span. Next, there is **divide-and-conquer scan**. Divide-and-conquer is an approach that benefits from parallelism, as we can compute the result for both halves of the sequence in parallel - resulting in a span bound of $O(\log n)$. However, because at each level we need to do $O(n)$ work to recombine the sequences, and because we have two recursive calls at each level, we have the work recurrence $W(n) = 2W(n/2) + O(n)$, which solves to a work bound of $O(n \log n)$ - worse than the bruteforce approach.

Finally, there is the contraction approach. Contraction algorithms generally consist of a contraction step, where we create a smaller instance of our problem, recursively solve that smaller instance, and then expand out that recursive result to solve our original problem. This gives us a work recurrence $W(n) = W(n/2) + O(n)$, and a span recurrence of $S(n) = S(n/2) + O(1)$. Notably, this preserves the improved span bound seen from the divide-and-conquer approach of $O(\log n)$, while also improving our work to be $O(n)$.

Implementation	Work	Span
Bruteforce	$O(n)$	$O(n)$
Divide-and-conquer	$O(n \log n)$	$O(\log n)$
Contraction	$O(n)$	$O(\log n)$

Figure 1: Summary of asymptotic costs of **scan** implementations

1.3 Representation of Sequences

Because sequences are an abstract data type, it is necessary to define a concrete implementation for sequences when writing our implementation and cost proof. Furthermore, we must consider our model of cost.

For our formalization of sequences in Agda, we represented sequences as lists. To achieve the necessary parallelism, a common sequence implementation is to use arrays. However, there are a few advantages to representing sequences as lists for our proof. Due to the naturally recursive construction of lists, they are much easier to reason about than arrays. Furthermore, we are able to use Agda standard library properties on lists.

The issue with lists is that they are an inherently sequential datatype, and thus do not benefit from parallelism in any way. This runs directly contrary to the goal of sequences, which is to benefit from parallelism.

Take, for example, the higher-ordered function **map**. A common cost model for **map f S** would be to count the number of calls to **f**. On arrays, we can say that the work is the sum of the

work of every call to `f`, while the span is the maximum cost of any one call to `f`. But in any concrete implementation of `map` on lists, it is impossible to take the maximum of the calls to `f` when considering span, because the implementation is always sequential on lists!

The solution is to give an arbitrary cost annotation to sequence functions such as `map` within our implementation of `scan`. This methodology is similar to the one introduced in Abstraction Functions as Types [4], which introduces a phase distinction **abs** between the abstract phase and concrete phase. Similar to how the open modality \bigcirc hides cost details in the behavioral phase in **Calf**, the phase distinction **abs** hides implementation details and other private data in the abstract phase. In particular, the abstract phase hides all cost details of our implementation - allowing us to arbitrarily annotate the cost of our sequence functions with their abstract cost.

1.4 Overview

We present an implementation and mechanized proof of all three aforementioned implementations of `scan` within **Calf** embedded in Agda. We denote the three implementations of bruteforce scan, divide-and-conquer scan, and contraction scan respectively as `scan/bruteforce`, `scan/divconq`, and `scan/contract`, and the three mechanized proofs as `scan/bruteforce/cost`, `scan/divconq/cost`, and `scan/contract/cost`. Notably, we were able to formalize concrete bounds for both the work and span of all three implementations.

The three major aspects of this proof were to define the monoid under the open modality, to implement each of the scan algorithms in Agda, and to verify their sequential and parallel cost within **Calf** embedded in Agda. Notably, cost verification in both `scan/divconq/cost` and `scan/contract/cost` required partial verification of correctness criterion for `scan/divconq` and `scan/contract`, respectively. Furthermore, the inequational reasoning features of **Decalf** were necessary to complete the proof of `scan/contract/cost`.

2 Implementation

2.1 The open monoid

As previously mentioned, `scan` is required to take in a monoid, because both parallel implementations of `scan` make the assumption that if we have `scan f b S`, then `f` and `b` form a monoid. However, it is entirely possible that a monoid is valid with respect to *behavior*, but not *cost*. For example, consider $f(x, y) := \text{step}^{(x, x)}(\text{ret}(x + y))$, with the identity element 0. Observe that `f` and 0 form a monoid when only considering behavior, as behaviorally `f` is identical to the addition operation. However, `f` and 0 do not form a monoid with respect to cost, as for $x > 0$ $f(x, 0) \neq f(0, x)$, because the right identity $f(x, 0)$ has cost (x, x) , while the left identity $f(0, x)$ has cost $(0, 0)$.

Note that `f` and 0 are still valid inputs into `scan`. However, it does not form a monoid overall, because it is not a monoid with respect to cost. Thus, any standard library implementation of `isMonoid` is insufficient for our purposes. That said, it is still necessary for us to define some type `isMonoid` in order to enforce the correctness of all our inputs into `scan`. Therefore, we must define our own monoid for `scan` to take in.

It turns out that the *phase distinction* is quite useful for this purpose. Because we want to ignore cost when checking that the properties of a monoid hold, and the open modality isolates the behavioral aspect of programs, we essentially want to define a monoid under the open modality. Indeed, this is exactly what we do. We define `○-isMonoid` as follows:

```
record ○-isMonoid {A : tp+} (f : cmp (Π (A ×+ A) (λ _ → F A)))
  (ε : val A) : Set where
  field
    identityr : {a : val A} → ○ ( f(a , ε) ≡ ret a )
    identityl : {a : val A} → ○ ( f(ε , a) ≡ ret a )
    assoc : {a b c : val A} →
      ○ ((bind (F _) (f (a , b))
        λ left → f (left , c)) ≡
        (bind (F _) (f (b , c))
        λ right → f (a , right)))
```

Basically, we require the properties of a monoid - identity and associativity - to only hold when `○` is inhabited. We can then define `○-Monoid` as a record that includes an identity element `identity` and a function `f`, as well as a proof that `f` and `identity` form a monoid in the form of a term of type `○-isMonoid f identity`.

2.2 Divide and conquer implementation

We defined `scan/divconq` in terms of a helper function `scan/divconq/clocked`, in order to make use of a *clock* to simulate general recursion in Agda. The base cases for `scan/divconq/clocked` are fairly straightforward, so we will focus primarily on the recursive case in our analysis.

```

scan/divconq/clocked :
  (A : tp+) →
  cmp (Π (U (Π (A ×+ A) (λ _ → F A))) (λ _ →
    Π A (λ _ →
      Π nat λ k →
      Π (list A) (λ l →
        Π (meta+ (⌈log2 length l ⌋ Nat.≤ k)) λ _ →
        F (Σ+ (list A ×+ A) λ (l' , _) → meta+ (length l ≡ length l'))))))
scan/divconq/clocked A f e zero [] p = ret (([], e), refl)
scan/divconq/clocked A f e zero (x :: []) p = ret ((e :: [], x), refl)
scan/divconq/clocked A f e (suc k) l p =
  bind (F _) (split A l) λ ((l1 , l2) , length1 , length2 , l1↔l2) →
  bind (F _)
    (scan/divconq/clocked A f e k l1 (h1 l1 length1) ||
     scan/divconq/clocked A f e k l2 (h2 l2 length2))
    λ (((l1' , b') , |l1|≡|l1'|) , ((l2' , c') , |l2|≡|l2'|)) →
  step (F _) (length l2' , 1) ( -- cost of map
    bind (F _) (mapList {A} (λ x → f (b' , x)) l2') λ (r' , |l2'|≡|r'|) →
    bind (F _) (f (b' , c')) λ res →
    step (F _) (length l1' + length r' , 1) ( -- cost of append
      ret ((l1' ++ r' , res) ,
        lem {A} l1 l2 l1' l2' r' l1↔l2 |l1|≡|l1'| |l2|≡|l2'| |l2'|≡|r'|)))

```

2.2.1 Clocked recursion

In a programming language that supports general recursion (such as Standard ML), the core of the divide-and-conquer approach is to obtain two lists (`l1`, `l2`) from splitting the list into two, and then recursively calling `scan/divconq` on `l1` and `l2`. However, this is not allowed in Agda, because Agda programs are required to be total, whereas general recursion introduces the possibility of an infinite loop. Thus, in Agda only direct recursion is allowed, and it is not allowed to recursively call `scan/divconq` on `l1` and `l2`.

One common way to model general recursion in total type theory is the Bove-Capretta method, which involves the use of an accessibility predicate as a termination metric. The use of a *clock*, as introduced in **Calf**, provides an alternative to this[8]. The idea is that we parameterize our `scan/divconq` program with some `k : nat`, which is our clock. Essentially, `k` acts as a counter for our recursion depth. Because `k` decreases by 1 at each level of recursion, we are able to recurse on `k` in Agda.

Parameterization of our program by the clock is also advantageous for cost analysis. In particular, note that the clock must always be greater than or equal to the remaining recursion depth. Therefore, if the cost of the program depends on the number of levels of recursion, then the value of the clock must be an upper bound on the runtime of the program. Then, we can instantiate our clock with the smallest value possible in order to obtain a tight upper bound on the runtime of our program - exactly what we desire from cost analysis.

2.2.2 Correctness criteria

The other interesting result that came about in `scan/divconq` is that we ended up requiring a proof of part of the correctness criteria. In particular, both `scan/divconq` and `map` return a proof that they

preserve the length of their input sequence. This is necessary first for the cost proof, and second in order to prove that length is preserved in the recursive call.

For the cost proof, it is easy to see why it is necessary to require a proof of preservation of length within our return type. Suppose we had no proof that `scan/divconq/clocked` preserved the length of its input list. Then, after making our recursive call on l_1 and l_2 , we have no idea what the length of the result sequences l'_1 and l'_2 are! In particular, at the end, we append the two sequences back together, after mapping over the right subsequence. To be able to assert that this append has cost `length l`, we need to know that the length of the left subsequence l'_1 and the length of the right subsequence l'_2 sum to l - which we can only do if we prove the correctness criteria that `scan/divconq/clocked` preserves length.

The fact that our cost proof requires this partial proof of correctness is further evidence of ideas developed in **Calf**. In particular, it shows that similar to the static-dynamic phase distinction of ML languages where the dynamic phase depends on the static phase (but not vice versa), the cost phase depends on the behavioral phase (but the reverse cannot be true).

2.3 Bruteforce implementation

Similar to `scan/divconq`, we needed to use a helper function in order to implement `scan/bruteforce`. Here, the reason is that `scan` is required to take in a monoid, but the `scan/bruteforce` implementation uses the field for the identity element as an accumulator. This means that the monoid is not preserved, so the resulting function will not typecheck against the type signature of `scan/bruteforce`, which takes in a monoid.

The solution is to define a helper function `scan/bruteforce/help` which relaxes the monoid requirement, and having our main `scan/bruteforce` function instantiate `scan/bruteforce/help` with the monoid. This allows us to make use of an accumulator argument, while still maintaining the abstraction that `scan` must take in a monoid.

```
scan/bruteforce/help :
  {A : tp+} →
  cmp (Π (U (Π (A ×+ A) (λ _ → F A))) (λ _ →
    Π A (λ _ → (
      Π (list A) (λ _ →
        F (list A ×+ A))))))
scan/bruteforce/help f e [] = ret ([], e)
scan/bruteforce/help f e (x :: L) =
  step (F _) (1, 1) (bind (F _) (f (e, x)) (λ y →
    bind (F _) (scan/bruteforce/help f y L) λ { (ys, r) →
      ret (e :: ys, r)})))
```

2.3.1 A bug in iteratePrefixes

CMU's sophomore undergraduate algorithms course, 15-210, makes significant use of the sequence library, and in particular `scan`. In particular, one function defined in the 15-210 library is `iteratePrefixes`, which is essentially the bruteforce implementation of `scan`. Thus, we based our implementation for `scan/bruteforce/help` off of 15-210's implementation of `iteratePrefixes`, which is shown below:


```

fun iteratePrefixes f b [] = ([], b)
  | iteratePrefixes f b (x::xs) =
    let
      val y = f (b, x)
      val (ys, r) = iteratePrefixes f y xs
    in (y::ys, r)
    end

```

However, it turns out this implementation is buggy, because it is off by one application of f . Specifically, in the result we return $(y::ys, r)$, where $y \cong f(b, x)$. But this results in each element having f applied to it exactly one too many times. So to fix, we change the return to $(b::ys, r)$. Thus, we have the corrected code as follows:

```

fun iteratePrefixes f b [] = ([], b)
  | iteratePrefixes f b (x::xs) =
    let
      val y = f (b, x)
      val (ys, r) = iteratePrefixes f y xs
    in (b::ys, r)
    end

```

2.4 Contraction Implementation

For the most part, the `scan/contract` implementation uses similar ideas to `scan/divconq`. In particular, `scan/contract` also uses a helper function `scan/contract/clocked` to parameterize the function by a clock, and also returns a partial correctness criteria in regards to proving the length is preserved.

However, unlike `scan/divconq`, which relies on the sequence library functions `map` and `append`, `scan/contract` relies on the custom helper functions `contract` and `expand`.

```

scan/contract/clocked : {A : tp+} →
  cmp (Π (U (Π (A ×+ A) (λ _ → F A))) λ f →
    Π A (λ e →
      Π nat (λ k →
        Π (list A) (λ l →
          Π (meta+ (⌈log2 length l⌉ Nat.≤ k)) (λ p →
            F (Σ+ (list A ×+ A) λ (l', _) → meta+ (length l ≡ length l'))))))))
scan/contract/clocked f e zero [] p = ret ( ([], e), refl)
scan/contract/clocked f e zero (x :: []) p = bind (F _) (f (e, x)) (λ x1 → ret ((e :: [], x1), refl))
scan/contract/clocked f e (suc k) [] p = ret ( ([], e), refl)
scan/contract/clocked f e (suc k) (x :: []) p = bind (F _) (f (e, x)) (λ x1 → ret ((e :: [], x1), refl))
scan/contract/clocked {A} f e (suc k) l@(x :: y :: _) p =
  step (F _) (2 * ⌈ length l / 2 ⌉, 2) -- since we need to do 2 lookups
  (bind (F _) (contract f l) λ (cs, p1) →
    bind (F _) (scan/contract/clocked f e k cs (h cs (Eq.sym p1))) λ ((rs, res), p2) →
    step (F _) (2 * length l, 2) -- since we need to do max of 2 lookups
    (bind (F _) (expand f l rs (Eq.sym (Eq.trans p1 p2))) λ (es, p3) →
      ret ((es, res), p3)))

```

`contract` and `expand` represent the contraction step and expansion step of the contraction algorithm, respectively. Because we can specify `contract` and `expand` in terms of the primitive sequence operations `tabulate` and `lookup`, we can say that these exist in the abstract phase and implement them directly using the list implementation while giving them fictitious cost annotation.

```

contract : {A : tp+} →
  cmp (Π (U (Π (A ×+ A) (λ _ → F A))) λ _ →
    Π (list A) (λ l →
      F (Σ+ (list A) λ l' → meta+ ( ⌈ length l / 2 ⌉ ≡ length l' ) )))
expand : {A : tp+} →
  cmp (Π (U (Π (A ×+ A) (λ _ → F A))) λ _ →
    Π (list A) (λ l2 →
      Π (list A) (λ l1 →
        Π (meta+ ( length l1 ≡ ⌈ length l2 / 2 ⌉ )) (λ p →
          F (Σ+ (list A) λ l' → meta+ ( length l2 ≡ length l' ) ))))

```

The other notable aspect about `contract` and `expand` is that they both return a partial correctness criteria in regards to their length. Specifically, `contract` returns a proof that $\lceil \text{length } l / 2 \rceil \equiv \text{length } l'$, where l is the input list and l' is the result, and `expand` returns a proof that $\text{length } l_2 \equiv \text{length } l'$, where l_2 is the original list and l' is the result. Just like `scan/divconq`, the correctness criteria is necessary here to reason about the cost of the recursive call, and to complete the proof that `scan/contract/clocked` preserves the length in the recursive case.

3 Cost Verification

3.1 Bruteforce Cost

The main interesting part of the formalization of cost for `scan/bruteforce` is that because we used a helper function that takes in an accumulator argument, we ended up also needing to prove the cost with respect to this accumulator. Specifically, our cost proof is essentially that the cost is independent of the value of the accumulator. Otherwise, the proof of `scan/bruteforce/cost` is pretty straightforward.

It should also be noted that the bound in this case is entirely strict - that is, the entire proof is done through equivalences, without the use of inequational reasoning from **Decalf**. Thus, we were able to prove a tight bound of `length l` work and `length l` span.

3.2 Divide-and-conquer Cost

As mentioned previously, the formalization of the cost of `scan/divconq` requires the proof of part of the correctness criteria - in particular, that `length` is preserved by `mapList` and `scan/divconq/clocked`. Furthermore, the clock is also useful in proving an upper bound, because we can instantiate the clock with the value $\lceil \log_2 \text{length } l \rceil$ and use this in our upper bound, since the recursion depth is part of the cost for `scan/divconq/clocked`.

There are a few more important things to note. First, a significant part of our proof steps involve stepping through the fictitious cost annotations on sequence functions (`split`, `mapList`, etc), which we made in the vein of Abstraction Functions as Types, as previously mentioned.

Second, there is a step where the equational reasoning of **Calf** is insufficient, and we must use weakening as in **Decalf**. Suppose that we get the pair of lists (l_1, l_2) from `split A l`, where l is our original sequence. Note that from the result of `split` we know that `length l1 + length l2 ≡ length l`. Then, our non-recursive work at each level is `length l1 + 2 * length l2`, since we map over the right sublist and then append both sublists together. We can rewrite this as `(length l1 + length l2) + length l2`, which is equivalent to `length l + length l2`. Then, we can say that $0 \leq \text{length } l_1$ and that `length l2 ≡ 0 + length l2` to bound `length l2` on the right side by `length l1 + length l2`. From there we can easily simplify the entire expression to `2 * length l`. On the other hand, the span bound only uses strict equivalences, since the span of the call to both `map` and `append` is $O(1)$.

3.3 Contraction Cost

The formalization of the proof of cost for `scan/contract` is interesting for a few reasons. When we prove the cost of `scan/contract` using big-O, we end up with the work recurrence $W(n) = W(n/2) + cn$ for some constant c . This expands out to the geometric sum $n + n/2 + n/4 + \dots$, which is provably bounded by $2n$. However, there are several difficulties that arise when formalizing this in **Decalf**.

First, because we cannot assume that the length of the sequence is even, we cannot use the term $n/2$ in our recurrence relation. Instead, the best bound on the result `cs` of `contract f l` is that $\lceil \text{length } l / 2 \rceil \equiv \text{length } cs$. This means we have a recurrence of $W(n) = W(\lceil n/2 \rceil) + 2 * \lceil n/2 \rceil + 2 * n$. Suppose that we say that $W(n)$ is bounded by cn , where c is some constant, in our result. Then we end up with wanting to prove that $(c + 2) * \lceil n/2 \rceil + 2 * n \leq c * n$.

It turns out that this statement only holds for $n = 0$ and $n \geq 2$. If we substitute in $n = 1$, we end up with the statement that $c + 4 \leq c$ - an obvious contradiction. If we instead suppose $n = 2$, we have $4 \leq c$ - not a contradiction. Thus, we know that we will end up needing a lemma that $2 \leq n$. There are two ways to enforce this. Either we ensure that our clock k is precisely the recursion depth, or we create base cases for when the list is empty or singleton but k is nonzero (i.e.,

we create a special case to handle the case where k is instantiated with a value that is larger than the actual recursion depth). Regardless, we must solve for c .

First, we can derive a lemma that $2 * \lceil n/2 \rceil \leq 1 + n$. We can use this lemma when solving for c .

$$\begin{aligned}
(c+2) * \lceil n/2 \rceil + 2 * n &\leq c * n \\
(c+2) * \lceil n/2 \rceil &\leq (c-2) * n \\
2 * \left(\frac{c+2}{2}\right) * \lceil n/2 \rceil &\leq (c-2) * n && \text{(this is imprecise, but it works)} \\
\frac{c+2}{2} * n + \frac{c+2}{2} &\leq (c-2) * n && \text{(Lemma)} \\
c + 2 + \frac{c}{2} + 1 &\leq 2c - 4 && \text{(substitute } n = 2 \text{ as worst case)} \\
7 &\leq \frac{c}{2} \\
14 &\leq c
\end{aligned}$$

Thus we end up with a “guess” that $c = 14$. If we plug this back into our recurrence, we end up with wanting to prove that $16 * \lceil n/2 \rceil + 2 * n \leq 14 * n$. Indeed, this is provable.

$$\begin{aligned}
16 * \lceil n/2 \rceil + 2 * n & \\
= 8 * (2 * \lceil n/2 \rceil) + 2 * n & \\
\leq 8 * (1 + n) + 2 * n & \quad \text{(Lemma)} \\
= 8 + 10 * n & \\
= (4 * 2) + 10 * n & \\
\leq (4 * n) + 10 * n & \quad \text{(if we assume } 2 \leq n) \\
= 14 * n &
\end{aligned}$$

So we’re able to complete the proof as long as we can prove $2 \leq n$. As previously stated, there are two ways to handle this. Firstly, we could enforce that k , our “clock” for induction, is exactly our recursion depth, such as by having `scan/contract/cost` take in an additional argument $q : 1 \leq k \rightarrow 2^{k-1} < \text{length } l$. (Note that since in the recursive case we know that k is of the form $\text{succ } k'$, this is equivalent to $2^{k'} < \text{length } l$, which in turn is stronger than the statement $2 \leq \text{length } l$).

However, it turns out that enforcing this strict bound on k is rather difficult. Instead, if we create additional base cases for the case where $k > 0$ but where the list is either empty or singleton, it is fairly simple to derive a lemma that $2 \leq \text{length } l$ (because if the length is 0 or 1 it will always fall into a base case).

Within this proof, it is also important to note the use of the inequational reasoning features of **Decalf**. Here, because we can only provide an approximate bound on the sum of the geometric series induced by our recurrence relation, it is clear that we need inequational reasoning from the beginning.

Implementation	Work	Span
Bruteforce	$\text{length } l$	$\text{length } l$
Divide-and-conquer	$(2 * \lceil \log_2 \text{length } l \rceil + 2) * \text{length } l$	$2 * \lceil \log_2 \text{length } l \rceil$
Contraction	$14 * \text{length } l$	$4 * \lceil \log_2 \text{length } l \rceil$

Figure 2: Summary of concrete costs of `scan` implementations

4 Future Work

4.1 Sequence Library as Postulates

As mentioned previously, we currently implement all of our sequence functions explicitly on lists and give them fictitious cost annotation (in particular, fictitious span annotation) in line with Abstraction Functions as Types. However, this is a rather inelegant solution in Agda. A more elegant way to express the abstract phase in Agda would be to write the primitive sequence operations `SeqAppend`, `SeqMap`, and `SeqTabulate` as postulates.

```

postulate
  SeqAppend :
    {A : tp+} →
    (l1 l2 : val (list A)) →
    cmp (F (Σ+ (list A) λ l' → meta+ (l' ≡ l1 ++ l2)))

  SeqAppend/cost :
    {A : tp+} →
    (l1 l2 : val (list A)) →
    lsBounded (Σ+ (list A) λ l' → meta+ (l' ≡ l1 ++ l2))
      (SeqAppend {A} l1 l2)
      (length l1 + length l2 , 1)

```

When we write `SeqAppend` as a postulate, we must modify the return type. Instead of `append` directly returning the appended list, `SeqAppend` must also return a proof that its result l' is equivalent to $l_1 ++ l_2$. Essentially, we are using the standard list append `++` as a *specification function* for `SeqAppend`, since we are not able to reason about the implementation of `SeqAppend`. Then, we reason about cost using the cost postulates (such as `SeqAppend/cost` here), rather than going through fictitious cost annotations located directly within the `scan` implementations.

4.2 Proof of Correctness

Significant prior work has been done in regards to verifying the correctness of `scan`. Safari et al. verified `scan` using the VerCors verifier based on contracts for correctness [10]. Chong et al. presented a method that allows the correctness of `scan` to be verified by running a single test case [3]. However, to our knowledge, there is no published formal verification of the parallel prefix sum algorithm within a dependently typed proof assistant.

Within **Decalf**, our proof of correctness is to begin by taking `scan/bruteforce` as a specification function. Then, we want to prove that $\bigcirc(\text{scan/bruteforce} \equiv \text{scan/divconq})$ and

$\bigcirc(\text{scan/bruteforce} \equiv \text{scan/contract})$. This is similar to how we can prove that

$\bigcirc(\text{insertionSort} \equiv \text{mergeSort})$.

There are a few difficulties that stand in the way of such a proof. When proving that

$\bigcirc(\text{insertionSort} \equiv \text{mergeSort})$, we were able to use the fact that both sorting algorithms return a sorted permutation of the original sequence. However, we do not have any properties to reason about the resulting sequence for **scan**, so we must reason directly about the behavior of the **scan** algorithms. This is problematic because **scan/bruteforce** proceeds by direct recursion on the input list, whereas both **scan/divconq** and **scan/contract** proceed by recursion on a clock, which makes it unclear how a proof of correctness should proceed.

5 Acknowledgements

The author is grateful for the mentorship and support of Runming Li throughout the course of this research. The author would also like to thank Robert Harper for providing insightful high-level guidance, and Lukas Kebuladze for fruitful discussions on various details.

This material is based upon work supported by the United States Air Force Office of Scientific Research under grant number FA9550-21-0009 and FA9550-23-1-0434 (Tristan Nguyen, program manager). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

References

- [1] ACAR, U. A., AND BLELLOCH, G. E. *Algorithms: Parallel and Sequential*. 2019.
- [2] BLELLOCH, G. E. *NESL: a nested data parallel language*. School of Computer Science, Carnegie Mellon University Pittsburgh, PA, 1992.
- [3] CHONG, N., DONALDSON, A. F., AND KETEMA, J. A sound and complete abstraction for reasoning about parallel prefix sums. *SIGPLAN Not.* 49, 1 (Jan. 2014), 397–409.
- [4] GRODIN, H., LI, R., AND HARPER, R. Abstraction functions as types, 2025.
- [5] GRODIN, H., NIU, Y., STERLING, J., AND HARPER, R. Decalf: A directed, effectful cost-aware logical framework. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 273–301.
- [6] HARRIS, M., SENGUPTA, S., AND OWENS, J. D. Parallel prefix sum (scan) with cuda. *GPU gems* 3, 39 (2007), 851–876.
- [7] LEVY, P. B. *Call-by-push-value: A Functional/imperative Synthesis*, vol. 2. Springer Science & Business Media, 2012.
- [8] NIU, Y., STERLING, J., GRODIN, H., AND HARPER, R. A cost-aware logical framework. *Proc. ACM Program. Lang.* 6, POPL (Jan. 2022).
- [9] NORELL, U. *Dependently Typed Programming in Agda*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 230–266.
- [10] SAFARI, M., OORTWIJN, W., JOOSTEN, S., AND HUISMAN, M. Formal verification of parallel prefix sum. In *NASA Formal Methods* (Cham, 2020), R. Lee, S. Jha, A. Mavridou, and D. Giannakopoulou, Eds., Springer International Publishing, pp. 170–186.