A type checker for a dependently-typed list language

ABD-EL-AZIZ ZAYED, McGill University, Canada
LOUIS HILDEBRAND, McGill University, Canada

1 ABSTRACT

Functions that operate on lists often have pre- and post-conditions related to the list length. These are typically checked only at run time. We present a minimal dependently-typed list language along with a bidirectional type checker which can instead enforce these constraints at compile time. [1] The language includes a "vector" type representing lists indexed by their length. The length of a vector is restricted to be a number, variable, or sum. Despite this restriction, we show that many useful list functions can be implemented in a safe way.

2 INTRODUCTION

Many list-related functions have pre-conditions involving list lengths. For example, consider the following functions from OCaml's `List` module [1].

- `List.tl` takes as input a *non-empty* list.
- `List.nth` takes as input an index n and a list *whose length is at least* $n + 1$.
- `List.map2` takes as input two lists *of the same length*.

These constraints are all checked at runtime. However, it is possible to enforce them at compile-time instead. Similarly, list-related functions make certain guarantees about their outputs—i.e., post-conditions. For example, in OCaml, `List.tl` returns a list that is one element shorter than the input. Mistakes in the implementation can sometimes be detected by the fact that they violate these post-conditions. For example, consider this faulty implementation of the `drop` function. `drop` expects a list of length $k + n$ and should remove its first k elements to get a vector of length n.

```
1  let rec drop (k : int) (xs : 'a list) : 'a list =
2      match k with
3      | 0 -> []
4      | k -> let k' = k - 1 in
5             drop k (List.tl xs)
```

In the `0` branch, we return `[]` (a list of length zero) when we should be returning `xs` (a list of length n). [2] In the non-zero branch, we drop k elements from the tail (which would yield a vector of length $n - 1$) when we should be dropping k' elements (which would yield a vector of length n). Both of these mistakes can be detected at compile time in a dependently-typed language.

Dependently-typed languages and proof assistants, such as Agda [2] and Idris [3], commonly support defining "vectors"—that is, lists indexed by their length. In this project, we focus on a greatly simplified version of these languages which *only* supports vectors.

Our concrete contributions are as follows.

- We define the syntax of a minimal dependently-typed list language (section 3.1.1).
- We define a type system for the language (section 3.1.3). The type system can enforce both pre- and post-conditions of functions that operate on lists.
- We demonstrate that the language can easily be extended with new types (section 3.2).

[1] The implementation is available on GitHub at https://github.com/louis-hildebrand/chick.
[2] One of us actually made this typo while testing the type checker!

Authors' addresses: Abd-El-Aziz Zayed, McGill University, Montreal, Canada, abd-el-aziz.zayed@mail.mcgill.ca; Louis Hildebrand, McGill University, Montreal, Canada, louis.hildebrand@mail.mcgill.ca.

- We implement the type checker as an OCaml program (section 3.4).
- We provide examples of safe list programs written in our language (section 3.3).
- We document some challenges we encountered in designing and implementing the type system (section 3.5).

3 DEVELOPMENT

3.1 Minimal Language

We start by defining a small, very limited language and gradually extend it.

3.1.1 Syntax. The initial language has only one base type—`Nat`—representing natural numbers. It also supports "vectors"—denoted `Vec` $A\ \ell$—which represent lists of length ℓ containing elements of type A. Finally, dependent functions are represented with the Π type. This is a strict generalization of the usual arrow type; $A_1 \to A_2$ is equivalent to $\Pi(x\colon A_1).A_2$, where x is a variable that does not occur free in A_2. In this report, we use $A_1 \to A_2$ as shorthand for such a "constant" Π type.

The length of a vector is restricted to be a natural number, a variable, or a sum of terms which are valid lengths. This is to make checking equality of lengths decidable (section 3.1.3).

Type	A	::=	`Nat`	Natural numbers
		|	`Vec` $A\ \ell$	Fixed-size lists
		|	$\Pi(x\colon A_1).A_2$	Dependent functions
Length	ℓ	::=	$n \mid x \mid \ell_1 + \ldots + \ell_k$	
Numeral	n	$\in$	$\mathbb{N}$	

We use a bidirectional type system, so terms are separated into checkable terms t and synthesizable terms s.

Chk.	t	::=	s	Synth. term
		|	$\lambda x.t$	Abstraction
		|	`fix` $x.t$	Recursion
		|	n	Numeral
		|	$t_1 + \ldots + t_k$	Sum
		|	`nil`	Empty vec
		|	`cons` $\ell\ t_1\ t_2$	Prepend to vec
		|	(`match` s `with` $\mid 0 \to t_1 \mid x+1 \to t_2$)	Pattern match on nat
		|	(`match` s `with` $\mid$ `nil` $\to t_1 \mid$ `cons` $x_1\ x_2\ x_3 \to t_2$)	Pattern match on vec
Syn.	s	::=	x	Variable
		|	$s\ t$	Application

Note that the programmer can actually omit unreachable branches of `match` expressions.

A program is a sequence of declarations, each of which must be accompanied by a type annotation.

3.1.2 Typing Judgements. Our type system involves the following four judgements.

- $\boxed{\Gamma|\Delta \vdash t \Longleftarrow A}$ (In context Γ and with assumptions Δ, t has type A.)
- $\boxed{\Gamma|\Delta \vdash s \Longrightarrow A}$ (In context Γ and with assumptions Δ, s synthesizes type A.)
- $\boxed{\Delta \vdash A_1 \equiv A_2}$ (With assumptions Δ, the types A_1 and A_2 are equivalent.)
- $\boxed{\Delta \vdash \ell_1 \equiv \ell_2}$ (With assumptions Δ, the lengths ℓ_1 and ℓ_2 are equivalent.)

As usual, Γ denotes the typing context, which maps variable names to types.

$$\text{Type context} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x\colon A$$

Furthermore, there is a context Δ which records assumptions about lengths. New assumptions are introduced when entering a branch of a `match` expression.

$$\text{Assumptions} \quad \Delta \quad ::= \quad \cdot \mid \Delta, \ell_1 = \ell_2$$

For example, the types `Vec Nat` n and `Vec Nat 0` are not equivalent in general but are equivalent if n is assumed to be zero. That is, $\cdot \nvdash$ `Vec Nat` $n \equiv$ `Vec Nat 0` but $(\cdot, n = 0) \vdash$ `Vec Nat` $n \equiv$ `Vec Nat 0`.

3.1.3 *Typing Rules.*

Synthesizable terms. The rules for synthesizable terms are as follows.

$$\frac{\Gamma(x) = A}{\Gamma|\Delta \vdash x \Longrightarrow A} \qquad \frac{\Gamma|\Delta \vdash s \Longrightarrow \Pi(x\colon A).B \quad \Gamma|\Delta \vdash t \Longleftarrow A}{\Gamma|\Delta \vdash s\ t \Longrightarrow [t/x]B}$$

To see why the substitution $[t/x]B$ is necessary in the rule for applications, consider the function `reverse` $: \Pi(n\colon$ `Nat`$).$`Vec Nat` $n \to$ `Vec Nat` n. We expect `reverse` 42 to have type `Vec Nat` 42 $\to$ `Vec Nat` 42.

Checkable terms. The rules for checkable terms are as follows.

$$\frac{\Gamma|\Delta \vdash s \Longrightarrow B \quad \Delta \vdash A \equiv B}{\Gamma|\Delta \vdash s \Longleftarrow A} \qquad \frac{(\Gamma, x : A)|\Delta \vdash t \Longleftarrow B}{\Gamma|\Delta \vdash \lambda x.t \Longleftarrow \Pi(x\colon A).B} \qquad \frac{(\Gamma, x : A)|\Delta \vdash t \Longleftarrow A}{\Gamma|\Delta \vdash \texttt{fix}\ x.t \Longleftarrow A}$$

$$\frac{}{\Gamma|\Delta \vdash n \Longleftarrow \texttt{Nat}} \qquad \frac{\forall 1 \le i \le k, \Gamma|\Delta \vdash t_i \Longleftarrow \texttt{Nat}}{\Gamma|\Delta \vdash t_1 + \ldots + t_k \Longleftarrow \texttt{Nat}}$$

$$\frac{\Delta \vdash \ell \equiv 0}{\Gamma|\Delta \vdash \texttt{nil} \Longleftarrow \texttt{Vec}\ A\ \ell}$$

$$\frac{\Gamma|\Delta \vdash \ell \Longleftarrow \texttt{Nat} \quad \Gamma|\Delta \vdash t_1 \Longleftarrow A \quad \Gamma|\Delta \vdash t_2 \Longleftarrow \texttt{Vec}\ A\ \ell \quad \Delta \vdash \ell' \equiv \ell + 1}{\Gamma|\Delta \vdash \texttt{cons}\ \ell\ t_1\ t_2 \Longleftarrow \texttt{Vec}\ A\ \ell'}$$

$$\frac{\Gamma|\Delta \vdash x \Longrightarrow \texttt{Nat} \quad \Gamma|(\Delta, x = 0) \vdash t_0 \Longleftarrow A \quad (\Gamma, y : \texttt{Nat})|(\Delta, x = y + 1) \vdash t_1 \Longleftarrow A}{\Gamma|\Delta \vdash \texttt{match}\ x\ \texttt{with} \mid 0 \to t_0 \mid y + 1 \to t_1 \Longleftarrow A}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma, x_1 : \texttt{Nat}, x_2 : B, x_3 : \texttt{Vec}\ B\ x_1 \\ \Gamma|\Delta \vdash s \Longrightarrow \texttt{Vec}\ B\ \ell \quad \Gamma|(\Delta, \ell = 0) \vdash t_0 \Longleftarrow A \quad \Gamma'|(\Delta, \ell = x_1 + 1) \vdash t_1 \Longleftarrow A\end{array}}{\Gamma|\Delta \vdash \texttt{match}\ s\ \texttt{with} \mid \texttt{nil} \to t_0 \mid \texttt{cons}\ x_1\ x_2\ x_3 \to t_1 \Longleftarrow A}$$

Notice how the rule for `nil` does not directly say `nil` $\Longleftarrow$ `Vec` A 0 but instead checks that the length is *equivalent* to zero. This is to handle cases like checking `nil` against `Vec` A $(0 + 0)$, or against `Vec` A n when $\Delta \vdash n \equiv 0$. However, the rule $\Gamma|\Delta \vdash n \Longleftarrow$ `Nat` is valid because the only type that is equivalent to `Nat` is `Nat` itself, so syntactic equality suffices.

Type equality. The rules for type equality are as follows.

$$\frac{}{\Delta \vdash \texttt{Nat} \equiv \texttt{Nat}} \qquad \frac{\Delta \vdash A \equiv B \quad \Delta \vdash n \equiv m}{\Delta \vdash \texttt{Vec}\ A\ n \equiv \texttt{Vec}\ B\ m} \qquad \frac{\Delta \vdash A_1 \equiv A_2 \quad \Delta \vdash B_1 \equiv B_2}{\Delta \vdash \Pi(x\colon A_1).B_1 \equiv \Pi(x\colon A_2).B_1}$$

In practice, Π types with different variable names can be compared by replacing the variables to a fresh variable.

Length equality. To check equality of lengths ($\Delta \vdash \ell \equiv \ell'$), we construct a proposition as follows. Let $\vec{x} = x_1, \ldots, x_k$ be the set of all the variables in ℓ, ℓ', and Δ. Let $\Delta = \cdot, \ell_1 = \ell_1', \ldots, \ell_j = \ell_j'$. We want to check that the equation holds given all the assumptions, so the proposition is

$$\forall \vec{x}, \left(\forall 1 \le i \le j, \ell_i = \ell_i'\right) \implies \ell = \ell'$$

However, suppose the programmer omits a branch. Then we want to check whether there *exists* any instantiation of the variables such that all the assumptions hold (i.e., we get to the current match expression) *and* we take the missing branch. In that case, the proposition is instead

$$\exists \vec{x}, \left(\forall 1 \le i \le j, \ell_i = \ell_i'\right) \wedge \ell = \ell'$$

We can use the same technique to check whether any of the branches that *were* implemented are unreachable, in which case the type checker can ask the programmer to omit the unreachable branch.

Since we restrict the length of a vector to only allow addition, these propositions are in the language of Presburger arithmetic. Presburger arithmetic is decidable [4, 5], which means that type checking our language is also decidable.

Substitutions. Some rules involve substitution in a type or in a length. Substitution for types is defined as follows.

$$\begin{cases} [t/x]\texttt{Nat} = \texttt{Nat} \\ [\ell'/x](\texttt{Vec}\ A\ \ell) = \texttt{Vec}\ ([\ell'/x]A)\ ([\ell'/x]\ell) \\ [t/x](\texttt{Vec}\ A\ \ell) = \texttt{Vec}\ ([t/x]A)\ \ell & \text{if } x \notin \mathrm{FV}(\ell) \text{ and } t \text{ is not a length} \\ [t/x](\Pi(y\colon A).B) = \Pi(y\colon [t/x]A).[t/x]B & \text{if } y \neq x \text{ and } y \notin \mathrm{FV}(t) \end{cases}$$

Notice that substitution into a vector is only possible if the variable does not occur within the type or the new term is a valid length. For example, $[42/x](\texttt{Vec}\ A\ x)$ is clearly valid. We also need to be able to pass arguments that are not lengths—as long as they are for a non-dependent function—so $[\texttt{nil}/v](\texttt{Vec}\ A\ 0)$ should also be valid. However, a substitution like $[n \cdot k/x](\texttt{Vec}\ A\ x)$ is *not* valid (even if $n \cdot k$ was a valid term). See section 4.1 for a discussion of this limitation.

Substitution into a length is straightforward.

$$\begin{cases} [\ell/x]x = \ell \\ [\ell/x]y = y & y \neq x \\ [\ell/x]n = n \\ [\ell/x](\ell_1 + \ldots + \ell_k) = ([\ell/x]\ell_1) + \ldots + ([\ell/x]\ell_k) \end{cases}$$

In practice, it is also sometimes necessary to rename variables within a checkable or synthesizable term. For example, this is used to avoid shadowing in `match` expressions, which may lead to contradictory assumptions in Δ. Renaming is similar to standard substitution.

3.2 Adding New Types

Adding new types to the language is straightforward.

3.2.1 Booleans. To add booleans, we start by adding one new base type (`Bool`), constructors (`true` and `false`), and a destructor (pattern matching, which is equivalent to if-then-else).

Type	A	::=	... | `Bool`
Check. term	t	::=	...
		|	`true` | `false`
		|	(`match` s `with` | `true` $\to t_1$ | `false` $\to t_2$)

We define substitution for the new type and renaming for the new terms.

$$\begin{cases} \ldots \\ [t/x]\texttt{Bool} = \texttt{Bool} \end{cases}$$

$$\begin{cases} \ldots \\ [x'/x]\texttt{true} = \texttt{true} \\ [x'/x]\texttt{false} = \texttt{false} \\ [x'/x](\texttt{match}\ s\ \texttt{with} \mid \texttt{true} \to t_1 \mid \texttt{false} \to t_2) \\ \qquad = \texttt{match}\ [x'/x]s\ \texttt{with} \mid \texttt{true} \to [x'/x]t_1 \mid \texttt{false} \to [x'/x]t_2 \end{cases}$$

We add a typing rule for each new term.

$$\frac{}{\Gamma|\Delta \vdash \texttt{false} \Longleftarrow \texttt{Bool}} \qquad \frac{}{\Gamma|\Delta \vdash \texttt{true} \Longleftarrow \texttt{Bool}}$$

$$\frac{\Gamma|\Delta \vdash s \Longrightarrow \texttt{Bool} \quad \Gamma|\Delta \vdash t_1 \Longleftarrow A \quad \Gamma|\Delta \vdash t_2 \Longleftarrow A}{\Gamma|\Delta \vdash (\texttt{match}\ s\ \texttt{with} \mid \texttt{true} \to t_1 \mid \texttt{false} \to t_2) \Longleftarrow A}$$

Finally, we add a new rule for equality of types.

$$\frac{}{\Delta \vdash \texttt{Bool} \equiv \texttt{Bool}}$$

3.2.2 Dependent Pairs. In some functions, such as `filter`, the output length cannot be computed from the input lengths. Such functions require "dependent pairs." We use $\Sigma(x\colon A_1).A_2$ to denote a dependent pair whose first element has type A_1 and in which the type of the second element, A_2, may depend on the value x of the first element.

$$\text{Type} \quad A \quad ::= \quad \ldots \mid \Sigma(x\colon A_1).A_2$$

Similarly to how Π types strictly generalize arrow types, Σ is a strict generalization of the standard "product" type. In general, $A_1 \times A_2 = \Sigma(x\colon A_1).A_2$, where $x \notin \mathrm{FV}(A_2)$.

We also add a constructor and destructor for dependent pairs.

Check. term	t	::=	...	
		|	$\langle t_1, t_2\rangle$	Pair
		|	$(\texttt{match}\ s\ \texttt{with} \mid \langle x_1, x_2\rangle \to t)$	Pattern match on pair

We use pattern matching as a general-purpose destructor for pairs; `fst` and `snd` can be implemented using pattern matching. For example, to access the first element, use $\texttt{match}\ x\ \texttt{with} \mid \langle y, \_\rangle \to y$.

We need to implement type substitution for Σ types. This is similar to the rule for Π types.

$$\begin{cases} \ldots \\ [t/x](\Sigma(y\colon A).B) = \Sigma(y\colon [t/x]A).[t/x]B \quad \text{if } y \neq x \text{ and } y \notin \mathrm{FV}(t) \end{cases}$$

We must also implement renaming for the new terms.

$$\begin{cases} \ldots \\ [x'/x]\langle t_1, t_2\rangle = \langle [x'/x]t_1, [x'/x]t_2\rangle \\ [x'/x]\ (\texttt{match}\ s\ \texttt{with} \mid \langle y_1, y_2\rangle \to t) \\ \qquad = \texttt{match}\ [x'/x]s\ \texttt{with} \mid \langle y_1, y_2\rangle \to [x'/x]t \quad \text{if } x \notin \{y_1, y_2\} \text{ and } \{y_1, y_2\} \cap \mathrm{FV}(t) = \emptyset \end{cases}$$

We add new typing rules for pairs and pattern matching.

$$\frac{\Gamma|\Delta \vdash t_1 \Longleftarrow A_1 \quad \Gamma|\Delta \vdash t_2 \Longleftarrow [t_1/x]A_2}{\Gamma|\Delta \vdash \langle t_1, t_2\rangle \Longleftarrow \Sigma(x\colon A_1).A_2}$$

$$\frac{\Gamma|\Delta \vdash s \Longrightarrow \Sigma(x\colon A_1).A_2 \quad \Gamma' = \Gamma, x_1 : A_1, x_2 : [x_1/x]A_2 \quad \Gamma'|\Delta \vdash t \Longleftarrow A}{\Gamma|\Delta \vdash (\texttt{match}\ s\ \texttt{with} \mid \langle x_1, x_2\rangle \to t) \Longleftarrow A}$$

Finally, we add a rule for equality of Σ types, similarly to the one for Π types.

$$\frac{\Delta \vdash A_1 \equiv A_2 \quad \Delta \vdash B_1 \equiv B_2}{\Delta \vdash \Sigma(x\colon A_1).B_1 \equiv \Sigma(x\colon A_2).B_2}$$

3.3 Examples

The following programs are all well-typed in our type system.

It is impossible to call `tail_nat` on an empty list because the vector has length $n+1$ and $\nexists n \in \mathbb{N}, n+1=0$:

```
let tail_nat : Π(n:Nat) . Vec Nat (n+1) -> Vec Nat n =
    λn.λv.match v with
          | cons _ _ xs -> xs
```

The length of the output of `concat` is guaranteed to be the sum of the lengths of the inputs:

```
let concat : Π(n:Nat) . Π(m:Nat) . Vec Nat n -> Vec Nat m -> Vec Nat (n+m) =
    fix concat.λn.λm.λv1.λv2.
        match v1 with
        | nil -> v2
        | cons n' x v1' ->
            cons (n' + m) x (concat n' m v1' v2)
```

As mentioned in section 3.2.2, filtering requires dependent pairs:

```
let filter_nat : Π(n:Nat) . Vec Nat n -> (Nat -> Bool) -> Σ(m:Nat) . Vec Nat m =
    fix filter_nat.λn.λv.λf.
        match v with
        | nil -> (0, nil)
        | cons n' x xs ->
            match (f x) with
            | true  -> cons n' x (filter_nat n' xs f)
            | false -> filter_nat n' xs f
```

Since the `Vec` type family is generic, it is possible to work with matrices:

```
let row_heads : Π(n:Nat) . Π(m:Nat) . Vec (Vec Nat (m+1)) n -> Vec Nat n =
    fix row_heads.λn.λm.λv.
        match v with
        | nil -> nil
        | cons n' row v' ->
            cons n' (head m row) (row_heads n' m v')
let row_tails :
            Π(n:Nat) . Π(m:Nat) . Vec (Vec Nat (m+1)) n -> Vec (Vec Nat m) n =
    fix row_tails.λn.λm.λv.
        match v with
        | nil -> nil
        | cons n' row v' ->
            cons n' (tail m row) (row_tails n' m v')
let transpose : Π(n:Nat) . Π(m:Nat) . Vec (Vec Nat m) n -> Vec (Vec Nat n) m =
    fix transpose.λn.λm.λv.
        match m with
        | 0 -> nil
        | m' + 1 ->
            cons m' (row_heads n m' v) (transpose n m' (row_tails n m' v))
```

3.4 Implementation

We implement the type checker as an OCaml program. For example, the judgements $\boxed{\Gamma|\Delta \vdash t \Longleftarrow A}$ and $\boxed{\Gamma|\Delta \vdash s \Longrightarrow A}$ are implemented as mutually-recursive functions with the following signatures.

```
let rec check (gamma : context) (delta : equation list)
        (tm : chk_tm) (typ : tp) : unit =
    (* ... *)
and synth (gamma : context) (delta : equation list) (s : syn_tm) : tp =
    (* ... *)
```

When comparing lengths and checking reachability, we construct propositions in the language of Presburger arithmetic (as explained in section 3.1.3). To decide whether these propositions are true or false, we use the OCaml bindings for the Z3 SMT solver [6, 7].

The type checker is available on GitHub at https://github.com/louis-hildebrand/chick. More examples of valid and invalid programs are in test/test.ml.

3.5 Challenges

At first, we used the assumptions introduced in each branch of a `match` expression by substitution. For example, for pattern matching on natural numbers, we used the following rule:

$$\frac{\Gamma \vdash x \Longrightarrow \mathtt{Nat} \quad [0/x]\Gamma \vdash [0/x]t_0 \Longleftarrow [0/x]A \quad \Gamma' = (\Gamma, y : \mathtt{Nat}) \quad [y+1/x]\Gamma' \vdash [y+1/x]t_1 \Longleftarrow [y+1/x]A}{\Gamma \vdash (\mathtt{match}\ x\ \mathtt{with} \mid 0 \to t_0 \mid y+1 \to t_1) \Longleftarrow A}$$

This approach has the advantage of being conceptually straightforward. To check whether two lengths are equivalent, it suffices to normalize each one and compare syntactically. Sums can be normalized by flattening, combining numerals, and sorting the terms. There is no need to construct propositions or invoke an SMT solver. It may even be possible to allow multiplication of lengths in this scheme, if we apply distributivity during normalization.

However, this strategy also has significant limitations.

First, it only works if the target of pattern matching is a natural number variable or a vector whose length is a variable. This excludes some important functions—most notably, `head` and `tail`. We would like `tail` to have type $\Pi(n : \mathtt{Nat}).\mathtt{Vec}\ A\ (n+1) \to \mathtt{Vec}\ A\ n$. The fact that the input vector has length $n+1$ guarantees it is not empty, but it also prevents us from using `match`.

Second, it requires implementing more general substitution methods. This includes substitution in a context, which is apparently prone to subtle bugs [8]. Furthermore, substitution into a checkable or synthesizable term may introduce non-normal expressions, which are syntactically invalid. For example, naively performing the substitution $[42/x](\mathtt{match}\ x\ \mathtt{with} \mid 0 \to 0 \mid \_ + 1 \to 1)$ yields the syntactically invalid term $\mathtt{match}\ 42\ \mathtt{with} \mid 0 \to 0 \mid \_ + 1 \to 1$. It may be possible to handle this by adapting hereditary substitution [9] or by introducing a synthesizable term $t : A$ for a checkable term with a type annotation. However, our approach described in section 3.1.3 sidesteps this issue altogether because it only requires *renaming*, which clearly cannot introduce non-normal terms.

Third, substitution can lead to confusing error messages. For example, suppose we try to check whether the expression $\mathtt{match}\ x\ \mathtt{with} \mid 0 \to \mathtt{cons}\ 0\ 0\ \mathtt{nil} \mid y+1 \to \mathtt{count}\ x$ has type $\mathtt{Vec}\ \mathtt{Nat}\ (x+1)$. In the "cons" branch we would try to check $\mathtt{count}\ (y+1)$ against type $\mathtt{Vec}\ \mathtt{Nat}\ (y+1+1)$. Confusingly, this would fail with a message like "$\mathtt{count}\ (y+1)$ does not have type $\mathtt{Vec}\ \mathtt{Nat}\ (y+1+1)$," despite the fact that the original source code does not include the term $\mathtt{count}\ (y+1)$ or the type $\mathtt{Vec}\ \mathtt{Nat}\ (y+1+1)$.

4 CONCLUSION

4.1 Limitations

Vector lengths. The length of a vector must be a numeral, a variable, or a sum of valid lengths. There are a few concrete limitations that stem from this rule.

First, arguments to dependent functions must abide by this rule. For example, if `count` : $\Pi(n\colon \texttt{Nat}).\texttt{Vec Nat } n$, then we cannot call `count` $(n \cdot k)$, since this would make the output have the invalid type `Vec Nat` $(n \cdot k)$. A workaround for this limitation is to first assign $n \cdot k$ to a new variable m and then call `count` m.

Second, this limitation makes it impossible to properly implement functions whose types involve multiplication or division, such as `split` or `join`. On one hand, `join` can at least be implemented so as to return a Σ type. That is, we tell the type checker that `join` returns a vector of some unknown length, not necessarily $n \cdot k$. The disadvantages of this workaround are that (1) the type checker will not prevent the programmer from returning a vector of the wrong size and (2) the type checker will not know that joining an $(n+1) \times (k+1)$ vector yields a non-empty vector. (Indeed, it may not if the programmer returns a vector of the wrong size.) On the other hand, it does not seem to be possible to enforce the pre-condition that the length of the input to `split` must be a product.

If this restriction were relaxed by allowing multiplication of lengths, type checking might become undecidable. Peano arithmetic, the theory of natural numbers with addition and multiplication, is undecidable. However, perhaps the propositions we wish to prove or disprove are simple enough that it would be acceptable, in practice, to try to prove or disprove them and fail if some timeout is exceeded.

Polymorphism. Although the `Vec` type is generic, it is not possible to write polymorphic functions like `map` : $\Pi(n\colon \texttt{Nat}).\texttt{Vec } \alpha\ n \to (\alpha \to \beta) \to \texttt{Vec } \beta\ n$. Instead, the programmer must manually write a variant of `map` for each input and output type.

One option for implementing polymorphism is the Hindley-Milner algorithm [10]. Rather than raising type errors directly, we return a set of constraints which are separately solved using "unification." For example, we might introduce a constraint set C to our judgements and then update the rule for pattern matching on a vector like this:

$$\frac{\begin{array}{c}\Delta_z = \Delta, y = 0 \quad \Gamma_s = \Gamma, x_1 : \forall.\texttt{Nat}, x_2 : \forall.\alpha, x_3 : \forall.\texttt{Vec } \alpha\ x_1 \quad \Delta_s = \Delta, y = x_1 + 1 \\ \Gamma|\Delta \vdash s \Longrightarrow_{C_1} B \quad \Gamma|\Delta_z \vdash t_0 \Longleftarrow_{C_2} A \quad \Gamma_s|\Delta_s \vdash t_1 \Longleftarrow_{C_3} A\end{array}}{\Gamma|\Delta \vdash (\texttt{match } s \texttt{ with } \texttt{|}\ \texttt{nil} \to t_0\ \texttt{|}\ \texttt{cons } x_1\ x_2\ x_3 \to t_1) \Longleftarrow_{C_1 \cup C_2 \cup C_3 \cup \{B = \texttt{Vec } \alpha\ y\}} A}$$

where the length equality check is handled during unification. However, it is not entirely clear how the assumptions Δ, which are necessary for the length equality judgement, can be made available to the unification algorithm. Furthermore, this rule introduces a fresh variable y which would need to be existentially quantified—in our existing type checker, all variables are universally quantified when comparing lengths. Perhaps it would be possible to interleave unification with type checking such that all the information necessary to compare lengths is available during checking rather than unification.

Explicit parameters. All inputs to dependent functions must be provided explicitly, which is not ergonomic. For example, although the list $[10, 20, 30]$ would intuitively be written

```
cons 10 (cons 20 (cons 30 nil))
```

the programmer must actually write

```
cons 2 10 (cons 1 20 (cons 0 30 nil))
```

Pattern matching. Our language has a separate pattern matching construct for each type. It would be more flexible to have one general pattern matching construct that includes a list of "patterns" with free variables. This would require some extra work to type check the patterns and bind the free variables. Another challenge would be to ensure all cases are covered. However, it should be possible to do so with a similar strategy to the one we already use for reachability (e.g., construct a proposition that, with the given assumptions, *at least* one of the given branches will match).

Runtime complexity. The runtime of checking propositions in Presburger arithmetic is at least doubly exponential in the length of the proposition [11]. Therefore, the type checker may run slowly if the programmer writes functions with deeply-nested `match` expressions or complex expressions for the vector lengths. However, this does not appear to be a serious limitation in practice. Functions like `map`, `fold_left`, and `tail` are small and involve simple propositions.

Typos. Although the type checker can detect errors that result in incorrectly-sized vectors, it cannot catch *all* typos. For example, the following two functions have an identical type signature and are both well-typed, but return different results. The type checker cannot stop the programmer from writing one when they meant to implement the other.

```
1 // count_down 3 = [3, 2, 1]
2 let count_down : Π(n:Nat) . Vec Nat n =
3   fix count_down.λn.
4     match n with
5     | 0 -> nil
6     | n' + 1 ->
7       cons n' n (count_down n')
```

```
// count_down 3 = [2, 1, 0]
let count_down : Π(n:Nat) . Vec Nat n =
  fix count_down.λn.
    match n with
    | 0 -> nil
    | n' + 1 ->
      cons n' n' (count_down n')
```

4.2 Related Work

Notable examples of dependently-typed programming languages and proof assistants include Agda [12], Idris [13], and Rocq [14]. Vectors are a classic use case for dependent types, and these languages all allow the programmer to define vectors as a user-defined datatype [2, 3, 15]. Furthermore, they allow dependencies on types other than natural numbers. Using the Curry-Howard correspondence, this lets programmers construct types representing propositions such as the equality of natural numbers or the associativity of addition [16].

As mentioned in section 4.1, it is cumbersome to always explicitly specify lengths when calling dependent functions. Agda, Idris, and Rocq all support "implicit parameters," which the compiler will try to infer automatically [17–19].

4.3 Summary

We presented the design and implementation of a bidirectional type checker for a small, monomorphic, dependently-typed list language. The language includes dependent pairs and products, vectors, natural numbers, and booleans. The type checker can enforce both pre-conditions and post-conditions for functions operating on vectors at compile time rather than runtime. To guarantee decidability of type checking, we restrict the lengths of vectors to the language of Presburger arithmetic. Assumptions about the lengths of vectors are introduced when pattern matching and are verified with an SMT solver. Despite the restriction on lengths, the language can be used to safely implement non-polymorphic versions of many useful list functions, such as `tail`, `concat`, and `filter`. Future work includes relaxing the limitation on lengths, making the language polymorphic, and supporting implicit parameters.

REFERENCES

[1] "Module List." https://ocaml.org/manual/5.1/api/List.html.

[2] "Indexed datatypes." https://agda.readthedocs.io/en/v2.6.1/language/data-types.html#indexed-datatypes. Accessed: 14 April 2025.

[3] "The Idris tutorial: Types and functions." https://docs.idris-lang.org/en/latest/tutorial/typesfuns.html#vectors. Accessed: 14 April 2025.

[4] M. Presburger, "Uber die vollstandigkeiteines gewissen systems der arithmetik ganzer zahlen, in welchen die addition als einzige operation hervortritt," in *Comptes-rendus du ler congres des mathematiciens des pays slavs*, 1929.

[5] C. Haase, "A survival guide to Presburger arithmetic," *ACM SIGLOG News*, vol. 5, p. 67–82, July 2018.

[6] "z3." https://github.com/Z3Prover/z3.

[7] "z3." https://opam.ocaml.org/packages/z3/.

[8] C. Jang. Personal communication, 2025.

[9] J. Errington, "COMP523 lecture: Dependent types." 26 March 2025.

[10] J. Errington, "COMP523 lecture: Hindley-milner." 12 March 2025.

[11] M. J. Fischer and M. O. Rabin, "Super-exponential complexity of Presburger arithmetic," in *Quantifier Elimination and Cylindrical Algebraic Decomposition* (B. F. Caviness and J. R. Johnson, eds.), (Vienna), pp. 122–135, Springer Vienna, 1998.

[12] "Welcome to Agda's documentation!." https://agda.readthedocs.io/en/latest/. Accessed: 17 April 2025.

[13] "Idris: A language for type-driven development." https://www.idris-lang.org/. Accessed: 17 April 2025.

[14] "Rocq." https://rocq-prover.org/. Accessed: 17 April 2025.

[15] "Library Stdlib.Vectors.Vector." https://rocq-prover.org/doc/master/stdlib/Stdlib.Vectors.Vector.html. Accessed: 17 April 2025.

[16] J. Errington, "COMP523 lecture: Agda." 28 March 2025.

[17] "Implicit arguments." https://agda.readthedocs.io/en/v2.6.1/language/implicit-arguments.html. Accessed: 14 April 2025.

[18] "The Idris tutorial: Types and functions." https://docs.idris-lang.org/en/latest/tutorial/typesfuns.html#implicit-arguments. Accessed: 14 April 2025.

[19] "Implicit arguments." https://rocq-prover.org/doc/master/refman/language/extensions/implicit-arguments.html. Accessed: 17 April 2025.