CPSC 449 Honours Thesis Proposal

Jonathan Chan

1 Introduction

Coq and other dependently-typed proof assistants rely on termination-checking to ensure soundness of proofs. Unfortunately, the current termination checkers are fragile and sensitive to the syntactic structure of code rather than the algorithm it computes. We propose to make them more robust and intuitive by using *sized types*.

1.1 Termination in Coq

Part of what makes Coq's termination checker fragile and unintuitive is that it requires that recursive functions are *guarded by destructors* [3]. This requires that the argument to a recursive call must be a structurally-recursive component of the corresponding parameter of the function, guaranteeing that recursive calls occur only on ever-smaller inductive terms. For example, given a definition of the natural numbers as the following:

```
Inductive Nat :=
  | O: Nat          (* zero *)
  | S: Nat -> Nat.  (* + 1  *)
```

The following function `shrink` is accepted by Coq, as `m` is a component of `n = S m`, but `grow` is not accepted.

```
Fixpoint shrink n :=
  match n with
    | O   => O
    | S m => shrink m
  end.
```

```
Fixpoint grow n := grow (S n).
```

Calls to other functions inside a recursive call are *unfolded* by replacing them with their definitions as needed when performing the termination check. For instance, if `shrink` were instead defined as follows:

```
Fixpoint shrink n :=
  match n with
  | O   => O
  | S m => shrink (ident m)
  end.
```

```
Definition ident n :=
  match n with
  | O   => n
  | S m => n
  end.
```

Unfolding might lead to:

```
Fixpoint shrink' n :=
  match n with
  | O    => O
  | S n' =>
    match n' with
    | O   => shrink' n'
    | S m => shrink' n'
    end
  end.
```

Coq accepts this since `n'` is a component of `n = S n'`. However, suppose we defined `ident` as the following:

```
Definition ident n :=
  match n with
  | O   => O
  | S m => S m
  end.
```

Then the recursive calls in `shrink` would be `shrink O` and `shrink (S m)`, but Coq does not accept this since `O` and `S m` are not explicitly structurally-smaller components of `n`. Thus the termination checker is fragile in that slightly changing the definition of `ident` in a semantically-equivalent way breaks the acceptance of the program, and is unintuitive in that users need to understand details of how the termination checker works in order to choose the correct definition of `ident` that will be accepted despite the two being semantically equivalent.

1.2 Termination-Checking using Sized Types

The key idea behind structural termination checking is that inductive objects have some size to them and that recursive calls are done on objects with a smaller size. Sized types [4] formalizes this notion of size and separates it from the structure of the object, instead attaching size information to its type. The size of a type counts the maximum number of layers of constructors that objects of that type have. For instance, for objects of type `Nat`, the sized type $\texttt{Nat}_i$ would contain all natural numbers less than i. Equivalently, an object with i layers of constructors has type of size at least i, so we would have:

$$\begin{aligned} \texttt{O}\colon\ & \texttt{Nat}_1, \texttt{Nat}_2, \texttt{Nat}_3, \ldots \\ \texttt{S O}\colon\ & \texttt{Nat}_2, \texttt{Nat}_3, \ldots \\ \texttt{S (S O)}\colon\ & \texttt{Nat}_3, \ldots \end{aligned}$$

And so on, with $\texttt{Nat}_\infty$ being the type of *all* natural numbers. Similarly, if we define polymorphic lists as follows:

```
Inductive List t :=
  | Nil: List₁ t
  | Cons: ∀i. t -> Listᵢ t -> Listᵢ₊₁ t.
```

Giving objects as precise a type as possible by choosing the smallest allowable size, lists of zeroes would then have types:

`Nil:`	$\texttt{List}_1$ `Nat`
`Cons O Nil:`	$\texttt{List}_2$ `Nat`
`Cons O (Cons O Nil):`	$\texttt{List}_3$ `Nat`

Constructor and function types are annotated with size indices (like `Cons` above), which can only be size variables, sums of size variables, constant multiples of size variables, or ∞. As an example, the type of a function that appends two lists might be:

$$\texttt{append:}\quad \forall i, j.\ \forall \texttt{t}.\ \texttt{List}_i\ \texttt{t} \to \texttt{List}_{j+1}\ \texttt{t} \to \texttt{List}_{i+j}\ \texttt{t}$$

Notice that the size of the type of the resulting list is one smaller than the sum of the sizes of the input lists since each input list ends in a `Nil` constructor and the appended list only needs one `Nil`. Functions that produce values whose type sizes cannot be expressed in terms of additions or constant multiples would require a return type indexed by ∞, such as the `factorial` function.

$$\texttt{factorial:}\quad \forall i.\ \texttt{Nat}_i \rightarrow \texttt{Nat}_\infty$$

Termination checking is then done by ensuring that the type of the argument to a recursive call has a sized type smaller than that of the parameter of the function. For instance, the types of `shrink` and `ident` from section 1.1 would be:

$$\begin{aligned}\texttt{shrink:}\quad & \forall i.\ \texttt{Nat}_i \rightarrow \texttt{Nat}_1\\ \texttt{ident:}\quad & \forall i.\ \texttt{Nat}_i \rightarrow \texttt{Nat}_i\end{aligned}$$

Suppose that we assume the type of the argument `n` of `shrink` to be $\texttt{Nat}_{i+1}$. Then the component `m` would have type $\texttt{Nat}_i$, the value from `ident m` would also have type $\texttt{Nat}_i$, and the argument to the recursive call is indeed smaller. Notice that we were able to determine this without needing to unfold the definition of `ident`; as long as size constraints are satisfied, termination is ensured. Since size indices are linear integer polynomials of size variables, solving these constraints is implemented as solving a system of integer linear inequalities, which can be done in $O(mn^2)$ [5], where m is the number of constraints and n is the number of size variables.

1.3 Inferring Sizes with Successor Sized Types

With sized types, structural termination checking is replaced by sized termination checking, reducing the fragility of the checker and easing reasoning about the termination of programs. However, it introduces the burden of size annotation on the user, who must now learn to add sizes to the types of functions correctly and with sufficient precision. It would be far preferable to have size information inferred rather than be specified by annotations to maintain usability. To do this, we can restrict the arithmetic on size variables to only three families of size indices: arbitrary sizes variables s for finite sizes, their successors $\hat{s}$, and ∞ [1, 2, 6]. With these families, the return type of constructors must have a successor size $\hat{s}$, while its recursive arguments must have a type with base size s. For example, `Cons` from section 1.2 now has the type:

$$\texttt{Cons:}\quad \forall s.\ \forall \texttt{t}.\ \texttt{t} \rightarrow \texttt{List}_s \rightarrow \texttt{List}_{\hat{s}}$$

A significant disadvantage of allowing only a successor function on size variables and removing addition is that it reduces the precision of the types of functions. For instance, append from section 1.2 now has the type:

$$\texttt{append:}\quad \forall s, r.\ \forall \texttt{t}.\ \texttt{List}_s\ \texttt{t} \rightarrow \texttt{List}_r\ \texttt{t} \rightarrow \texttt{List}_\infty\ \texttt{t}$$

Nevertheless, `append` remains typeable and passes sized termination checking, as do `shrink` and `ident`, which now have the following types:

$$\begin{aligned}\texttt{shrink:}\quad & \forall s.\ \texttt{Nat}_s \rightarrow \texttt{Nat}_s\\ \texttt{ident:}\quad & \forall s.\ \texttt{Nat}_s \rightarrow \texttt{Nat}_s\end{aligned}$$

Reasoning about their termination follows similarly:

1. The argument `n` to `shrink` is assumed to have type $\texttt{Nat}_{\hat{s}}$.
2. The component `m` must have type $\texttt{Nat}_s$.
3. By its function type and the type of its argument, the value from `ident m` must have type $\texttt{Nat}_s$.

Then the argument to the recursive call in `shrink` has a sized type smaller than the parameter of the function. Notably, this procedure is done entirely *without* explicit size annotations – the sizes of the function types are all inferred. Furthermore, solving the size constraints of successor sizes is also more efficient and can be done in $O(n^2)$ time [1], where n is again the number of size variables.

1.3.1 Example: Quicksort

The benefit of sized types is not restricted to reducing the fragility of termination checking. Terminating recursive functions which are not accepted using structural termination checking may instead be accepting when using sized termination checking. Consider the following implementation of quicksort:

```
Fixpoint filter {X} (p: X -> bool) l :=
  match l with
  | nil => nil
  | h :: t =>
    if   p h
    then h :: filter p t
    else filter p t
  end.
```

```
Fixpoint quicksort l :=
  match l with
  | nil => nil
  | h :: t =>
    quicksort (filter (<=? h) t) ++
    h :: quicksort (filter (>? h) t)
  end.
```

By inspection, we can see that `quicksort` is terminating, but this definition is not accepted by Coq because it expects the recursive call to take `t` or something structurally smaller than `t`; it cannot tell that `filter` will return something structurally equal or smaller in size without being provided a proof. On the other hand, these functions can be assigned successor sized types as follows:

$$\texttt{filter:}\quad \forall s.\ \forall \texttt{t}.\ (\texttt{t} \to \texttt{Bool}) \to \texttt{List}_s\ \texttt{t} \to \texttt{List}_s\ \texttt{t}$$

$$\texttt{quicksort:}\quad \forall s.\ \texttt{List}_s\ \texttt{Nat} \to \texttt{List}_\infty\ \texttt{Nat}$$

Notice that `filter` terminates because if `l` has size $\hat{s}$, then `t` must have size s. Then `quicksort` also terminates, since if `l` has size $\hat{s}$, then `t` must have size s, and by the type of `filter`, the argument to the recursive call of `quicksort` also has size s. Again, these annotations are inferred by the termination checker, so no additional code is required.

1.4 Objectives

The goal of this thesis is to reduce the fragility and the complexity of Coq's termination checker by implementing successor sized types in a non-trivial subset of Coq, replacing the existing syntactic termination checking. We will follow the typing and sizing rules in [6] and implement size inference using the algorithms presented its preceding works [1, 2]. This will reduce the task of termination-checking to type-checking and solving size constraints without requiring any additional user annotations, and accept some additional terminating recursive functions which may not currently be accepted by Coq.

1.5 Timeline

Task	Month (2019)
Set up development environment and become familiar with codebase	May
Add size fields to internal representation	June
Implement size inference algorithm	July
Modify termination checker to use sized types	September
Final fixes and wrap-up	November
Write-up	mid-November

References

[1] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Practical inference for type-based termination in a polymorphic setting. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications*, pages 71–85, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[2] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. CIC$\hat{}$: Type-based termination of recursive definitions in the calculus of inductive constructions. In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 257–271, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[3] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, pages 39–59, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[4] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 410–423, New York, NY, USA, 1996. ACM.

[5] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 4–13, Nov 1991.

[6] Jorge Luis Sacchini. *On type-based termination and dependent pattern matching in the calculus of inductive constructions.* Theses, École Nationale Supérieure des Mines de Paris, June 2011.