Total (Co)Programming
with Guarded Recursion

Andrea Vezzosi

Dept. of Comput. Sci. and Engin., Chalmers University of Technology, Sweden
vezzosi@chalmers.se

The standard way to ensure the totality of recursive definitions in systems like Coq or Agda relies on syntactic checks that mainly ensure that the recursive calls are made on structurally smaller arguments. However, such syntactic checks get in the way of code reuse and compositionality. This is especially the case when using higher-order combinators, because they can appear between a recursive call and the actual data it will process. The following definition of a map function for rose trees is an example of this.

```
data RoseTree A = Node A (List (RoseTree A))
mapRT : (A → B) → RoseTree A → RoseTree B
mapRT f (Node a ts) = Node (f a) (map (λ t → mapRT f t) ts)
```

The definition of `mapRT` is not accepted because syntactically `t` has no relation to `Node a ts`: we need to pay attention to the semantics of `map` to accept this definition as terminating. In fact a common workaround is to essentially inline the code for `map`. Such a system then actively fights abstraction, and offers few recourses other than sticking to some specific syntactic forms for (co)recursive definitions.

Recently there has been a fair amount of research into moving the information about how functions consume and produce data to the type level, so that totality checking can be more modular [3, 5, 1]. In particular, the previous work on Guarded Recursion [3, 5] has handled the case of ensuring totality for corecursion, i.e. manipulating infinite data. The issue of ensuring totality in a modular way for recursion, over well-founded data, was however left open. We address that problem by presenting a calculus that supports both corecursion and recursion with a single guarded fixed point combinator.

\[\Gamma, i : Time \vdash A(i) : Type\]
\[\Gamma \vdash \forall i. (\forall j < i. A(j)) \rightarrow A(i)\]
\[\Gamma \vdash \text{fix} f : \forall i. A(i)\]

The guarded fixpoint `fix` is presented here as well-founded recursion on the abstract type `Time`. Our bounded quantification `\forall j < i` takes the role of the delay modality of previous works.

The name `Time` comes from thinking of such values as representing the amount of time left to perform our computation.

Taking a fixed point on an universe `U` we can define both guarded and standard coinductive streams.

```
gStream A = fix (λ i. λ (X : ∀ j < i. U). A × (∀ j < i. X j))
Stream A = ∀ i. gStream A i
```

The induction hypothesis only provides an element of `U` when given a smaller `Time`, however we want to guard recursive occurrences with `∀ j < i`, so we have the smaller time `j` available.

\[1\text{If the definition of } map \text{ is available, Coq will attempt this automatically.}\]
The type $\forall i. \text{gStream } A i$ then corresponds to the standard coinductive stream type, since we can choose an $i$ big enough to inspect the stream as deeply as we need.

The above technique reproduces previous results [3, 5], however in our language we also obtain inductive types, by making use of the existential rather than universal quantification over $\text{Time}$.

$$g\text{Nat} = \text{fix } (\lambda i (X : \forall j < i. \ U) \cdot \top + (\exists j < i. \ X j))$$

$$\text{Nat} = \exists i. \ g\text{Nat} \ i$$

An element of $g\text{Nat} \ i$ is a Peano natural bounded in height by $i$. An element of $\text{Nat}$ is then an arbitrary natural number since we can pick a large enough bound.

It is however not enough to quantify over the time to get the right type, for example we risk having too many representations of $0$ if we can tell the difference between $(i, \text{inl } \text{tt})$ and $(j, \text{inl } \text{tt})$ for two different times $i$ and $j$.

The key idea is that values of type $\exists i. \ A$ must keep abstract the specific time they were built with, exactly like weak sums in System F. Intuitively $\text{Nat}$ will not be the initial algebra of $(\top +)$ unless $\text{Nat} \cong \top + \text{Nat}$ holds, so $\text{Nat}$ must be able to support both an interface and an equational theory where times play no role.

We characterize this invariance by a suitable interpretation of $\text{Time}$ and the time quantifiers in the relational parametric model of type theory of [2]. In particular while $\forall i$ is just a $\Pi$ type, the existential quantification is not simply a $\Sigma$ type but requires an interpretation analogous to the one of the existential types of the polymorphic lambda calculus [4]. In the calculus we provisionally internalize this invariance as type isomorphisms. In the specific case of $\text{Nat}$, both $(i, \text{inl } \text{tt})$ and $(j, \text{inl } \text{tt})$ get sent to $\text{inl } \text{tt}$ by the isomorphism, so we can conclude they are equal.

An example that highlights the expressivity of the resulting system is the ability to give a safe type to the $\text{unfold}$ combinators for lists:

$$\text{unfold} : (\forall i. \ S i \to \top + (A \times \exists j < i. \ S j)) \to \forall i. \ S i \to \text{List } A$$

$$\text{unfold } f = \text{fix } \lambda i \text{unfold}' \ s. \ \text{case } f i s \ \text{of}$$

$$\text{inl } \to \ [ ]$$

$$\text{inr } (a, (j, s')) \to \ a :: \text{unfold}' \ j \ s'$$

References


---

2This gives a sound denotational model for our calculus and we are currently working on deriving operational semantics that enjoy strong normalization and decidable typechecking by constraining the unfolding of the fix operator.