Rewrite Semantics for Guarded Recursion with Universal Quantification over Clocks

Aleš Bizjak\(^1\) and Rasmus Ejlers Møgelberg\(^2\)

\(^1\)Department of Computer Science, Aarhus University, Denmark, abizjak@cs.au.dk
\(^2\)IT University of Copenhagen, Denmark, mogel@itu.dk

Guarded recursion\(^7\) is an approach to solving recursive type equations where the type variable appears \textit{guarded} by a \(\triangleright\) (pronounced “later”) modal type operator. In particular the type variable could appear positively or negatively or both, e.g. the equation \(D = 1 + \triangleright(D \to D)\) has a unique solution\(^3\). On the term level the \textit{guarded fixed point combinator} \textit{fix} \(\forall\tau.\triangleright\tau\to\tau\) satisfies the equation \(f(\text{next}(\text{fix}\_A\_f)) = \text{fix}\_A\_f\) for any \(f: \triangleright\_A\to\_A\). Here \textit{next}: \_A\to\triangleright\_A\) is an operation that “freezes” an element that we have available now so that it is only available in the next time step.

One situation where guarded recursive types are useful is when faced with an unsolvable type equation. These arise for example when modelling advanced programming languages. In this case a solution to a guarded version of the equation often turns out to suffice, cf.\(^4\).

But guarded recursive versions of polymorphic type equations are also useful in type theory, even in settings where inductive and coinductive solutions to these equations are assumed to exist. To see this, consider the coinductive type of streams \(\text{Str}\), i.e., the final coalgebra for the functor \(\text{S}(X) = N \times X\). The proof assistants Coq and Agda allow programmers to construct streams using recursive definitions, but to ensure normalisation (and thereby consistency), these recursive definitions must be \textit{productive}, i.e., one must be able to compute the \(n\) first elements of a stream in finite time. Coq and Agda inspect recursive definitions for productivity by a syntactic property that does not interact well with higher-order functions.

Using the type of \textit{guarded streams} \(\text{Str}_g\), i.e., the unique type satisfying the equation \(\text{Str}_g = N \times \triangleright\text{Str}_g\), one can encode productivity in types: a productive recursive stream definition is exactly a term of type \(\triangleright\text{Str}_g\to\text{Str}_g\). To combine the benefits of coinductive and guarded recursive types, Atkey and McBride\(^8\) suggested a simply typed calculus with clock variables \(\kappa\) representing time streams, each with associated \(\triangleright\kappa\) type constructors, and universal quantification over clocks \(\forall\kappa\). If we think of the type \(\tau\) as being time-indexed along \(\kappa\), then the type \(\forall\kappa.\tau\) contains only elements which are available for all time steps. The relationship between the two notions of streams can then be captured by the encoding of the coinductive stream type as \(\text{Str} = \forall\kappa.\text{Str}_g^\kappa\). This encoding works for a general class of coinductive types including those given by polynomial functors, and these results were since extended to the dependently typed setting by Møgelberg\(^6\). In both cases the encodings were proved sound with respect to a denotational model and no rewrite semantics was given. This work is strongly related to sized types\(^2\,\,1\).

\textbf{Rewrite semantics} In this talk we will present our ongoing work on giving computational meaning to guarded recursion and \(\forall\kappa\). One of the challenges is to give operational meaning to a series of type isomorphisms assumed in both Atkey and McBride\(^8\) and Møgelberg\(^6\). These type isomorphisms are crucial to prove correctness of the encodings of coinductive types.

One of these isomorphisms is \(\forall\kappa.\triangleright^\kappa\tau) ≅ \forall\kappa.\tau\) stating that time steps on universally quantified clocks can be ignored. We suggest a partial eliminator for \(\triangleright^\kappa\) which suffices to encode the isomorphism. Note that an unrestricted eliminator would cause an inconsistency: if we allow a term of type \(\triangleright^\kappa\tau\to\tau\) for all \(\tau\), the fixed point of these maps give inhabitants of any type.
Another type isomorphism is $\forall \kappa. \tau \cong \tau$, valid whenever $\kappa$ is not free in $\tau$. This is used in the encoding of coinductive types to give the head map type $\text{Str} \rightarrow \mathbb{N}$ rather than $\text{Str} \rightarrow \forall \kappa. \mathbb{N}$. There is a natural term $\lambda x. \Lambda \kappa. x$ of type $\tau \rightarrow \forall \kappa. \tau$ and we want to reflect in the calculus that this term has an inverse. We suggest a new term construct $af$ (apply at fresh) with the typing rule

$$\Delta, | \Gamma \vdash t : \forall \kappa. \tau \quad \kappa \not\in \tau$$

Here $\Delta$ is a clock context (a finite set of clocks) and $\Gamma$ is an ordinary context of term variables whose types only contain clocks in the set $\Delta$. Intuitively, $af t$ should generate a fresh name for a clock variable and apply $t$ to it. Since fresh name generation is an effect it is non-trivial to give semantics to it in type theory, see e.g. [9, 5]. In this setting, however, we believe that it suffices to have a rule $af(\Lambda \kappa. t) \mapsto t$ applicable only when $\kappa$ is not free in $t$, along with a family of “commuting conversions” analogous to those of the $\nu$ construct of Nominal System T [8].

We will show that just the two type isomorphisms listed above suffice for constructing the remaining type isomorphisms listed in [3, 6]. In particular we can show $\forall \kappa$ commutes over binary sums, if those sums are encoded using dependent sums, universes and booleans in the standard way.

**Clock synchronisation** In the talk we will also address the problem of clock synchronisation, which was disallowed in previous work. This restriction arose in the elimination rule for $\forall \kappa$:

$$\Delta, | \Gamma \vdash t : \forall \kappa. \tau \quad \kappa' \in \Delta$$

which in [3, 6] had freshness side conditions on $\kappa'$. The reason for such a restriction was that the models did not support clock substitution, only clock permutation. In this talk we present a new model that does allow clock substitution, and thus proves the unrestricted version of (1) sound. This greatly simplifies the language and the rewrite semantics.

**References**


---

1 Compare this to the situation in, e.g. System F where the types $\mathbb{N}$ and $\forall \alpha. \mathbb{N}$ should be isomorphic, but the $\beta\nu$-laws are not sufficient to derive this; parametricity is needed.