Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 319 (2015) 83–101

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Model of Guarded Recursion With Clock Synchronisation

Aleš Bizjak [1]

Department of Computer Science, Aarhus University, Denmark

Rasmus Ejlers Møgelberg[2]

IT University of Copenhagen, Denmark

Abstract

Guarded recursion is an approach to solving recursive type equations where the type variable appears guarded by a modality to be thought of as a delay for one time step. Atkey and McBride proposed a calculus in which guarded recursion can be used when programming with coinductive data, allowing productivity to be captured in types. The calculus uses *clocks* representing time streams and *clock quantifiers* which allow limited and controlled elimination of modalities. The calculus has since been extended to dependent types by Møgelberg. Both works give denotational semantics but no rewrite semantics.
In previous versions of this calculus, different clocks represented separate time streams and *clock synchronisation* was prohibited. In this paper we show that allowing clock synchronisation is safe by constructing a new model of guarded recursion and clocks. This result will greatly simplify the type theory by removing freshness restrictions from typing rules, and is a necessary step towards defining rewrite semantics, and ultimately implementing the calculus.

Keywords: Guarded recursion, coinductive types, type theory, categorical semantics.

1 Introduction

Guarded recursion [17] is an approach to solving recursive type equations where the type variable appears *guarded* by a $\blacktriangleright$ (pronounced "later") modal type operator. In particular the type variable could appear positively or negatively or both, e.g. the equation $\sigma = 1 + \blacktriangleright(\sigma \to \sigma)$ has a unique solution [6]. On the term level the *guarded fixed point combinator* $\mathsf{fix}_\tau : (\blacktriangleright\tau \to \tau) \to \tau$ satisfies the equation $f\,(\mathsf{next}\,(\mathsf{fix}_\tau f)) = \mathsf{fix}_\tau f$ for any $f : \blacktriangleright\tau \to \tau$. Here $\mathsf{next} : \tau \to \blacktriangleright\tau$ is an operation that "freezes" an element that we have available now so that it is only available in the next time step.

[1] Email: abizjak@cs.au.dk

[2] Email: mogel@itu.dk

http://dx.doi.org/10.1016/j.entcs.2015.12.007

One situation where guarded recursive types are useful is when faced with an unsolvable type equation. These arise for example when modelling programming languages with sophisticated features. In this case a solution to a guarded version of the equation often turns out to suffice, as shown in [6].

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 Str, i.e., the final coalgebra for the functor $\mathbb{S}(X) = \mathbb{N} \times X$. Proof assistants like Coq [14] and Agda [18] allow programmers to construct streams using recursive definitions, but to ensure consistency, these must be *productive*, i.e., one must be able to compute the first n elements of a stream in finite time. Coq and Agda inspect recursive definitions for productivity by a *syntactic property* that is often overly conservative and does not interact well with higher-order functions.

Using the type of *guarded streams* Str_g, i.e., the *unique* type satisfying the equation $\mathsf{Str}_\mathsf{g} = \mathbb{N} \times \blacktriangleright \mathsf{Str}_\mathsf{g}$, one can encode productivity in types: a productive recursive stream definition is exactly a term of type $\blacktriangleright \mathsf{Str}_\mathsf{g} \to \mathsf{Str}_\mathsf{g}$. To combine the benefits of coinductive and guarded recursive types, Atkey and McBride [3] suggested a simply typed calculus with clock variables κ representing time streams, each with associated $\blacktriangleright^\kappa$ type constructors, and universal quantification over clocks $\forall\kappa$. If we think of the type τ as being time-indexed along κ, 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 $\mathsf{Str} = \forall\kappa.\, \mathsf{Str}_\mathsf{g}^\kappa$. 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 [16]. In both cases the encodings were proved sound with respect to a denotational model and no rewrite semantics was given. This paper is part of ongoing work to construct just that.

Clock synchronisation

In the calculus for guarded recursion with clocks, typing judgements are given in a context of clocks Δ, which is just a finite set of names for clocks, as well as a context of term variables Γ. Clock variables κ are simply names, there are no constants or operations on them, and there is no type of clocks. The introduction and elimination rules for $\forall\kappa$ as defined by Atkey and McBride [3] are

$$\frac{\Delta, \kappa \mid \Gamma \vdash t : \tau}{\Delta \mid \Gamma \vdash \Lambda\kappa.t : \forall\kappa.\tau} \qquad \frac{\Delta, \kappa' \mid \Gamma \vdash t : \forall\kappa.\tau \qquad \kappa' \notin \forall\kappa.\tau}{\Delta, \kappa' \mid \Gamma \vdash t[\kappa'] : \tau[\kappa'/\kappa]} \tag{1}$$

These rules are very similar to those for polymorphic types in System F [8], except for the freshness side condition on the elimination rule ensuring that the clocks κ and κ' are not synchronised in τ. The side condition makes the rule syntactically not well-behaved. For instance it is not clear that the β-rule for clock application preserves types.

This becomes a more serious problem in dependent type theory. The rule