Djinn, Monotonic
(extended abstract)

Conor McBride
Department of Computer and Information Sciences
University of Strathclyde
conor@cis.strath.ac.uk

Abstract

Dyckhoff's algorithm for contraction-free proof search in intuitionistic propositional logic (popularized by Augustsson as the type-directed program synthesis tool, *Djinn*) is a simple program with a rather tricky termination proof [4]. In this talk, I describe my efforts to reduce this program to a steady structural descent. On the way, I shall present an attempt at a compositional approach to explaining termination, via a uniform presentation of memoization.

1 Introduction

Let us grasp the problem. In order to focus on termination issues, I shall consider only the implicational fragment of the logic: higher-order implication is the termination troublemaker. In a language like Haskell, we might declare a type formulae—atoms closed under implication.

$$\text{data}\ \mathbf{Fmla}\ =\ \mathsf{Atom}\ \mathbf{String} \mid \mathbf{Fmla} \supset \mathbf{Fmla}$$

Again, for simplicity, let us consider the task merely of checking *whether* (rather than *how*) one formula holds, given hypotheses. The first step is to introduce hypotheses, until an atomic goal remains.

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{fmla}\ ::\ [\mathbf{Fmla}] \to \mathbf{Fmla} \to \mathbf{Bool}} \\ \mathsf{fmla}\ hs\ (h \supset g) & = & \mathsf{fmla}\ (h : hs)\ g \\ \mathsf{fmla}\ hs\ (\mathsf{Atom}\ a) & = & \mathsf{atom}\ hs\ a \end{array}$$

Next, we scan the hypotheses in the hope that one will deliver the goal.

$$\begin{array}{l} \mathsf{atom}\ ::\ [\mathbf{Fmla}] \to \mathbf{String} \to \mathbf{Bool} \\ \mathsf{atom}\ hs\ a\ \ =\ \ \mathsf{try}\ []\ hs\ \ \text{where} \\ \quad \mathsf{try}\ ::\ [\mathbf{Fmla}] \to [\mathbf{Fmla}] \to \mathbf{Bool} \\ \quad \mathsf{try}\ js\ []\qquad\quad =\ \ \mathsf{False} \\ \quad \mathsf{try}\ js\ (h : hs)\ \ =\ \ \mathsf{from}\ h\ a\ (js +\!\!+ hs)\ \vee\ \mathsf{try}\ (h : js)\ hs \end{array}$$

Note that try retains the list js of the hypotheses tried already. When we attempt to derive a from a chosen hypothesis h, we may need the other hypotheses $js +\!\!+ hs$ to solve any subgoals which may arise in the process, implemented as follows. Each premise of the hypothesis in use becomes a subgoal.

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{from}\ ::\ \mathbf{Fmla} \to \mathbf{String} \to [\mathbf{Fmla}] \to \mathbf{Bool}} \\ \mathsf{from}\ (\mathsf{Atom}\ b)\ a\ hs & = & b \equiv a \\ \mathsf{from}\ (g \supset h)\quad a\ hs & = & \mathsf{from}\ h\ a\ hs\ \wedge\ \mathsf{fmla}\ hs\ g \end{array}$$

Each time the algorithm backchains on a hypothesis, the context shrinks, but as the resulting subgoals are decomposed, the context grows. There is no apparent structural descent: Dyckhoff shows termination by appeal to a carefully crafted measure. The key point is that each step of backchaining and introduction eliminates a hypothesis of higher *order* than those added. Correspondingly, a lexicographic recursion structure lurks latently within this algorithm. Let us expose and develop it.

2 Memo Structures and Recursion Operators

One way to legitimize forms of recursion over some set X is by means of a *memo structure*—a record with two components (here in Agda notation):

$$
\begin{array}{l}
\text{record } \mathbf{Memo}\ (X : \mathbf{Set}) : \mathbf{Set}_1 \text{ where field} \\
\quad \mathsf{Below} : (X \to \mathbf{Set}) \to (X \to \mathbf{Set}) \\
\quad \mathsf{below} : (P : X \to \mathbf{Set}) \to ((x : X) \to \mathsf{Below}\ P\ x \to P\ x) \ \to\ ((x : X) \to \mathsf{Below}\ P\ x)
\end{array}
$$

Given a memo structure, we acquire its recursion operator

$$
\begin{array}{l}
\mathsf{rec} : (M : \mathbf{Memo}\ X) \to (P : X \to \mathbf{Set}) \to ((x : X) \to \mathsf{Below}\ M\ P\ x \to P\ x) \ \to\ ((x : X) \to P\ x) \\
\mathsf{rec}\ M\ P\ p\ x \ =\ p\ x\ (\mathsf{below}\ M\ P\ p\ x)
\end{array}
$$

In effect, $\mathsf{rec}\ M$ helps you to solve a problem P for any given x by offering you whatever information $\mathsf{Below}\ M$ remembers about x—typically that P holds for values which are in some sense 'below' x. Indeed, a popular choice for Below is

$$\mathsf{Below}\ P\ x \ =\ (y : X) \to y < x \to P\ y$$

for some well founded relation, $<$. This choice effectively packages Nordström's generic approach to terminating general recursion in type theory [9].

We are always free to make the trivial choice $\mathsf{M1} : \mathbf{Memo}\ X$ with

$$\mathsf{Below}\ P\ x \ =\ \mathbf{1}$$

which gives no useful information. Whilst the trivial memo structure supports only non-recursive programming, it proves helpful to have a 'nil' when composing memo structures.

For the natural numbers, consider $\mathsf{NatStep} : \mathbf{Memo}\ \mathbf{Nat}$, choosing

$$
\begin{array}{lcl@{\qquad}lcl}
\mathsf{Below}\ P\ \mathsf{zero} & = & \mathbf{1} & \mathsf{below}\ P\ \mathsf{zero}\quad p & = & \_ \\
\mathsf{Below}\ P\ (\mathsf{suc}\ n) & = & P\ n & \mathsf{below}\ P\ (\mathsf{suc}\ n)\ p & = & p\ n\ (\mathsf{below}\ P\ n)
\end{array}
$$

This gives $\mathsf{rec}\ \mathsf{NatStep}$ the one-step reach of Peano's induction principle. If *case analysis* exposes a top-level suc constructor, Below responds by offering an inductive hypothesis. For a two-step reach (perhaps to write Fibonacci's function), choose

$$
\begin{array}{lcl}
\mathsf{Below}\ P\ \mathsf{zero} & = & \mathbf{1} \\
\mathsf{Below}\ P\ (\mathsf{suc}\ \mathsf{zero}) & = & P\ \mathsf{one} \\
\mathsf{Below}\ P\ (\mathsf{suc}\ (\mathsf{suc}\ n)) & = & P\ n\ \times\ P\ (\mathsf{suc}\ n)
\end{array}
$$

I leave below as an exercise in this case. One can imagine constructing just the right memo structure to deliver the calls required by a particular function, and in this way to emulate the method of Bove and Capretta [2]

Alternatively, one might seek to build more reusable kit. For many-step constructor-guarded recursion in general, we may use a construction which dates back to my doctoral research with Goguen and McKinna [7], defining Below thus:

$$
\begin{array}{lcl@{\qquad}lcl}
\mathsf{Below}\ P\ \mathsf{zero} & = & \mathbf{1} & \mathsf{below}\ P\ \mathsf{zero}\quad p & = & \_ \\
\mathsf{Below}\ P\ (\mathsf{suc}\ n) & = & \mathsf{Below}\ P\ n\ \times\ P\ n & \mathsf{below}\ P\ (\mathsf{suc}\ n)\ p & = & (ps, p\ n\ ps)\ \text{ where} \\
 & & & & & \quad ps\ =\ \mathsf{below}\ P\ p\ n
\end{array}
$$

In this way, many-step recursion reduces to one-step recursion. We use this presentation as the basis for recursive computation in the Epigram language [8]. Termination checking in Epigram amounts to elaborating recursive calls as projections from such memo structures—a naïve search, constructing a an object in an underlying theory validated by type checking alone. We use type theory as a language of evidence. By contrast, Agda and Coq both rely on syntactic termination criteria, documented primarily by their implementations and invulnerable to reason. We may hope to develop a compositional library of memo structures and with it, a flexible method of accounting for termination.

3 Lexicographic Memo Structures

Given some $S : \mathbf{Set}$ and a family $T : S \to \mathbf{Set}$, we may form the type of dependent pairs $\Sigma\ S\ T$. If, moreover, we have memo structures $MS : \mathbf{Memo}\ S$ and $MT : (s\!:\!S) \to \mathbf{Memo}\ (T\ s)$, we may form their lexicographic combination:

$$
\begin{array}{l}
\mathsf{M\Sigma}_{S,T}\ MS\ MT : \mathbf{Memo}\ (\Sigma\ S\ T) \\
\mathsf{M\Sigma}_{S,T}\ MS\ MT\ =\ \text{record}\ \{ \\
\quad \mathsf{Below}\ P\ (s,t) = \mathsf{Below}\ (MT\ s)\ (\lambda t' \mapsto P\ (s,t'))\ t \times \mathsf{Below}\ MS\ (\lambda s' \mapsto (t'\!:\!T\ s') \to P\ (s',t'))\ s \\
\quad \mathsf{below}\ P\ p\ (s,t)\ =\ \{-\text{implementation details}-\} \\
\}
\end{array}
$$

That is, below (s,t) we may make recursive reference to (s,t') for any t' below t, or to (s',t') for any s' below s and any t' at all—we may blow t up if we reduce s. The implementation is easy in a type-directed setting, because the problem is so abstract!

4 Formulae and Contexts Revisited

The crucial observation on which proof search termination relies is that backchaining is strictly order-reducing. It is correspondingly useful to index formulae by an upper bound on their order. The strategy of turning a measure into an index has a track record of success [6, 3]!

$$
\begin{array}{l}
\text{data}\ \mathbf{Fmla}\ :\ \mathbf{Nat} \to \mathbf{Set}\ \text{where} \\
\quad \mathsf{atom} : \forall\{n\} \to \mathbf{String} \to \mathbf{Fmla}\ n \\
\quad \_ \supset \_ : \forall\{n\} \to \mathbf{Fmla}\ n \to \mathbf{Fmla}\ (\mathbf{suc}\ n) \to \mathbf{Fmla}\ (\mathbf{suc}\ n)
\end{array}
$$

We now have the information we need to refine the notion of context by dividing it into buckets according to order. We may take

$$
\begin{array}{l}
\mathsf{Ctxt}\ :\ \mathbf{Nat} \to \mathbf{Set} \\
\mathsf{Ctxt}\ \mathbf{zero}\quad = \mathbf{1} \\
\mathsf{Ctxt}\ (\mathbf{suc}\ n) = \mathsf{Bucket}\ (\mathbf{Fmla}\ n)\ \times\ \mathbf{Ctxt}\ n
\end{array}
$$

where, for our purposes, a Bucket is a list of *known* length

$$\mathsf{Bucket}\ X\ =\ \Sigma\ \mathbf{Nat}\ \lambda i \mapsto \mathbf{Vec}\ X\ i$$

Correspondingly, deleting any element from a Bucket makes its length structurally smaller. Lexicographic combination of numerical recursion with trivial vector recursion

$$
\begin{array}{l}
\mathsf{MBucket}\ :\ \mathbf{Memo}\ (\mathsf{Bucket}\ X) \\
\mathsf{MBucket}\ =\ \mathsf{M\Sigma}\ \mathsf{NatStep}\ (\lambda n \mapsto \mathsf{M1})
\end{array}
$$

captures the idea that recursion makes sense for *any* vector whenever the length decreases by one.

Contexts, meanwhile, are iterated products, so they also support iterated lexicographic recursion.

$$\begin{array}{l} \mathsf{MCtxt} \;:\; (n\!:\!\mathbf{Nat}) \to \mathbf{Memo}\,(\mathsf{Ctxt}\;n) \\ \mathsf{MCtxt}\;\mathbf{zero} \quad\;\; = \mathsf{M1} \\ \mathsf{MCtxt}\;(\mathbf{suc}\;n) = \mathsf{M\Sigma}\;\mathsf{MBucket}\;(\lambda_{\_} \mapsto \mathsf{MCtxt}\;n) \end{array}$$

Crucially, this allows us to take out a higher-order formula from an earlier bucket and backchain on it, adding formulae to lower-order buckets. We have thus justified the recursion strategy for Dyckhoff's method in structural terms.

5 Overview of Talk

In my talk, I shall show the program which arises from this analysis of formulae and contexts. It falls outside the class readily acepted by Agda's termination oracle [1] but is codable 'Epigram-style' by direct appeal to rec. I shall consider how memo structures might give rise to a more flexible economy of termination explanation, using the typechecker as the basis for trust.

References

[1] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *J. Funct. Program.*, 12(1):1–41, 2002.

[2] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

[3] Ana Bove and Thierry Coquand. Formalising bitonic sort in type theory. In Filliâtre et al. [5], pages 82–97.

[4] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 57(3):795–807, 1992.

[5] Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors. *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*. Springer, 2006.

[6] Conor McBride. First-order unification by structural recursion. *J. Funct. Program.*, 13(6):1061–1075, 2003.

[7] Conor McBride, Healfdene Goguen, and James McKinna. A few constructions on constructors. In Filliâtre et al. [5], pages 186–200.

[8] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[9] Bengt Nordström. Terminating general recursion. *BIT*, 28(3):605–619, 1988.