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.

```haskell
data Fmla = Atom String | Fmla ⊃ 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.

```haskell
fmla :: [Fmla] → Fmla → Bool
fmla hs (h ⊃ g) = fmla (h : hs) g
fmla hs (Atom a) = atom hs a
```

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

```haskell
atom :: [Fmla] → String → Bool
atom hs a = try [] hs where
try :: [Fmla] → [Fmla] → Bool
try js [] = False
try js (h : hs) = from h a (js ++ hs) ∨ try (h : js) hs
```

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.

```haskell
from :: Fmla → String → [Fmla] → Bool
from (Atom b) a hs = b ≡ a
from (g ⊃ h) a hs = from h a hs ∧ fmla hs g
```

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 latent 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):

\[
\text{record } \text{Memo} \ (X : \text{Set}) : \text{Set} \_ \text{where field}
\begin{align*}
\text{Below} & : (X \to \text{Set}) \to (X \to \text{Set}) \\
\text{below} & : (P : X \to \text{Set}) \to ((x : X) \to \text{Below} P x \to P x) \to ((x : X) \to \text{Below} P x)
\end{align*}
\]

Given a memo structure, we acquire its recursion operator

\[
\text{rec} : (M : \text{Memo} \ X) \to (P : X \to \text{Set}) \to ((x : X) \to \text{Below} M P x \to P x) \to ((x : X) \to P x)
\]

\[
\text{rec} M P px = px \ (\text{below} M P px)
\]

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

\[
\text{Below} P x = (y : X) \to y < x \to P y
\]

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 \( M1 : \text{Memo} \ X \) with

\[
\text{Below} P x = 1
\]

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 \( \text{NatStep} : \text{Memo} \ \text{Nat} \), choosing

\[
\begin{align*}
\text{Below} P \text{ zero} & = 1 \\
\text{below} P \text{ zero} & = - \\
\text{Below} P \text{ (suc } n\text{)} & = P n \\
\text{below} P \text{ (suc } n\text{)} & = p n \ (\text{below} P n)
\end{align*}
\]

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

\[
\begin{align*}
\text{Below} P \text{ zero} & = 1 \\
\text{Below} P \text{ (suc zero)} & = P \text{ one} \\
\text{Below} P \text{ (suc } (\text{suc } n)) & = P n \times P \text{ (suc } n\text{)}
\end{align*}
\]

I leave \( \text{below} \) 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 \( \text{Below} \) thus:

\[
\begin{align*}
\text{Below} P \text{ zero} & = 1 \\
\text{Below} P \text{ (suc } n\text{)} & = \text{Below} P n \times P n \\
\text{below} P \text{ zero} & = - \\
\text{below} P \text{ (suc } n\text{)} & = \text{below} P \text{ (suc } n\text{)}
\end{align*}
\]

\[
\begin{align*}
\text{below} P \text{ (suc } (\text{suc } n)) & = (ps, p n \ ps) \ \text{where} \\
ps & = \text{below} P p n
\end{align*}
\]
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 $S : \text{Set}$ and a family $T : S \rightarrow \text{Set}$, we may form the type of dependent pairs $\Sigma S T$. If, moreover, we have memo structures $MS : \text{Memo } S$ and $MT : (s : S) \rightarrow \text{Memo } (T s)$, we may form their lexicographic combination:

$$M_{\Sigma S,T} \; MS \; MT : \text{Memo } (\Sigma S T)$$

$$M_{\Sigma S,T} \; MS \; MT = \text{record} \{$$

$\quad \text{Below } P (s,t) = \text{Below } (MT s) (\lambda t' \rightarrow P (s,t')) \times \text{Below } MS (\lambda s' \rightarrow (t':T s') \rightarrow P (s',t')) \; s$

$\quad \text{below } P p (s,t) = \{ \text{—implementation details—} \}$

$$\}\}

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]!

```
data Fmla : Nat \rightarrow \text{Set} where
  atom : \forall \{n\} \rightarrow \text{String} \rightarrow Fmla n
  ⊃ : \forall \{n\} \rightarrow Fmla n \rightarrow Fmla (suc n) \rightarrow Fmla (suc n)
```

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

$$\begin{align*}
\text{Ctx} & : \text{Nat} \rightarrow \text{Set} \\
\text{Ctx} \; \text{zero} & = 1 \\
\text{Ctx} \; (\text{suc } n) & = \text{Bucket } (\text{Fmla } n) \times \text{Ctx } n
\end{align*}$$

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

$$\text{Bucket } X = \Sigma \text{Nat } \lambda i \rightarrow \text{Vec } X i$$

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

$$\begin{align*}
\text{MBucket} & : \text{Memo } (\text{Bucket } X) \\
\text{MBucket} & = M_{\Sigma \text{NatStep } (\lambda n \rightarrow \text{M1})}
\end{align*}$$
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.

\[
\text{MCtxt} : (n : \text{Nat}) \to \text{Memo} (\text{Ctx} n)
\]

\[
\text{MCtxt} \text{ zero} = \text{M1}
\]

\[
\text{MCtxt} (\text{suc} n) = \text{M} \Sigma \text{MBucket (\lambda \_ \to \text{MCtxt} n)}
\]

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 accepted by Agda’s termination oracle but is codable ‘Epigram-style’ by direct appeal to \texttt{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


