Verifying Program Optimizations in Agda
Case Study: List Deforestation

Andreas Abel
16 July 2009

This is a case study on proving program optimizations correct. We prove the \textit{foldr-unfold} fusion law, an instance of deforestation. As a result we show that the summation of the first \( n \) natural numbers, implemented by producing the list \( n :: \ldots :: 1 :: 0 :: [] \) and summing up its elements, can be automatically optimized into a version which does not use an intermediate list.

\begin{verbatim}
module Fusion where
open import Data.Maybe
open import Data.Nat
open import Data.Product
open import Data.List hiding (downFrom)
open import Relation.Binary.PropositionalEquality
import Relation.Binary.EqReasoning as Eq

From Data.List we import \texttt{foldr} which is the standard iterator for lists.

\begin{verbatim}
foldr : {a b : Set} → (a → b → b) → b → List a → b
foldr c n [] = n
foldr c n (x :: xs) = c x (foldr c n xs)
\end{verbatim}

Further, \texttt{sum} sums up the elements of a list by replacing \([\ ]\) by \(0\) and \(\_::\_\) by \(+\).

\begin{verbatim}
sum : List N → N
sum = foldr (+) 0
\end{verbatim}

Finally, \texttt{unfold} is a generic list producer. It takes two parameters, \( f : B \rightarrow \text{Maybe} \ (A \times B) \), the transition function, and \( s : B \), the start state. Now \( f \) is iterated on the start state. If the result of applying \( f \) on the current state is \texttt{nothing}, an empty list is output and the list production terminates. If the application of \( f \) yields just \((x, s')\) then \( x \) is taken to be the next element of the list and \( s' \) the new state of the production.

In Agda, everything needs to terminate, so we add a (hidden) parameter \( n : N \) which is an upper bound on the number of elements to be produced. Each iteration decreases
this number. Consequently the type $B : \mathbb{N} \rightarrow \text{Set}$ is now parameterized by $n$, and $f : \forall \{n\} \rightarrow B (\text{suc } n) \rightarrow \text{Maybe } (A \times B n)$ can only be applied to a state $B (\text{suc } n)$ where still an element could be output.

$$\text{unfold} : \{A : \text{Set}\} (B : \mathbb{N} \rightarrow \text{Set})$$

$$\forall \{n\} \rightarrow B n \rightarrow \text{List } A$$

$$\text{unfold } B f \{n = \text{zero}\} s = []$$

$$\text{unfold } B f \{n = \text{suc } n\} s \text{ with } f s$$

$$\text{... | nothing } = []$$

$$\text{... | just } (x, s') = x :: \text{unfold } B f s'$$

A typical instance of unfold is the function downFrom from the standard library with the behavior $\text{downFrom } 3 = 2 :: 1 :: 0 :: []$. We reimplement it here, avoiding local definitions as used in the standard library.

$$\text{data Singleton} : \mathbb{N} \rightarrow \text{Set where}$$

$$\text{wrap} : (n : \mathbb{N}) \rightarrow \text{Singleton } n$$

$$\text{downFromF} : \forall \{n\} \rightarrow \text{Singleton } (\text{suc } n) \rightarrow \text{Maybe } (\mathbb{N} \times \text{Singleton } n)$$

$$\text{downFromF} \{n\} (\text{wrap } \circ (\text{suc } n)) = \text{just } (n, \text{wrap } n)$$

$$\text{downFrom} : \mathbb{N} \rightarrow \text{List } \mathbb{N}$$

$$\text{downFrom } n = \text{unfold } \text{Singleton } \text{downFromF} (\text{wrap } n)$$

$$\text{sumFrom} : \mathbb{N} \rightarrow \mathbb{N}$$

$$\text{sumFrom } \text{zero} = \text{zero}$$

$$\text{sumFrom } (\text{suc } n) = n + \text{sumFrom } n$$

Our goal is to show the theorem $\forall n \rightarrow \text{sum } (\text{downFrom } n) \equiv \text{sumFrom } n$.

The theorem follows from general considerations:

- sum is a foldr, it consumes a list.
- downFrom is a unfold, it produces a list.

The list is only produced to be consumed again. Can we optimize away the intermediate list?

Removing intermediate data structures is called deforestation, since data structures are tree-shaped in the general case.

In our case, we would like to fuse an unfold followed by a foldr into a single function foldUnfold which does not need lists. We observe that a foldr after an unfold satisfies the following equations:

$$\text{foldr } c n (\text{unfold } B f \{\text{zero}\} s) = n$$

$$\text{foldr } c n (\text{unfold } B f \{\text{suc } m\} s \mid f s = \text{nothing}) = n$$

$$\text{foldr } c n (\text{unfold } B f \{\text{suc } m\} s \mid f s = \text{just } (x, s'))$$

$$2$$
In the recursive case, the pattern \( \text{foldr}\ c\ n \circ \text{unfold}\ B\ f \) resurfaces, and it contains all the recursive calls to \( \text{foldr} \) and \( \text{unfold} \). Hence, we can introduce a new function \( \text{foldUnfold} \) as

\[
\text{foldUnfold}\ c\ n\ B\ f = \text{foldr}\ c\ n \circ \text{unfold}\ B\ f
\]

\( \text{foldUnfold} \) does not produce an intermediate list.

It is easy to show that the definition of \( \text{foldUnfold} \) is correct.

\[
\text{foldr-unfold} : \{ A : \text{Set} \} \rightarrow (c : A \rightarrow C \rightarrow C) \rightarrow (n : C) \rightarrow (B : \text{Set}) \rightarrow (f : \forall \{ n \} \rightarrow B (\text{suc}\ n) \rightarrow \text{Maybe}\ (A \times B n)) \rightarrow \\
\{ m : \text{ℕ} \} \rightarrow (s : B m) \rightarrow \\
\text{foldr}\ c\ n\ (\text{unfold}\ B\ f\ s) \equiv \text{foldUnfold}\ c\ n\ B\ f\ s
\]

\( \text{foldr-unfold} \) is a special case of \( \text{foldUnfold} \).

\[
\text{lem1} : \forall \{ n \} \rightarrow \text{foldUnfold}_{+}\ 0\ \text{Singleton} \downarrow\ \text{sumFrom}\ (\text{wrap}\ n) \equiv \text{sumFrom}\ n
\]

Our theorem follows by composition of the two lemmata.
sumFrom n

where open ≡ Reasoning

That’s it!