Verifying Program Optimizations in Agda
Case Study: List Deforestation

Andreas Abel
3 July 2012

This is a case study on proving program optimizations correct. We prove the 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 \( n :: ... :: 1 :: 0 :: [] \) and summing up its elements, can be automatically optimized into a version which does not use an intermediate list.

```agda
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 foldr which is the standard iterator for lists.

```
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)
```

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

```
sum : List N → N
sum = foldr _+_ 0
```

Finally, \( \text{unfold} \) 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 \( \text{nothing} \), 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 $B : N \to \text{Set}$ is now parameterized by $n$, and $f : \forall \{n\} \to B (\text{suc } n) \to \text{Maybe } (A \times B \ n)$ can only be applied to a state $B (\text{suc } n)$ where still an element could be output.

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

$(f : \forall \{n\} \to B (\text{suc } n) \to \text{Maybe } (A \times B \ n)) \to$

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

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

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

... | nothing = []

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

A typical instance of $\text{unfold}$ is the function $\text{downFrom}$ 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.

$$\text{data } \text{Singleton} : N \to \text{Set where}$$

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

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

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

$$\text{downFrom} : N \to \text{List } N$$

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

$$\text{sumFrom} : N \to N$$

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

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

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

The theorem follows from general considerations:

- $\text{sum}$ is a $\text{foldr}$, it consumes a list.

- $\text{downFrom}$ is a $\text{unfold}$, 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 $\text{deforestation}$, since data structures are tree-shaped in the general case.

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

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

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

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

In the recursive case, the pattern \(\text{foldr}\ c\ n\ \cdot\ \text{unfold}\ B\ f\ \text{resurfaces, and it contains all the recursive calls to foldr and unfold.}

Hence, we can introduce a new function \(\text{foldUnfold}\) as

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

\[
\begin{align*}
\text{foldUnfold} & : \{A, C : \text{Set}\} \to (A \to C \to C) \to C \\
& \quad \to (B : \mathbb{N} \to \text{Set}) \to (\forall \{n\} \to B (\text{suc } n) \to \text{Maybe } (A \times B n)) \to \\
& \quad \to \{n : \mathbb{N}\} \to B n \to C \\
\text{foldUnfold}\ c\ n\ B\ f\ \{\text{zero}\}\ s & = n \\
\text{foldUnfold}\ c\ n\ B\ f\ \{\text{suc}\ m\}\ s\ \text{with } f\ s \\
& \quad \mid \text{nothing} = n \\
& \quad \mid \text{just } (x, s') = c x (\text{foldUnfold}\ c\ n\ B\ f\ \{m\}\ s')
\end{align*}
\]

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

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

\[
\begin{align*}
\text{foldr-unfold} & : \{A, C : \text{Set}\} \to (c : A \to C \to C) \to (n : C) \to \\
& \quad \to (B : \mathbb{N} \to \text{Set}) \to (f : \forall \{n\} \to B (\text{suc } n) \to \text{Maybe } (A \times B n)) \to \\
& \quad \to \{m : \mathbb{N}\} \to (s : B m) \to \\
& \quad \text{foldr}\ c\ n\ (\text{unfold}\ B\ f\ s) \equiv \text{foldUnfold}\ c\ n\ B\ f\ s \\
\text{foldr-unfold}\ c\ n\ B\ f\ \{\text{zero}\}\ s & = \text{refl} \\
\text{foldr-unfold}\ c\ n\ B\ f\ \{\text{suc}\ m\}\ s\ \text{with } f\ s \\
& \quad \mid \text{nothing} = \text{refl} \\
& \quad \mid \text{just } (x, s') = \text{cong } (c x) (\text{foldr-unfold}\ c\ n\ B\ f\ \{m\}\ s')
\end{align*}
\]

\(\text{sumFrom}\) is a special case of \(\text{foldUnfold}\).

\[
\begin{align*}
\text{lem1} & : \forall \{n\} \to \text{foldUnfold}\ _+_\ 0\ \text{Singleton}\ \text{downFromF}\ (\text{wrap } n) \equiv \text{sumFrom } n \\
\text{lem1}\ \{\text{zero}\} & = \text{refl} \\
\text{lem1}\ \{\text{suc } n\} & = \text{cong } (\lambda m \to n + m) (\text{lem1}\ \{n\})
\end{align*}
\]

Our theorem follows by composition of the two lemmata.

\[
\begin{align*}
\text{thm} & : \forall \{n\} \to \text{sum } (\text{downFrom } n) \equiv \text{sumFrom } n \\
\text{thm}\ \{n\} & = \begin{align*}
\text{sum } (\text{downFrom } n) \\
& \equiv \langle \text{refl} \rangle \\
\text{foldr}\ _+_\ 0\ (\text{unfold}\ \text{Singleton}\ \text{downFromF}\ (\text{wrap } n)) \\
& \equiv \langle \text{foldr-unfold}\ _+_\ 0\ \text{Singleton}\ \text{downFromF}\ (\text{wrap } n) \rangle \\
\text{foldUnfold}\ _+_\ 0\ \text{Singleton}\ \text{downFromF}\ (\text{wrap } n) \\
& \equiv \langle \text{lem1}\ \{n\} \rangle
\end{align*}
\]
sumFrom n

where open ≡ -Reasoning

That's it!