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 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 the its elements, can be automatically optimized into a version which does not use an intermediate list.

```
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, sum sums up the elements of a list by replacing [] by 0 and _::_ by +.

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

Finally, unfold is a generic list producer. It takes two parameters, f : B → Maybe (A × 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 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 $\mathsf{B} : \mathbb{N} \to \mathsf{Set}$ is now parameterized by n, and $\mathsf{f} : \forall \{\mathsf{n}\} \to \mathsf{B}\ (\mathsf{suc}\ \mathsf{n}) \to \mathsf{Maybe}\ (\mathsf{A} \times \mathsf{B}\ \mathsf{n})$ can only be applied to a state $\mathsf{B}\ (\mathsf{suc}\ \mathsf{n})$ where still an element could be output.

```
unfold : {A : Set} (B : ℕ → Set)
  (f : ∀ {n} → B (suc n) → Maybe (A × B n)) →
  ∀ {n} → B n → List A
unfold B f {n = zero} s = []
unfold B f {n = suc n} s with f s
... | nothing = []
... | just (x, s') = x :: unfold B f s'
```

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

```
data Singleton : ℕ → Set where
  wrap : (n : ℕ) → Singleton n

downFromF : ∀ {n} → Singleton (suc n) → Maybe (ℕ × Singleton n)
downFromF {n} (wrap ◦ (suc n)) = just (n, wrap n)

downFrom : ℕ → List ℕ
downFrom n = unfold Singleton downFromF (wrap n)


sumFrom : ℕ → ℕ
sumFrom zero    = zero
sumFrom (suc n) = n + sumFrom n
```

Our goal is to show the theorem ∀ n → sum (downFrom n) ≡ 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:

```
foldr c n (unfold B f {zero} s) = n
foldr c n (unfold B f {suc m} s | f s = nothing) = n
foldr c n (unfold B f {suc m} s | f s = 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 foldr c n ∘ unfold B f resurfaces, and it contains all the recursive calls to foldr and unfold. Hence, we can introduce a new function foldUnfold as

```
foldUnfold c n B f = foldr c n ∘ unfold B f
```

```
foldUnfold : {A C : Set} → (A → C → C) → C →
  (B : ℕ → Set) → (∀ {n} → B (suc n) → Maybe (A × B n)) →
  {n : ℕ} → B n → C
foldUnfold c n B f {zero} s = n
foldUnfold c n B f {suc m} s with f s
... | nothing    = n
... | just (x, s') = c x (foldUnfold c n B f {m} s')
```

foldUnfold does not produce an intermediate list.

It is easy to show that the definition of foldUnfold is correct.

```
foldr-unfold : {A C : Set} → (c : A → C → C) → (n : C) →
  (B : ℕ → Set) → (f : ∀ {n} → B (suc n) → Maybe (A × B n)) →
  {m : ℕ} → (s : B m) →
  foldr c n (unfold B f s) ≡ foldUnfold c n B f s
foldr-unfold c n B f {zero} s = refl
foldr-unfold c n B f {suc m} s with f s
... | nothing = refl
... | just (x, s') = cong (c x) (foldr-unfold c n B f {m} s')
```

sumFrom is a special case of foldUnfold.

```
lem1 : ∀ {n} → foldUnfold _+_ 0 Singleton downFromF (wrap n) ≡ sumFrom n
lem1 {zero} = refl
lem1 {suc n} = cong (λ m → n + m) (lem1 {n})
```

Our theorem follows by composition of the two lemmata.

```
thm : ∀ {n} → sum (downFrom n) ≡ sumFrom n
thm {n} = begin
  sum (downFrom n)
    ≡⟨ refl ⟩
  foldr _+_ 0 (unfold Singleton downFromF (wrap n))
    ≡⟨ foldr-unfold _+_ 0 Singleton downFromF (wrap n) ⟩
  foldUnfold _+_ 0 Singleton downFromF (wrap n)
    ≡⟨ lem1 {n} ⟩
```

```
sumFrom n
  ∎
where open ≡-Reasoning
```

That's it!