Lifting Definition Option*

René Thiemann

May 28, 2015

Abstract

We implemented a command, `lift-definition-option`, which can be used to easily generate elements of a restricted type \{x :: 'a. P x\}, provided the definition is of the form \(\lambda y_1 \ldots y_n. \text{if check } y_1 \ldots y_n \text{ then Some (generate } y_1 \ldots y_n :: 'a) \text{ else None and check } y_1 \ldots y_n \implies P \text{ (generate } y_1 \ldots y_n)\) can be proven.

In principle, such a definition is also directly possible using one invocation of `lift-definition`. However, then this definition will not be suitable for code-generation. To this end, we automated a more complex construction of Joachim Breitner which is amenable for code-generation, and where the test `check y_1 \ldots y_n` will only be performed once. In the automation, one auxiliary type is created, and Isabelle's lifting- and transfer-package is invoked several times.

Contents

1 Introduction 2
2 Usage and limitations 4
3 Examples 4
  3.1 A simple restricted type without type-parameters . . . . . . . 4
  3.2 Examples with type-parameters in the restricted type. . . . . 5
  3.3 Example from `IsaFoR/Cifa` . . . . . . . . . . . . . . . . . . . 5
  3.4 Code generation tests and derived theorems . . . . . . . . . . 6

theory Lifting-Definition-Option-Explanation
imports
  Lifting-Definition-Option
  Rat
begin

*This research is supported by FWF (Austrian Science Fund) project Y 757.
1 Introduction

Often algorithms expect that their input satisfies some specific property $P$. For example, some algorithms require lists as input which have to be sorted, or numbers which have to be positive, or programs which have to be well-typed, etc. Here, there are at least two approaches how one can reason about these algorithms.

The first approach is to guard all soundness properties of the algorithm by the additional precondition $P$ input. So, as an example, a binary search algorithm might take arbitrary lists as input, but only if the input list is sorted, then the result of the search is meaningful. Whereas for binary search, this approach is reasonable, there might be problems that the restriction on the input is even crucial for actually defining the algorithm, since without the restriction the algorithm might be non-terminating, and thus, cannot be easily defined using Isabelle’s function-package [4]. As an example, consider an approximation algorithm for $\sqrt{x}$, where the algorithm should stop once the current approximation $y$ satisfies $|x^2 - y^2| < \delta$. Imagine now, that $\delta$ is a negative number.

To this end, in the second approach the idea is to declare restricted types, so that the algorithms are only invoked with inputs which satisfy the property $P$. For example, using Isabelle’s lifting- and transfer-package [3], one can easily define a dedicated type for positive numbers, and a function which accesses the internal number, which is then guaranteed to be positive.

```isar
typedef 'a pos-num = { x :: 'a :: linordered-field. x > 0 } morphisms num pos
⟨proof⟩
setup-lifting type-definition-pos-num
lemma num-positive: num x > 0
⟨proof⟩
Using these restricted types, it is often possible to define the desired algorithms where non-termination because of invalid inputs is no longer possible, and where soundness properties can be stated without additional preconditions. E.g., for the approximation algorithm, one takes $\delta$ as input, and the approximation stops, once $|x^2 - y^2| < \text{num } \delta$ is satisfied.

One question in the second approach is how to actually generate elements of the restricted type. Although one can perform the following definition,

```isar
lift-definition create-pos-1 :: 'a :: linordered-field ⇒ 'a pos-num option is
λ x. if x > 0 then Some x else None
⟨proof⟩
the problem is that corresponding defining equation create-pos-1 ?x = map-option pos (if (0::?'a) < ?x then Some ?x else None) is not amenable for code-generation, as it uses the abstraction function pos in a way which is not admitted for code-equations [1, 2].
To overcome this problem, Joachim Breitner proposed the following workaround\(^1\), which requires an additional type definition, and some auxiliary definitions, to in the end define \(\text{create-pos}\) in a way that is amenable for code-generation.

```
typedef 'a num-bit = { (x :: 'a :: linordered-field, b). b → x > 0 } ⟨proof⟩
```

```
setup-lifting type-definition-num-bit
```

```
lift-definition num-bit-bit :: ('a :: linordered-field) num-bit ⇒ bool is snd ⟨proof⟩
```

```
lift-definition num-bit-num :: ('a :: linordered-field) num-bit ⇒ 'a pos-num is
λ (x,b). if b then x else 42 ⟨proof⟩
```

```
lift-definition num-bit :: 'a :: linordered-field ⇒ 'a num-bit is
λ x. if x > 0 then (x, True) else (42, False) ⟨proof⟩
```

```
definition create-pos-2 :: 'a :: linordered-field ⇒ 'a pos-num option where
create-pos-2 x ≡ let nb = num-bit x
in if num-bit-bit nb then Some (num-bit-num nb) else None
```

```
lemma create-pos-2: create-pos-2 x = Some p ⇒ num p = x
⟨proof⟩
```

```
export-code create-pos-2 in Haskell
```

Breitner’s construction has the advantage that the invariant \((0::'a) < x\) only has to be evaluated once (when invoking \(\text{num-bit}\)). Hence, the construction allows to create data for types with invariants in an efficient, executable, and canonical way.

In this AFP entry we now turned this canonical way into a dedicated method \((\text{lift-definition-option})\) which automatically generates the types and auxiliary functions of Breitner’s construction. As a result it suffices to write:

```
lift-definition-option create-pos :: 'a :: linordered-field ⇒ 'a pos-num option is
λ x :: 'a. if x > 0 then Some x else None
⟨proof⟩
```

 Afterwards, we can directly generate code.

```
export-code create-pos in Haskell
```

Moreover, we automatically generate two soundness theorems, that the generated number is the intended one: \(\text{create-pos '?x} = (\text{if} (0::'?a) < '?x \text{then Some '?x}) \text{else None})\) and \(\text{create-pos '?x} = \text{Some '?r} \implies \text{num '?r} = '?x\). Here, the morphisms from the type-definitions reappear, i.e., \(\text{num}\) and \(\text{pos}\) in the example.

\(^1\)http://stackoverflow.com/questions/16273812/working-with-isabelles-code-generator-data-refinement-and-higher-order-functio
2 Usage and limitations

The command lift-definition-option is useful to generate elements of some restricted type (say 'restricted) which has been defined as \{ x. P x \} for some property P of type 'base ⇒ bool. It expects three arguments, namely

- The name of the definition, e.g., create-pos.
- The type of the definition, which must be of the form 'a1 ⇒ 'a2 ⇒ 'a-dots ⇒ 'a-n ⇒ 'restricted option
- The right-hand side of the definition which must be of the shape \( \lambda x1 x2 x-dots x-n. \) if check \( x1 x2 x-dots x-n \) then Some (generate \( x1 x2 x-dots x-n \)) else None where generate is of type 'a1 ⇒ 'a2 ⇒ 'a-dots ⇒ 'a-n ⇒ 'base option

After providing the three arguments, a proof of \( \text{check } x1 x2 x-dots x-n \) \( \Rightarrow P (\text{generate } x1 x2 x-dots x-n) \) has to be provided. Then, code-equations will be derived and registered, and in addition two soundness theorems are generated. These are accessible under the names def-name and def-name-Some, provided that the lifting definition uses def-name as first argument.

Note, that \( P \) is automatically extracted from the type-definition of 'restricted. Similarly, the default value (42 in the 'a pos-num-example) is generated automatically.

To mention a further limitation besides the strict syntactic structure for the right-hand side, it is sometimes required, to add explicit type-annotations in the right-hand-side and the selector, e.g., the 'a in \( \lambda x. \) if (0::a) < x then Some x else None.

3 Examples

3.1 A simple restricted type without type-parameters

typedef restricted = \{ i :: int. i mod 2 = 0 \} morphisms base restricted

(\text{proof})

setup-lifting type-definition-restricted

Let us start with just using a sufficient criterion for testing for even numbers, without actually generating them, i.e., where the generator is just the identity function.
lift-definition-option restricted-of-simple :: int ⇒ restricted option is
\[ \lambda x :: \text{int. if } x \in \{0, 2, 4, 6\} \text{ then Some } x \text{ else None} \]
(proof)

We can also take several input arguments for the test, and generate a more complex value.

lift-definition-option restricted-of-many-args :: nat ⇒ int ⇒ bool ⇒ restricted option is
\[ \lambda x y (b :: \text{bool}). \text{if } \text{int } x + y = 5 \text{ then Some } ((\text{int } x + 1) * (y + 1)) \text{ else None} \]
(proof)

No problem to use type parameters.

lift-definition-option restricted-of-poly :: 'b list ⇒ restricted option is
\[ \lambda xs :: 'b \text{ list. if length } xs = 2 \text{ then Some } (\text{int (length } (xs))) \text{ else None} \]
(proof)

3.2 Examples with type-parameters in the restricted type.

typedef 'f restrictedf = \{ xs :: 'f list. length xs < 3 \} morphisms basef restrictedf
(proof)

setup-lifting type-definition-restrictedf

It does not matter, if we take the same or different type-parameters in the lift-definition.

lift-definition-option test1 :: 'g ⇒ nat ⇒ 'g restrictedf option is
\[ \lambda (e :: 'y) x. \text{if } x < 2 \text{ then Some } (\text{replicate } x e) \text{ else None} \]
(proof)

lift-definition-option test2 :: 'f ⇒ nat ⇒ 'f restrictedf option is
\[ \lambda (e :: 'f) x. \text{if } x < 2 \text{ then Some } (\text{replicate } x e) \text{ else None} \]
(proof)

Tests with multiple type-parameters.

typedef ('a,'f) restr = \{ (xs :: 'a list,ys :: 'f list). length xs = length ys \} morphisms base' restr
(proof)

setup-lifting type-definition-restr

lift-definition-option restr-of-pair :: 'g ⇒ 'e list ⇒ nat ⇒ nat ⇒ ('e,nat) restr option is
\[ \lambda (z :: 'g) (xs :: 'e list) (y :: nat) n. \text{if length } xs = n \text{ then Some } (xs,\text{replicate } n y) \text{ else None} \]
(proof)

3.3 Example from IsaFoR/CeTA

An argument filter is a mapping \( \pi \) from \( n \)-ary function symbols into lists of positions, i.e., where each position is between 0 and \( n-1 \). In IsaFoR, (Isabelle’s
Formalization of Rewriting) and CeTA [5], the corresponding certifier for term rewriting related properties, this is modelled as follows, where a partial argument filter in a map is extended to a full one by means of an default filter.

typedef ′f af = \{ (\pi : ′f \times \text{nat} \Rightarrow \text{nat list}). (\forall f n. \text{set} (f, n)) \subseteq \{0 ..< n\}\} 

morphisms af Abs-af ⟨proof⟩

setup-lifting type-definition-af

type-synonym ′f af-impl = ((′f \times \text{nat}) \times \text{nat list})\text{list}

fun fun-of-map-fun :: (′a ⇒ ′b option) ⇒ (′a ⇒ ′b ⇒ (′a ⇒ ′b)) where 
fun-of-map-fun m f a = (case m a of Some b ⇒ b | None ⇒ f a)

lift-definition-option af-of :: ′f af-impl ⇒ ′f option is 
λ s :: ′f af-impl. if (\forall fidx ∈ set s. (\forall i ∈ set (snd fidx). i < snd (fst fidx)))
then Some (fun-of-map-fun (map-of s) (λ (f, n). [0 ..< n])) else None
⟨proof⟩

3.4 Code generation tests and derived theorems

export-code
  restricted-of-many-args
  restricted-of-simple
  restricted-of-poly
  test1
  test2
  restr-of-pair
  af-of

in Haskell

thm
  restricted-of-many-args Some restricted-of-many-args
  restricted-of-simple Some restricted-of-simple
  restricted-of-poly Some restricted-of-poly
  test1 Some test1
  test2 Some test2
  restr-of-pair Some restr-of-pair
  af-of Some af-of

end

Acknowledgements

We thank Andreas Lochbihler for pointing us to Joachim’s solution, and we thank Makarius Wenzel for explaining us, how we can go back from states to local theories within Isabelle/ML.
References


