Lab 3: building proofs using Welder

Rodrigo Raya Castellano
École Polytechnique Fédérale de Lausanne
rodrigo.raya@epfl.ch

1 Acknowledgements

The following persons and materials have been useful for completing this lab.

For practical issues, we have to thank users Nicolas Voirol and Romain Edelmann on Stackoverflow and user D.W. on cs.stackexchange. The relevant links are [1],[2] and [3]. Also, for understanding how extractors work in Scala we used [4].

Exercise 1

The solution for this exercise follows the one that was introduced in class. Basically, it consists of proving an auxiliary lemma by induction given the hypothesis on f:

Lemma:

$$\forall l : List[A].\forall x : A.(foldl(f, x, l) = f(x, foldl(f, z, l)))$$

Finally, the theorem looks as:

Theorem:

$\forall z : A, f : (A, A) => A, l : List[A].$

$((\forall x : A, y : A, z : A.(f(x, f(y, z)) = f(f(x, y), z)) \wedge \forall x : A.(f(x, z) = x \wedge f(z, x) = x))$

$\implies (foldl(f, z, l) = foldr(f, z, l))))$

Overall, this exercise was a good introduction to the constructs and syntax of Welder. We learned how to formulate structural induction on lists, instantiate lemmas and introduce hypotheses.

Exercise 2

The first approach to solve this exercise comes from the experimentation with small examples in the style of Polya. Indeed, three observations are important here. First, we can take a $fold(f, z, l)$ to be a repeated application of function f to elements of list l. Instead of noting $f(f(f(z, x_1)x_2)x_3)$ it is useful to just write $((z, x_1)x_2), x_3)$. Second, we can observe that the commutativity and associativity of operator f allows us to move elements x_i forward and backward in this onion-like structure. Finally, using idempotence we can show that moving backward and forward different elements of the list we can get a list with no repeated elements. Now, it is clear at least from the formal point of view that the fold are equals.

Proving the theorem with Welder it actually more complicated. In this sense, I think is worth mentioning an idea that came up in class the last day of the assignment. Basically, we want to reorder in some sense two lists represented as trees so that they become equal. The bright idea here is that any finite set has an order. In the case of lists this can be the order induced by the bijection that gives a place to the elements. The problem would actually reduce to proof that some sorting algorithm is correct.

However, we did not use this approach in our solution and instead we preferred a straight forward induction attempt. The following is a wrong approach to do so, we omit the *nil* case as trivial:

$$fold(f, z, l_1) = fold(f, z, x :: xs) = fold(f, z, x :: without(x, xs)) = fold(f, f(z, x), without(x, xs))$$

note that here the only difficult step is step two. The *without* function removes all the occurrences of the first argument x from the second argument xs and it is defined as follows:

Listing 1: Scala code for the without function

```
1 def without(x: A, xs: List[A]) = xs match{
2  case Nil() => Nil()
3  case y :: ys if(x == y) => without(x,ys)
4  case y :: ys if(x != y) => y :: without(x,ys)
5 }
```

For the right hand side we have:

$$fold(f, z, l_2) = fold(f, z, x :: without(x, l_2)) = fold(f, f(z, x), without(x, l_2))$$

Here the non trivial step is of course the first one. The idea would be to apply induction on xs as clearly $without(x, xs)$ and $without(x, l_2)$ have the same contents and it would be easy to proof that $without(x, xs)$ is a of less size. However, it turns out this is not a suitable proof for structural induction as $without(x, xs)$ needs not to be a tail of the original list on which we induct. Seeing a list as a tree this can be restated as not being a sub-tree of the original tree. [1]

It is also worth mentioning another approach to solve the exercise in order to avoid having $without(x, xs)$ as in the last term of the left hand side above so that we could perform induction.

Basically, we decided to relax our $without(x, l)$ function that removes all the occurrences of element x in list l and introduce instead a function $withoutOne(x, l)$ that would take just one x from the list. With this approach we used *Bags* instead of *Sets* and unconvered a bug in Inox. Namely, writing:

$$content(without(x, l)) === BagDifference(content(l), singleton(x))$$

would lead into a run time error:

Error: map expects all arguments to have the same array domain, this is not the case for argument 1

that appears to come from the Z3 prover. Also we may note that the error that the system currently provides for using $--$ instead of $BagDifference$ is pretty cryptic being:

Error: expecting boolean range

Also, the resulting proof suspiciously omitted the use of the idempotence assumption and we realized it was wrong.

Nevertheless, after figuring out this last approach was bad we could reuse its ideas to finally fix our first proof. It goes as follows (again omitting the nil case):

We distinguish to cases. For the first case assume that xs contains x. Then we would have easily that $contents(xs) = content(l_2)$ and so using the induction hypothesis:

$$fold(f, z, x :: xs) = fold(f, f(z, x), xs) = fold(f, f(z, x), l_2) = fold(f, z, x :: l_2) = fold(f, z, l_2)$$

For the second case, assume that xs does not contain x. Then we would have easily that $contents(xs) = contents(l_2) \setminus \{x\}$. And so:

$fold(f, z, x :: xs) = fold(f, f(z, x), xs) = fold(f, f(z, x), without(x, l_2)) = fold(f, z, x :: without(x, l_2)) = fold(f, z, l_2)$

The non-trivial steps here can be solved by the introduction of lemmas 1 and 3. These are indeed the most important lemmas of our proof as they are the only ones that use the hypothesis on f and the contents of the list. For reference purposes we now list all the lemmas introduced in the proof:

Lemma 1:

$$\forall xs : List[A].\forall x : A, z : A.(fold(f, z, Cons(x, xs)) = fold(f, z, Cons(x, without(x, xs))))$$

Lemma 2:

$$\forall x : A, y : A, xs : List[A].((x \neq y) \implies (x \in content(xs) = x \in content(Cons(y, xs))))$$

Lemma 3:

$$\forall xs : List[A].\forall x : A, z : A.(x \in content(xs) \implies (fold(f, z, xs) = fold(f, z, Cons(x, xs))))$$

Lemma 4:

$\forall x : A, n_1 : List[A], n_2 : List[A].$

$$((content(n_1) = content(n_2)) \implies (x \in content(n_1) = x \in content(n_2)))$$

Lemma 5:

$$\forall l : List[A].\forall x : A.(content(without(x, l)) = content(l) \setminus \{x\})$$

Lemma 6:

$\forall x : A, l_1 : List[A], l_2 : List[A].((content(l_1) = content(l_2)) \implies (content(without(x, l_1)) = content(without(x, l_2))))$

Lemma 7:

$\forall x : A, xs : List[A], l_2 : List[A].$

$$((content(Cons(x, xs)) = content(l_2) \wedge x \in content(xs)) \implies (content(xs) = content(l_2)))$$

Lemma 8:

$\forall x : A, xs : List[A], l2 : List[A].$

$((content(Cons(x, xs)) == content(l2) \wedge x \notin content(xs)) \implies (content(xs) == content(l2) \setminus \{x\}))$

Lemma 9:

$\forall x : A, xs : List[A].x \in content(Cons(x, xs))$

The final result is then:

$\forall f : (A, A) => A, z : A, l1 : List[A], l2 : List[A].$

$((((\forall x : A, y : A, z : A.(f(x, f(y, z)) = f(f(x, y), z)) \wedge$

$\forall x : A, y : A.(f(x, y) = f(y, x))) \wedge$

$\forall x : A, y : A.(f(x, y) = f(f(x, y), y))) \wedge$

$content(l1) = content(l2)) \implies (fold(f, z, l1) = fold(f, z, l2)))$

We are sure this proof can be shorten both in size, as there are trivial steps that could be omitted, and in the number of lemmas used. However, we stick to this solution for the time being. For the future, we would like to experiment with proofs that perform natural induction on the size of lists and proofs that use *Bags* in their solution.

References

[1] Nicolas Voirol. *Is there a notion of strong induction in Welder?* 2017. URL: http://stackoverflow.com/questions/43020382/is-there-a-notion-of-strong-induction-in-welder (visited on 03/27/2017).

[2] Romain Edelmann. *Proving properties of sets in Inox/Welder*. 2017. URL: http://stackoverflow.com/questions/43011252/proving-properties-of-sets-in-inox-welder (visited on 03/27/2017).

[3] D.W. *Formally proving properties of fold function*. 2017. URL: http://cs.stackexchange.com/questions/71916/formally-proving-properties-of-fold-function (visited on 03/27/2017).

[4] Emir B. et alii. "Matching objects with patterns". In: *European Conference on Object-Oriented Programming* (2007), pp. 273–298.