unfolder : User Manual

José M. Rey Poza

August 14, 2014

Contents

Chapter 1

What is unfolder?

unfolder is an experimental toolkit that enables the user to unfold full fledged functional programs (i.e. programs that use higher order, function composition and lazy evaluation with partially undefined terms).

unfolder is written in Prolog (more specifically, Ciao Prolog). In its current state the programs to be unfolded are read from the Prolog code itself, where they are represented as Prolog facts. These facts represent guarded rules belonging to a functional program written in a generic functional language. That is, unfolder does not currently read functional programs (written in Haskell or any other existing functional language); instead, it reads the source to be unfolded from an internal representation.

1.1 Unfolding

Unfolding is the process of replacing a function invocation by the definition of that very same function. By repeating the process as many times as necessary, what we get is a program that has the same meaning as the original *folded* but that is simpler than the original one in terms of function composition (a completely unfolded program has no function composition at all but this is not possible in general). That simpler form of the program can be seen as a set of facts that plainly express the meaning of the original program (its semantics).

Unfolding generates valid functional code that can be run just as the original one to produce the same result (provided full unfolding has been possible).

Unfolding is an iterative process: Starting with an empty code, every iteration produces a better approximation to the meaning of the original code. These successive approximations draw closer to the real meaning of the program by having more rules that deal with more input values or by letting existing rules handle better approximations of large terms.

Every approximation to the final meaning of the functional program under examination is nothing more than a set of functional rules. The set of functional rules contained within a given approximation is called **an interpretation**. The unfolding process begins with an empty interpretation (denoted as I_0) and every iteration of unfolding takes current interpreation along with the original program rules to generate the next interpretation (that is, jumping from I_n to I_{n+1}).

Chapter 2

Writing Programs for unfolder

As stated in chapter 1, unfolder reads the programs it must unfold from Prolog source code. Programs to be unfolded are made up of a number of rules. All rules must adhere to the following format:

```
rule(Function_Name,Patterns,Guard,Body,Where,RuleId)
```

where each argument of `rule` describes a part of a given rule. For example, the two rules for the well known `append` function are written as follows:

```
rule(append,[nil,X],_,X,[]).
rule(append,[cons(X,Xs),Ys],_,cons(X,append(Xs,Ys)),[]).
```

These two Prolog facts denote the following two Haskell rules:

```
append [] x = x
append (x:xs) ys = (x:(append xs ys))
```

The example above shows how program rules must be written:

- Free functional variables are represented by free logic variables.
- Constructors are written as Prolog atoms (that is, they start by a lowecase letter). Lists are represented by constractors `nil` (for `[]`) and `cons` (for `(_:_)`).
- No type declaration is necessary. The unfolder does not currently know anything about types, so it is up to the user to ensure that all programs are well typed.

In turn, each part of the rule must obey the following restrictions:

- The head must be an atom representing the name of the function. All rules defining the same function must bear the same atom at this position.
- The second argument for `rule` represents the pattern of every rule. A pattern is written as a list of terms. All functions defined are considered to be *fully currified*. If tuples have to be used, the can be written as `(`$term_1, \ldots, term_n$ `)`.
- The guard of the rule is next. An empty guard is represented by a free Prolog variable. If the guard is not empty, it must be written using normal applicative notations (that is, $f(1,2)$ must be written as `f(1,2)`). Boolean conjunction is written as `and(_,_)`.
- The fourth argument of `rule` contains the body of the rule. It has to be written according the same rules that govern the writing of guards.
- The next argument is an experimental one intended to contain local (*where*) declarations. It is currently unused and must contain an empty list (`[]`).
- Finally, the last argument is an optional rule identifier that can be used to let unfolder track what rules have been used to generate every fact. If a rule has to have an identifier, this argument must hold such an identifier. The format to be used is `[rule(<ID>)]` where `<ID>` is a Prolog atom or string containing the rule name.

Chapter 3

Running unfolder

3.1 Execution

unfolder is currently included in one Prolog file that must be run from within Ciao Prolog. Therefore, Ciao must be started and then `unfolding.pl` must be loaded:

```
$ ciao
Ciao 1.15.0-14760: mi jun 27 18:36:00 CEST 2012 [install]
?- [unfolding].

yes
?-
```

The rules that represent the functional program to be unfolded must, at the time of writing this, be present inside the Prolog code.

As said before, unfolding is an iterative process. Basically, the process of unfolding a functional program involves running an iteration and examining its result until needed.

In unfolder , an iteration is run by invoking the Prolog predicate `unfolding_operator/0`. This takes the functional program rules (represented as Prolog facts) and current interpretation to generate the next interpretation.

Once `unfolding_operator/0` has been run, the content of current interpretation can be examined by means of `show_int/0`.

The unfolding of the `append` function is shown below:

```
$ ciao
Ciao 1.15.0-14760: mi jun 27 18:36:00 CEST 2012 [install]
?- [unfolding].

yes
```

```
?- unfolding_operator,show_int.
* append(Nil,b) = b
* append(Cons(b,c),d) = Cons(b,Bot)

yes
?- unfolding_operator,show_int.
* append(Nil,b) = b
* append(Cons(b,Nil),c) = Cons(b,c)
* append(Cons(b,Cons(c,d)),e) = Cons(b,Cons(c,Bot))

yes
?- unfolding_operator,show_int.
* append(Nil,b) = b
* append(Cons(b,Nil),c) = Cons(b,c)
* append(Cons(b,Cons(c,Nil)),d) = Cons(b,Cons(c,d))
* append(Cons(b,Cons(c,Cons(d,e))),f) = Cons(b,Cons(c,Cons(d,Bot)))

yes
?-
```

Three unfolding iterations are shown above (I_1, I_2 and I_3). Note that every step involves invoking `unfolding_operator/0` to create a new interpretation and invoking `show_int/0` just after that to see what the new interpretation contains. Of course, there is no problem in invoking `unfolding_operator/0` any number of times between two invocations of `show_int/0`.

3.2 Understanding the Output of unfolder

The output of unfolder is the one generated by `show_int/0`. This predicate dumps current interpretation to the screen. Every functional rule that belongs to current interpretation is written in a somewhat beautified form. Although the result of unfolding is valid functional code, unfolder does not generate valid Haskell code yet (in particular the rules are still uncurried). This is work for the future.

The *beautification* of every interpretation element (every generated functional rule) comprises the following steps:

- Every interpretation element is preceded by an asterisk (`'*'`) to ease the reading of rules.
- Constructors are shown with uppercase initials. Lists are shown as `Nil` (for `[]`) and `Cons` (for `(_:_)`). Note that `Bot` (*bottom*) is an special constructor denoting the absence of information (usally written as $\perp$ in books).

- Variables are shown with lowercase initials.
- Partial or higher order applications are represented by <*function*>`@`<*arguments_list*> where <*function*> can be a free variable or the name of a function and <*arguments_list*> is a list of arguments already applied to the function.

The three iterations run above show how the unfolding process moves closer and closer to the final meaning of the program until the final full meaning is eventually found (which does not happen in this case since `append` has an infinite semantics). Every iteration includes the previous one and adds more elements to it. Those new elements are able to deal with successively longer lists.

The next example shows that unfolding does not always add more rules to an interpretation even if the final result has not been found. Consider the following code:

```
data Nat = Zero | Suc Nat

ones :: [Int]
ones = (1:ones)

take_n :: Nat -> [Int] -> Int
take_n Zero _ = []
take_n Suc(n) (x:xs) = (x:(take_n n xs))

main5 :: [Int]
main5 = take_n Suc(Suc(Zero)) ones
```

Its first unfolding iterations are as follows:

```
$ ciao
Ciao 1.15.0-14760: mi jun 27 18:36:00 CEST 2012 [install]
?- [unfolding].

yes
?- unfolding_operator,show_int.
* ones = Cons(1,Bot)
* take_n(Zero,b) = Nil
* take_n(Suc(b),Cons(c,d)) = Cons(c,Bot)

yes
?- unfolding_operator,show_int.
* take_n(Zero,b) = Nil
* ones = Cons(1,Cons(1,Bot))
```

```
* take_n(Suc(Zero),Cons(b,c)) = Cons(b,Nil)
* take_n(Suc(Suc(b)),Cons(c,Cons(d,e))) = Cons(c,Cons(d,Bot))
* main5 = Cons(1,Bot)

yes
?- unfolding_operator,show_int.
* take_n(Zero,b) = Nil
* take_n(Suc(Zero),Cons(b,c)) = Cons(b,Nil)
* ones = Cons(1,Cons(1,Cons(1,Bot)))
* take_n(Suc(Suc(Zero)),Cons(b,Cons(c,d))) = Cons(b,Cons(c,Nil))
* take_n(Suc(Suc(Suc(b))),Cons(c,Cons(d,Cons(e,f)))) =
    Cons(c,Cons(d,Cons(e,Bot)))
* main5 = Cons(1,Cons(1,Bot))

yes
?- unfolding_operator,show_int.
* take_n(Zero,b) = Nil
* take_n(Suc(Zero),Cons(b,c)) = Cons(b,Nil)
* take_n(Suc(Suc(Zero)),Cons(b,Cons(c,d))) = Cons(b,Cons(c,Nil))
* ones = Cons(1,Cons(1,Cons(1,Cons(1,Bot))))
* take_n(Suc(Suc(Suc(Zero))),Cons(b,Cons(c,Cons(d,e)))) =
    Cons(b,Cons(c,Cons(d,Nil)))
* take_n(Suc(Suc(Suc(Suc(b)))),Cons(c,Cons(d,Cons(e,Cons(f,g))))) =
    Cons(c,Cons(d,Cons(e,Cons(f,Bot))))
* main5 = Cons(1,Cons(1,Nil))
```

Take a look at function `ones`: It has just one rule in every iteration. By contrast, `take_n` gathers more rules as iterations go by. This is due to the fact that `ones` has just one pattern (the empty one) so all the rules are covered by a single case. Observe how the value returned by that rule grows bigger with every iteration (keep in mind that `Bot` is the infimum – least value – of the domain inside which interpretations are generated).

3.3 Other Examples

The code of unfolder contains many additional examples covering the main features of functional programming. They all are written as Prolog rules belonging to the `rule/5` predicate. All rules must be commented out except for those that are about to be tested. The steps for a test are the following ones:

1. Uncomment the rules to be tested. Comment all the others.
2. Load `unfolding.pl` into Ciao Prolog.

3. Run the iterations as described above.

The examples included in `unfolding.pl` usually contain a `mainN` function where `N` is a number. This function can be seen as the one that generates the answer that is being sought with the particular test case.

Topics covered by the examples include:

- Functions that need the `match` operator (see the relavent paper).
- Guarded rules.
- Experimenting with constraints within guards.
- Dealing with infinite structures and lazy evaluation.
- Higher order and partial applications.
- Treatment of predefined functions (i.e. those with no rules).
- Function composition.

Chapter 4

Cleaning the Interpretations

It was mentioned in the last Example of Section 3.2 that some functions (like `ones`) generate successive facts that overlap, so the *smaller*, older facts are made redundant by newer facts.

unfolder contains code that *cleans* interpretations. A portion of this code works automatically while some other portion has to be manually invoked by the user.

The code that works automatically removes facts that are completelly overlapped by existing facts. This was the case with function `ones`. However, this behaviour is not always valid and can cause loss of information with some programs. The programs that tend to cause these problems are those containing functions that are only partially defined or have rules that can never generate a fact.

This is why this automatic cleaning capability can be disabled and replaced by some other methods that must be explicitly invoked by the user. This way of working has been chosen because, although the manual cleaning methods are valid for every program, they may generate cumbersome interpretations (i.e. interpretations that are large and whose facts contain large, complex guards).

As it is, unfolder has its automatic cleaning code enabled. Every interpretation is cleaned right after it is calculated. Such a cleaning process is responsible for removing the old facts of `ones` in Section 3.2.

On the opposite side, the code below shows when the automatic behaviour is not desirable:

```
data Nat = Zero | Suc Nat

f Zero = Zero

g x = Suc (f x)

h (Suc x) = Zero
```

Note that `f` and `h` are incomplete functions while `g` is complete. Programs with incomplete functions may arise this incorrect behaviour of *clean*. In this case, unfolder generates the following interpretations (I_0 is empty, as usual):

```
Ciao 1.15.0-14760: mi jun 27 18:36:00 CEST 2012 [install]
?- [unfolding].

yes
?- unfolding_operator,show_int.
* f(Zero) = Zero
* g(b) = Suc(Bot)
* h(Suc(b)) = Suc(Zero)

yes
?- unfolding_operator,show_int.
* f(Zero) = Zero
* h(Suc(b)) = Suc(Zero)
* g(Zero) = Suc(Zero)
* main30 = Suc(Zero)

yes
?-
```

The first set of facts belong to I_1 while the second set represents I_2. Observe that `g` is complete in I_1 but it is only defined for `Zero` in I_2: *clean* has removed information that was essential for `g`.

If the automatic cleaning functionality is disabled (which currently requires a small code modification), I_1 is the same as before but I_2 changes slightly:

```
* f(Zero) = Zero
* g(b) = Suc(Bot)
* h(Suc(b)) = Suc(Zero)
* g(Zero) = Suc(Zero)
* main30 = Suc(Zero)
```

Comparing this interpretation to the former I_2, we can see that `g` now has two fact that overlap: `* g(b) = Suc(Bot)` and `* g(Zero) = Suc(Zero)`. Such an overlapping must be solved but neither fact can be removed completely or some information would be lost.

The predicate `reclean/1` can be called to see what facts overlap. To see the overlappings that may exist inside the current interpretation without modifying anything, it is called with a free variable as argument:

```
?- reclean(X).

* f(Zero) = Zero

* g(b) = Suc(Bot)
*** g(Zero) = Suc(Zero)
* g(Zero) = Suc(Zero)

* h(Suc(b)) = Suc(Zero)

* main30 = Suc(Zero)
```

This shows, by means of indentation, that `g(Zero) = Suc(Zero)` is overlapped by `* g(b) = Suc(Bot)`. To eliminate the overlapping from the interpretation, the most general fact of the overlapping pair must be modified so it can no longer be applied where the most specific fact of the pair can. The predicate `reclean/1` can also perform that change at the user request. This is done by invoking `reclean/1` with `yes` as argument:

```
reclean(yes),show_int.
* f(Zero) = Zero
* h(Suc(b)) = Suc(Zero)
* g(Zero) = Suc(Zero)
* main30 = Suc(Zero)
* g(b) | Nunif([b],[Zero]) = Suc(Bot)

yes
?-
```

The operator `Nunif/2` stands for *non-unification*. It returns true if its two argument cannot be unified.

You can see that this latest interpretation does not have overlappings: The last fact cannot be applied when its argument is `Zero`. At the same time, the other fact for `g` (third fact) cannot be applied to values other than `Zero`.

The invocation:

```
:- reclean(yes).
```

removes any overlapping that may exist inside the current interpretation (that is, `reclean` acts on all the functions of the interpretation).

Chapter 5

Outermost Unfolding

By default, unfolder performs unfolding steps in all the possible ways: That means that all the expressions that are headed by a user-defined function are tested against the facts known so far in order to check whether the pair $\langle$ *expression*, *fact* $\rangle$ is suitable for unfolding.

This, of course, can generate a large number of redundant, overlapping facts. These redundant facts will be removed by `clean` and `reclean` (or, alternatively, `clean` and `reclean` will modify the existing facts so that they do not overlap).

However, if the generation of redundant facts is avoided as much as possible, the necessity of removing or modifying facts is greatly reduced and thus the system will run more efficiently.

It is well known that *outermost unfolding* prevents the evaluation of expressions whose value is not needed to find the real value of a larger expression (evaluation proceeds from the outermost position, delving to innermost positions only if their value is strictly needed to find the value of the larger expression).

unfolder can simulate *outermost unfolding*. Its behaviour can be changed from the default one (full unfolding of all suitable positions) by using the following predicate:

```
:- assert_outermost_only.
```

After the predicate above is executed all calls to `unfolding_operator/0` use *outermost unfolding*. The default behaviour is recovered by executing the following predicate:

```
:- retract_outermost_only.
```

5.1 Example: Differences Between Full and Outermost Unfolding

Let us think of the following code:

```
data Nat = Zero | Suc Nat

leq :: Nat -> Nat -> Bool
leq Zero _ = True
leq (Suc _) Zero = False
leq (Suc x) (Suc y) = leq x y

g :: Nat -> Nat
g Zero = Zero

main22b :: Nat -> Bool
main22b x = leq Zero (g x)
```

This code translates into the following rules of unfolder (including names for all the rules):

```
rule(leq,[zero,_],_,true,[],[rule('L1')]).
rule(leq,[suc(_),zero],_,false,[],[rule('L2')]).
rule(leq,[suc(X),suc(Y)],_,leq(X,Y),[],[rule('L3')]).
rule(g,[zero],_,zero,[],[rule('G1')]).
rule(main22b,[X],_,leq(zero,g(X)),[],[rule('Main22b')]).
```

Using the unfolder's default behaviour, the following results are obtained before cleaning (note that, since `g` is incomplete, the optimized version of `clean` cannot be applied:

```
?- unfolding_operator,show_int.
* leq(Zero,b) = True  <L1>
* leq(Suc(b),Zero) = False  <L2>
* g(Zero) = Zero  <G1>

yes
?- unfolding_operator,show_int.
* leq(Zero,b) = True  <L1>
* leq(Suc(b),Zero) = False  <L2>
* g(Zero) = Zero  <G1>
* leq(Suc(Zero),Suc(b)) = True  <L3,L1>
* leq(Suc(Suc(b)),Suc(Zero)) = False  <L3,L2>
* main22b(b) = True  <Main22b,L1>
* main22b(Zero) = True  <Main22b,G1,L1>
```

```
yes
```

The execution above shows two interpretations (I_1 and I_2) for the program given. Note that there are two overlapping facts for `main22b`. Specifically, the second of those facts (`* main22b(Zero) = True <Main22b,G1,L1>`) owes its existence to the the fact that the unfolding process has evaluated `g x` (rule `G1`) before evaluating `leq` by means of rule `L1`. Since the value of `g x` is irrelevant for `main22b`, what that superfluous rule usage has achieved is to create a fact that is not needed (it is too specific).

Outermost unfolding would have avoided that superfluous evaluation after realizing that the value of the inner expression `g x` is not needed to find the value of `main22b` since only `leq`'s first rule is usable to unfold the body of `main22b` and that rule does not demand the value of the position occupied by `g x`. This causes *outermost unfolding* to *ignore* that position when unfolding. The result is as follows:

```
?- assert_outermost_only.

yes
?- unfolding_operator,show_int.
* leq(Zero,b) = True  <L1>
* leq(Suc(b),Zero) = False  <L2>
* g(Zero) = Zero  <G1>

yes
?- unfolding_operator,show_int.

* leq(Zero,b) = True  <L1>
* leq(Suc(b),Zero) = False  <L2>
* g(Zero) = Zero  <G1>
* leq(Suc(Zero),Suc(b)) = True  <L3,L1>
* leq(Suc(Suc(b)),Suc(Zero)) = False  <L3,L2>
* main22b(b) = True  <Main22b,L1>

yes
```

Observe that both interpretations are identical to the ones before except for the fact that was too specific: it is no longer there. This has been caused because unfolder has *purged out* the expression `g x` when considering the positions available for unfolding.