Mutually Recursive Partial Functions*

René Thiemann

March 17, 2025

Abstract

We provide a wrapper around the partial-function command which supports mutual recursion.

Our results have been used to simplify the development of mutually recursive parsers, e.g., a parser to convert external proofs given in XML into some mutually recursive datatype within Isabelle/HOL.

Contents

1 Introduction

The partial function command of Krauss [1] turns monotone monadic function specifications into equational theorems. Here, monadic means that the output type of the function must be a monad like the option-monad. This is required to prohibit specifications like

$$f\ x = 1 + f\ x$$

*This research is supported by FWF (Austrian Science Fund) project P22767-N13. We thank Makarius Wenzel for several hints on how to properly localize our wrapper.

which would immediately lead to a contradiction. Since the command produces unconditional equations, it is extremely helpful in writing possibly nonterminating functions which are amenable to code generation. For example, using *partial-function*, one can write a recursive parser in Isabelle/HOL and can then use it in several target languages—without having to struggle with a tedious termination proof which might have to reason about the internal state of the parser.

Unfortunately, the command currently does not support mutually recursive functions, which however would be a convenient feature when writing parsers for mutually recursive datatypes. To be more precise, a specification of a partial function has to be of the following shape

$$f\ \overrightarrow{xs} = F\ f\ \overrightarrow{xs} \tag{1}$$

where $\overrightarrow{xs}$ is a sequence of distinct variables and F is an arbitrary monotone functional that may depend on f and $\overrightarrow{xs}$.

For mutually recursive functions we would like to specify functions in the more general form

$$\begin{aligned} f_1\ \overrightarrow{xs_1} &= F_1\ \overrightarrow{fs}\ \overrightarrow{xs_1} \\ &\vdots \\ f_n\ \overrightarrow{xs_n} &= F_n\ \overrightarrow{fs}\ \overrightarrow{xs_n} \end{aligned} \tag{2}$$

where $\overrightarrow{fs} = f_1, \ldots, f_n$ and $\overrightarrow{xs_i}$ are the individual arguments to each of the functions f_i.

In the following, we describe our wrapper around the partial function command which supports mutual recursion. We first synthesize a global function g from the specifications in (2) which itself has a defining equation in the form of (1). Then we register g and derive the defining equation for g as theorem in Isabelle/HOL using *partial-function*. Afterwards, it will be easy to define each f_i in terms of g, and finally derive the equations in (2) as theorems.

Let us now consider the details. Assume each f_i has a type $in_{i,1} \Rightarrow \ldots \Rightarrow in_{i,ar(f_i)} \Rightarrow out_{f_i}\ monad$, where for each f, $ar(f)$ is the arity of f, and *monad* is the common monad. For g there will only be one input, and this input has type $(in_{f_1}) + \ldots + (in_{f_n})$: each sequence of input types $in_{i,1}, \ldots, in_{i,ar(f_i)}$ is first transformed into a single argument of type $(in_{f_i}) := in_{i,1} \times \ldots \times in_{i,ar(f_i)}$, and afterwards the sum type is used to distinguish between the inputs of the individual functions. Similarly, the output type of g will be $(out_{f_1} + \ldots + out_{f_n})\ monad$. Note that we did not choose $out_{f_1}\ monad + \ldots + out_{f_n} monad$ as output of g as it is not monadic, and thus, g would not be definable via *partial-function*.

Next, we define g via a single equation which can then be passed to *partial-function*. Here, we have to

- convert between tuples and sequences of arguments via currying and uncurrying. To this end, we use the predefined *curry*-function for currying and for uncurrying we perform pattern matching in expressions like $\lambda(xs).h\ \overrightarrow{xs}$ which take a tuple of variables as argument and then feed these variables sequentially to some function h.

- convert between argument and sum-types. To this end, we use constructors inj_i of type $\alpha_i \Rightarrow \alpha_1 + \ldots + \alpha_i + \ldots \alpha_n$, and destructors $proj_i$ which work in exactly the opposite direction. Moreover, we perform case-analyses via pattern matching on the inj_i's. Note that internally each inj_i is encoded via repeated usage of the constructors *Inl* and *Inr* of Isabelle/HOL's *sum*-type, and similarly we nest *Projl* and *Projr* to encode arbitrary $proj_i$-functions.

- work within the monad to combine the various result types into a single one. To this end, we demand that there is some *map-monad*-function which lifts an operation $\alpha \Rightarrow \beta$ to a function of type $\alpha\ monad \Rightarrow \beta\ monad$. In general, these mappings may also take several functions as input, depending on the number of type-variables of the monad-constructor. For each kind of monad that should be supported by our method, a user-defined *map-monad* function can be registered. It is important, to also register a monotonicity lemma of each *map-monad* function within the partial function package. Otherwise, monotonicity proofs for g will most likely fail.

Putting everything together, we setup the following equation

$$\begin{aligned} g\ x = case\ x\ of & \\ & inj_1 xs_t \Rightarrow map\text{-}monad\ inj_1\ ((\lambda(xs).F_1\ \overrightarrow{fs'}\ \overrightarrow{xs})\ xs_t) \\ & \mid \ldots \\ & \mid inj_n xs_t \Rightarrow map\text{-}monad\ inj_n\ ((\lambda(xs).F_n\ \overrightarrow{fs'}\ \overrightarrow{xs})\ xs_t) \end{aligned} \tag{3}$$

where $\overrightarrow{fs'}$ is the sequence of abbreviations $f'_1, \ldots, f'_n$ and where

$$f'_i = curry\ (\lambda xs_t.\ map\text{-}monad\ proj_i\ (g\ (inj_i\ xs_t))) \tag{4}$$

Once, g has been defined using *partial-function*, we obtain Equality (3) as a theorem. Afterwards, it is easy to define

$$f_i = curry\ (\lambda xs_t.\ map\text{-}monad\ proj_i\ (g\ (inj_i\ xs_t))) \tag{5}$$

and it remains to derive the equations in (2) as theorems. To this end, first note the difference in (4) and (5). In the former, g is a free variable which should be defined as a constant at that point, whereas g is already the newly defined constant in (5). Obviously, at this point one can now replace the abbreviations (4) in Equation (3) by the real constants f_i via the defining equations (5). This yields the following modified theorem for g where now $\overrightarrow{fs}$ is the sequence $f_1, \ldots, f_n$.

$$\begin{aligned} g\ x = \mathit{case}\ x\ \mathit{of} \\ & \mathit{inj}_1 \mathit{xs}_t \Rightarrow \mathit{map\text{-}monad}\ \mathit{inj}_1\ ((\lambda(\mathit{xs}).F_1\ \overrightarrow{fs}\ \overrightarrow{xs})\ \mathit{xs}_t) \\ & \mid \ldots \\ & \mid \mathit{inj}_n \mathit{xs}_t \Rightarrow \mathit{map\text{-}monad}\ \mathit{inj}_n\ ((\lambda(\mathit{xs}).F_n\ \overrightarrow{fs}\ \overrightarrow{xs})\ \mathit{xs}_t) \end{aligned} \tag{6}$$

Now it is indeed easy to derive the desired equations in (2):

$$\begin{aligned} f_i\ \overrightarrow{xs} & \overset{(5)}{=} (\mathit{curry}\ (\lambda \mathit{xs}_t.\ \mathit{map\text{-}monad}\ \mathit{proj}_i\ (g\ (\mathit{inj}_i\ \mathit{xs}_t))))\ \overrightarrow{xs} \\ & \overset{(\star)}{=} \mathit{map\text{-}monad}\ \mathit{proj}_i\ (g\ (\mathit{inj}_i\ (\overrightarrow{xs}))))) \\ & \overset{(6)}{=} \mathit{map\text{-}monad}\ \mathit{proj}_i\ (\mathit{map\text{-}monad}\ \mathit{inj}_i\ (F_i\ \overrightarrow{fs}\ \overrightarrow{xs})) \\ & \overset{(\star\star)}{=} F_i\ \overrightarrow{fs}\ \overrightarrow{xs} \end{aligned}$$

Here, $(\star)$ used the definition of *curry* and splitting of tuples, and for $(\star\star)$ we demand that *map-monad* is compositional and that *map-monad* applied on the identity function is the identity function itself.

2 Implementation

2.1 Known limitations

- The method does only provide equational theorems. It does not convert the induction rule for the global function g from the partial function command into an induction rule for the set of mutually recursive functions.

theory *Partial-Function-MR*
imports *Main*
keywords *partial-function-mr* :: *thy-decl*
begin

2.2 Register the *partial-function-mr* command

⟨*ML*⟩

2.3 Register the "option"-monad

Obviously, the map-function for the *option*-monad is *map-option*.

First, derive the required identity lemma.

lemma *option-map-id*: *map-option* $(\lambda\ x.\ x)\ x = x$
⟨*proof*⟩

Second, register *map-option* as being monotone.

lemma *option-map-mono*[*partial-function-mono*]:
assumes *mf*: *mono-option B*
shows *mono-option* $(\lambda f.$ *map-option* $h\ (B\ f))$
⟨*proof*⟩

And finally perform the registration. We need

- a constructor for map: it takes a monadic term *mt* of type *mtT*, a list of functions *t-to-ss* with corresponding types in *t-to-sTs*, a resulting monadic type *msT*, and it should return a monad term *ms* of type *msT* which is obtained by applying the functions on *mt*. Although for the *option*-monad, the lengths of the lists will always be one, there might be more elements for monads having more than one type-parameter.
- a function to perform type-construction for monads: it takes a list of fixed parameters and a list of flexible parameters and has to construct a monadic type out of these parameters. The user can freely choose which parameters should be fixed, and which are flexible. Only flexible parameters can be changes in the return type of each set of mutual recursive functions. Since in the *option*-monad we would like to be able to change the type-parameter, we ignore the fixed parameters here.
- a function to deconstruct monadic types into fixed and flexible type arguments.
- a compositionality theorem of the form $map\ f\ (map\ g\ x) = map\ (f \circ g)\ x$
- an identity theorem of the form $map\ (\lambda x.\ x)\ m = m$

⟨*ML*⟩

2.4 Register the "tailrec"-monad

For the "tailrec"-monad (which is the identity monad) we take the identity function as map, there are no flexible parameters, and the monadic type itself is the (only) fixed argument. As a consequence, we can only define tail-recursive and mutual recursive functions which share the same return type.

⟨*ML*⟩

end

3 Examples

theory *Partial-Function-MR-Examples*
imports
  Partial-Function-MR
  HOL−Library.Monad-Syntax
  HOL.Rat
begin

3.1 Collatz function

In the following, we define the Collatz function, which is artificially encoded via mutually recursive functions. As second argument we store the intermediate values. It is currently unknown whether this function is terminating for all inputs or not.

```
partial-function-mr (tailrec) collatz and even-case and odd-case where
  collatz (x :: int) xs =
    (if (x ≤ 1) then rev (x # xs) else
    (if (x mod 2 = 0) then even-case x (x # xs)
     else odd-case x xs))
| even-case x xs = collatz (x div 2) xs
| [simp]: odd-case x xs = collatz (3 ∗ x + 1) (x # xs)
```

The equations are registered as code-equations.

lemma *length* (*collatz 327* []) = *144* ⟨*proof*⟩

The equations are accessible via .simps, but are not put in the standard simpset.

lemma *collatz 5* [] = [*5*,*16*,*8*,*4*,*2*,*1*] ⟨*proof*⟩

3.2 Evaluating expressions

Note that we also provide a least fixpoint operator. Hence, the evaluation function will clearly be partial. The example also illustrates the usage of polymorphism and of different return types.

In the following datatype, *Mu b f a* encodes the least n such that $b(f^n(a))$.

```
datatype 'a bexp =
  BConst bool
| Less 'a aexp 'a aexp
| Eq 'a aexp 'a aexp
| And 'a bexp 'a bexp
and 'a aexp =
  Plus 'a aexp 'a aexp
| Div 'a aexp 'a aexp
| IfThenElse 'a bexp 'a aexp 'a aexp
| AConst 'a
| Mu 'a ⇒ 'a bexp 'a ⇒ 'a aexp 'a aexp
```

```
partial-function-mr (option)
  b-eval and a-eval and mu-eval where
  b-eval bexp = (case bexp of
     BConst b ⇒ Some b
   | Less a1 a2 ⇒ do {
        x1 ← a-eval a1;
        x2 ← a-eval a2;
        Some (x1 < x2)
     }
   | Eq a1 a2 ⇒ do {
        x1 ← a-eval a1;
        x2 ← a-eval a2;
        Some (x1 = x2)
     }
   | And be1 be2 ⇒ do {
        b1 ← b-eval be1;
        b2 ← b-eval be2;
        Some (b1 ∧ b2)
     }
  )
| a-eval aexp = (case aexp of
     AConst x ⇒ Some x
   | Plus a1 a2 ⇒ do {
        x1 ← a-eval a1;
        x2 ← a-eval a2;
        Some (x1 + x2)
     }
   | Div a1 a2 ⇒ do {
        x1 ← a-eval a1;
        x2 ← a-eval a2;
        if (x2 = 0) then None else Some (x1 / x2)
     }
   | IfThenElse bexp a1 a2 ⇒ do {
        b ← b-eval bexp;
        (if b then a-eval a1 else a-eval a2)
     }
   | Mu b f a ⇒ do {
        mu-eval b f a 0
     }
  )
| mu-eval b f a n = do {
      x ← a-eval a;
      check ← b-eval (b x);
      (if check then Some (of-nat n) else
       mu-eval b f (f x) (Suc n))
   }
```

definition

five-minus-two = *a-eval* (*Mu* (λ *x*. *Eq* (*AConst 5*) (*AConst x*)) (λ *x*. *Plus* (*AConst x*) (*AConst 1*)) (*AConst* (*2* :: *rat*)))

value *five-minus-two*

3.3 An example with contexts

Mutual recursive partial functions also work within contexts.

context
fixes *y* :: *int*
begin
partial-function-mr (*tailrec*) *foo* **and** *bar* **where**
foo x = (*if x* = *y then foo* (*x* − *1*) *else* (*bar x* (*y* − *1*)))
| *bar x z* = *foo* (*x* + (*1* :: *int*) + *y*)
end

end

References

[1] A. Krauss. Recursive definitions of monadic functions. In *Proc. of the International Workshop on Partiality and Recursion in Interactive Theorem Proving*, volume 43 of *EPTCS*, pages 1–13, 2010. doi:10.4204/EPTCS.43.1.