Automating side conditions in formalized partial functions

Cezary Kaliszyk

cek@cs.ru.nl
Institute for Computing and Information Sciences,
Radboud University Nijmegen, the Netherlands

Abstract. Assumptions about the domains of partial functions are necessary in state-of-the-art proof assistants. On the other hand when mathematicians write about partial functions they tend not to explicitly write the side conditions. We present an approach to formalizing partiality in real and complex analysis in total frameworks that allows keeping the side conditions hidden from the user as long as they can be computed and simplified automatically. This framework simplifies defining and operating on partial functions in formalized real analysis in HOL Light. Our framework allows simplifying expressions under partiality conditions in a proof assistant in a manner that resembles computer algebra systems.

1 Introduction

1.1 Motivation

When mathematicians write partial function they tend not to explicitly write assumptions about their domains. It is common for mathematical texts to include expressions like:

$$\dots \frac{1}{x} \dots$$

without specifying the type of the variable x and without giving any assumptions about it.

On the other hand these assumptions are necessary in proof assistants. Since most proof assistants are total frameworks, a similar formula expressed there looks like:

$$\forall x \in \mathbb{C}. x \neq 0 \Rightarrow \dots \frac{1}{x} \dots$$

The assumptions about the domain are obvious for any mathematician, in fact they can be generated by an algorithm. All names that have not been defined previously are considered to be universally quantified variables and all applications of partial functions give raise to conditions about their arguments. Inferring the types of variables is something that proof assistants are already good at. Giving the type of just one of the terms in an expression is often enough for a proof assistant to infer the types of the other. Mathematicians often work in a particular setting, where arithmetic operations and constants are assumed to be of

particular types. Some proof assistants have mechanisms that allow to achieve a similar effect, eg. prioritizing a type in HOL Light or using a local scope in Coq [5].

There are many examples of statements in libraries of theorems for proof assistants that include assumptions which are often omitted in mathematical practice. In particular the HOL Light library part concerning real analysis includes statements like `EXP_LN`:

$$\forall x.0 < x \Rightarrow exp(\ln(x)) = x$$

Here the type of x is inferred automatically as real from the type of applied functions (the complex versions of the exponent and logarithm functions have different names in the library), but the domain conditions are not taken care of. The real logarithm is defined only for positive numbers, but the positivity assumption is not only in the statement of the theorems that include it, but also appears many times in the proofs that use this fact.

Computer algebra systems allow applying partial functions to terms and some of them have assumptions about variables computed automatically. This might be one of the reasons why for mathematicians computer algebra systems are usually more appealing than proof assistants. Unfortunately the way assumptions are handled in those systems is often approximate, and this is one of the reasons computer algebra systems sometimes give erroneous answers [2]. Therefore handling assumptions cannot be done in the same way in theorem proving.

In [11] we show, that it is possible to implement a prototype computer algebra system in HOL Light and that proof assistants are already able to perform many simplification operations that one would expect from computer algebra. The prototype is able to perform many computations that involve total functions[1], but even simplest operations that require understanding partiality fail, since HOL Light is a total framework:

```
 In1 := diff (diff (\x. &3 * sin (&2 * x) + &7 + exp (exp x)))
Out1 := \x. exp x pow 2 * exp (exp x) + exp x * exp (exp x) +
        -- &12 * sin (&2 * x)
 In2 := diff (\x. &1 / x)
Out2 := diff (\x. &1 / x)
```

The problem with the above example is that the function $\frac{1}{x}$ is mathematically a partial function that is not defined in zero. Still computer algebra systems asked for the derivative of it reply with $-\frac{1}{x^2}$. This answer is correct since the original function is differentiable on the whole domain where it is defined, and its derivative has the same domain. The proposed approach will let the framework correctly compute this kind of expressions.

We would also like to check whether approach for handling partiality in an automated way can be useful not only in formalizing partiality but might generalize to formalizing functions that operate on more complicated data structures, like when formalizing multivaluedness.

[1] The `&` operator is the coercion from natural numbers

1.2 Approach

The domain of the function can be often inferred from the function itself. For example the domain of $1 + \frac{1}{x}$ can be computed to be $\lambda x.x \neq 0$. In such circumstances the domain can be represented by the function itself relieving the user from typing unnecessary expressions. This is not always the case. For example if the function $\lambda x.\frac{1}{x} - \frac{1}{x}$ is simplified to $\lambda x.0$ deriving the domain $\lambda x.x \neq 0$ is not possible. When a singularity point is not removed from the domain, the domain can be recomputed from the expression itself. Expressions in which singularity points are removed occur rarely in practical examples.

When we apply an operation in a CAS system[2] to a function f in a domain D, a function f' and its domain D' are returned. If the system can prove that D' represents the same domain as the one which we can compute from f' we can discard D'. We will be able to recompute it whenever it will be needed.

Our approach is to let the user input the partial functions as values from and to the `option` type and show them to the user as such, but to perform all operations on a total function of the underlying proof assistant with keeping the domain predicate alongside with the function. To do this we have two representations for functions and convert between them. The first representation is functions that operate on values in the option type and the second is pairs of total functions and domain predicates. We show how higher order functions (differentiation) can be defined in this framework and how terms involving it can be treated automatically.

1.3 Related work

There are multiple approaches and frameworks for formalizing partial recursive functions. Ana Bove and Venanzio Capretta [4] introduce an approach to formalizing partial recursive functions and show how to apply it in the Coq proof assistant. Normally recursive functions are defined directly using `Fixpoint`, but that allows only primitive recursion. They propose to create an inductive definition that has a constructor for every recursive definition and create a `Fixpoint` that recurses over this definition. Alexander Krauss [12] has developed a framework for defining partial recursive functions in Isabelle/Hol, that formally proves termination by searching for lexicographic combinations of size measures. William Farmer [9] proposes a scheme for defining partial recursive functions and implements it in IMPS. The main difference is that those approaches and frameworks compute the domains of partial recursive functions whereas we concentrate on functions in analysis which cannot be obtained by recursion and where the domain is limited because there are no values of the functions that would match their intuitive definition or that would allow properties like continuity.

The existing libraries for proof assistants contain formalized properties of functions in real and complex analysis. There are common approaches to partiality in existing libraries. It is common to define every function total. This is

[2] We refer to the computer algebra functionality embedded in HOL as the CAS system

the case for the HOL Light [10] library. Division is defined to return zero when dividing by zero. The resulting theory is consistent, but to make some standard theorems true assumptions are required. For example REAL_DIV_REFL:

$$\forall x.x \neq 0 \Rightarrow \frac{x}{x} = 1.$$

Another common approach is to require proofs that arguments applied to partial functions are in their domains. This is the case for the CoRN library [7] of formalized real and complex analysis for Coq. There division takes three arguments, the third one is a proof that the second argument is different from zero.

There are approaches to include partiality in the logic of the proof assistant. Those unfortunately complicate the logic and are already complicated for first order logics [16]. Some proof assistants are based on logics that support partial functions. An example is PVS [15] where partial functions are obtained by subtyping and IMPS [8] where there is a built-in notion of definedness of objects in the logic.

Olaf Müller and Konrad Slind [14] present an approach for lifting functions with the option monad that is closest to the one presented here. Their approach is aimed at partial recursive functions where computation of the domains of functions is not possible. Our approach is similar to applying the option monad to the real and complex values, but since particular functions need to have their domains reduced, we explicitly compute and keep the domains of functions and be able to transform these values back to original ones.

Finally computer algebra systems have their own approaches to partiality, eg "provisos" [6]. The main difference is they are intended to obtain maximum usability, sometimes at the cost of correctness. This is why those approaches cannot be used in a theorem proving environment.

1.4 Contents

This paper is organized as follows: in Section 2 we give the basic definitions of the two representations of partial functions and we define the operations used to convert between those representations. We also show a simplified example of a computation with partial functions. In Section 3 we present the design decisions and the details of our formalization. We show how does the automation work and show its limitations. Finally in Section 4 we present a conclusion and possible future work.

2 Proposed Approach

2.1 Basic definitions

Our approach involves two representations of partial functions. The first representation is: as pair of a total extension of the original function and a domain

predicate. The second representation is: a function from an option type to an option type. The first representation will be used in all automated calculations and the latter will be used in the user input and if possible in the output since it resembles better mathematical notation.

An option type is a type built on another type. The option type has two constructors. One denoting that the variable has a value and one used for no value. In proof assistants they are usually written as `SOME` α and `NONE`. We will denote those with $\overline{\alpha}$ and $-$. To simplify reading of the types, variables of the option type will be denoted as z and real variables as x both in the paper and in the shown examples from the system. The approach works for partial values of different types, but since HOL Light does not have dependent types we cannot generalize over types, so we present our approach for a single type of partial values. We chose real numbers and not complex numbers since there are more decision procedures available in HOL Light for real numbers and we make use of them.

We define two operations to convert between the two representations. Creating operations that work on the option type from the operations on the underlying proof assistant type is similar to applying the `option` monad operations *bind* composed with *return* to the functions in the proof assistant. In fact this is equivalent to the presented approach for functions that are really total. For functions that are undefined on a part of their original domain we additionally require the desired domain predicate so we create an operation that will additionally require the domain predicate and check it in the definition. We define @ that converts functions from the pair representation to the option representation (written as `papp` in the HOL Light formalization) and $@^{-1}$ that converts a function in the option type to a pair (`punapp` in HOL Light). The definition of @ is straightforward:

$$(f, D)@z = \begin{cases} \overline{fx} \text{ if } z = \overline{x} \wedge D(x) \\ - \quad \text{otherwise} \end{cases}$$

The inverse operation can be defined using the Hilbert operator (which we will denote as ϵ). This operator takes a property and returns an element that satisfies this property if such an element exists. It returns an undefined value when applied to a property that is not satisfied for any element. The inverse operation is defined as:

$$@^{-1}f = (\lambda x.\epsilon v.f(\overline{x}) = \overline{v}, \lambda x.\exists v.f(\overline{x}) = \overline{v})$$

The $@^{-1}$ function is the left inverse of @, (in fact we prove this in our formalization that for any F, D and z)[3]:

$$@^{-1}(\lambda z.(f, D)@z) = (f, D)$$

[3] The $@^{-1}$ function is not the right inverse of @. This would require that for any function f and any z, $(@^{-1}f)@z = f(z)$. But this equality is not true for $z = \text{NONE}$ and $f(\text{NONE}) = \text{SOME}(0)$.

With the two operations definitions of the translations of the standard arithmetic operations are simple. The @ operator will check that the arguments applied to plus are defined. Note that in HOL Light the syntax and semantics of expressions are very close, namely when syntactic expressions are applied to values they can be reduced to its syntax. This is why we will not distinguish between syntax and semantics in the paper:

$$a + b =_{\text{def}} @(\lambda xy.(x + y), \lambda xy.\top)$$

We can also define higher order functions that operate on partial functions by embedding the existing higher order operators from the proof assistant, first in the pair representation:

$$(f, D)' =_{\text{def}} (f', \lambda x.D(x) \wedge f \text{ is differentiable in } x)$$
$$f'(z) =_{\text{def}} (@^{-1}(f)')@z$$

2.2 Example in mathematical notation

With the definitions from the previous section it is possible to automatically simplify the side conditions in partial functions, we will first show it in the example and then show the full HOL Light definitions and the algorithm for simplification in Section 3.2.

We will show a simplified example of automatically computing a derivative of a partial function in our framework. We will denote the derivative of a function $f(x)$ as $f(x)'$. The user types an expression:

$$(\lambda z.\pi z^2 + cz + \frac{2}{z})'$$

The expression that the user sees is written with standard mathematical operators. All the operator symbols are overloaded, and they are understood as the operations on partial functions, that is functions of type (ℝ)option → (ℝ)option → ... → (ℝ)option. In the above expression z is the only variable of the (ℝ)option type. All the other constants and expressions are their translations from the underlying total functions or constants. The only functions that are really partial (that is undefined on a part of the proof assistant domain) are division and differentiation and they are defined by providing their domain. The system unfolds the translation of all operators and constants, and computes a total function and its domain[4]:

$$(\lambda z.\langle \lambda x.\pi x^2 + cx + \frac{2}{x}, \lambda x.x \neq 0\rangle @z)'$$

We finally translate the derivative. For the obtained function we add the requirement that the derivative of the original function exists in the given point,

[4] In some proof assistants all computation is really simplification done by rewrite rules. This is the case in HOL Light in which we will be formalizing this example, but we will refer to those simplifications as computation in the text.

otherwise a function defined in one point would always be differentiable there. This domain condition will be often combined with the assumptions about the domain of the original function:

$$\lambda z.(\langle(\lambda x.\pi x^2 + cx + \frac{2}{x})', \lambda x.x \neq 0 \wedge (\lambda x.\pi x^2 + cx + \frac{2}{x}) \text{ is differentiable in } x)\rangle @ z)$$

We can then apply the decision procedure for computing derivatives of total functions in the underlying proof assistant. It was not possible earlier, since the result of the procedure is a predicate with additional assumptions. It is possible with the use of the `papp` operator, since its definition ensures that the result does not depend on the function outside its desired domain. Since we also know the set on which the reciprocal is differentiable the domain can be simplified:

$$\lambda z.(\langle(\lambda x.2\pi x + c - \frac{2}{x^2}), \lambda x.x \neq 0\rangle @ z)$$

Finally we try to return to the partial representation. This is done by reconstructing a partial function with the same symbols and recomputing its domain.

$$\lambda z.2\pi z + c - \frac{2}{z^2} = \lambda z.(\langle(\lambda x.2\pi x + c - \frac{2}{x^2}), \lambda x.x \neq 0\rangle @ z)$$

Since the domains agree we can convert back and display the left hand side of the above equation as the final result to the user.

Returning from the representation of the function as a total function and its domain to the option type representation is not always possible, since a partial expression does not need to have an original form (can be expressed in the option representation). On the other hand the simplification is often possible and when it is possible it is desirable since it allows for greater readability. An example where it is not possible is:

$$\lambda z.\frac{1}{z} - \frac{1}{z} = \lambda z.(\langle \lambda x.0, \lambda x.x \neq 0\rangle @ z)$$

The value is not equal to the constant function zero, since the expression does not have a value when x is zero. Furthermore for values of the option type even the term $z - z$ is not equal to zero if $z =$ `NULL`, therefore even after simplification to zero its value will depend on the variable z.

There are two approaches of treating this kind of terms. One can either simplify it to zero while leave the domain condition or not simplify the expression at all. We currently do not simplify expressions for which we cannot find a valid partial representation to return to. This is to avoid showing the user the complicated representation with the domain conditions. A possible approach that allows those simplifications and displays results in the option representation will be mentioned in Section 4.1.

3 The implementation

3.1 Design decisions

For our formalization we chose HOL Light. The factors that influenced our choice were: a good library of real and complex analysis, as well as the possibility to write conversions in the same language as the language of the prover itself. HOL Light is written in OCaml and is provided as an extension of it. This is very convenient for developing since it allows generating definitions and simplification rules by a programs and immediately using them in the prover.

In the representation with option types we use the vector type $\mathbb{R}^n \to \mathbb{R}$ instead of the curried types $\mathbb{R} \to \mathbb{R} \to \ldots \to \mathbb{R}$ to represent functions. One can convert between these two representations and the latter representation is often preferred since it allows partial application. The reason why we chose to work with the vector representation is that HOL Light does not have general dependent types. Instead it has a bit less powerful mechanism that only allows proving theorems that reason about $\mathbb{A}^n$ for any n. We will use this to prove theorems about n-ary functions. With this approach some definitions (`papp` mentioned below and its properties) will have to be defined for multiple arities. On the other hand the theorems that are hard to prove will be only be needed to be proved once. Otherwise they would be needed for all versions of curried functions.

3.2 HOL Light implementation details

In this section we will give the formalization details. To understand them knowledge of HOL Light [10] is required. We will show an example of automatically computing the derivative of the partial function

$$f(z) = \pi z^2 + cz + \frac{2}{z}.$$

When the user inputs this function in the correct syntax in the main loop of the CAS, the system responds with the correct answer:

```
  In1 := pdiff (\z. SOME pi * z * z + SOME c * z + & 2 / z)
 Out1 := \z. & 2 * SOME pi * z + SOME c + --& 2 / (z * z)
```

The system computed this derivative automatically, but we will look at the conversions performed step by step. First lets examine the types in the entered expression. The variable z used in the function definitions is of the type `(real)option`. We overload all the standard arithmetic operators to their versions that take arguments of the `(real)option` type and produce results of this type. The coercion from naturals operator `&` creates values of this type. We decided not to overload the `&` operator to the coercion from real numbers (`SOME`), since this would lead to typing ambiguity and would require some types to be explicitly given in expressions.

The semantics of the standard arithmetic operations is to return a value if all arguments have a value and `NONE` if any of the arguments is `NONE`. For real

partial functions we define an operation (called `papp`) that will create a partial function of type `(real)option` → `(real)option` → ... → `(real)option` from a pair of a HOL Light total function $\texttt{real}^n \rightarrow \texttt{real}$ and a predicate expressing its domain $\texttt{real}^n \rightarrow \texttt{bool}$. We show below the definitions of `papp` for one and two variables. In the formalization we see them as `papp1`, `papp2`, ..., but in the text we will refer to all those definitions together as `papp`:

```
new_definition `(papp1 (f, d) (SOME x) = if (d (lambda i.x)) then
 (SOME (f (lambda i.x))) else NONE) /\
 (papp1 ((f:A^1->A), (d:A^1->bool)) NONE = NONE)`

new_definition `(papp2 ((f:A^2->A), (d:A^2->bool)) (SOME x) (SOME y) =
 if (d (lambda i.if i = 1 then x else y)) then
   (SOME (f (lambda i.if i = 1 then x else y))) else NONE) /\
 (papp2 (f, d) NONE v = NONE) /\ (papp2 (f, d) v NONE = NONE)`;;
```

In the above definitions we see the usage of `lambda` and below we see the usage of `$`. Those are used to create vectors and refer to vector elements. The reasons for using the vector types instead of curried type for functions was discussed in Section 3.1.

The total binary operations can be defined by applying a common operator, that defines binary operators in terms of `papp` for two variables. The types of all defined binary operations is `(real)option` → `(real)option` → `(real)option`. We show only the definition of addition on partial values:

```
new_definition `pbinop (f:A->A->A) x y =
  papp2 ((\x:A^2. (f:A->A->A) (x$1) (x$2)),(\x:A^2.T)) x y`;;

new_definition `padd = pbinop real_add`;;
```

The first partial function is division defined in terms of the reciprocal.

```
new_definition `pinv = papp1 (partial ((\x:real^1. inv (x$1)),
  \x:real^1. ~((x$1) = &0)))`;;

new_definition `pdiv x y = pmul x (pinv y)`;;
```

`pdiff` is the unary differentiation operator. It takes partial functions of the type `(real)option`→`(real)option` and returns functions of the same type. Since the derivative may not always exist it is defined using the Hilbert operator. Given a (partial) function it returns a partial function being a derivative of the given one on the intersection of its domain and the set on which it is differentiable. We will again define it in terms of `papp` applied to a total function and its domain. Since we are given a function and need to find its underlying total function and domain to apply the original differentiation predicate we will define `punapp` that returns this pair. For our definition it returns a pair of `real`→`real` and `real`→`bool`:

```
new_definition ‘punapp1 f = ((\x:real^1. @v:real. (f(SOME (x$1))) =
  (SOME v)), (\x:real^1. ?v. (f (SOME (x$1))) = (SOME v)))‘;;

new_definition ‘pdiff_proto (f:real^1->real, d:real^1->bool) =
  ( (\x:real^1. if d x /\ ?v. ((\x. f (lambda i. x)) diffl v) (x$1)
  then @v. ((\x. f (lambda i. x)) diffl v) (x$1) else &0) ,
  (\x:real^1. d x /\ ?v. ((\x. f (lambda i. x)) diffl v) (x$1)) )‘;;

new_definition ‘pdiff f = papp1 (pdiff_proto (punapp1 f))‘;;
```

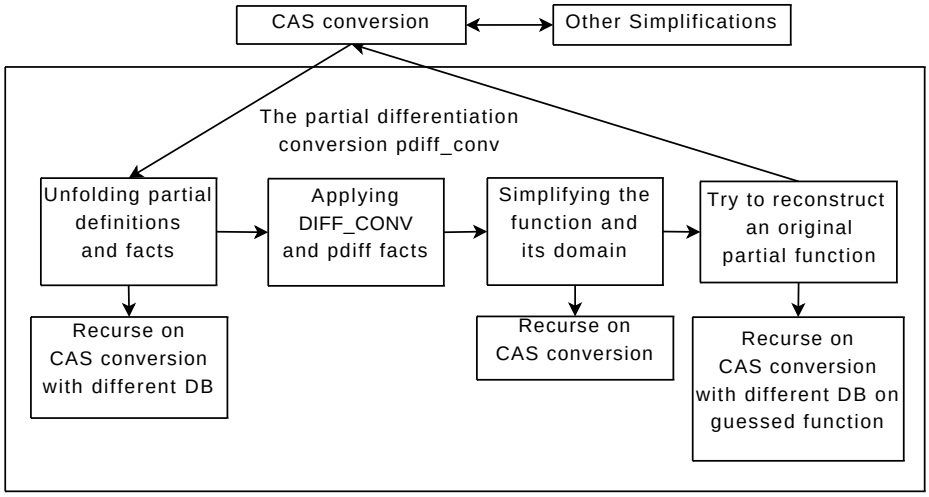

Fig. 1. A schematic view of the simplification performed by the partial differentiation conversion.

The simplification of the term will be performed by a partial differentiation conversion `pdiff_conv` (Fig. 1). This conversion is a part of the knowledge base of the CAS and will be called by the CAS framework when the term has a `pdiff` term in it. To simplify the implementation of the partial differentiation conversion it will recursively call the CAS conversion with a modified database to simplify terms. The first step is a simplification performed by the main CAS conversion with the database of theorems for extended to include the above definitions of the partial operators and some basic facts, that will be described below. The conversion proves:

```
|- pdiff (\z. SOME pi * z * z + SOME c * z + & 2 / z) =
papp1 ((\x. @v. ((\x. x pow 2 * pi + c * x + &2 * inv x) diffl v) (x$1)),
(\x. ~(x$1 = &0) /\ (?v. ((\x. if ~(x = &0)
 then x pow 2 * pi + c * x + &2 * inv x else @v. F) diffl v) (x$1))))
```

All the partial operators and the `pdiff` operator were unfolded to their definitions. We notice that the partiality included in division (reciprocal) and differentiation have been propagated to the term. All occurrences of variables are pulled inside the `papp` terms and consecutive `papp` applications are combined by a set of reduction rules. This set includes a number of theorems, for the categories we give only single examples for one variable:

- rewrite rules that reduce the number of `papp` applications for `SOME` terms for arbitrary numbers of variables. An example for the second of two variables:

```
# papp2_beta_right;;
val it : thm = |- papp2 (f, d) (a:(A)option) (SOME b) =
  papp1 ((\x. f (lambda i. if i = 1 then x$1 else b)),
  (\x. d (lambda i. if i = 1 then x$1 else b))) a
```

- rewrite rules that combine multiple occurrences of the same variable:

```
# papp2_same;;
val it : thm =
|- papp2 (f, d) x x = papp1 ((\x:real^1. f ((lambda i. x$1):real^2)),
 \x:real^1. d ((lambda i. x$1):real^2)) x
```

- rewrite rules that combine consecutive applications of `papp` possibly with different numbers of abstracted variables:

```
# papp1_papp1;;
val it : thm = |- papp1 (f1, d1) (papp1 (f2, d2) (x:(A)option)) =
   papp1 ((\x. f1 (lambda i.(f2 x))),
   (\x. d2 x /\ d1 (lambda i.(f2 x)))) x
```

The next step performed by the partial differentiation conversion extracts the function to which the `diffl` term is applied. The HOL Light `DIFF_CONV` is applied to this term. For total functions it produces a `diffl` theorem with no additional assumptions. For partial functions `DIFF_CONV` produces conditional theorems that have additional assumptions about the domain. For our example:

```
# DIFF_CONV `(\x. x pow 2 * pi + x * &c + &2 * inv x)`;;
val it : thm = |- !x. ~(x = &0) ==>
  ((\x. x pow 2 * pi + x * &c + &2 * inv x) diffl
  (((&2 * x pow (2 - 1)) * &1) * pi + &0 * x pow 2) +
  (&1 * &c + &0 * x) + &0 * inv x + --(&1 / x pow 2) * &2) x
```

Our formalization includes certain theorems about derivatives of partial functions where the derivative exists depending on some condition. For the example case the used theorem is about derivatives of functions that are not differentiable in a single point. We provide some similar theorems for inequalities which may arise in differentiating more complicated functions. The exact statement of the theorem used here is:

```
# pdiff_but_for_point;;
val it : thm = |- (!x. ~(x = w) ==> (f diffl (g x)) x) ==>
  papp1((\(x:real^1). @v. ((\x:real. f x) diffl v) (x$1)),
        (\(x:real^1). (~(x$1 = w) /\ d (x$1)) /\
    ?v. ((\x:real. if ~(x = w) then f x else @v. F) diffl v) (x$1))) =
  papp1((\(x:real^1). g (x$1)), \(x:real^1). ~(x$1 = w) /\ d (x$1))
```

The partial differentiation conversion combines the above facts to prove:

```
|- pdiff (\z. SOME pi * z * z + SOME c * z + & 2 / z) =
papp1 ((\x. (((&2 * x$1 pow (2 - 1)) * &1) * pi + &0 * x$1 pow 2) +
  (&0 * x$1 + &1 * c) + &0 * inv (x$1) + --(&1 / x$1 pow 2) * &2),
 (\x. ~(x$1 = &0)))
```

The above function can be easily simplified, and this simplification is performed by recursively calling the CAS conversion both on the function and on the domain. For our example only the function can be reduced. For the recursive call to the CAS conversion we do not include the facts about partiality to prevent looping. The conversion proves:

```
|- pdiff (\z. SOME pi * z * z + SOME c * z + & 2 / z) =
  papp1 ((\x. &2 * pi * x$1 + c + -- &2 * inv (x$1 * x$1)),
  (\x. ~(x$1 = &0)))
```

The last part of `pdiff_conv` tries to convert the term back to the original representation. As described is Section 3.1 this is not always possible, but it will be possible in our case. The algorithm for computing the original form examines the tree structure of the total function and reconstructs a partial function with the same structure. In our case:

```
# pconvert `(&2 * pi * (x:real^1)$1 + c + -- &2 * inv (x$1 * x$1))`;;
val it : term = `& 2 * SOME pi * x + SOME c + --& 2 * pinv (x * x)`
```

We now check if the domain of the guessed partial function is the same as the original real one. To do this we apply the CAS conversion to the guessed term with the partial function definitions and facts about them again:

```
# cas_conv it;;
val it : thm = |- & 2 * SOME pi * z + SOME c + --& 2 * pinv (z * z) =
  papp1 ((\x. &2 * pi * x$1 + c + -- &2 * inv (x$1 pow 2)),
   (\x. ~(x$1 pow 2 = &0))) z
```

The domain of the converted function is the same as the domain of the function we that was computed by differentiation[5]. Therefore we can compose this theorem with the previous result arriving at the final proved theorem:

[5] The two domains can be expressed in a slightly different way, thus there may be some theorem proving involved to show that they are equal. In our implementation the only thing performed is the CAS conversion, that internally tries HOL Light decision procedures for reals and tautology solving.

```
|- pdiff (\z. SOME pi * z * z + SOME c * z + & 2 / z) =
   (\z. & 2 * SOME pi * z + SOME c + --& 2 / (z * z))
```

And the user is presented with the right hand side of the equation.

3.3 How to extend the system

In this section we will show examples that the system cannot handle automatically. We will then show how the user can add theorems to the knowledge base to add automation for simplification of those terms. Consider adding a new partial function being the real square root:

```
new_definition `psqrt = papp1 ((\x. sqrt (x$1)), (\x. (x$1) >= &0))`;;
```

The original HOL Light differentiation conversion `DIFF_CONV` is able to differentiate the real square root producing a differentiation predicate with a condition:

```
# DIFF_CONV `\x. sqrt x`;;
val it : thm =
|- !x. &0 < x ==> ((\x. sqrt x) diffl inv (&2 * sqrt x) * &1) x
```

The partial differentiation conversion can not simplify the derivative of the partial square root automatically without additional facts in its knowledge base. This is because the result of the original differentiation conversion is only a condition for the function to be differentiable. It does not prove that the function is not differentiable elsewhere (namely in zero). To be able to simplify this function the user needs to prove an additional theorem that would show that the function is differentiable if and only if the variable is greater than zero. Namely:

```
|- (?v. ((\x. if x >= &0 then sqrt x else @v. F) diffl v) ((x:real^1)$1))
   = x$1 > &0
```

Adding this to the knowledge base allows the partial differentiation conversion to handle automatically the partial square root function.

4 Conclusion

The presented approach and formalized framework allow the automation of side-conditions. Simple expressions with partial functions can be simplified transparently to the user. More complicated partiality conditions still appear in the expressions.

The approach allows mathematical expressions in proof assistants to resemble those seen in computer algebra. The language for writing equations and for calculations (rewriting in HOL Light) becomes simpler.

It can be useful for formalizing partial functions that we encounter in engineering books, for example in Abramowitz and Stegun [1] or in the NIST DLMF project [13].

4.1 Future Work

Our primary goal is to check how easily our approach can be extended to more complicated partial operations. For example with integration it is hard to check whether the objects are defined. Of course even then our approach gives a response, but the existential expression in the result may be hard to simplify.

It is important to note, that the standard HOL Light equality does not take into account the option type, so any objects that do not exist will be equal. Defining an equality that is not true for `NONE` is possible, and this is what has been done in IMPS. On the other hand it leads to two separate notions of equality, which makes the expressions more complicated.

We would like to add more automation. All the simplifications that we perform can be done with functions of arbitrary number of variables. Those can be proved on the fly by special conversions. Our formalization currently has all simplifications rules proved for functions with at most two optional variables. Also the `papp` definitions for more variables and facts about them are analogous to their simpler version and their definitions can be created automatically by a ML function that calls HOL Light's definition primitives.

We are looking for a policy for simplifying expressions. Currently when an expression is simplified in the total representation, but we cannot find an original partial representation, the whole conversion fails and the expression is returned unchanged. The same conversions would succeed with assumptions about the domains of variables present in the CAS environment. It would be therefore desirable to suggest assumptions about variables that would allow further simplification of terms [3].

It would be most interesting to see if the presented approach can be extended to address multivaluedness. Multivalued functions are rarely treated in proof assistants. On the other hand multivalued expressions tend to be one of the common sources of mistakes performed by computer algebra systems. There are not too many theorems in prover libraries that concern multivalued functions. The representation of multivalued functions could be done in a similar way as partiality is done in our approach.

References

1. Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables*, volume 55 of *National Bureau of Standards Applied Mathematics Series*. United States Department of Commerce, Washington, D.C., June 1964. 9th Printing, November 1970, with corrections.
2. H. Aslaksen. Multiple-valued complex functions and computer algebra. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 30(2):12–20, June 1996.
3. Michael Beeson. Using nonstandard analysis to ensure the correctness of symbolic computations. *Int. J. Found. Comput. Sci.*, 6(3):299–338, 1995.
4. Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

5. Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.1*. INRIA-Rocquencourt, 2006.
6. Robert M. Corless and David J. Jeffrey. Well ... it isn't quite that simple. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 26(3):2–6, 1992.
7. Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *MKM*, volume 3119 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2004.
8. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System (system abstract). In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 653–654. Springer-Verlag, 1990.
9. William M. Farmer. A scheme for defining partial higher-order functions by recursion. In Andrew Butterfield and Klemens Haegele, editors, *IWFM*, Workshops in Computing. BCS, 1999.
10. John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *LNCS*, pages 265–269. Springer-Verlag, 1996.
11. Cezary Kaliszyk and Freek Wiedijk. Certified computer algebra on top of an interactive theorem prover. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Calculemus/MKM*, volume 4573 of *Lecture Notes in Computer Science*, pages 94–105. Springer, 2007.
12. Alexander Krauss. Partial recursive functions in higher-order logic. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 589–603. Springer, 2006.
13. Daniel W. Lozier. Nist digital library of mathematical functions. *Ann. Math. Artif. Intell.*, 38(1-3):105–119, 2003.
14. Olaf Müller and Konrad Slind. Treating partiality in a logic of total functions. *Comput. J.*, 40(10):640–652, 1997.
15. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Berlin, Heidelberg, New York, 1992. Springer-Verlag.
16. Freek Wiedijk and Jan Zwanenburg. First order logic with domain conditions. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2003.