Partial Functions in Induction Theorem Proving*
— Extended Abstract —

Jürgen Giesl

Dept. of Computer Science, Darmstadt University of Technology, Alexanderstr. 10, 64283 Darmstadt, Germany. E-mail: giesl@informatik.tu-darmstadt.de

Abstract. We present an approach for automated induction proofs with partial functions. Most well-known techniques developed for (explicit) induction theorem proving are unsound when dealing with partial functions. But surprisingly, by slightly restricting the application of these techniques, it is possible to develop a calculus for automated induction proofs with partial functions. In particular, under certain conditions one may even generate induction schemes from the recursions of non-terminating algorithms. The need for such induction schemes and the power of our approach have been demonstrated on a large collection of non-trivial theorems (including Knuth and Bendix’ critical pair lemma). In this way, existing induction theorem provers can be directly extended to partial functions without changing their logical framework.

1 Introduction

The most important proof method for software verification is induction. Therefore, several techniques\(^1\) have been developed to compute suitable induction relations and to perform induction proofs automatically, cf. e.g. [BM79, ZKK88, Bu93, Wal94, KS96]. However, most of these approaches are only sound if all occurring functions are total.

In this paper we show that by slightly modifying the prerequisites of these techniques it is nevertheless possible to use them for partial functions, too. In particular, the successful heuristic of deriving induction relations from the recursions of algorithms can also be applied for partial functions. In fact, under certain conditions one may even perform inductions w.r.t. non-terminating algorithms. Hence, with our approach the well-known existing techniques for automated induction proofs can be directly extended to partial functions.

In [Gie96] we already presented a first approach for induction proofs with partial functions. This approach did not require any reasoning about definedness and it was already very successful for a certain class of conjectures (in particular,

\(^*\) Appeared in Proceedings of the Workshop on the Mechanization of Partial Functions, held in conjunction with the 15th International Conference on Automated Deduction (CADE-15), Lindau, Germany, 1998.

\(^1\) There are two paradigms for the automation of induction proofs, viz. *explicit* and *implicit* induction (e.g. [KM87, BR95]), where we only focus on the first one.
conjectures containing at most one occurrence of a partial function). But to increase the power of our approach, in this paper we suggest a refinement where definedness is made explicit.

In Sect. 2 we introduce our programming language and in Sect. 3 we define the notion of truth used for statements about partial functions. Then in Sect. 4 we sketch how the basic rules usually applied in automated induction theorem proving can be extended to partial functions. In Sect. 5 we discuss some application areas where reasoning about partial functions is required. Finally, we give a comparison with related work in Sect. 6 and end up with a short conclusion.

2 The Programming Language

We consider a first order functional language with eager (i.e. call-by-value) semantics, non-parameterized and free algebraic data types, and pattern matching. As an example regard the algorithms minus and div. They operate on the data type nat for naturals whose objects are built with the constructors 0 and s (where we often write “1” instead of “s(0)” etc.).

\[
\begin{align*}
  &\text{function } \text{minus} : \text{nat} \times \text{nat} \to \text{nat} \\
  &\text{minus}(x, 0) = x \\
  &\text{minus}(s(x), s(y)) = \text{minus}(x, y)
\end{align*}
\]

\[
\begin{align*}
  &\text{function } \text{div} : \text{nat} \times \text{nat} \to \text{nat} \\
  &\text{div}(0, s(y)) = 0 \\
  &\text{div}(s(x), y) = \text{if}(\geq(s(x), y), \\
  &\quad s(\text{div}(\text{minus}(s(x), y), y)), \\
  &\quad 0)
\end{align*}
\]

In general, an algorithm \( f \) is defined by a set of orthogonal (i.e. non-overlapping and left-linear) equations of the form \( f(t_1, \ldots, t_n) = r \) where the terms \( t_i \) are built from constructors and variables only and where all variables of \( r \) also occur in \( t_1, \ldots, t_n \).

We restrict ourselves to well-sorted terms and substitutions, i.e. variables of type \( \tau \) are only replaced by terms of the same data type \( \tau \). Now the operational semantics of our programming language can be defined by regarding each defining equation as a rewrite rule, where however the variables in these rewrite rules may only be instantiated with data objects, i.e. with constructor ground terms. This restriction is due to the eager nature of our programming language. So for example, div’s first defining equation cannot be applied directly to evaluate the term \( \text{div}(0, s(\text{minus}(1, 0))) \), because one argument of div is no constructor ground term. Therefore, \( \text{minus}(1, 0) \) has to be evaluated to 1 first. Afterwards a defining equation of div can be used to evaluate the resulting term \( \text{div}(0, 2) \) to 0.

Our programming language has a pre-defined conditional function \( \text{if} : \text{bool} \times \tau \times \tau \to \tau \) for each data type \( \tau \) (where bool is the data type with the constructors true and false). These conditionals are the only functions with non-eager semantics, i.e. when evaluating \( \text{if}(t_1, t_2, t_3) \), the (boolean) term \( t_1 \) is evaluated first and depending on the result of its evaluation either \( t_2 \) or \( t_3 \) is evaluated afterwards yielding the result of the whole conditional.
Obviously, both algorithms \texttt{minus} and \texttt{div} compute \emph{partial} functions. The defining equations of \texttt{minus} do not cover all possible inputs, i.e., the algorithm \texttt{minus} is \emph{incomplete} and hence, \texttt{minus}(x, y) is only defined if \(x\) is not smaller than \(y\). The algorithm \texttt{div} for truncated division uses a (total) auxiliary function \texttt{ge} to check whether the first argument is greater than or equal to the second one before performing the recursive call. It is not only incomplete, but there are also inputs which lead to a \emph{non-terminating} evaluation (e.g. \texttt{div}(1, 0)). In fact, \texttt{div}(x, y) is only defined if \(y\) is not 0. In general, we say that (evaluation of) a ground term is \emph{defined}, if it can be evaluated to a constructor ground term.

\section{Truth of Statements about Partial Functions}

Now our goal is to verify statements concerning a given collection of algorithms and data types. For instance, we may try to verify that the multiplication of \texttt{div}(n, m) with the divisor \(m\) yields a number \(\leq n\) whenever \texttt{div}(n, m) is defined.

\[ \forall n, m : \text{nat} \; \text{def}(\text{div}(n, m)) = \text{true} \rightarrow \text{ge}(n, \times(m, \text{div}(n, m))) = \text{true} \quad (1) \]

Here, we use an appropriate (total) algorithm \texttt{times} and in order to reason about definedness, we introduce a definedness function \texttt{def} : \(\tau \rightarrow \text{bool}\) for each data type \(\tau\). For any ground term \(t\), \texttt{def}(t) is \texttt{true} iff evaluation of \(t\) is defined.

We only consider universally closed formulas of the form \(\forall x \varphi\) where \(\varphi\) is quantifier free and we often omit the quantifiers to ease readability. So for example, \(\varphi_1 \rightarrow \varphi_2\) is an abbreviation for \(\forall x (\varphi_1 \rightarrow \varphi_2)\), where \(\varphi_1\) and \(\varphi_2\) are quantifier free. We sometimes write \(\varphi(x^*)\) to indicate that \(\varphi\) contains at least the variables \(x^*\) (where \(x^*\) is a tuple of pairwise different variables \(x_1, \ldots, x_n\)) and \(\varphi(t^*)\) denotes the result of replacing the variables \(x^*\) in \(\varphi\) by the terms \(t^*\).

Intuitively, a formula \(\forall x^* \varphi(x^*)\) is \emph{inductively true}, if it holds for all instantiations of \(x^*\) with data objects \(q^*\). For example, formula (1) is true, because the term \texttt{ge}(n, \times(m, \text{div}(n, m))) evaluates to \texttt{true} for all those natural numbers \(n\) and \(m\) where \texttt{div}(n, m) is defined. In the following we will often speak of “truth” instead of “inductive truth”.

For a formal definition of \emph{truth} for statements about partial functions, we use a model theoretic approach. For \emph{total} functions, the notion of inductive truth generally used in the literature is equivalent to validity in the initial model of the defining equations \texttt{Eq}. cf. e.g. [ZKK88, Wa94, WG94, BR95]. However, due to the occurrence of partial functions, now the initial model of \texttt{Eq} is no longer the specific intended model. The reason is that the defining equations do not represent the \emph{eager} evaluation strategy of our programming language. For example, \texttt{div}(0, \text{div}(1, 0)) = 0 is valid in the initial model of the defining equations\footnote{when extended with the equations if(true, \(x, y\)) = \(x\) and if(false, \(x, y\)) = \(y\)} although (innermost) evaluation of \texttt{div}(0, \text{div}(1, 0)) is not terminating.

In our language, a defining equation \(f(t) = r\) may only be applied to evaluate a term \(\sigma(f(t))\) if evaluation of the argument \(\sigma(t)\) is \emph{defined}. i.e., if \texttt{def}(\sigma(t)) is \texttt{true}. Thus, instead of a defining equation \(f(t) = r\) we use the equation \(f(t) = \)

3
if(def(t), r, f(t)). To handle functions with several arguments, in the following let
def(t_1, \ldots, t_n) be an abbreviation for the term if(def(t_1), def(t_2, \ldots, t_n), false). So intuitively, def(t_1, \ldots, t_n) is true iff def(t_i) is true for all i. For the empty tuple (where n = 0), def() is defined to be true. This leads to the following definition of inductive truth for conjectures about partial functions.

Definition (Inductive Truth). Let I be the initial model of

\{ f(t) = \text{if}(\text{def}(t^*), r, f(t^*)) | \text{for each defining equation } f(t^*) = r \}

\cup \{ \text{if}(\text{true}, x, y) = x, \text{if}(\text{false}, x, y) = y \}

\cup \{ \text{def}(c(x^*)) = \text{def}(x^*) | \text{for each constructor } c \}.

Then a formula is inductively true iff it is valid in I.

For terminating and completely defined algorithms, this notion of inductive truth is equivalent to validity in the initial model of the defining equations. Moreover, now the model theoretic semantics of def corresponds to the operational semantics of “definedness”. So for any ground term t, the conjecture def(t) = true is inductively true iff evaluation of t is defined. To verify partial correctness of an algorithm w.r.t. a specification \varphi, one has to prove the conjecture

def(t^*) = true \rightarrow \varphi,

where t^* are the (top-level) terms of \varphi. Thus, an algorithm is partially correct w.r.t. \varphi if \varphi holds for those instantiations where evaluation of all its terms is defined. This notion of partial correctness is widely used in program verification, cf. e.g. [Man74, LS87].

4 Induction Theorem Proving for Partial Functions

Numerous techniques have been developed to perform induction proofs automatically. As (1) contains a call of the function div, this call suggests a plausible induction. For instance, we can apply an induction w.r.t. the recursions of the algorithm div and use the variables n and m as induction variables. For that purpose we perform a case analysis according to the defining equations of div (i.e. n and m are instantiated by 0 and s(y) and by s(x) and y, respectively). In the recursive equation of div we perform another case analysis w.r.t. the condition ge(s(x), y) of the if-term. In the case ge(s(x), y) = true we assume that (1) already holds for the arguments (\text{minus}(s(x), y), y) of div’s recursive call. So instead of (1) it is sufficient to prove the following formulas where we underlined instantiations of the induction variables. Here, \varphi(n, m) abbreviates formula (1).

\varphi(0, s(y)) \hspace{2cm} (2)

ge(s(x), y) = false \rightarrow \varphi(s(x), y) \hspace{2cm} (3)

ge(s(x), y) = true \rightarrow (\varphi(s(x), y) \rightarrow \varphi(s(x), y)) \hspace{2cm} (4)
The technique of performing inductions w.r.t. the recursions of algorithms (like \( \text{div} \)) is commonly used in induction theorem proving, cf. e.g. [BM79, ZKK88, Bun89, Wa94]. However, induction proofs are only sound if the induction relation used is well founded (i.e. if there is no infinite descending chain \( t_1^* \succ t_2^* \succ \ldots \) w.r.t. the induction relation \( \succ \)). Here, the well-foundedness of the induction relation corresponds to the termination of the algorithm \( \text{div} \). Because when proving a statement for the inputs of a recursive defining equation, we assume as induction hypothesis that the statement holds for the arguments of the recursive call.

Hence, inductions w.r.t. non-terminating algorithms like \( \text{div} \) must not be used in an unrestricted way. For example, by induction w.r.t. the non-terminating algorithm \( f \) with the defining equation \( f(x) = f(x) \) one could prove any formula, e.g. false conjectures like \( \neg x = x \).

However, for formula (1) the induction w.r.t. the recursions of \( \text{div} \) is nevertheless sound, i.e. inductive truth of (2), (3), and (4) in fact implies inductive truth of (1). To see this, assume that \( \varphi(n, m) \) is false. Recall that \( \varphi(n, m) \) has the form ‘\( \text{def}(\text{div}(n, m)) = \text{true} \rightarrow \varphi'(n, m) \)’. Thus, there must be a counterexample, i.e. two numbers \( p \) and \( q \) such that \( \text{div}(p, q) \) is defined, but \( \varphi'(p, q) \) is false.

Let \( \succ_{\text{div}} \) be the relation where \( \langle p_1, q_1 \rangle \succ_{\text{div}} \langle p_2, q_2 \rangle \) holds for two pairs of data objects iff evaluation of \( \text{div}(p_1, q_1) \) is defined and leads to the recursive call \( \text{div}(p_2, q_2) \). This relation is well founded although \( \text{div} \) is partial. Hence, there also exists a minimal counterexample \( \langle p, q \rangle \) w.r.t. \( \succ_{\text{div}} \).

By (2) and (3), \( \langle p, q \rangle \) corresponds to a recursive case of \( \text{div} \). Thus due to (4), \( \varphi(\text{minus}(p, q), q) \) is also false, i.e. \( \langle \text{minus}(p, q), q \rangle \) is also a counterexample. Note that by the eager nature of our language, evaluation of \( \text{div}(p, q) \) necessarily leads to evaluation of \( \text{div}(\text{minus}(p, q), q) \). Hence, \( \langle \text{minus}(p, q), q \rangle \) is a smaller counterexample than \( \langle p, q \rangle \) which contradicts the minimality of \( \langle p, q \rangle \).

So due to the eager nature of our programming language, an induction w.r.t. a (possibly partial) algorithm \( f \) using the induction variables \( x^* \) proves a conjecture \( \varphi(x^*) \) for those instantiations where \( f(x^*) \) is defined. Hence, in addition one also has to verify \( \varphi(x^*) \) for those instantiations where \( f(x^*) \) is not defined, i.e. one also has to prove the permissibility conjecture

\[
-\text{def}(f(x^*)) = \text{true} \rightarrow \varphi(x^*).
\]

Thus, by adding this permissibility conjecture to the premises of the induction inference rule, the successful technique of deriving induction relations from the recursions of algorithms may also be used for partial functions. In our example the permissibility conjecture obtained is the following tautology.

\[
-\text{def}(\text{div}(n, m)) = \text{true} \rightarrow (1)
\]

In a similar way, other techniques typically used in automated induction theorem proving can also be extended to partial functions. For example, analogously to induction w.r.t. algorithms, a structural induction using the induction variable \( x \) proves \( \varphi(x) \) for all instantiations of \( x \) with defined terms. In other words, structural induction may also be used in the presence of partial functions, if in
addition we also prove the permissibility conjecture

\[ \neg \text{def}(x) = \text{true} \rightarrow \varphi(x). \]

Next we consider the well-known technique of symbolic evaluation, i.e. the application of defining equations as rewrite rules. Due to our eager evaluation strategy, now one has to take into account that a defining equation \( f(t^*) = r \) can only be applied to evaluate the term \( \sigma(f(t^*)) \) if the arguments \( \sigma(t^*) \) are defined, i.e. if \( \text{def}(\sigma(t^*)) = \text{true} \) holds. Hence, when evaluating the term \( \sigma(f(t^*)) \) in a formula \( \varphi \), one also has to prove the permissibility conjecture

\[ \neg \text{def}(\sigma(t^*)) = \text{true} \rightarrow \varphi. \]

For example, in this way any formula \( \varphi(\text{div}(0, \text{minus}(\ldots))) \) can be transformed into \( \varphi(0) \) and the permissibility conjecture

\[ \neg \text{def}(0, \text{minus}(\ldots)) = \text{true} \rightarrow \varphi(\text{div}(0, \text{minus}(\ldots))). \]

Of course, as if is the only function symbol with non-eager semantics, to evaluate a term \( f(t_1, t_2, t_3) \) it is sufficient if just \( t_1 \) is defined.

Finally, first-order inference rules can be applied to simplify or to verify resulting proof obligations. In particular, one may also use axioms \( \text{Ax}^{\text{def}} \) about definedness, which state how \text{def} operates on terms built with algorithms, conditionals, and constructors.

\[
\text{Ax}^{\text{def}} = \{ \text{def}(f(x^*)) = \text{true} \rightarrow \text{def}(x^*) = \text{true} \mid \text{for all algorithms } f \} \\
\cup \{ \text{def(if}(x, y, z)) = \text{true} \rightarrow \text{def}(x) = \text{true} \} \\
\cup \{ \text{def}(c(x^*)) = \text{def}(x^*) \mid \text{for all constructors } c \}.
\]

In this way, the conjecture (1) about \text{div} can be easily be proved.

By modifying the standard inference rules of induction theorem proving as described above, we developed a calculus for induction proofs with partial functions in [Gie98a]. The only difference between the rules of this calculus and the rules typically used for induction theorem proving (with total functions) is the function symbol \text{def}, the axioms \( \text{Ax}^{\text{def}} \), and an additional permissibility conjecture which has to be proved whenever induction or symbolic evaluation is applied. Hence, the existing induction theorem provers can easily be extended to this calculus and thus, these systems can be directly used to reason about partial functions. In particular, they may even perform an induction w.r.t. partial functions whenever the corresponding permissibility conjecture can be verified.

Apart from partial correctness statements (of the form “\( \varphi \) holds if its evaluation is defined”), our calculus also verifies “definedness conjectures” (e.g. statements about termination) which are often needed in both partial and total correctness proofs. Moreover, it can also verify undefinedness. For instance, by induction w.r.t. the partial algorithm \text{div} one can prove that \text{div} is always undefined if its second argument is 0, i.e.

\[ y = 0 \rightarrow \neg \text{def}(\text{div}(x, y)) = \text{true}. \]
A refinement of our approach is obtained by combining it with techniques to approximate the domains of partial functions. More precisely, for every algorithm \( f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau \), a (total) algorithm \( \theta_f : \tau_1 \times \ldots \times \tau_n \rightarrow \text{bool} \) (a domain predicate for \( f \)) is generated, such that the truth of \( \theta_f(t^*) \) implies that evaluation of \( f(t^*) \) is defined. Thus, \( \theta_f \) is a total function specifying the domain of \( f \).

To benefit from these domain approximations, in our calculus one may now use additional axioms \( \text{Ax}^{\text{dom}} \). For every algorithm \( f \), \( \text{Ax}^{\text{dom}} \) contains the axioms

\[
\begin{align*}
\theta_f(x^*)& = \text{true} \rightarrow \text{def}(f(x^*)) = \text{true} \\
\text{def}(x^*)& = \text{true} \rightarrow \text{def}(\theta_f(x^*)) = \text{true}
\end{align*}
\]

which state that the truth of \( \theta_f \) is sufficient for definedness of \( f \), (i.e. domain predicates are partially correct) and that domain predicates are total functions.

To generate domain predicate algorithms \( \theta_f \) automatically, together with J. Brumbruger we developed a method for termination analysis of partial functions which proved successful on a large collection of examples. For details on this work see [BG96, GWB98, BG98].

The approach of present paper is a refinement of the technique suggested in [Gie96]. The technique of [Gie96] had the advantage that one could perform proofs about partial functions (and even inductions w.r.t. partial functions) without reasoning about definedness. However, in this technique induction w.r.t. partial functions was only allowed for statements containing at most one occurrence of a partial function. The reason for this restriction was that definedness was not made explicit and hence, the calculus had to ensure that definedness of the induction conclusion implied definedness of the induction hypothesis. Thus, there exist conjectures which could not be verified with this technique, because their proofs require reasoning about definedness. An example is the proof that

\[
\text{minus}(\text{minus}(x, y), z) = \text{minus}(\text{minus}(x, z), y)
\]

holds whenever its evaluation is defined. This formula can be proved by induction w.r.t. the partial function \( \text{minus} \) using \( x \) and \( y \) as induction variables. However, the technique of [Gie96] does not allow this induction, because \( \text{minus}(x, y) \) is not the only term with a partial root function in the conjecture. On the other hand, with the method of the present paper the proof is easily possible, because by explicit reasoning about definedness one can show that definedness of the induction conclusion indeed implies definedness of the induction hypothesis.

To conclude, while the new calculus performs more refined inference steps than the one in [Gie96], it also imposes more proof obligations, since now definedness conditions have to be checked explicitly, whereas this was not necessary in the former calculus. Hence, for statements containing just one occurrence of a partial function, it is often advantageous to use the calculus of [Gie96] instead.

5 Applications

In this section we analyze areas for applications of our results. One could guess that for those partial functions whose domain can be determined automatically,
techniques for handling partiality are not necessary any more. Indeed, such a function \( f(x^*) \) could be replaced by a new total function \( f'(x^*) \) which first tests whether the corresponding domain predicate \( \theta_f(x^*) \) holds and only executes its body if \( \theta_f(x^*) \) is true. Otherwise, \( f'(x^*) \) returns some default value. However, this transformation of partial functions into total ones leads to several problems.

The first problem is that this approach may result in unintuitive semantics. Moreover, to transform partial functions \( f \) into total extensions \( f' \) one has to construct \( f \)'s domain predicate. However, for many algorithms with nested or mutual recursion, the generation of domain predicates already requires reasoning about (possibly) partial functions, cf. [Gie97].

But the main problem with the transformation of partial functions \( f \) into total ones is that in general the synthesized domain predicate \( \theta_f \) is only sufficient, but not necessary for definedness of \( f \), i.e. it only returns true for a subset of \( f \)'s domain. To determine whether a generated domain predicate indeed describes the exact domain of a function, one may again apply our calculus. For example, then a statement like \( \text{def}(\text{div}(x, y)) = \text{true} \rightarrow \theta_{\text{div}}(x, y) = \text{true} \) can be verified by induction w.r.t. \( \text{div} \). Hence, even for a partial function where an exact domain predicate can be synthesized, one still needs an induction proof w.r.t. a partial function in order to verify this exactness.

However, there are many interesting algorithms where an exact domain predicate cannot be generated automatically. In particular, as the halting problem is undecidable (and as totality is not even semi-decidable), there are even many important total algorithms where totality cannot be verified automatically. For example, the well-known unification algorithm by J. A. Robinson is total, but its termination is a “deep theorem” [Pau85] and none of the current methods for automated termination analysis succeeds with this example. Hence, such functions cannot be handled by (fully) automated theorem provers without the ability of reasoning about possibly partial functions.

To show that our approach indeed can be used to prove relevant theorems about (possibly) partial functions, in [Gie98b] we applied our calculus on more than 400 conjectures from the area of term rewriting systems. As demonstrated there, in contrast to previous approaches (e.g. [MW81, Pau85]), our calculus can prove the soundness of the unification algorithm by induction w.r.t. its recursions without having to verify its termination. So the ability to use induction relations without ensuring their well-foundedness is needed for algorithms where the automated methods fail in determining the domains. But moreover, this ability also allows us to prove conjectures about algorithms like the famous “\( 3x + 1 \)” problem where totality is still an open question, i.e. algorithms whose domain has not even be determined manually.

Even worse, there are numerous practically relevant algorithms with undecidable domain, i.e. there does not exist any exact domain predicate. Typical examples for such algorithms include interpreters for programming languages and algorithms for automated reasoning (e.g. any implementation of a sound and complete first order calculus). For instance, our collection in [Gie98b] contains algorithms which check whether one term rewrites to another in arbitrary
many steps and algorithms for joinability. The domains of such algorithms are obviously undecidable. Nevertheless, we showed that induction w.r.t. such algorithms can be used to prove numerous important theorems\cite{KnuthBendix}. In particular, with our calculus we also proved D. E. Knuth and P. B. Bendix' critical pair lemma [KB70] which states that if all critical pairs of a term rewriting system are joinable, then the system is locally confluent.

Note that apart from reasoning about given partial functions, our approach is also required for program schemes where termination of the program depends on the instantiation of the auxiliary functions which were left unspecified. Moreover, partial algorithms can also result from total ones during program transformations, e.g. when transforming imperative programs into functional ones. This transformation is often necessary for the verification of imperative programs as most existing induction provers are restricted to functional languages.

6 Related Work

In this section we give a short survey on related work. We first discuss alternative notions of "truth" for partial functions in Sect. 6.1. Then in Sect. 6.2 we comment on other techniques for automated reasoning with partial functions.

6.1 Notions of Truth for Partial Functions

Essentially, there are two main possibilities for a formal handling of partial functions. One possibility is to incorporate partiality into the logic itself. In algebraic specifications, partiality is often modelled by partial algebras and different appropriate semantics of equality have been suggested in that framework (see e.g. [Kre87, Rei87] for an overview and alternatives).

In some of these approaches formulas still are either true or false (e.g. by considering all atomic formulas containing undefined terms as false, cf. [Far90]). But one may also use a formalization with a three-valued logic [Kle52], where the truth value of formulas depending on undefined terms is "undefined". See [KK95] for a mechanization of this approach and for a discussion of other alternatives.

The other main possibility to handle partiality is to define an appropriate notion of "truth" in a classical two-valued logic where all terms denote and where all algebras are total. (This is also the approach we used, as our aim was to extend existing induction theorem provers to partial functions, i.e. we did not want to change the underlying logic.)

Our notion of inductive truth corresponds to one of the definitions of inductive validity proposed in [WG94, "Type E"]; Alternative notions of truth have been suggested in [KM86, KM87, Wal94]. Here, an incompletely specified function is interpreted as the set of all possible complete and consistent extensions, cf. also [WG94, "Type D"].

\footnote{In that respect, our proofs differ from other case studies in related areas (e.g. the proofs of the Church-Rosser theorem for the λ-calculus in [Sha88, Nip96]).}
a function is not really partial, but it is a total function with (partly) unknown behaviour. Hence, this approach cannot be used for non-terminating functions like \( f(x) = s(f(x)) \) which do not have a complete consistent extension. In contrast, in our approach every specification is consistent. Thus, we can handle non-termination without any consistency checks. For a further discussion on the differences between the semantics see e.g. [KM86, WG94, AM93].

6.2 Automated Induction Proofs with Partial Functions

We suggested an approach to perform inductions on the objects of the data structures. However, many general purpose tools for reasoning about programs use techniques based on denotational semantics instead. The classical technique for proofs about denotational semantics is computational induction (e.g. D. Scott's fixpoint induction [Sco69]). A full formalization of denotational semantics requires a higher order logic (as it is for instance used in LCF [Pau87]), but an alternative formalization of an LCF-like calculus with fixpoint induction using first order logic can be found in [Sha89].

However, while fixpoint induction is a powerful tool for reasoning about programs, it is less suitable for automation. For that reason, virtually all (explicit) induction provers (i.e. systems with powerful heuristics especially designed for induction like NQTHM [BM79], RRL [ZKK88, KS96], CLAM [Bu93], INKA [Wal94, HS96]) perform inductions on the values of the program variables instead. To find suitable induction relations automatically, a successful heuristic is to use relations which correspond to the recursions of the algorithms occurring in the conjecture. This approach has also been implemented in systems like HOL, LAMBDA, and ISABELLE, cf. [Bou93, Bus93, Si97]. This demonstrates that even in provers for higher order logics, Noetherian induction on the data structure is better suitable for automation than computational induction (see also [Pau85]).

However, a drawback is that up to now the derivation of induction schemes from the recursions of algorithms was just considered to be a good heuristic. But their soundness had to be guaranteed separately, i.e. one had to verify that these induction relations were indeed well founded. To ensure this, in the existing provers, induction relations could only be generated from the recursions of terminating algorithms.

Here, our main observation is that in partial correctness proofs, induction relations do not have to be checked for well-foundedness any more if they are obtained from the recursions of algorithms occurring in the conjecture. So this choice is not just a successful heuristic, but it already guarantees the soundness of the induction schemes. Now the restriction only to derive induction relations from terminating algorithms is no longer necessary. Thus, induction proofs w.r.t. partial functions can be automated without using proof techniques based on denotational semantics. Hence, the existing induction provers and their powerful heuristics can also be applied for partial functions without adapting them to a new logical framework.

\[4\] This is also true for all previous extensions of induction theorem provers to partial functions, e.g. [BK84, KM86, BM88, KS96, Kap97].

10
7 Conclusion

Partial functions are important in many areas, but the techniques implemented in most induction provers rely on the termination of the occurring algorithms. However, we showed that by introducing a few appropriate restrictions, these techniques can be applied for partial functions, too. Based on this observation, we developed a calculus for induction proofs with partial functions in [Gie98a].

To demonstrate its applicability, we tested our approach on a large benchmark of examples and used it to prove numerous theorems about partial functions with undecidable domains [Gie98b]. Our calculus corresponds to the basic rules used in induction theorem proving. So in this way, the existing induction provers and their heuristics to control the application of these rules can be directly extended to partial functions. Thus, induction theorem proving for partial functions may now become as powerful as it is for total functions.

Acknowledgements. I would like to thank J. Brauburger, D. Kapur, M. Kaufmann, T. Kolbe, N. Shankar, C. Walther, and C.-P. Wirth for helpful comments and fruitful discussions. This work was supported by the DFG under grant Wa 652/2-2.

References


