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, Bu$^+$93, 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* $\mathsf{0}$ and s (where we often write "1" instead of "$\mathsf{s}(\mathsf{0})$" etc.).

$$\begin{array}{ll}
\textit{function}\ \mathsf{minus} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} \\
\quad \mathsf{minus}(x, \mathsf{0}) \ = \ x \\
\quad \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \ = \ \mathsf{minus}(x, y)
\end{array}$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textit{function}\ \mathsf{div} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}} \\
\quad \mathsf{div}(\mathsf{0}, \mathsf{s}(y)) & = & \mathsf{0} \\
\quad \mathsf{div}(\mathsf{s}(x), y) & = & \mathsf{if}(\,\mathsf{ge}(\mathsf{s}(x), y), \\
& & \quad \mathsf{s}(\mathsf{div}(\mathsf{minus}(\mathsf{s}(x), y), y)), \\
& & \quad \mathsf{0})
\end{array}$$

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 τ are only replaced by terms of the same data type τ. 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 $\mathsf{div}(\mathsf{0}, \mathsf{s}(\mathsf{minus}(\mathsf{1}, \mathsf{0})))$, because one argument of div is no constructor ground term. Therefore, $\mathsf{minus}(\mathsf{1}, \mathsf{0})$ has to be evaluated to $\mathsf{1}$ first. Afterwards a defining equation of div can be used to evaluate the resulting term $\mathsf{div}(\mathsf{0}, \mathsf{2})$ to $\mathsf{0}$.

Our programming language has a pre-defined conditional function $\mathsf{if} : \mathsf{bool} \times \tau \times \tau \to \tau$ for each data type τ (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 $\mathsf{if}(t_1, t_2, t_3)$, 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 minus and div compute *partial* functions. The defining equations of minus do not cover all possible inputs, i.e. the algorithm minus is *incomplete* and hence, $\mathsf{minus}(x, y)$ is only defined if x is not smaller than y. The algorithm div for truncated division uses a (total) auxiliary function 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 *non-terminating* evaluation (e.g. $\mathsf{div}(1, 0)$). In fact, $\mathsf{div}(x, y)$ is only defined if y is not $\mathsf{0}$. In general, we say that (evaluation of) a ground term is *defined*, if it can be evaluated to a constructor ground term.

3 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 $\mathsf{div}(n, m)$ with the divisor m yields a number $\leq n$ whenever $\mathsf{div}(n, m)$ is defined.

$$\forall n, m : \mathsf{nat} \quad \mathsf{def}(\mathsf{div}(n, m)) = \mathsf{true} \;\;\rightarrow\;\; \mathsf{ge}(n, \mathsf{times}(m, \mathsf{div}(n, m))) = \mathsf{true} \qquad (1)$$

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

We only consider universally closed formulas of the form $\forall ... \varphi$ where φ is quantifier free and we often omit the quantifiers to ease readability. So for example, "$\varphi_1 \rightarrow \varphi_2$" is an abbreviation for "$\forall ... (\varphi_1 \rightarrow \varphi_2)$", where φ_1 and φ_2 are quantifier free. We sometimes write $\varphi(x^*)$ to indicate that φ 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 φ by the terms t^*.

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

For a formal definition of *truth* for statements about partial functions, we use a model theoretic approach. For *total* functions, the notion of inductive truth generally used in the literature is equivalent to validity in the initial model of the defining equations Eq, cf. e.g. [ZKK88, Wal94, WG94, BR95]. However, due to the occurrence of partial functions, now the initial model of Eq is no longer the specific intended model. The reason is that the defining equations do not represent the *eager* evaluation strategy of our programming language. For example, $\mathsf{div}(\mathsf{0}, \mathsf{div}(1, \mathsf{0})) = \mathsf{0}$ is valid in the initial model of the defining equations[2] although (innermost) evaluation of $\mathsf{div}(\mathsf{0}, \mathsf{div}(1, \mathsf{0}))$ 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 *defined*, i.e. if $\mathsf{def}(\sigma(t))$ is true. Thus, instead of a defining equation $f(t) = r$ we use the equation $f(t) =$

[2] when extended with the equations $\mathsf{if}(\mathsf{true}, x, y) = x$ and $\mathsf{if}(\mathsf{false}, x, y) = y$

$\mathsf{if}(\mathsf{def}(t), r, f(t))$. To handle functions with several arguments, in the following let $\mathsf{def}(t_1, \ldots, t_n)$ be an abbreviation for the term $\mathsf{if}(\mathsf{def}(t_1), \mathsf{def}(t_2, \ldots, t_n), \mathsf{false})$. So intuitively, $\mathsf{def}(t_1, \ldots, t_n)$ is true iff $\mathsf{def}(t_i)$ is true for all i. For the *empty* tuple (where $n = 0$), $\mathsf{def}()$ 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*

$$\begin{aligned} &\{f(t^*) = \mathsf{if}(\mathsf{def}(t^*), r, f(t^*)) \mid \textit{for each defining equation } f(t^*) = r\} \\ \cup\ &\{\mathsf{if}(\mathsf{true}, x, y) = x,\ \mathsf{if}(\mathsf{false}, x, y) = y\} \\ \cup\ &\{\mathsf{def}(c(x^*)) = \mathsf{def}(x^*) \mid \textit{for each constructor } c\}. \end{aligned}$$

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 $\mathsf{def}(t) = \mathsf{true}$ is inductively true iff evaluation of t is defined. To verify *partial correctness* of an algorithm w.r.t. a specification φ, one has to prove the conjecture

$$\mathsf{def}(t^*) = \mathsf{true} \ \rightarrow\ \varphi,$$

where t^* are the (top-level) terms of φ. Thus, an algorithm is partially correct w.r.t. φ, if φ 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 $\mathsf{0}$ and $\mathsf{s}(y)$ and by $\mathsf{s}(x)$ and y, respectively). In the recursive equation of div we perform another case analysis w.r.t. the condition $\mathsf{ge}(\mathsf{s}(x), y)$ of the if-term. In the case $\mathsf{ge}(\mathsf{s}(x), y) = \mathsf{true}$ we assume that (1) already holds for the arguments $\langle \mathsf{minus}(\mathsf{s}(x), y),\ y\rangle$ 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).

$$\varphi(\underline{\mathsf{0}}, \underline{\mathsf{s}(y)}) \tag{2}$$

$$\mathsf{ge}(\mathsf{s}(x), y) = \mathsf{false} \ \rightarrow\ \varphi(\underline{\mathsf{s}(x)}, \underline{y}) \tag{3}$$

$$\mathsf{ge}(\mathsf{s}(x), y) = \mathsf{true} \ \rightarrow\ (\,\varphi(\underline{\mathsf{minus}(\mathsf{s}(x), y)}, \underline{y}) \ \rightarrow\ \varphi(\underline{\mathsf{s}(x)}, \underline{y})\,) \tag{4}$$

The technique of performing inductions w.r.t. the recursions of algorithms (like div) is commonly used in induction theorem proving, cf. e.g. [BM79, ZKK88, Bun89, Wal94]. 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 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 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 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 "$\mathsf{def}(\mathsf{div}(n, m)) = \mathsf{true} \;\rightarrow\; \varphi'(n, m)$". Thus, there must be a counterexample, i.e. two numbers p and q such that $\mathsf{div}(p, q)$ is defined, but $\varphi'(p, q)$ is false.

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

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

$$\neg\mathsf{def}(f(x^*)) = \mathsf{true} \;\rightarrow\; \varphi(x^*).$$

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.

$$\neg\mathsf{def}(\mathsf{div}(n, m)) = \mathsf{true} \;\rightarrow\; (1)$$

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

$$\neg\mathsf{def}(x) = \mathsf{true} \to \varphi(x).$$

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 $\mathsf{def}(\sigma(t^*)) = \mathsf{true}$ holds. Hence, when evaluating the term $\sigma(f(t^*))$ in a formula φ, one also has to prove the permissibility conjecture

$$\neg\mathsf{def}(\sigma(t^*)) = \mathsf{true} \to \varphi.$$

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

$$\neg\mathsf{def}(\mathsf{0}, \mathsf{minus}(\ldots)) = \mathsf{true} \;\to\; \varphi(\mathsf{div}(\mathsf{0}, \mathsf{minus}(\ldots))).$$

Of course, as if is the only function symbol with non-eager semantics, to evaluate a term $\mathsf{if}(t_1, t_2, t_3)$ 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 Ax^{def} about definedness, which state how def operates on terms built with algorithms, conditionals, and constructors.

$$\begin{aligned} Ax^{\mathrm{def}} = {} & \{\mathsf{def}(f(x^*)) = \mathsf{true} \;\to\; \mathsf{def}(x^*) = \mathsf{true} \mid \text{for all algorithms } f\} \\ & \cup \{\mathsf{def}(\mathsf{if}(x, y, z)) = \mathsf{true} \;\to\; \mathsf{def}(x) = \mathsf{true}\} \\ & \cup \{\mathsf{def}(c(x^*)) = \mathsf{def}(x^*) \mid \text{for all constructors } c\}. \end{aligned}$$

In this way, the conjecture (1) about 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 def, the axioms Ax^{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 "φ 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 div one can prove that div is always undefined if its second argument is $\mathsf{0}$, i.e.

$$y = \mathsf{0} \;\to\; \neg\mathsf{def}(\mathsf{div}(x, y)) = \mathsf{true}.$$

A refinement of our approach is obtained by combining it with techniques to approximate the *domains* of partial functions. More precisely, for every algorithm $f : \tau_1 \times \ldots \times \tau_n \to \tau$, a (total) algorithm $\theta_f : \tau_1 \times \ldots \times \tau_n \to \mathsf{bool}$ (a *domain predicate* for f) is generated, such that the truth of $\theta_f(t^*)$ implies that evaluation of $f(t^*)$ is defined. Thus, θ_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 Ax^{dom}. For every algorithm f, Ax^{dom} contains the axioms

$$\theta_f(x^*) = \mathsf{true} \to \mathsf{def}(f(x^*)) = \mathsf{true}$$
$$\mathsf{def}(x^*) = \mathsf{true} \to \mathsf{def}(\theta_f(x^*)) = \mathsf{true}$$

which state that the truth of θ_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 θ_f automatically, together with *J. Brauburger* 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

$$\mathsf{minus}(\mathsf{minus}(x, y), z) = \mathsf{minus}(\mathsf{minus}(x, z), y)$$

holds whenever its evaluation is defined. This formula can be proved by induction w.r.t. the partial function minus using x and y as induction variables. However, the technique of [Gie96] does not allow this induction, because $\mathsf{minus}(x, y)$ 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 θ_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 $\mathsf{def}(\mathsf{div}(x, y)) = \mathsf{true} \to \theta_{\mathsf{div}}(x, y) = \mathsf{true}$ can be verified by induction w.r.t. 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[3]. 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′"]. This corresponds to the intuition that such

[3] 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 $f(x) = \mathsf{s}(f(x))$ 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, AM95].

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 [Bu$^+$93], 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, Sli97]. 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[4].

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].

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

[AM95] Avenhaus, J. and Madlener, K., Theorem Proving in Hierarchical Clausal Specifications, in *Advances in Algorithms, Languages, and Complexity* (eds. Du, Ko), Kluwer Academic Publishers, 1997.

[BR95] Bouhoula, A. and Rusinowitch, M., Implicit Induction in Conditional Theories, *Journal of Automated Reasoning* **14**, 189-235, 1995.

[Bou93] Boulton, R. J., Boyer-Moore Automation for the HOL System, *6th Int. Workshop HOL Th. Pr. App.*, Vancouver, Canada, Elsevier, 1993.

[BM79] Boyer, R. S. and Moore, J S., *A Computational Logic*, Academic Press, 1979.

[BK84] Boyer, R. S. and Kaufmann, M., On the Feasibility of Mechanically Verifying SASL Programs, Tech. Rep. ARC 84-16, Burroughs Research Center, 1984.

[BM88] Boyer, R. S. and Moore, J S., The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover, *Journal of Automated Reasoning* **4**, 117-172, 1988.

[BG96] Brauburger, J. and Giesl, J., Termination Analysis for Partial Functions, *Proc. SAS '96*, Aachen, Germany, LNCS 1145, 1996.

[BG98] Brauburger, J. and Giesl, J., Termination Analysis by Inductive Evaluation, *Proc. CADE-15*, Lindau, Germany, LNAI, 1998.

[Bun89] Bundy, A., A Rational Reconstruction and Extension of Recursion Analysis, *Proc. IJCAI '89*, Detroit, MI, Morgan Kaufmann, 1989.

[Bu+93] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., and Smaill, A., Rippling: A Heuristic for Guiding Inductive Proofs, *AI* **62**, 185-253, 1993.

[Bus93] Busch, H., Unification-Based Induction, *Proc. 6th Int. Workshop Higher Order Logic Theorem Proving Appl.*, Vancouver, Canada, Elsevier, 1993.

[Far90] Farmer, W. M., A Partial Function's Version of Church's Simple Theory of Types, *Journal of Symbolic Logic* **55**, 1269-1291, 1990.

[Gie96] Giesl, J., Proving Partial Correctness of Partial Functions, *Proc. CADE-Workshop Mechanization of Partial Functions*, New Brunswick, NJ, 1996.

[Gie97] Giesl, J., Termination of Nested and Mutually Recursive Algorithms, *Journal of Automated Reasoning* **19**, 1-29, 1997.

[Gie98a] Giesl, J., Induction Proofs with Partial Functions, Technical Report IBN 98/48, TU Darmstadt, 1998.

[Gie98b] Giesl, J., The Critical Pair Lemma: A Case Study for Induction Proofs with Partial Functions, Technical Report IBN 98/49, TU Darmstadt, 1998.

[GWB98] Giesl, J., Walther, C., and Brauburger, J., Termination Analysis for Functional Programs, in *Automated Deduction – A Basis for Applications, Vol. 3* (eds. W. Bibel and P. Schmitt), Kluwer Academic Publishers, 1998.

[HS96] Hutter, D. and Sengler, C., INKA: The Next Generation, *Proc. CADE-13*, New Brunswick, NJ, LNAI 1104, 1996.

[KM86] Kapur, D. and Musser, D. R., Inductive Reasoning with Incomplete Specifications, *Proc. LICS '86*, 1986.

[KM87] Kapur, D. and Musser, D. R., Proof by Consistency, *Artificial Intelligence* **31**, 125-157, 1987.

[KS96] Kapur, D. and Subramaniam, M., New Uses of Linear Arithmetic in Automated Theorem Proving by Induction, *J. Aut. Reasoning* **16**, 39-78, 1996.

[Kap97] Kapur, D., Constructors can be Partial, too, *Automated Reasoning and Its Applications – Essays in Honor of L. Wos* (ed. R. Veroff), MIT Press, 1997.

[KK95] Kerber, M. and Kohlhase, M., A Tableau Calculus for Partial Functions, *Collegium Logicum – Annals of the Kurt Gödel-Society* **2**, 21-49, 1996.

[Kle52] Kleene, S. C., *Introduction to Metamathematics*, Van Nostrand, 1952.

[KB70] Knuth, D. E. and Bendix, P. B., Simple Word Problems in Universal Algebras, in *Computational Problems in Abstract Algebra* (ed. J. Leech), Pergamon Press, Oxford, 1970.

[Kre87] Kreowski, H.-J., Partial Algebras flow from Algebraic Specifications, *Proc. ICALP '87*, Karlsruhe, Germany, LNCS 267, 1987.

[LS87] Loeckx, J. and Sieber, K., *The Foundations of Program Verification*, Wiley-Teubner, 1987.

[Man74] Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[MW81] Manna, Z. and Waldinger, R., Deductive Synthesis of the Unification Algorithm, *Science of Computer Programming* **1**, 5-48, 1981.

[Nip96] Nipkow, T., More Church-Rosser Proofs (in ISABELLE/HOL), *Proc. CADE-13*, New Brunswick, NJ, LNAI 1104, 1996.

[Pau85] Paulson, L. C., Verifying the Unification Algorithm in LCF, *Science of Computer Programming* **5**, 143-169, 1985.

[Pau87] Paulson, L. C., *Logic and Computation*, Cambridge University Press, 1987.

[Rei87] Reichel, H., *Initial Computability, Algebraic Specifications and Partial Algebras*, Oxford University Press, 1987.

[Sco69] Scott, D. S., A Type-Theoretic Alternative to CUCH, ISWIM, PWHY, Notes, Oxford (1969). Annotated version in *Theor. Comp. Sc.* **121**, 411-440, 1993.

[Sha88] Shankar, N., A Mechanical Proof of the Church-Rosser Theorem, *Journal of the ACM* **35**, 475-522, 1988.

[Sha89] Shankar, N., A Logical Basis for Functional Programming, Draft, Stanford University, 1989.

[Sli97] Slind, K., Derivation and Use of Induction Schemes in Higher-Order Logic, *Proc. 10th Int. Conf. on Theorem Proving in Higher Order Logics*, Murray Hill, NJ, LNCS 1275, 1997.

[Wal94] Walther, C., Mathematical Induction, in *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 2* (eds. D. M. Gabbay, C. J. Hogger, and J. A. Robinson), Oxford University Press, 1994.

[WG94] Wirth, C.-P. and Gramlich, B., On Notions of Inductive Validity for First-Order Equational Clauses, *Proc. CADE-12*, Nancy, France, LNAI 814, 1994.

[ZKK88] Zhang, H., Kapur, D., and Krishnamoorthy, M. S., A Mechanizable Induction Principle for Equational Specifications, *Proc. CADE-9*, LNCS 310, 1988.