Proving Partial Correctness of Partial Functions*

Jürgen Giesl

FB Informatik, TH Darmstadt, Alexanderstr. 10, 64283 Darmstadt, Germany,
E-mail: giesl@inferenzsysteme.informatik.th-darmstadt.de

Abstract. We present a method for automated induction proofs about partial functions. This method cannot only be used to verify the partial correctness of functional programs, but it also solves some other challenge problems where reasoning about partial functions is necessary. For a further analysis of partial functions we also developed a method to determine (non-trivial subsets of) their domains automatically.

1 Introduction

Induction is the essential proof method for the verification of functional programs. For that reason, several techniques[1] have been developed to perform induction proofs automatically, cf. e.g. [BM79, Bu+93, Wa94a]. However, most of these techniques are only sound if all occurring functions are total.

In this paper we show that by slightly restricting the prerequisites of these techniques it is nevertheless possible to use them for partial functions, too. In particular, the successful proof technique of performing *inductions w.r.t. algorithms* can also be applied for partial functions, i.e. (under certain conditions) one may even perform inductions w.r.t. non-terminating algorithms.

Hence, with our approach the well-known techniques for automated induction proofs can be *directly* extended to partial functions (i.e. we do not follow the suggestion of [BM88] to treat partial functions only as inputs to an interpreter function). Previous suggestions for the mechanization of partial functions either did not focus on induction [KK94] or they could not deal with non-terminating algorithms [WG94, Wa94a].

2 Partial Correctness

In this section we introduce the notion of *partial correctness* used in the paper. We regard an eager first order functional language with (non-parameterized and free) algebraic data types and pattern matching (where the patterns must be exclusive). As an example consider the algorithms minus and quot. They operate

* To appear in the *Proceedings of the Workshop on the Mechanization of Partial Functions, held in conjunction with the 13th International Conference on Automated Deduction*, New Brunswick, NJ, USA, 1996.

This work was supported by the Deutsche Forschungsgemeinschaft under grant no. Wa 652/7-1 as part of the focus program "Deduktion".

[1] In fact there are two research paradigms in the automation of induction proofs, viz. *implicit* and *explicit* induction, where we will only focus on the latter one.

on the algebraic data type nat for natural numbers whose objects are built with the *constructors* 0 and s (where we sometimes write "1" instead of "s(0)" etc.).

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

Obviously, both algorithms minus and quot compute *partial* functions. The reason is that the defining equations of minus do not cover all possible inputs, i.e. the algorithm minus is *incomplete* and hence, the result of $\mathsf{minus}(x, y)$ is only defined if the number x is not smaller than the number y. The algorithm quot is not only incomplete, but there are also inputs which lead to a *non-terminating* evaluation (e.g. $\mathsf{quot}(1, 0)$). Hence, the result of $\mathsf{quot}(x, y)$ is only defined if the number y is a divisor of the number x (and $y \neq 0$). So if we want to "verify" programs like minus and quot which compute *partial* functions we can at most verify their *partial correctness*. For instance, suppose that the specifications for minus and quot are

$$\forall n, m : \mathsf{nat} \quad \mathsf{plus}(m, \mathsf{minus}(n, m)) = n, \tag{1}$$

$$\forall n, m : \mathsf{nat} \quad \mathsf{times}(m, \mathsf{quot}(n, m)) = n, \tag{2}$$

where plus and times are defined by the obvious algorithms. Then minus and quot are in fact *partially correct* w.r.t. these specifications. So for quot we have

for all natural numbers n and m: if evaluation of $\mathsf{quot}(n, m)$ is defined, then $\mathsf{times}(m, \mathsf{quot}(n, m)) = n$.

In this paper we only regard universally closed formulas of the form $\forall \ldots \varphi$ where φ is quantifier free and we often omit the quantifiers to ease readability. We sometimes write $\varphi(x^*)$ to indicate that φ contains (at least) the variables x^* (where x^* abbreviates 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^*. We say that a formula $\forall x^*\ \varphi(x^*)$ is *partially correct*, if $\varphi(t^*)$ is true for all those data objects t^* where evaluation of all terms in $\varphi(t^*)$ is defined.

While this notion of partial correctness is widely used in program verification [LS87] several other definitions for "correctness" of statements about partial functions have been suggested in the literature, cf. e.g. [KK94].

Methods to prove the partial correctness of partial functions are not only essential for the verification of functional programs, but they are also necessary to solve some further challenge problems in automated deduction:

2.1 Termination of Nested and Mutually Recursive Algorithms

In the area of *automated termination analysis*, termination proofs for algorithms with *nested* or *mutual recursion* are regarded as one of the main challenge problems. The reason is that if an algorithm f has nested recursion, then f's *own semantics* have to be considered in its termination proof (and a similar problem occurs with mutual recursion).

To prove the termination of a functional program f there has to be a well-founded ordering $\succ$ such that the arguments in each recursive call are smaller than the corresponding inputs. Hence, if evaluation of $f(t)$ leads to a (nested) recursive call $f(f(r))$, then we have to show that both the argument r of the *inner* recursive call *and* the argument $f(r)$ of the *outer* recursive call are smaller than the corresponding input t, i.e. we have to prove $t \succ r$ and $t \succ f(r)$. But the statement $t \succ f(r)$ contains the function f which may possibly be partial (as we have not yet verified the termination of its algorithm). For that reason previously developed methods for automated termination proofs of functional programs usually failed for algorithms with nested recursion [BM79, Wa94b, Gie95].

However, using the techniques to be presented in Section 3, it will be possible to verify *partial* correctness of statements like $t \succ f(r)$. Note that (surprisingly), *partial* correctness of these statements is already sufficient for the termination of the algorithm f. Hence, a method for partial correctness proofs allows us to prove termination of algorithms with nested or mutual recursion without having to prove the correctness of the algorithms simultaneously. This enables automated termination proofs for well-known challenge problems such as *J. McCarthy*'s f_91 function. For a detailed description of these results see [Gie96a].

2.2 Reasoning about Imperative Programs

Although imperative languages are almost exclusively used in practice, up to now most systems for automated induction proofs are restricted to the verification of *functional* languages.

Therefore one attempt for automated reasoning about imperative programs is to translate imperative programs into functional programs. In this translation every *while*-loop is transformed into a separate function [Hen80]. But note that in general these functions are *partial*, because in imperative programs, termination of *while*-loops often depends on their contexts (i.e. on the preconditions that hold before entering a *while*-loop). Hence, to apply existing systems for automated program verification to imperative programs, one needs a method to prove partial correctness of statements involving partial functions.

3 Induction Proofs with Partial Functions

After having illustrated why one is interested in partial correctness, in this section we will sketch a method for proving partial correctness automatically. For the partial correctness of a formula $\varphi(x^*)$ we have to verify infinitely many instantiations $\varphi(t^*)$. As data types are constructed inductively, this can often be reduced to a finite proof by using induction.

Several techniques have been developed for the automation of induction proofs. But unfortunately, statements "proved" with these techniques are only correct provided that all occurring functions are total. However, in the following we will show that by slightly restricting the application of these techniques one in fact obtains a sound calculus for induction proofs with partial functions. A more detailed description of our calculus can be found in [Gie96b].

3.1 Induction w.r.t. Algorithms

One of the key ideas in automated induction theorem proving is to perform inductions *w.r.t. the recursions of the algorithms*. For example, as (2) contains a call of the function quot, this call suggests a plausible induction, i.e. we use an induction w.r.t. the algorithm quot and choose the variables n and m as induction variables. For that purpose one performs a case analysis w.r.t. the cases of quot and in its recursive case one can assume that (2) already holds for the arguments of quot's recursive call. So instead of (2) one has to prove the following formulas where we have underlined instantiations of the induction variables n and m to ease readability.

$$\mathsf{times}(\underline{\mathsf{s}(y)}, \mathsf{quot}(\underline{0}, \underline{\mathsf{s}(y)})) = \underline{0}, \tag{3}$$

$$\mathsf{times}(\underline{y}, \mathsf{quot}(\underline{\mathsf{minus}(\mathsf{s}(x), y)}, \underline{y})) = \underline{\mathsf{minus}(\mathsf{s}(x), y)} \rightarrow \mathsf{times}(\underline{y}, \mathsf{quot}(\underline{\mathsf{s}(x)}, \underline{y})) = \underline{\mathsf{s}(x)}. \tag{4}$$

But induction proofs are only *sound* if the induction relation used is *well founded*. Here, the well-foundedness of the induction relation corresponds to the termination of the algorithm quot. So in general, by inductions w.r.t. non-terminating algorithms like quot one can easily "prove" false facts. For example, by induction w.r.t. the algorithm f with the defining equation $f(x) = f(x)$ one can prove formulas like $\neg x = x$ which are not partially correct.

However, for formula (2) the induction w.r.t. the recursions of quot is nevertheless sound, i.e. partial correctness of (3) and (4) in fact implies partial correctness of (2). The reason is that the only occurrence of a partial function in (2) is the term $\mathsf{quot}(n, m)$. Hence, for all natural numbers n and m, evaluation of "$\mathsf{times}(m, \mathsf{quot}(n, m)) = n$" is defined iff evaluation of "$\mathsf{quot}(n, m)$" is defined.

Partial correctness of (3) and (4) implies that "$\mathsf{times}(m, \mathsf{quot}(n, m)) = n$" holds for all numbers n and m where $\mathsf{quot}(n, m)$ is defined, provided that it also holds for those numbers n' and m', where evaluation of $\mathsf{quot}(n, m)$ leads to the recursive call $\mathsf{quot}(n', m')$. Hence, the original induction proof w.r.t. the recursions of quot can be regarded as an induction proof where the induction relation is restricted to those inputs where evaluation of quot is defined. As this restricted induction relation is *well founded* (although quot is not always terminating), the partial correctness of (3) and (4) is indeed sufficient for the partial correctness of (2).

Therefore by restricting the prerequisites of the technique for "inductions w.r.t. algorithms", this technique can also be applied to perform inductions w.r.t. partial functions like quot: In the proof of $\varphi(x^*)$ one may perform an induction w.r.t. the partial function f using x^* as induction variables, if $\varphi(x^*)$ contains the subterm $f(x^*)$ and if $\varphi(x^*)$ does not contain any other occurrences of partial functions.

3.2 Using Defining Equations of Algorithms

Another important technique often used in induction proofs is *symbolic evaluation*, i.e. the defining equations of an algorithm are used as rewrite rules. For instance, by symbolic evaluation of quot and times, (3) can be transformed into

the tautology $0 = 0$. Note that, while the defining equations of partial functions may indeed be used for symbolic evaluation, they must not be used as ordinary axioms. The reason is that defining equations of non-terminating algorithms may be inconsistent with the axioms for the data types used. For example, consider a theory where $\neg x = \mathsf{s}(x)$ holds and let f have the defining equation $f(y) = \mathsf{s}(f(y))$. Together with the axiom $\neg x = \mathsf{s}(x)$, this defining equation is inconsistent. Hence, if the defining equations of non-terminating algorithms were to be used as ordinary axioms, one could prove anything (e.g. FALSE).

3.3 Other Inference Steps

In automated induction theorem proving one applies rules of the form[2] $\frac{\psi}{\varphi}$ in backwards direction. "Soundness" of these rules guarantees that φ holds for all data objects, provided that ψ holds for all data objects.

However, in general these rules are no longer sound when considering partial functions. For example, $\frac{\varphi_1 \wedge \varphi_2}{\varphi_1}$ is a sound rule for total functions, but it becomes unsound when handling partial functions. The reason is that φ_1 could be FALSE and φ_2 could contain an undefined term like $\mathsf{quot}(1, 0)$.

Therefore a rule $\frac{\psi}{\varphi}$ may only be used in partial correctness proofs, if "definedness" of φ implies "definedness" of the corresponding instantiation of ψ. Assume that for each formula φ we know a *definition formula* (which we denote by $\varphi\!\downarrow$) such that φ and $\varphi\!\downarrow$ contain the same variables x^* and such that evaluation of $\varphi(t^*)$ is defined iff $\varphi\!\downarrow\,(t^*)$ is true. Then a rule $\frac{\psi}{\varphi}$ may only be applied if for all data objects t^* there exist data objects s^* such that

$$\psi(s^*) \rightarrow \varphi(t^*) \text{ and} \tag{5}$$

$$\varphi\!\downarrow\,(t^*) \rightarrow \psi\!\downarrow\,(s^*). \tag{6}$$

For certain rules (e.g. *symbolic evaluation* or *instantiation* $\frac{\varphi}{\sigma(\varphi)}$) both these conditions are always fulfilled. But for other rules in automated induction theorem proving, one has to check these conditions in each rule application.

One method to check condition (5) is to test whether $\psi(t^*) \rightarrow \varphi(t^*)$ holds (i.e. to choose $s^* = t^*$) and to check condition (6) one could examine whether every term with a partial root function in ψ also occurs in φ.

4 Termination Analysis for Partial Functions

The techniques presented in Section 3 allow us to prove partial correctness of statements like (1) and (2) automatically by performing inductions w.r.t. partial functions as sketched in Section 3.1. Moreover, these techniques are also sufficient for the partial correctness proofs needed for termination analysis of nested and mutually recursive functions, cf. Section 2.1.

However, for certain proofs one really needs to generate *definition formulas* $\varphi\!\downarrow$ to check condition (6). In other words, one has to determine the domains of partial functions. For that purpose, together with *J. Brauburger* we have developed a method to synthesize a *termination predicate* algorithm θ_f for each

[2] Corresponding statements hold for rules $\frac{\psi_1,\ldots,\psi_k}{\varphi}$ with several premises.

functional program f, i.e. θ_f computes a total function which only returns TRUE for inputs where the original program is terminating.

As we want to generate termination predicates automatically, we can only demand that a termination predicate θ_f represents a *sufficient* criterion for the termination of f's algorithm. But when testing our method with numerous examples we found that it is often able to synthesize termination predicates which describe the *whole* domain of a function. For instance, for minus our method synthesizes the termination predicate "greater-equal" and for quot it synthesizes the algorithm divides. For details on our work on termination analysis for partial functions see [BG96].

5 Conclusion

We presented a method to extend the existing techniques for automated induction proofs to partial functions. In this way, partial correctness of partial functional programs can be proved automatically and moreover, our result can also be used for the verification of imperative programs and for termination proofs of nested and mutually recursive algorithms. For further automated reasoning we have also developed a method for termination analysis of partial functions.

References

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

[BM88] R. S. Boyer & J S. Moore. 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] J. Brauburger & J. Giesl. Termination Analysis for Partial Functions. In *Proc. 3rd International Static Analysis Symposium*, Aachen, Germany, LNCS, 1996.

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

[Gie95] J. Giesl. Termination Analysis for Functional Programs using Term Orderings. *Pr. 2nd Int. Static Analysis Symp.*, Glasgow, Scotland, LNCS 983, 1995.

[Gie96a] J. Giesl. Termination of Nested and Mutually Recursive Algorithms. *Journal of Automated Reasoning.* To appear.

[Gie96b] J. Giesl. Induction Proofs with Partial Functions. Technical Report IBN 96/35, Technische Hochschule Darmstadt, Germany, 1996.

[Hen80] P. Henderson. *Functional Programming.* Prentice-Hall, London, 1980.

[KK94] M. Kerber & M. Kohlhase, A Mechanization of Strong Kleene Logic for Partial Functions. In *Proc. 12th CADE*, Nancy, France, LNAI 814, 1994.

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

[Wa94a] C. Walther. Mathematical Induction. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 2, Oxford University Press, 1994.

[Wa94b] C. Walther. On Proving the Termination of Algorithms by Machine. *Artificial Intelligence*, 71(1):101-157, 1994.

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