Induction Proofs with Partial Functions*

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

Abstract. In this paper we present a method for automated induction proofs about partial functions. We show that 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 calculus 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 major changes of their logical framework.

Key words: induction, automated theorem proving, partial functions

1. Introduction

Induction is the essential proof method for the verification of functional programs. For that reason, several techniques[1] have been developed to compute suitable induction relations and to perform induction proofs automatically, cf. e.g. [6, 17, 42, 75, 79]. 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 Section 2 we present a calculus for induction proofs which consists of the basic rules usually applied in automated induction theorem proving. But unfortunately, this calculus requires all occurring functions to be total. Therefore, by restricting its rules in an appropriate way, in Section 3 we develop a new calculus for induction proofs with

* Technical Report IBN 98/48, TU Darmstadt, Germany. Final version to appear in the *Journal of Automated Reasoning*.

[1] There are two research paradigms for the automation of induction proofs, viz. *explicit* and *implicit* induction (e.g. [4, 40]), where we only focus on the first one.

partial functions. We first regard algorithms defined by unconditional equations only, but in Section 4 we show how to extend our calculus to handle algorithms with conditionals.

While the calculus of Section 3 and 4 is already sufficient for many conjectures, certain proofs require reasoning about the *definedness* of partial functions. For that purpose we introduce a refinement of our calculus in Section 5. For some proofs it is even necessary to compute (or at least to approximate) the *domains* of partial functions. Therefore a method for automatic domain analysis is presented in Section 6.

In Section 7 we discuss some application areas where reasoning about partial functions is required and illustrate the power of our approach with several examples. Finally, we give a detailed comparison with related work in Section 8 and end up with a short conclusion.

2. Automated Induction Theorem Proving

Before dealing with the special problems arising with partial functions, in this section we first sketch the standard approach typically used for automated (explicit) induction proofs. 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 consider the algorithms plus and times. 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.).

function plus : nat × nat → nat	*function* times : nat × nat → nat
$\mathsf{plus}(0, y) = y$	$\mathsf{times}(x, 0) = 0$
$\mathsf{plus}(\mathsf{s}(x), y) = \mathsf{s}(\mathsf{plus}(x, y))$	$\mathsf{times}(x, \mathsf{s}(y)) = \mathsf{plus}(x, \mathsf{times}(x, y))$

In general, an algorithm f is defined by a set of orthogonal[2] 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 do not impose any restrictions on the form of f's recursions, i.e., algorithms may also have nested or mutual recursion.

We always restrict ourselves to well-sorted terms and substitutions, i.e., variables of the data 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

[2] A set of equations is called *orthogonal*, if it is non-overlapping (i.e., there are no critical pairs) and left-linear (i.e., left-hand sides may not contain multiple occurrences of the same variable).

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, the first defining equation of times cannot be applied directly to evaluate the term $\mathsf{times}(\mathsf{plus}(\mathsf{0},\mathsf{1}),\mathsf{0})$, because one argument of times is not a constructor ground term. Therefore, the argument $\mathsf{plus}(\mathsf{0},\mathsf{1})$ has to be evaluated to $\mathsf{1}$ first. Afterwards a defining equation of times can be used to evaluate the resulting term $\mathsf{times}(\mathsf{1},\mathsf{0})$ to $\mathsf{0}$.

For a formal definition, let R^{op} be the (infinite) term rewriting system with the rules $\sigma(s_1) \to \sigma(s_2)$ for every defining equation $s_1 = s_2$ and for every substitution σ which instantiates all variables of s_1 with constructor ground terms. Then we say that a ground term t *evaluates* to t' iff $t \to^*_{R^{\mathrm{op}}} t'$ holds. Note that R^{op} is orthogonal and hence, confluent [34] (where in fact, R^{op}'s confluence already follows from innermost confluence, and thus, from the fact that the rules are non-overlapping). Thus, every ground term can evaluate to at most one constructor ground term (i.e., all our algorithms are deterministic).

In this section we restrict ourselves to algorithms that are *terminating* and *completely defined* (i.e., the patterns have to be exhaustive). In other words, the corresponding term rewriting system R^{op} terminates and every non-constructor ground term is R^{op}-reducible. Due to the special form of R^{op}, this is equivalent to *sufficient completeness* [33] (i.e., to the requirement that for every ground term t there exists a constructor ground term q with $t \leftrightarrow^*_{R^{\mathrm{op}}} q$). As every ground term evaluates to a (unique) constructor ground term, all algorithms compute total functions. Now our goal is to verify statements concerning a given collection of algorithms and data types. For instance, we may try to verify the associativity of plus (where we wrote "+" instead of plus).

$$\forall u, v, w : \mathsf{nat} \quad u + (v + w) = (u + v) + w \tag{1}$$

In this paper we only consider universally closed formulas of the form $\forall \ldots \varphi$ where φ is quantifier-free and we often omit the quantifiers to ease readability. So for example, "$\varphi_1 \Rightarrow \varphi_2$" always is an abbreviation for "$\forall \ldots (\varphi_1 \Rightarrow \varphi_2)$", 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 φ holds for all instantiations of x^* with data objects q^*. For example, formula (1) is true, because for all natural numbers u, v, and w, $\mathsf{plus}(\mathsf{plus}(u, v), w)$ and $\mathsf{plus}(u, \mathsf{plus}(v, w))$ evaluate to the same number. In the following we will often speak of "truth" instead of "inductive truth".

More precisely, $\forall x^*\ \varphi(x^*)$ is *true*, if for all data objects q^* we have $Eq \cup Ax^{\mathrm{data}} \models \varphi(q^*)$. Here, "$\models$" denotes first order consequence and

Eq is the set of all defining equations of the algorithms. So for plus, Eq contains the equations $\mathsf{plus}(\mathsf{0}, y) = y$ and $\mathsf{plus}(\mathsf{s}(x), y) = \mathsf{s}(\mathsf{plus}(x, y))$. In order to prove non-atomic formulas φ, we need additional axioms Ax^{data} which guarantee that different constructor ground terms represent different objects. For that purpose Ax^{data} states that constructors are injective and that terms built with different constructors are not equal, cf. [75]. Hence, for each constructor c, Ax^{data} contains the axiom

$$c(x_1, \ldots, x_n) = c(y_1, \ldots, y_n) \Rightarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n.$$

Moreover, if c_1 and c_2 are different constructors of the same data type, then Ax^{data} also contains the axiom

$$\neg c_1(x_1, \ldots, x_n) = c_2(y_1, \ldots, y_m).$$

So for nat, we obtain the axioms $\mathsf{s}(x) = \mathsf{s}(y) \Rightarrow x = y$ and $\neg \mathsf{0} = \mathsf{s}(x)$.

Our definition of "truth" is equivalent to validity in the initial model of the defining equations Eq, i.e., it corresponds to the notion of inductive truth generally used in the literature, cf. e.g. [4, 30, 75, 77, 79].

So for the truth of a formula $\varphi(x^*)$ we have to verify infinitely many instantiations $\varphi(q^*)$. But as data types are constructed inductively, this can often be reduced to a finite proof by using induction.

Several techniques have been developed to perform induction proofs automatically. In the following we present a calculus for induction proofs to give a precise and compact formalization of the basic techniques usually applied in induction theorem proving. As will be shown in Section 3, this formalization is especially suitable for an extension of induction theorem proving to partial functions. Of course, the calculus can also be refined by additional rules (e.g., rules for the use of more sophisticated induction relations), cf. Section 5.

As (1) contains calls of the function plus, these calls suggest plausible inductions. For instance, we can apply an *induction w.r.t. the recursions of the algorithm* plus and use the variables u and v as *induction variables*. For that purpose we perform a case analysis according to the defining equations of plus (i.e., u and v are instantiated by $\mathsf{0}$ and y and by $\mathsf{s}(x)$ and y, respectively). In the recursive case of plus we assume that (1) already holds for the arguments x, y of plus' recursive call. So instead of (1) it is sufficient to prove the following formulas where we underlined instantiations of the induction variables.

$$\underline{\mathsf{0}} + (\underline{y} + w) = (\underline{\mathsf{0}} + \underline{y}) + w \qquad (2)$$

$$\underline{x} + (\underline{y} + w) = (\underline{x} + \underline{y}) + w \Rightarrow \underline{\mathsf{s}(x)} + (\underline{y} + w) = (\underline{\mathsf{s}(x)} + \underline{y}) + w \qquad (3)$$

In general, the following rule is used for inductions w.r.t. algorithms (where rules of the calculus have to be applied in backwards direction).

1. Induction w.r.t. Algorithms

$$\frac{\{\varphi(s^*_{i,1}) \wedge \ldots \wedge \varphi(s^*_{i,n_i}) \Rightarrow \varphi(t^*_i) \mid i = 1, \ldots, k\}}{\varphi(x^*)}$$

if x^* are variables of the appropriate data types (the *induction variables*) and if f is an algorithm with the defining equations $f(t^*_i) = r_i$ $(i = 1, \ldots, k)$, where r_i contains the f-terms $f(s^*_{i,1}), \ldots, f(s^*_{i,n_i})$.

In this rule, we always assume that apart from x^*, the patterns t^*_i contain no variables from φ (otherwise the variables have to be renamed).

The technique of performing inductions w.r.t. the recursions of algorithms (like plus) is commonly applied in induction theorem proving, cf. e.g. [6, 15, 75, 79]. However, induction proofs are only sound if the induction relation used is well founded (i.e., if there exists 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 plus, because when proving a statement for the inputs of a recursive defining equation, we assume as induction hypothesis that the statement already holds for the arguments of the recursive call. Hence, one may already guess that Rule 1 leads to problems when dealing with partial functions.

Apart from inductions w.r.t. algorithms there is also a rule for *structural* inductions according to the definitions of data types. So for $\forall x : \mathsf{nat}\ \varphi(x)$ it is sufficient to prove $\varphi(\mathsf{0})$ and $\forall x : \mathsf{nat}\ \ \varphi(x) \Rightarrow \varphi(\mathsf{s}(x))$.

2. Structural Induction

$$\frac{\{\varphi(x_{i,1}) \wedge \ldots \wedge \varphi(x_{i,n_i}) \Rightarrow \varphi(c_i(x^*_i)) \mid i = 1, \ldots, k\}}{\varphi(x)}$$

if x is a variable of a data type τ with the constructors $c_1, \ldots, c_k$, and if $x_{i,1}, \ldots, x_{i,n_i}$ are the variables of the data type τ occurring in x^*_i.

For example, if we have a data type list with the constructors empty : list and add : nat $\times$ list $\rightarrow$ list, then instead of $\forall x : \mathsf{list}\ \varphi(x)$ one may prove $\varphi(\mathsf{empty})$ and $\forall x_1 : \mathsf{nat}, x_2 : \mathsf{list}\ \ \varphi(x_2) \Rightarrow \varphi(\mathsf{add}(x_1, x_2))$.

To continue our proof of the associativity of plus (1), the terms in formula (2) can be *symbolically evaluated*, i.e., the first defining equation of plus can be used as a rewrite rule which yields $y + w = y + w$. In general, the following rule is used for symbolic evaluation.

3. Symbolic Evaluation

$$\frac{\varphi(\quad \sigma(r) \quad)}{\varphi(\ \sigma(\ f(t^*)\ )\ )}$$

if σ is a substitution and $f(t^*) = r$ is a defining equation.

The formula $y+w = y+w$ resulting from symbolic evaluation of (2) is a (trivial) first order theorem. Symbolic evaluation of (3) also results in a first order theorem. In this way, the truth of the associativity law (1) can be verified. For that purpose the following fourth rule is introduced. It states that it is sufficient to prove lemmata $\psi_1, \ldots, \psi_n$ instead of the original conjecture φ, if $\psi_1 \wedge \ldots \wedge \psi_n \Rightarrow \varphi$ can be shown by a first order calculus. If φ is a trivial theorem, then (by choosing $n = 0$) one obtains $\frac{}{\varphi}$ as a special case of this rule, i.e., then the proof of φ is completed.

4. First Order Consequence

$$\frac{\psi_1, \ldots, \psi_n}{\varphi}$$

if $Ax^{\text{data}} \vdash \psi_1 \wedge \ldots \wedge \psi_n \Rightarrow \varphi$, where $n \geq 0$ and "$\vdash$" denotes derivability by a first order calculus.

This fourth rule can also be used to "apply" lemmata or induction hypotheses (e.g., for "cross-fertilization" [6]). For example, consider the verification of the distributivity law (where "$*$" abbreviates times)

$$u * (v + w) = u * v + u * w \tag{4}$$

by induction w.r.t. plus(v, w). The formula resulting from plus' non-recursive equation is easily proved. The formula corresponding to plus' recursive equation has the form (IH) $\Rightarrow$ (IC), where the induction hypothesis is

$$u * (x + y) = u * x + u * y \tag{IH}$$

and the induction conclusion is symbolically evaluated to

$$u + u * (x + y) = (u + u * x) + u * y. \tag{IC}$$

With the fourth rule we can "apply" the induction hypothesis and replace $u * (x + y)$ by $u * x + u * y$ in the induction conclusion (IC). So instead of (IH) $\Rightarrow$ (IC) it suffices to prove

$$u + (u * x + u * y) = (u + u * x) + u * y \tag{5}$$

as $Ax^{\text{data}} \vdash (5) \Rightarrow [(\text{IH}) \Rightarrow (\text{IC})]$. Note that (5) is an instantiation of the associativity law. The following fifth rule of the calculus allows us to

generalize formula (5) by replacing $u * x$ and $u * y$ by new variables and to prove the generalized conjecture instead. Hence, as the associativity of plus (1) has already been verified, the distributivity (4) is also proved.

5. Generalization

$$\frac{\varphi}{\sigma(\varphi)} \quad \text{where } \sigma \text{ is a substitution.}$$

(Instead of Rule 4 and 5 one could also use a different version of Rule 4, where instead of "$Ax^{\text{data}} \vdash \forall_{\ldots}\,(\psi_1 \wedge \ldots \wedge \psi_n \Rightarrow \varphi)$" one only requires "$Ax^{\text{data}} \vdash \forall_{\ldots}\,(\psi_1 \wedge \ldots \wedge \psi_n) \Rightarrow \forall_{\ldots}\,\varphi$". But the advantage of our formulation will become obvious when extending Rule 4 to partial functions in Section 3.)

Recall that if all occurring algorithms are terminating and completely defined then these algorithms compute total functions. In this case, the above calculus is sound.

THEOREM 1. *Let all algorithms be terminating and completely defined. If $\varphi(x^*)$ can be derived with the rules 1 – 5, then $\forall x^*\ \varphi(x^*)$ is inductively true, i.e., $\varphi(q^*)$ holds for all data objects q^*.*

Proof. For each inference rule $\frac{\varphi_1,\ldots,\varphi_n}{\varphi}$ of the calculus, the truth of $\varphi_1, \ldots, \varphi_n$ implies the truth of φ. The first two rules are sound because they perform a (Noetherian) induction w.r.t. a well-founded relation. The third rule is sound because the defining equations of total algorithms are true. The soundness of Rule 4 and 5 is obvious. □

3. A Calculus for Induction Proofs with Partial Functions

After illustrating the usual approach for induction proofs with *total* functions now we regard algorithms which define *partial* functions.

function $\mathsf{minus} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$	*function* $\mathsf{quot} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$
$\mathsf{minus}(x, \mathsf{0}) = x$	$\mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) = \mathsf{0}$
$\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) = \mathsf{minus}(x, y)$	$\mathsf{quot}(\mathsf{s}(x), y) = \mathsf{s}(\mathsf{quot}(\mathsf{minus}(\mathsf{s}(x), y), y))$

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, $\mathsf{minus}(x, y)$ 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. $\mathsf{quot}(\mathsf{1}, \mathsf{0})$). In fact, $\mathsf{quot}(x, y)$ is only defined if the number y is a divisor of the number

x (and $y \neq 0$). In general, we say that (evaluation of) a ground term is *defined*, if it can be evaluated to a constructor ground term using the operational semantics given in Section 2.

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

$$\forall n, m : \mathsf{nat} \quad \mathsf{plus}(m, \mathsf{minus}(n, m)) = n, \tag{6}$$

$$\forall n, m : \mathsf{nat} \quad \mathsf{times}(m, \mathsf{quot}(n, m)) = n. \tag{7}$$

Then minus and quot are in fact *partially correct* w.r.t. these specifications. So for quot we have:

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

To formalize the handling of partial correctness, we define a new notion of "partial inductive truth" for formulas like (6) and (7) which may contain partial functions. A formula $\forall x^*\ \varphi(x^*)$ is *partially true*, if $Eq \cup Ax^{\mathrm{data}} \models \varphi(q^*)$ holds for all those data objects q^* *where evaluation of all terms in* $\varphi(q^*)$ *is defined*. Again, Eq is the set of all defining equations and Ax^{data} is defined as in Section 2.

Note that as the defining equations are *orthogonal*, $Eq \cup Ax^{\mathrm{data}}$ is still *consistent*, even if Eq contains the defining equations of partial functions.[3] Here the requirement of *linear* patterns is necessary to ensure that the defining equations are confluent, and thus, they never contradict the freeness of the constructors. For example, let bool be the data type with the constructors true and false. Then the (non-overlapping, but non left-linear) defining equations $\mathsf{same}(x, x) = \mathsf{true}$, $\mathsf{same}(x, \mathsf{s}(x)) = \mathsf{false}$, and $\mathsf{f}(x) = \mathsf{s}(\mathsf{f}(x))$ would imply $\mathsf{true} = \mathsf{false}$ [34] and hence, $Eq \cup Ax^{\mathrm{data}}$ would not be consistent any more.

To ensure that evaluation of *all* terms in $\varphi(q^*)$ is defined, one only has to check whether the *top-level* terms of $\varphi(q^*)$ are defined. The reason is that due to the eager nature of our programming language, definedness of a term implies definedness of all its subterms. Here, the top-level terms of an equality $s = t$ are defined to be s and t, the top-level terms of $\varphi_1 \wedge \varphi_2$ are the union of φ_1's and φ_2's top-level terms, etc. For the sake of brevity, in the following we often speak of "evaluation of φ" instead of "evaluation of all (top-level) terms in φ". Now we say that an algorithm is *partially correct* w.r.t. a specification formula, if

[3] For instance, Eq's initial model is also a model of Ax^{data}. The reason is that for all ground terms t^* and s^*, validity of $c(t^*) = c(s^*)$ in the initial model implies Eq-joinability of $c(t^*)$ and $c(s^*)$ by Birkhoff's Theorem [3] and by the confluence of Eq (regarded as a term rewrite system). But as Eq is a constructor system, this implies that for all i, $t_i = s_i$ is valid in the initial model, too. Similarly, $c_1(t^*) = c_2(t^*)$ cannot be valid in the initial model, since these terms are not Eq-joinable.

this formula is partially true. For instance, (6) and (7) are partially true and hence, minus and quot are partially correct w.r.t. these formulas.

This notion of partial truth resp. of partial correctness is widely used in program verification, cf. e.g. [52, 54]. If a conjecture only contains terminating and completely defined algorithms, then partial truth coincides with the notion of truth introduced in Section 2. A model theoretic characterization of partial truth (which requires an explicit object-level representation of definedness) can be found in Section 5 and for a comparison with alternative definitions see Section 8.

For partial truth we again have to verify a statement about infinitely many data objects and hence, we intend to perform induction again. As (7) contains a call of quot, for the proof of (7) one would like to use an induction w.r.t. the algorithm quot (according to Rule 1). To ease readability, let $\varphi(n, m)$ denote the conjecture (7) (i.e., $\varphi(n, m)$ is "$\mathsf{times}(m, \mathsf{quot}(n, m)) = n$") and let "$x - y$" abbreviate $\mathsf{minus}(x, y)$. Then instead of (7) one would have to verify

$$\varphi(\underline{0}, \underline{\mathsf{s}(y)}), \tag{8}$$

$$\varphi(\underline{\mathsf{s}(x) - y}, \underline{y}) \Rightarrow \varphi(\underline{\mathsf{s}(x)}, \underline{y}). \tag{9}$$

But recall that induction proofs are only *sound* if the induction relation used is *well founded*. Hence, inductions w.r.t. non-terminating algorithms like quot must not be used in an unrestricted way. For example, by induction w.r.t. the non-terminating algorithm f with the defining equation $\mathsf{f}(x) = \mathsf{f}(x)$ one could prove *any* formula, e.g., conjectures like $\mathsf{f}(x) = x$ or $\neg x = x$. However, while $\mathsf{f}(x) = x$ is indeed partially true (as it holds for every instantiation of x where evaluation of $\mathsf{f}(x)$ is defined), the conjecture $\neg x = x$ is not partially true. Thus, for partial functions we can no longer use the calculus of Section 2, since this would enable the proof of false facts.

However, for formula (7) the induction w.r.t. the recursions of quot is nevertheless sound, i.e., partial truth of (8) and (9) in fact implies partial truth of (7). The reason is that the only occurrence of a partial function in (7) is $\mathsf{quot}(n, m)$. Hence, for all natural numbers n and m, evaluation of $\varphi(n, m)$ is defined iff evaluation of $\mathsf{quot}(n, m)$ is defined.

Partial truth of (8) and (9) implies that $\varphi(n, m)$ holds for all numbers n and m where $\mathsf{quot}(n, m)$ is defined, provided that it also holds for the numbers $n - m$ and m, if evaluation of $\mathsf{quot}(n, m)$ leads to the recursive call $\mathsf{quot}(n - m, m)$. 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. This restricted induction relation is *well founded* although quot is not always terminating. We formalize this result with the following lemma.

LEMMA 2 (Induction w.r.t. Partial Functions). *Let the term $f(x^*)$ be the only occurrence of a possibly partial function in the conjecture $\varphi(x^*)$. For each defining equation $f(t^*) = r$ where r contains the f-terms $f(s_1^*), \ldots, f(s_n^*)$, let $\varphi(s_1^*) \wedge \ldots \wedge \varphi(s_n^*) \Rightarrow \varphi(t^*)$ be partially true. Then $\varphi(x^*)$ is also partially true.*

Proof. Suppose that $\varphi(x^*)$ is not partially true.[4] Then there exists a "counterexample", i.e., a tuple of data objects q^* such that evaluation of $\varphi(q^*)$ is defined, but $Eq \cup Ax^{\text{data}} \not\models \varphi(q^*)$. As $\varphi(x^*)$ contains the term $f(x^*)$, this implies that $f(q^*)$ is also defined.

Let $\succ_f$ be the relation where $q_1^* \succ_f q_2^*$ holds for two tuples of data objects iff evaluation of $f(q_1^*)$ is defined and leads to the recursive call $f(q_2^*)$. This relation is well founded even if f is partial. Hence, there also exists a minimal counterexample q^* w.r.t. $\succ_f$.

As evaluation of $f(q^*)$ is defined, there must be a defining equation $f(t^*) = r$ such that q^* is an instantiation of the pattern t^*, i.e., $q^* = \sigma(t^*)$ for some substitution σ. Let $f(s_1^*), \ldots, f(s_n^*)$ be the f-terms in r. Due to the definedness of $f(q^*)$, each $\sigma(s_i^*)$ evaluates to some data objects p_i^* where we have $q^* \succ_f p_i^*$ for all $1 \leq i \leq n$. Moreover, evaluation of each $f(p_i^*)$ is defined and as $\varphi(x^*)$ does not contain any other occurrences of partial functions besides the term $f(x^*)$, evaluation of $\varphi(p_i^*)$ is defined, too.

So by the partial truth of $\varphi(s_1^*) \wedge \ldots \wedge \varphi(s_n^*) \Rightarrow \varphi(t^*)$, we have $Eq \cup Ax^{\text{data}} \models \varphi(p_1^*) \wedge \ldots \wedge \varphi(p_n^*) \Rightarrow \varphi(q^*)$. Thus, $Eq \cup Ax^{\text{data}} \not\models \varphi(q^*)$ implies $Eq \cup Ax^{\text{data}} \not\models \varphi(p_i^*)$ for some i. But then p_i^* is a smaller counterexample than q^*, which leads to a contradiction. □

Therefore by restricting the first rule of the calculus in a suitable way, one may perform *inductions w.r.t. partial functions* like quot, too.

1′. Induction w.r.t. Algorithms
Rule 1, where either f must be total and φ may contain total functions only or f may be partial and the only occurrence of a possibly partial function in φ must be the term $f(x^*)$.

Note that for this rule, the *reason* for partiality is not crucial, i.e., incompleteness and non-termination are treated in the same way.

So inductions w.r.t. partial functions may be used for verifying conjectures containing partial functions. Note that actually this is the *only* kind of induction which is possible for such conjectures, i.e., for statements about partial functions the rules for well-founded induction are

[4] To ease readability, we only prove the lemma for the the case where φ contains no other variables than x^*. The extension of the proof to the general case where φ may contain additional variables y^* is straightforward.

no longer sound (and a similar problem appears with conjectures containing *several* occurrences of partial functions). For instance, if half has the defining equations $\mathsf{half}(0) = 0$ and $\mathsf{half}(\mathsf{s}(\mathsf{s}(x))) = \mathsf{s}(\mathsf{half}(x))$, then by structural induction one could verify the statement

$$\forall x : \mathsf{nat} \quad \mathsf{half}(x) = \mathsf{half}(\mathsf{half}(x)) \tag{10}$$

although it is not partially true (e.g. $\mathsf{half}(4) = 2$, but $\mathsf{half}(\mathsf{half}(4)) = 1$). Here, structural induction would transform (10) into the formulas $\mathsf{half}(\underline{0}) = \mathsf{half}(\mathsf{half}(\underline{0}))$ and $\mathsf{half}(\underline{x}) = \mathsf{half}(\mathsf{half}(\underline{x})) \Rightarrow \mathsf{half}(\underline{\mathsf{s}(x)}) = \mathsf{half}(\mathsf{half}(\underline{\mathsf{s}(x)}))$, both of which are partially true. The reason is that there does not exist a data object q such that evaluation of both $\mathsf{half}(q)$ and $\mathsf{half}(\mathsf{s}(q))$ is defined. (More precisely, the problem is that definedness of the induction conclusion does not imply definedness of the induction hypothesis.) Similarly, the false conjecture (10) could also be proved by induction w.r.t. half using x as induction variable.

For that reason, we only allow induction w.r.t. a partial function f if $f(x^*)$ is the *only* term with a partial root function in the conjecture and well-founded inductions are only permitted for statements containing total functions *only* (see Section 5 for an extension of our calculus). This results in the following rule for structural induction.

2′. Structural Induction
Rule 2, where all functions in φ must be total.

To continue the proof of (8) and (9), we now apply symbolic evaluation according to the third rule of the calculus. Symbolic evaluation may also be done for partial functions because if $f(t^*) = r$ is a defining equation, then replacing $\sigma(f(t^*))$ by $\sigma(r)$ does not "decrease the definedness" of the formula. Thus, $\sigma(f(t^*))$ and $\sigma(r)$ evaluate to the same result, whenever evaluation of $\sigma(f(t^*))$ is defined.

3′. Symbolic Evaluation Rule 3

In this way, (8) is transformed into the trivial theorem $0 = 0$ and (9) is transformed into the formula

$$\begin{aligned} y * \mathsf{quot}(\mathsf{s}(x) - y, y) &= \mathsf{s}(x) - y \quad \Rightarrow \\ y + y * \mathsf{quot}(\mathsf{s}(x) - y, y) &= \mathsf{s}(x). \end{aligned} \tag{11}$$

Now one would like to use the fourth rule to "apply" the induction hypothesis, i.e., to transfer (11) into

$$y + (\mathsf{s}(x) - y) = \mathsf{s}(x). \tag{12}$$

However, this rule may no longer be used to perform arbitrary first order inferences if we deal with partial functions. As an example, the fourth rule allows us to conclude $\frac{\varphi_1 \wedge \varphi_2}{\varphi_1}$. This is a sound inference for

total functions, but it becomes unsound when handling partial functions, i.e., $\varphi_1 \wedge \varphi_2$ may be partially true, although φ_1 is not partially true. For instance, φ_1 could be $\neg x = x$ and φ_2 could contain an undefined term like $\mathsf{quot}(1, 0)$.

Therefore the fourth rule $\frac{\psi_1,\ldots,\psi_n}{\varphi}$ may only be used in proofs about partial functions, if definedness of φ implies definedness of $\psi_1, \ldots, \psi_n$ (resp. of their corresponding instantiations). For that reason we demand that every term with a partial root function in $\psi_1, \ldots, \psi_n$ must also occur in φ.[5] In this way, (11) can indeed be transformed into (12), because the only term with a partial root symbol in (12) (viz. $\mathsf{minus}(\mathsf{s}(x), y)$) was already present in (11).

4′. First Order Consequence
Rule 4, where all terms $f(t^*)$ with a possibly partial root function f in $\psi_1, \ldots, \psi_n$ must also occur in φ.

To conclude the proof of (7), we use the generalization rule and replace the term $\mathsf{s}(x)$ in (12) by a new variable z. Obviously, this rule can also be used for partial functions, because if φ is partially true then any instantiation of $\sigma(\varphi)$ must hold if its evaluation is defined.

5′. Generalization Rule 5

Generalization of (12) results in $y + (z - y) = z$. This is the specification of minus (6) (with a variable renaming), i.e., in this way the partial correctness proof of quot is reduced to the partial correctness proof of minus. (Subsequently, partial truth of (6) can also be proved with our calculus by induction w.r.t. the algorithm minus.)

Summing up, the restricted rules $1' - 5'$ constitute a calculus for induction proofs which is also sound for partial functions. Thus, by imposing some slight restrictions, the inference rules implemented in most induction theorem provers and their heuristics for the application of these rules can now also be used for partial functions.

THEOREM 3. *If $\varphi(x^*)$ can be derived with the rules $1' - 5'$, then $\forall x^*\ \varphi(x^*)$ is partially true, i.e., $\varphi(q^*)$ holds for all those data objects q^*, where evaluation of all top-level terms in $\varphi(q^*)$ is defined.*

Proof. The soundness of Rule $1'$ is proved in Lemma 2 and for the remaining rules it is obvious. Hence, for each inference rule $\frac{\varphi_1,\ldots,\varphi_n}{\varphi}$ of the calculus, *partial* truth of $\varphi_1, \ldots, \varphi_n$ implies *partial* truth of φ. □

[5] This condition is sufficient, because in our formulation of Rule 4 we required "$Ax^{\mathrm{data}} \vdash \forall\ldots\ (\psi_1 \wedge \ldots \wedge \psi_n \Rightarrow \varphi)$" instead of "$Ax^{\mathrm{data}} \vdash \forall\ldots(\psi_1 \wedge \ldots \wedge \psi_n) \Rightarrow \forall\ldots\varphi$". So for each instantiation σ, $\sigma(\varphi)$ is a consequence of $\sigma(\psi_1), \ldots, \sigma(\psi_n)$. Hence, now it is sufficient if for each σ, definedness of $\sigma(\varphi)$ implies definedness of $\sigma(\psi_1), \ldots, \sigma(\psi_n)$.

4. Extensions for Algorithms with Conditionals

While up to now we restricted ourselves to algorithms defined by *unconditional* equations, in this section we extend our results to algorithms with conditions. For that purpose, our programming language uses a pre-defined conditional function $\mathsf{if} : \mathsf{bool} \times \tau \times \tau \to \tau$ for each data type τ. 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. As an example regard the following algorithm div for *truncated* division (i.e., $\mathsf{div}(n, m)$ computes $\lfloor \frac{n}{m} \rfloor$). In contrast to quot, div is defined whenever its second argument is not 0. It uses the (total) auxiliary function

$$\begin{aligned} &\mathit{function}\ \mathsf{ge} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool} \\ &\quad \mathsf{ge}(x, \mathsf{0}) \ = \ \mathsf{true} \\ &\quad \mathsf{ge}(\mathsf{0}, \mathsf{s}(y)) \ = \ \mathsf{false} \\ &\quad \mathsf{ge}(\mathsf{s}(x), \mathsf{s}(y)) \ = \ \mathsf{ge}(x, y) \end{aligned}$$

to compute the usual "greater-equal" relation on naturals. Now the algorithm for div reads as follows.

$$\begin{aligned} &\mathit{function}\ \mathsf{div} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} \\ &\quad \mathsf{div}(\mathsf{0}, \mathsf{s}(y)) \ = \ \mathsf{0} \\ &\quad \mathsf{div}(\mathsf{s}(x), y) \ = \ \mathsf{if}(\,\mathsf{ge}(\mathsf{s}(x), y),\ \mathsf{s}(\mathsf{div}(\mathsf{minus}(\mathsf{s}(x), y), y)),\ \mathsf{0}) \end{aligned}$$

The operational semantics of our extended programming language is again obtained by regarding all constructor ground instantiations of the defining equations as rewrite rules. However, now in R^{op} we have additional rewrite rules

$$\mathsf{if}(\mathsf{true}, x, y) \to x \quad \text{and} \quad \mathsf{if}(\mathsf{false}, x, y) \to y$$

for the conditionals. This captures their non-eager semantics, as the variables x, y in these rewrite rules may be instantiated by arbitrary terms. For example, "$\mathsf{if}(\mathsf{false}, t, \mathsf{0})$" can be evaluated to 0 (i.e., its evaluation is *defined*), even if t cannot be evaluated to a constructor ground term. So for terms with conditionals, a term may be defined although it contains undefined subterms.

Note that now R^{op} is no longer suitable as an interpreter for our language, because a term like "$\mathsf{if}(\mathsf{false}, t, \mathsf{0})$" has *both* finite and infinite R^{op}-reductions. To avoid unnecessary infinite reductions, we have to use a *context-sensitive* rewriting strategy where reductions may never take place in the second or third argument of an if [29, 53]. Let "$\to_{R^{\mathrm{op}},\mathsf{if}}$" denote this restricted rewrite relation. Now we say that a ground term

t evaluates to another ground term t' iff $t \to^*_{R^{\rm op},\mathsf{if}} t'$ holds. However, the following lemma shows that to define the operational semantics (i.e., to determine whether t eventually evaluates to a *constructor* ground term q), this restriction on the rewrite relation is not necessary.

LEMMA 4 (Operational Semantics). *A ground term t evaluates to a constructor ground term q (i.e., $t \to^*_{R^{\rm op},\mathsf{if}} q$) iff $t \to^*_{R^{\rm op}} q$.*

Proof. The "only if" direction is clear. For the "if" direction, we use an induction w.r.t. the relation where a ground term t_1 is greater than another one t_2 iff either the minimal $R^{\rm op}$-reduction of t_2 to a constructor ground term is shorter than the one of t_1 or else, the minimal $R^{\rm op}$-reductions of t_1 and t_2 to constructor ground terms have the same length, but t_2 is a subterm of t_1. Here, the *minimal* $R^{\rm op}$-reduction is used to ensure well-foundedness of the induction relation, since $\to_{R^{\rm op}}$ itself is not even well founded for terms t with $t \to^*_{R^{\rm op}} q$.

Let $t \to^*_{R^{\rm op}} q$ be a minimal $R^{\rm op}$-reduction of t to a constructor ground term. If $t = f(t^*)$ for some algorithm f, then the reduction must have the form $f(t^*) \to^*_{R^{\rm op}} f(q^*) \to_{R^{\rm op}} r \to^*_{R^{\rm op}} q$, where $f(q^*) = r$ is a constructor ground instantiation of a defining equation. The induction hypothesis implies $t^* \to^*_{R^{\rm op},\mathsf{if}} q^*$ and $r \to^*_{R^{\rm op},\mathsf{if}} q$ and so we obtain $f(t^*) \to^*_{R^{\rm op},\mathsf{if}} f(q^*) \to_{R^{\rm op},\mathsf{if}} r \to^*_{R^{\rm op},\mathsf{if}} q$, as desired.

If $t = c(t^*)$ for some constructor c, then the conjecture immediately follows from the induction hypothesis. Finally, if $t = \mathsf{if}(b, t_1, t_2)$, then $\mathsf{if}(b, t_1, t_2) \to^*_{R^{\rm op}} q$ implies $b \to^*_{R^{\rm op}} \mathsf{true}$ or $b \to^*_{R^{\rm op}} \mathsf{false}$. Without loss of generality, we assume $b \to^*_{R^{\rm op}} \mathsf{true}$. Thus, the reduction of $\mathsf{if}(b, t_1, t_2)$ has the form

$$\mathsf{if}(b, t_1, t_2) \to^*_{R^{\rm op}} \mathsf{if}(\mathsf{true}, t_1', t_2') \to_{R^{\rm op}} t_1' \to^*_{R^{\rm op}} q$$

(where in fact we have $t_2 = t_2'$, as the reduction should be minimal). By the induction hypothesis this implies $b \to^*_{R^{\rm op},\mathsf{if}} \mathsf{true}$ and $t_1 \to^*_{R^{\rm op},\mathsf{if}} q$. Thus, we obtain $\mathsf{if}(b, t_1, t_2) \to^*_{R^{\rm op},\mathsf{if}} \mathsf{if}(\mathsf{true}, t_1, t_2) \to_{R^{\rm op},\mathsf{if}} t_1 \to^*_{R^{\rm op},\mathsf{if}} q$, which proves the lemma. □

Similar to the partial correctness statement (7) about quot, we may now want to verify partial truth of the following conjecture.

$$\forall n, m : \mathsf{nat} \quad \mathsf{ge}(n, \mathsf{times}(m, \mathsf{div}(n, m))) = \mathsf{true} \tag{13}$$

Recall that $\forall x^*\ \varphi(x^*)$ is partially true if $Eq \cup Ax^{\rm data} \models \varphi(q^*)$ holds for all those data objects q^*, where evaluation of all top-level terms in $\varphi(q^*)$ is defined. Note that with the function symbol if, the restriction to the definedness of *top-level* terms is important, since an if-term may be

defined, even if some of its subterms are not. Of course, now we have to extend Eq by the new axioms "$\mathsf{if}(\mathsf{true}, x, y) = x$" and "$\mathsf{if}(\mathsf{false}, x, y) = y$" for all conditionals.

To prove the partial truth of (13) automatically, we intend to proceed in a similar way as in the partial correctness proof of quot. Hence, (13) should be proved by an induction w.r.t. div. However, to handle functions defined with conditionals, we have to change the rule for inductions w.r.t. algorithms slightly. First of all, recall that the soundness of Rule $1'$ relied on the fact that definedness of a formula ensured definedness of all its subterms. Hence, for the partial truth of a formula φ containing $f(x^*)$, it was sufficient to prove it just for those inputs where f is defined. However, this does not necessarily hold if f occurs in an argument of a conditional. Therefore, we have to demand that Rule $1'$ may only be applied to conjectures without if-terms.

Up to now, when proving the conjecture $\varphi(x^*)$ by induction w.r.t. f, we obtained an induction formula

$$\varphi(s^*) \Rightarrow \varphi(t^*) \tag{14}$$

if $f(t^*) = r$ is a defining equation and r contains a recursive call $f(s^*)$ at some position π. For functions without conditionals, this induction is sound, because then evaluation of $f(t^*)$ always leads to evaluation of $f(s^*)$. Hence, if $f(t^*)$ is defined, then $f(s^*)$ is also defined and its evaluation takes fewer steps than evaluation of $f(t^*)$.

However, if the defining equation has the form $f(t^*) = \mathsf{if}(b, f(s^*), \ldots)$, then evaluation of $f(t^*)$ only leads to evaluation of $f(s^*)$, if the condition b is true. Hence, when proving $\varphi(t^*)$ one may only use the induction hypothesis $\varphi(s^*)$ if b evaluates to true.

In general, if π is a position in a term r, then its subterm $r|_\pi$ is only evaluated under the condition $\textsc{con}(r, \pi)$, where $\textsc{con}(r, \pi)$ is defined as follows:

$$\textsc{con}(r,\pi) = \begin{cases} \mathsf{true}, & \text{if } \pi \text{ is the top position of } r \\ \mathsf{if}(r_1, \textsc{con}(r_2, \pi'), \mathsf{false}), & \text{if } r = \mathsf{if}(r_1, r_2, r_3) \text{ and } \pi = 2\pi' \\ \mathsf{if}(r_1, \mathsf{false}, \textsc{con}(r_3, \pi')), & \text{if } r = \mathsf{if}(r_1, r_2, r_3) \text{ and } \pi = 3\pi' \\ \textsc{con}(r_j, \pi'), & \text{otherwise (where } r = g(r_1 \ldots r_k) \\ & \text{and } \pi = j\pi') \end{cases}$$

For example, in the result of div's second equation the recursive call is at position 21 and we obtain $\textsc{con}(\mathsf{if}(\ldots), 21) = \mathsf{if}(\mathsf{ge}(\mathsf{s}(x), y), \mathsf{true}, \mathsf{false})$. Note that due to the use of the function "if" in the definition of con, definedness of r implies definedness of $\textsc{con}(r, \pi)$ for all instantiations with data objects. Thus, instead of the induction formula (14) we have to use the following two formulas which allow a use of the induction hypothesis $\varphi(s^*)$ only under the condition of its evaluation.

$$\begin{array}{ll} \textsc{con}(r,\pi) = \mathsf{true} \ \wedge \ \varphi(s^*) & \Rightarrow \ \varphi(t^*) \\ \textsc{con}(r,\pi) = \mathsf{false} & \Rightarrow \ \varphi(t^*) \end{array}$$

Let us abbreviate the conjecture (13) by $\varphi(n, m)$. Then induction w.r.t. div transforms this conjecture into the following three formulas.

$$\varphi(\mathsf{0}, \mathsf{s}(y)) \tag{15}$$

$$\mathsf{if}(\mathsf{ge}(\mathsf{s}(x), y), \mathsf{true}, \mathsf{false}) = \mathsf{true} \ \wedge \ \varphi(\mathsf{s}(x) - y, y) \ \Rightarrow \ \varphi(\mathsf{s}(x), y) \tag{16}$$

$$\mathsf{if}(\mathsf{ge}(\mathsf{s}(x), y), \mathsf{true}, \mathsf{false}) = \mathsf{false} \ \Rightarrow \ \varphi(\mathsf{s}(x), y) \tag{17}$$

Of course, a defining equation $f(t^*) = r$ may contain several recursive calls in r. Hence in general, Rule $1'$ now reads as follows.

$1'$. Induction w.r.t. Algorithms (extended to conditionals)

$$\frac{\{\, \textsc{con}(r_i, \pi_{i,j}) = \mathsf{true} \ \wedge \ \varphi(s^*_{i,j}) \ \Rightarrow \ \varphi(t^*_i) \mid i = 1, \ldots, k, \ j = 1, \ldots, n_i \,\} \\ \{\, \textsc{con}(r, \pi_{i,1}) = \mathsf{false} \wedge \ldots \wedge \textsc{con}(r, \pi_{i,n_i}) = \mathsf{false} \ \Rightarrow \ \varphi(t^*_i) \mid i = 1, \ldots, k \,\}}{\varphi(x^*)}$$

if f has the defining equations $f(t^*_i) = r_i$ for $i = 1, \ldots, k$, where $r_i|_{\pi_{i,j}} = f(s^*_{i,j})$ for $j = 1, \ldots n_i$, and either f and all functions in φ are total or else, φ contains no occurrence of "if" and the only occurrence of a possibly partial function in φ is $f(x^*)$.

A refinement of this approach is obtained by combining those induction formulas which have the same condition $\textsc{con}(r, \pi)$, cf. e.g. [6, 75, 79]. So if $\textsc{con}(r, \pi_{i,j}) = \textsc{con}(r, \pi_{i,j'})$ then instead of two separate induction formulas for $s^*_{i,j}$ and $s^*_{i,j'}$, it is preferable to use the formula

$$\textsc{con}(r, \pi_{i,j}) = \mathsf{true} \ \wedge \ \varphi(s^*_{i,j}) \ \wedge \ \varphi(s^*_{i,j'}) \ \Rightarrow \ \varphi(t^*_i).$$

In general, this weaker induction formula is easier to prove, as one may use *both* induction hypotheses $\varphi(s^*_{i,j})$ and $\varphi(s^*_{i,j'})$ together in order to verify the induction conclusion $\varphi(t^*_i)$.

For the proof of formulas $\varphi(\,\mathsf{if}(t_1, t_2, t_3)\,)$ containing conditionals, we need an additional rule which performs a case analysis w.r.t. the condition t_1. In this way, we obtain two new formulas "$t_1 = \mathsf{true} \Rightarrow \varphi(t_2)$" and "$t_1 = \mathsf{false} \Rightarrow \varphi(t_3)$". Note that such a case analysis may only be done for *top-level* conditionals. For example, let t_1 be a term whose evaluation is undefined. Nevertheless, the formula

$$\mathsf{if}(\mathsf{false}, \mathsf{if}(t_1, \ldots, \ldots), \mathsf{false}) = \mathsf{true} \tag{18}$$

is not partially true, because $\mathsf{if}(\mathsf{false}, \ldots, \mathsf{false})$ evaluates to false. However, if one would perform a case analysis w.r.t. the condition t_1 of the

inner conditional, then (18) would be transformed into the formulas "$t_1 = \mathsf{true} \Rightarrow \ldots$" and "$t_1 = \mathsf{false} \Rightarrow \ldots$". These formulas are both partially true, since both contain the undefined top-level term t_1.

6′. Case Analysis

$$\frac{t_1 = \mathsf{true} \;\Rightarrow\; \varphi(t_2) \qquad t_1 = \mathsf{false} \;\Rightarrow\; \varphi(t_3)}{\varphi(\,\mathsf{if}(t_1, t_2, t_3)\,)}$$

if φ contains no if-term at positions above the term $\mathsf{if}(t_1, t_2, t_3)$.

By repeated application of this rule, all occurring conditionals can be eliminated. For example by case analysis, formula (16) from the div example is transformed into the two formulas

$$\begin{array}{l} \mathsf{ge}(\mathsf{s}(x), y) = \mathsf{true} \;\Rightarrow\; \big[\,\mathsf{true} = \mathsf{true} \wedge \varphi(\mathsf{s}(x) - y, y) \;\Rightarrow\; \varphi(\mathsf{s}(x), y)\,\big] \\ \mathsf{ge}(\mathsf{s}(x), y) = \mathsf{false} \Rightarrow\; \big[\,\mathsf{false} = \mathsf{true} \wedge \varphi(\mathsf{s}(x) - y, y) \;\Rightarrow\; \varphi(\mathsf{s}(x), y)\,\big]. \end{array}$$

Using Rule 4′, the second formula can be proved and the first one is transformed into

$$\mathsf{ge}(\mathsf{s}(x), y) = \mathsf{true} \wedge \varphi(\mathsf{s}(x) - y, y) \;\Rightarrow\; \varphi(\mathsf{s}(x), y).$$

Similar to Rule 1′, in Rule 4′ we also have to demand that φ contains no if-terms (i.e., all occurring conditionals have to be eliminated by the case analysis rule first). The reason is that to ensure that definedness of φ implies definedness of ψ, in Rule 4′ we only check whether each term with a partial root function in ψ also occurs in φ. But of course, for terms with conditionals this criterion is no longer sufficient. This results in the following rule.

4′. First Order Consequence (extended to conditionals)

$$\frac{\psi_1, \ldots, \psi_n}{\varphi}$$

if $Ax^{\mathrm{data}} \vdash \psi_1 \wedge \ldots \wedge \psi_n \Rightarrow \varphi$, φ contains no occurrence of "if", and all terms with possibly partial root function in $\psi_1, \ldots, \psi_n$ also occur in φ.

Now the rules $1' - 6'$ constitute a sound calculus for induction proofs with partial functions for our extended *conditional* functional programming language. In this way, partial truth of the conjecture (13) about div can be proved similar to the partial correctness of quot in Section 3 (i.e., (13) can be reduced to the conjecture $\mathsf{ge}(u - v, w) = \mathsf{ge}(u, v + w)$ which is proved by induction w.r.t. minus).

5. Refinements by Reasoning about Definedness

With the calculus of Section 3 and 4 the techniques for automated induction proofs can be directly extended to partial functions. In this way it is possible to prove the partial truth of statements like (6), (7), and (13) automatically. The calculus has the advantage that one can perform proofs about partial functions (and even inductions w.r.t. partial functions) without having to deal with definedness explicitly. However, there exist conjectures which cannot be verified with our calculus, because their proofs require reasoning about definedness. As an example, consider the following conjecture.

$$\forall x, y, z : \mathsf{nat} \quad (x - y) - z = (x - z) - y \tag{19}$$

For this formula we attempt to perform induction w.r.t. the algorithm minus using x and y as induction variables. Then (19) would be transformed into the following formulas.

$$(\underline{x} - \underline{\mathsf{0}}) - z = (\underline{x} - z) - \underline{\mathsf{0}} \tag{20}$$

$$(\underline{x} - \underline{y}) - z = (\underline{x} - z) - \underline{y} \;\Rightarrow\; (\underline{\mathsf{s}(x)} - \underline{\mathsf{s}(y)}) - z = (\underline{\mathsf{s}(x)} - z) - \underline{\mathsf{s}(y)} \tag{21}$$

However, our calculus would only allow this induction if $\mathsf{minus}(x, y)$ were the *only* term with a partial root function in the conjecture (otherwise induction w.r.t. algorithms can be unsound, cf. Section 3). But as (19) contains four different minus-terms, it cannot be transformed into (20) and (21) with Rule $1'$.

In fact one may only perform induction, if for each induction step formula of the form "*induction hypothesis* $\Rightarrow$ *induction conclusion*" the induction hypothesis is always defined whenever the induction conclusion is defined. Then for every potential counterexample (corresponding to the induction conclusion) there would be a smaller counterexample (corresponding to the induction hypothesis), cf. Lemma 2. So in our example, for (19) one may indeed use induction w.r.t. minus, because for all values of x and y, definedness of the induction conclusion

$$(\underline{\mathsf{s}(x)} - \underline{\mathsf{s}(y)}) - z = (\underline{\mathsf{s}(x)} - z) - \underline{\mathsf{s}(y)} \tag{22}$$

implies definedness of the induction hypothesis

$$(\underline{x} - \underline{y}) - z = (\underline{x} - z) - \underline{y}. \tag{23}$$

But to check whether definedness of (22) really implies definedness of (23) we have to reason about definedness explicitly. For that purpose we now introduce a definedness function $\mathsf{def} : \tau \to \mathsf{bool}$ for each data

type τ where for any ground term t, $\mathsf{def}(t)$ is true iff evaluation of t is defined. Otherwise, $\mathsf{def}(t)$ is not defined either (i.e., the function def is recursive and eager). Then instead of proving partial inductive truth of a conjecture φ with the top-level term t, we will prove inductive truth of $\mathsf{def}(t) = \mathsf{true} \Rightarrow \varphi$. To ease readability, in the following we will often abbreviate formulas of the form "$t = \mathsf{true}$" by just writing the boolean term t and formulas of the form "$\neg t = \mathsf{true}$" are abbreviated by "$\neg t$".

In Section 5.1 we define an appropriately extended notion of inductive truth and in Section 5.2 our calculus is refined in order to reason about definedness.

5.1. Inductive Truth for Partial Functions

Up to now, *inductive truth* has only been defined if all occurring functions are total (Section 2), whereas for partial functions we only introduced the notion of *partial* truth. Hence, now we have to extend the definition of inductive truth to conjectures with partial functions. For that purpose we use a model theoretic approach. Recall that for *total* functions, inductive truth is equivalent to validity in the initial model of the defining equations Eq. However, due to the occurrence of partial functions, now the initial model of Eq does not correspond to the semantics of our programming language any more. The reason is that the defining equations do not represent the *eager* evaluation strategy of our programming language. For example, $\mathsf{times}(\mathsf{quot}(\mathsf{1},\mathsf{0}),\mathsf{0}) = \mathsf{0}$ is valid in the initial model of the defining equations although (innermost) evaluation of $\mathsf{times}(\mathsf{quot}(\mathsf{1},\mathsf{0}),\mathsf{0})$ is not terminating.

Due to the eager semantics, 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 $f(t) = \mathsf{if}(\mathsf{def}(t), r, f(t))$. To handle functions with several arguments, in the following $\mathsf{def}(t_1, \ldots, t_n)$ is 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 (which may also contain the function def).

DEFINITION 5 (Inductive Truth). Let I be the initial model of

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

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

To illustrate this definition, note that the carrier of the initial model I is (isomorphic to) the quotient algebra $\mathcal{T}/_{=_E}$. Here, $\mathcal{T}$ denotes the set of all ground terms and $=_E$ is the equivalence relation induced by the set E of equations in Definition 5. So for example, the equivalence class $[\mathsf{0}]_{=_E}$ also contains terms like $\mathsf{plus}(\mathsf{0},\mathsf{0})$ and $\mathsf{times}(\mathsf{s}(\mathsf{0}),\mathsf{0})$, etc. As all our algorithms are deterministic, every equivalence class $[t]_{=_E}$ contains at most one constructor ground term. However, there may also be equivalence classes without any constructor ground term. For example, $[\mathsf{minus}(\mathsf{1},\mathsf{2})]_{=_E}$ contains $\mathsf{minus}(\mathsf{0},\mathsf{1})$, but it does not contain any constructor ground term. Those equivalence classes which contain a constructor ground term are the *defined* ones, whereas those without a constructor ground term correspond to the *undefined* ones. Note that in the initial model I not all undefined terms are equal. For example, $\mathsf{minus}(\mathsf{0},\mathsf{1})$ and $\mathsf{minus}(\mathsf{0},\mathsf{2})$ are not in the same equivalence class (although they are both undefined). A formal definition of initial algebras can for instance be found in [30].

For terminating and completely defined algorithms, this notion of inductive truth is equivalent[6] to validity in the initial model of the defining equations, i.e., to the definition of inductive truth used in Section 2. However, Definition 5 extends this notion to partial functions.

In the following, let R be the term rewriting system resulting from E by orienting the equations from left to right. Note that for any ground term t and any constructor ground term q

$$\begin{aligned} & t = q \text{ is inductively true} \\ \text{iff } & I \models t = q \\ \text{iff } & E \models t = q \text{ (as } I \text{ is isomorphic to } \mathcal{T}/_{=_E}) \\ \text{iff } & t \leftrightarrow_R^* q \text{ (by Birkhoff's Theorem [3])} \\ \text{iff } & t \rightarrow_R^* q \text{ (as } R \text{ is a confluent constructor system).} \end{aligned}$$

To show the connection between inductive truth and *partial* inductive truth (i.e., the truth of a conjecture for all instantiations where its evaluation is defined, cf. Section 3), we need some lemmata about R-reductions. The first lemma shows that one may exchange subterms in arguments of if-terms whose condition $\textsc{con}(t,\pi)$ is false.

LEMMA 6 (Exchanging Subterms in if-Arguments). *Let t and s be ground terms which only differ on positions π with $I \models \textsc{con}(t,\pi) =$*

[6] For a formal justification of this claim, note that if all algorithms are total, then every ground term t is Eq- and E-equal to a constructor ground term q. Thus, then a formula $\forall x^*\ \varphi(x^*)$ without def is valid in the initial model I_{Eq} of Eq or in the initial model I of E, respectively, iff $I_{Eq} \models \varphi(q^*)$ resp. $I \models \varphi(q^*)$ holds for all constructor ground terms q^*. By induction on the structure of $\varphi(q^*)$ one can show that $I_{Eq} \models \varphi(q^*)$ iff $Eq \cup Ax^{\mathrm{data}} \models \varphi(q^*)$ iff $I \models \varphi(q^*)$ for all formulas $\varphi(q^*)$ without def, variables, and undefined terms (cf. conjecture (28) in the proof of Theorem 9).

false. *If there exists a number n and a constructor ground term q such that $t \to_R^n q$, then there exists an $m \leq n$ such that $s \to_R^m q$.*

Proof. The lemma is proved by induction on n. For $n = 0$ it is clear and otherwise we have $t \to_R t' \to_R^{n-1} q$. If the reduction $t \to_R t'$ was done on a position π with $I \models \text{CON}(t, \pi) = \mathsf{false}$, then the induction hypothesis implies $s \to_R^m q$ for some $m \leq n - 1$. Otherwise, there is a reduction $s \to_R s'$ which is analogous to $t \to_R t'$, such that t' and s' again only differ on positions π with $I \models \text{CON}(t', \pi) = \mathsf{false}$. Then the conjecture also follows from the induction hypothesis. □

Now we prove that if t reduces to a constructor ground term, then its undefined subterms are only on positions π where $\text{CON}(t, \pi)$ is false.

LEMMA 7 (Positions of Undefined Terms). *Let t be a ground term and $t \to_R^* q$ for some constructor ground term q. If $t|_\pi$ does not R-reduce to a constructor ground term, then we have $I \models \text{CON}(t, \pi) = \mathsf{false}$.*

Proof. The lemma can be shown by structural induction on t. Note that for the top position π we always have $t|_\pi \to_R^* q$.

If $t = \mathsf{if}(r_1, r_2, r_3)$, $\pi = 2\pi'$, and $r_1 \to_R^* \mathsf{false}$, then the lemma is trivial. Otherwise, we must have $r_1 \to_R^* \mathsf{true}$ and thus, $t \to_R^* r_2$. By confluence of R this implies $r_2 \to_R^* q$. Thus, the lemma is implied by the induction hypothesis. The case $\pi = 3\pi'$ is analogous.

Otherwise, if $t = g(r_1 \ldots r_j \ldots r_k)$ and $\pi = j\pi'$ then we also have $r_j \to_R^* q_j$ for some constructor ground term q_j, and thus, the lemma follows from the induction hypothesis. For constructors g and for $g = \mathsf{if}$ and $j = 1$ this is clear. Otherwise, we have $t \to_R \mathsf{if}(\mathsf{def}(r_1 \ldots r_j \ldots r_k), \ldots, \ldots)$. By confluence of R, this if-term must also R-reduce to q and thus, $\mathsf{def}(r_1 \ldots r_j \ldots r_k)$ reduces to a constructor ground term. By induction on the length of the minimal R-reduction of $\mathsf{def}(r_1, \ldots, r_k)$ one can show that there must be constructor ground terms $q_1, \ldots, q_k$ such that $r_i \to_R^* q_i$ for $1 \leq i \leq k$. □

Recall that by Lemma 4, a ground term t *evaluates* to a constructor ground term q iff $t \to_{R^{\text{op}}}^* q$ holds. To handle terms with the definedness function def, let R^{opd} be the extension of R^{op} by the rules $\{\mathsf{def}(q) \to \mathsf{true} \mid q \text{ is a constructor ground term}\}$. The resulting term rewriting system R^{opd} is still orthogonal and hence, confluent. In the following, we say that t evaluates to a constructor ground term q iff $t \to_{R^{\text{opd}}}^* q$, where obviously this definition of the operational semantics coincides with the old one for all terms t without def. The next lemma proves that using Definition 5, def has indeed the intended semantics, i.e., the model theoretic semantics of def corresponds to the operational semantics of "definedness".

LEMMA 8 (Semantics of def). *For any ground term t, $\mathsf{def}(t) = \mathsf{true}$ is inductively true iff evaluation of t is defined.*

Proof. The conjecture $\mathsf{def}(t) = \mathsf{true}$ is inductively true iff $t \to_R^* q$ holds for some constructor ground term q. Thus, for the lemma it is sufficient to prove

$$t \to_{R^{\mathrm{opd}}}^* q \text{ iff } t \to_R^* q. \tag{24}$$

We have $\to_{R^{\mathrm{opd}}} \subseteq \to_R^*$, i.e., every evaluation can also be done with the equations in Definition 5. This implies the "only if" direction.

For the "if" direction, we use an induction on the length of the *minimal* R-reduction of t to a constructor ground term. If t is already a constructor ground term, then the conjecture is trivial. Otherwise, the minimal reduction of t has the form

$$t \to_R t' \to_R^* q$$

where $t|_\pi = \sigma(s_1)$, $t' = t[\sigma(s_2)]_\pi$, and $s_1 \to s_2$ is a rule from R. Due to the minimality of the reduction, by Lemma 6 we have $I \models \neg\mathrm{CON}(t, \pi) = \mathsf{false}$ and thus, $I \models \neg\mathrm{CON}(t', \pi) = \mathsf{false}$, too. Now by Lemma 7, $t' \to_R^* q$ implies that there must also be an R-reduction from $t'|_\pi = \sigma(s_2)$ to some constructor ground term.

So if $s_1 = f(t^*)$ and $s_2 = \mathsf{if}(\mathsf{def}(t^*), r, f(t^*))$, then $\sigma(t^*)$ must be R-reducible to constructor ground terms. Hence, if x^* are the variables occurring in the patterns t^*, then $\sigma(x^*)$ are R-reducible to some constructor ground terms q^*, too. Due to the induction hypothesis these reductions can also be done with R^{opd}, i.e. $\sigma(x^*) \to_{R^{\mathrm{opd}}}^* q^*$. Hence,

$$t = t[\sigma(f(t^*))]_\pi \to_{R^{\mathrm{opd}}}^* t[f(t^*)[x^*/q^*]]_\pi \to_{R^{\mathrm{opd}}} t[r[x^*/q^*]]_\pi. \tag{25}$$

Moreover, $\sigma(\mathsf{def}(t^*)) \to_{R^{\mathrm{opd}}}^* \mathsf{true}$ implies

$$t' = t[\sigma(\mathsf{if}(\mathsf{def}(t^*), r, \ldots))]_\pi \to_{R^{\mathrm{opd}}}^* t[\sigma(r)]_\pi \to_{R^{\mathrm{opd}}}^* t[r[x^*/q^*]]_\pi. \tag{26}$$

On the other hand, by the induction hypothesis we have

$$t' \to_{R^{\mathrm{opd}}}^* q. \tag{27}$$

So by confluence of R^{opd}, (26) and (27) imply $t[r[x^*/q^*]]_\pi \to_{R^{\mathrm{opd}}}^* q$ and by (25) we obtain $t \to_{R^{\mathrm{opd}}}^* q$.

The case $s_1 = \mathsf{def}(c(x^*))$ and $s_2 = \mathsf{def}(x^*)$ can be proved in a similar way and the cases $s_1 = \mathsf{if}(\ldots)$ are trivial, because then the rule $s_1 \to s_2$ is also contained in R^{opd}. Thus, conjecture (24) is proved. □

Now we can show the connection between inductive truth and *partial* inductive truth, i.e., we give a model theoretic characterization which corresponds to our definition of partial truth from Section 3.

THEOREM 9 (Model Theoretic Characterization of Partial Truth). *Let φ be a quantifier-free formula without the function symbol* def *and let t^* be the top-level terms of φ. Then φ is a* partial *inductive truth iff* $\mathsf{def}(t^*) \Rightarrow \varphi$ *is inductively true.*

Proof. Let I be the initial model from Definition 5 and let Eq contain all defining equations and $\mathsf{if}(\mathsf{true}, x, y) = x$, $\mathsf{if}(\mathsf{false}, x, y) = y$.

We first prove that for any *variable-free* formula ψ which does *not contain* the function def and where evaluation of all top-level terms *is defined*, we have

$$Eq \cup Ax^{\mathrm{data}} \models \psi \;\text{ iff }\; I \models \psi. \tag{28}$$

The proof of conjecture (28) is done by induction on the structure of ψ. If ψ is an atomic formula $t_1 = t_2$, then by assumption t_1 and t_2 evaluate to constructor ground terms q_1 and q_2, respectively. Hence, we have $Eq \models t_1 = q_1$ and $Eq \models t_2 = q_2$. Moreover, (by Lemma 8 resp. conjecture (24)), $I \models t_1 = q_1$ and $I \models t_2 = q_2$ holds, too. Thus if $q_1 = q_2$, then we have $I \models \psi$ and $Eq \models \psi$. Otherwise, $q_1 \neq q_2$ implies $I \not\models \psi$ and $Eq \cup Ax^{\mathrm{data}} \not\models \psi$. (In fact, we have $Ax^{\mathrm{data}} \models \neg q_1 = q_2$ as can easily be proved by structural induction. As $Eq \cup Ax^{\mathrm{data}}$ is consistent, this implies $Eq \cup Ax^{\mathrm{data}} \not\models q_1 = q_2$.)

If ψ is of the form $\neg\psi'$, then we have $I \models \psi$ iff $I \not\models \psi'$. Due to the induction hypothesis, this is equivalent to $Eq \cup Ax^{\mathrm{data}} \not\models \psi'$. By structural induction one can prove that for any variable-free (and hence, quantifier-free) formula ψ' with defined top-level terms and without the function def we either have $Eq \cup Ax^{\mathrm{data}} \models \psi'$ or $Eq \cup Ax^{\mathrm{data}} \models \neg\psi'$. Hence, due to the consistency of $Eq \cup Ax^{\mathrm{data}}$, $Eq \cup Ax^{\mathrm{data}} \not\models \psi'$ is equivalent to $Eq \cup Ax^{\mathrm{data}} \models \neg\psi'$ (i.e., to $Eq \cup Ax^{\mathrm{data}} \models \psi$). For other non-atomic formulas ψ, conjecture (28) is proved in a similar way.

We also need the following conjecture which states that for the formula $\mathsf{def}(t^*) \Rightarrow \varphi$ it suffices to regard instantiations with *constructor* ground terms only. Here, x^* are the variables occurring in φ.

$$\begin{array}{l}\text{If } I \models \big(\mathsf{def}(t^*) \Rightarrow \varphi\big)[x^*/q^*] \text{ for all } \textit{constructor} \text{ ground terms } q^*,\\ \text{then } I \models \big(\mathsf{def}(t^*) \Rightarrow \varphi\big)[x^*/p^*] \text{ holds for all ground terms } p^*.\end{array} \tag{29}$$

Let p^* be arbitrary ground terms such that $I \models \mathsf{def}(t^*[x^*/p^*]) = \mathsf{true}$. We examine the positions π of those variables in the terms t from t^* which are replaced by "undefined" ground terms p, i.e., by ground terms p where $I \models \neg p = q$ holds for all constructor ground terms q. By Lemma 7, for all these π we have $I \models \textsc{con}(t[x^*/p^*], \pi) = \mathsf{false}$. So the "undefined" ground terms p are not needed for evaluation of $t[x^*/p^*]$. Hence, by Lemma 6 there exist *constructor* ground terms q^* such that $I \models t[x^*/p^*] = t[x^*/q^*]$. As the variables of φ only occur

within these top-level terms t, $I \models (\mathsf{def}(t^*) \Rightarrow \varphi)[x^*/q^*]$ indeed implies $I \models (\mathsf{def}(t^*) \Rightarrow \varphi)[x^*/p^*]$. Now we can prove Theorem 9:

$\forall x^*\ \varphi(x^*)$ is a partial inductive truth
iff $Eq \cup Ax^{\mathrm{data}} \models \varphi(q^*)$ for all constructor ground terms q^* where evaluation of $t^*[x^*/q^*]$ is defined
iff $I \models \varphi(q^*)$ for all constructor ground terms q^* where evaluation of $t^*[x^*/q^*]$ is defined (by (28))
iff $I \models \varphi(q^*)$ for all constructor ground terms q^* with $I \models \mathsf{def}(t^*[x^*/q^*]) = \mathsf{true}$ (by Lemma 8)
iff $I \models \big(\mathsf{def}(t^*) \Rightarrow \varphi\big)[x^*/p^*]$ for all ground terms p^* (by (29))
iff $I \models \big(\mathsf{def}(t^*) \Rightarrow \varphi\big)$, as I is isomorphic to $\mathcal{T}/_{=_E}$
iff $\mathsf{def}(t^*) \Rightarrow \varphi$ is inductively true. □

Hence, *partial* truth of (19) is equivalent to inductive truth of

$$\mathsf{def}((x - y) - z, (x - z) - y) \ \Rightarrow\ (x - y) - z = (x - z) - y. \qquad (30)$$

5.2. A Refined Induction Calculus for Partial Functions

Now we introduce a refinement of our calculus for induction proofs about partial functions. In contrast to the calculus from Section 3 and 4, this refined calculus proves inductive truth (instead of *partial* inductive truth). For that purpose, it allows an explicit reasoning about definedness. So in particular, it can also be used for total correctness proofs.

In classical Noetherian induction, in order to verify $\forall n \in N\ \varphi(n)$ for some set N, it is sufficient to prove the induction formula

$$\forall n \in N\ \ [\forall k \in N\ \ n > k \Rightarrow \varphi(k)] \Rightarrow \varphi(n), \qquad (31)$$

provided that the relation $> \subseteq N \times N$ is *well founded*. Essentially, our induction principle for partial functions now allows us to use arbitrary (possibly not well-founded) relations $> \subseteq N \times N$ for such induction proofs. The important observation is that (31) implies

$$\forall m \in M\ \ \varphi(m)$$

for every subset $M \subseteq N$ provided that $>$ is well founded on the subset M and that for all $m \in M$ and all $n \in N$, $m > n$ implies $n \in M$.

In our calculus, we choose N to be the set of all (tuples of) ground terms (resp. of equivalence classes from $\mathcal{T}/_{=_E}$), for the relation $>$ we use the ordering induced by the recursion structure of some algorithm f, and we define M to be the domain of f. With this choice, $>$ is always

well founded on M and $m > n$ indeed implies $n \in M$ for all $m \in M$. Thus, these conditions on M do not have to be checked any more. Now the advantage of our induction principle is that we can perform induction proofs without having to determine the actual form of M (since in the induction formula (31) we consider N instead of M).

However, this induction only proves $\varphi(m)$ for $m \in M$. If we want to verify that $\varphi(n)$ holds for all $n \in N$, then we also have to prove a *permissibility conjecture* which shows that $\varphi(n)$ also holds for all $n \in N \setminus M$. It will turn out that in most cases, these permissibility conjectures can also be proved without actually determining the set M. This enables us to perform induction proofs w.r.t. algorithms without having any knowledge about their domains.

In other words, an induction w.r.t. a (possibly partial) algorithm f using the induction variables x^* only 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

$$\neg\mathsf{def}(f(x^*)) \;\Rightarrow\; \varphi(x^*). \tag{32}$$

For instance, let φ be the flawed conjecture $\mathsf{def}(x) \Rightarrow \neg x = x$. If we intended to prove φ by induction w.r.t. the non-terminating function f with the defining equation $\mathsf{f}(x) = \mathsf{f}(x)$, then the (only) induction formula would be a tautology. However, the permissibility conjecture $\neg\mathsf{def}(\mathsf{f}(x)) \Rightarrow \varphi$ would not hold. Thus, permissibility conjectures are indeed needed to prevent the proof of false facts like φ. Thus, we obtain the following induction rule.

1″. Induction w.r.t. Algorithms

$$\frac{\begin{array}{c}\{\,\textsc{con}(r_i,\pi_{i,j}) = \mathsf{true} \;\wedge\; \varphi(s^*_{i,j}) \;\Rightarrow\; \varphi(t^*_i) \mid i = 1,\ldots,k,\; j = 1,\ldots,n_i\,\} \\ \{\,\textsc{con}(r,\pi_{i,1}) = \mathsf{false} \wedge \ldots \wedge \textsc{con}(r,\pi_{i,n_i}) = \mathsf{false} \;\Rightarrow\; \varphi(t^*_i) \mid i = 1,\ldots,k\,\} \\ \neg\mathsf{def}(f(x^*)) \;\Rightarrow\; \varphi(x^*)\end{array}}{\varphi(x^*)}$$

if f has the defining equations $f(t^*_i) = r_i$ for $i = 1,\ldots,k$, where $r_i|_{\pi_{i,j}} = f(s^*_{i,j})$ for $j = 1,\ldots,n_i$.

For example, induction w.r.t. minus using x and y as induction variables transforms (30) into the following conjectures. Here, $\varphi_{\mathrm{def}}(x,y,z)$ abbreviates the formula $\mathsf{def}((x-y)-z,(x-z)-y) = \mathsf{true}$ and $\varphi_{\mathrm{eq}}(x,y,z)$ abbreviates $(x-y)-z = (x-z)-y$. Moreover since $\textsc{con}(\mathsf{minus}(x,y),\epsilon)$ is true, we omitted it in the induction step formula and we also omitted the formula with the premise "$\textsc{con}(\mathsf{minus}(x,y),\epsilon) = \mathsf{false}$" (where ϵ denotes the top position).

$$\varphi_{\mathrm{def}}(\underline{x}, \underline{0}, z) \;\Rightarrow\; \varphi_{\mathrm{eq}}(\underline{x}, \underline{0}, z) \tag{33}$$

$$\big[\varphi_{\mathrm{def}}(\underline{x}, \underline{y}, z) \Rightarrow \varphi_{\mathrm{eq}}(\underline{x}, \underline{y}, z)\big] \Rightarrow \big[\varphi_{\mathrm{def}}(\underline{\mathsf{s}(x)}, \underline{\mathsf{s}(y)}, z) \Rightarrow \varphi_{\mathrm{eq}}(\underline{\mathsf{s}(x)}, \underline{\mathsf{s}(y)}, z)\big] \tag{34}$$

$$\neg\mathsf{def}(x - y) \;\Rightarrow\; \big[\varphi_{\mathrm{def}}(x, y, z) \Rightarrow \varphi_{\mathrm{eq}}(x, y, z)\big] \tag{35}$$

These formulas reflect the considerations from the beginning of Section 5. In fact, partial truth of the former step formula $\varphi_{\mathrm{eq}}(\underline{x}, \underline{y}, z) \Rightarrow \varphi_{\mathrm{eq}}(\underline{\mathsf{s}(x)}, \underline{\mathsf{s}(y)}, z)$, i.e.

$$(\underline{x} - \underline{y}) - z = (\underline{x} - z) - \underline{y} \;\Rightarrow\; (\underline{\mathsf{s}(x)} - \underline{\mathsf{s}(y)}) - z = (\underline{\mathsf{s}(x)} - z) - \underline{\mathsf{s}(y)}, \tag{21}$$

is only sufficient for the new induction step formula (34), if one can also prove that definedness of the (former) induction conclusion implies definedness of the (former) induction hypothesis, i.e. $\varphi_{\mathrm{def}}(\underline{\mathsf{s}(x)}, \underline{\mathsf{s}(y)}, z) \Rightarrow \varphi_{\mathrm{def}}(\underline{x}, \underline{y}, z)$ or, in other words,

$$\mathsf{def}((\underline{\mathsf{s}(x)} - \underline{\mathsf{s}(y)}) - z, (\underline{\mathsf{s}(x)} - z) - \underline{\mathsf{s}(y)}) \;\Rightarrow\; \mathsf{def}((\underline{x} - \underline{y}) - z, (\underline{x} - z) - \underline{y}).$$

Similar to induction w.r.t algorithms, a structural induction using the induction variable x only proves $\varphi(x)$ for instantiations of x with defined terms. In other words, in addition we also have to check whether the permissibility conjecture $\neg\mathsf{def}(x) \Rightarrow \varphi(x)$ holds.

2″. Structural Induction

$$\frac{\begin{array}{r}\{\varphi(x_{i,1}) \wedge \ldots \wedge \varphi(x_{i,n_i}) \;\Rightarrow\; \varphi(c_i(x_i^*)) \mid i = 1, \ldots, k\} \\ \neg\mathsf{def}(x) \;\Rightarrow\; \varphi(x)\end{array}}{\varphi(x)}$$

if x is a variable of a data type τ with the constructors $c_1, \ldots, c_k$, and if $x_{i,1}, \ldots, x_{i,n_i}$ are the variables of the data type τ occurring in x_i^*.

The rule for symbolic evaluation now 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 $\neg\mathsf{def}(\sigma(t^*)) \Rightarrow \varphi$.

3″. Symbolic Evaluation

$$\frac{\begin{array}{rl} & \varphi(\quad \sigma(r) \quad) \\ \neg\mathsf{def}(\sigma(t^*)) \;\Rightarrow & \varphi(\;\sigma(\,f(t^*)\,)\;)\end{array}}{\varphi(\;\sigma(\,f(t^*)\,)\;)}$$

if σ is a substitution and $f(t^*) = r$ is a defining equation.

Recall that for the base case in the induction proof of (30), we have to prove the formula

$$\mathsf{def}((\underline{x} - \underline{0}) - z,\ (\underline{x} - z) - \underline{0}) \ \Rightarrow\ (\underline{x} - \underline{0}) - z = (\underline{x} - z) - \underline{0}. \qquad (33)$$

For example, in order to evaluate the subterm $(x-z)-0$ to $x-z$, Rule $3''$ transforms (33) into

$$\mathsf{def}((x - 0) - z,\ x - z) \ \Rightarrow\ (x - 0) - z = x - z \quad \text{and}$$
$$\neg\mathsf{def}(x - z,\ 0) \ \Rightarrow\ (33).$$

In the rule for first order consequences, in addition to Ax^{data} we can now also use axioms about definedness, which state how def operates on terms built with algorithms, conditionals, and constructors. The soundness of these axioms follows from Lemma 7 and 8.

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

With these axioms about def, the permissibility conjectures and conjectures about definedness required by our calculus can often be proved without actually computing or even approximating the domain of f.

$4''$. First Order Consequence

$$\frac{\psi_1, \ldots, \psi_n}{\varphi}$$

if $Ax^{\mathrm{data}} \cup Ax^{\mathrm{def}} \vdash \psi_1 \wedge \ldots \wedge \psi_n \Rightarrow \varphi$.

The generalization rule again remains unchanged.

$5''$. Generalization

$$\frac{\varphi}{\sigma(\varphi)} \quad \text{where } \sigma \text{ is a substitution.}$$

Finally, for the conjecture $\varphi(\,\mathsf{if}(t_1, t_2, t_3)\,)$ a case analysis w.r.t. t_1 only proves the conjecture for instantiations where t_1 is defined. Hence, we also have to verify that the conjecture holds if $\mathsf{def}(t_1)$ is not true.

$6''$. Case Analysis

$$\frac{\begin{array}{rl} t_1 = \mathsf{true} & \Rightarrow\ \varphi(t_2) \\ t_1 = \mathsf{false} & \Rightarrow\ \varphi(t_3) \\ \neg\mathsf{def}(t_1) & \Rightarrow\ \varphi(\,\mathsf{if}(t_1, t_2, t_3)\,) \end{array}}{\varphi(\,\mathsf{if}(t_1, t_2, t_3)\,)}$$

Note that in particular, the rules $4''$ and $6''$ allow us to prove

$$\mathsf{def}(x_1, \ldots, x_n) \ \Rightarrow\ \mathsf{def}(x_i) \qquad (36)$$

for all $i = 1, \ldots, n$. Now we can finish the proof of (30), as symbolic

evaluation and Rule $4''$ transform all remaining proof obligations into trivial theorems, instantiations of (36), and the lemmata

$$\mathsf{def}(x - z) \;\Rightarrow\; \mathsf{s}(x) - z = \mathsf{s}(x - z), \tag{37}$$

$$\mathsf{def}((x - y) - z) \;\Rightarrow\; \mathsf{def}(x - z), \tag{38}$$

$$\mathsf{def}(x - z) \;\Rightarrow\; \mathsf{def}(\mathsf{s}(x) - z), \tag{39}$$

which in turn are verified by inductions w.r.t. minus. The following theorem proves the soundness of our calculus.

THEOREM 10. *If φ can be derived with the rules $1''$–$6''$, then $\forall x^*\ \varphi(x^*)$ is inductively true.*

Proof. Let I be the initial model from Definition 5. For each rule $\frac{\varphi_1,\ldots,\varphi_n}{\varphi}$ it can be shown that $I \models \varphi_1, \ldots, I \models \varphi_n$ implies $I \models \varphi$. For Rule $1''$, validity of its first premises implies $I \models \varphi(q^*)$ for all data objects (and hence, also for all ground terms) q^* where evaluation of $f(q^*)$ is defined (this can be proved similarly to Lemma 2). By Lemma 8, this is equivalent to $I \models \mathsf{def}(f(x^*)) \Rightarrow \varphi(x^*)$. Thus, validity of the permissibility conjecture $\neg\mathsf{def}(f(x^*)) \Rightarrow \varphi(x^*)$ implies $I \models \varphi(x^*)$. The soundness of the remaining rules is proved in an analogous way. □

Apart from partial correctness statements (of the form "φ holds if its evaluation is defined"), our calculus can also verify "*definedness conjectures*" like (38) and (39) which are often needed in both partial and total correctness proofs. In particular, the calculus can even perform termination proofs. For example, totality of the algorithm plus, i.e.

$$\mathsf{def}(x, y) \;\Rightarrow\; \mathsf{def}(\mathsf{plus}(x, y)), \tag{40}$$

can easily be proved by structural induction on x. But in addition to such definedness theorems the calculus can also verify *undefinedness*. For instance, the conjecture stating that div is always undefined if its second argument is 0,

$$y = \mathsf{0} \;\Rightarrow\; \neg\mathsf{def}(\mathsf{div}(x, y)) = \mathsf{true}, \tag{41}$$

can be proved by induction w.r.t. the partial algorithm div.

Moreover, the induction rules $1''$ and $2''$ can be strengthened further by allowing arbitrary instantiations of non-induction variables, cf. e.g. [6, 17, 42, 75]. So when proving $\varphi(x^*, y^*)$ by induction w.r.t. $f(x^*)$, in each induction formula

$$\varphi(s^*, y^*) \;\Rightarrow\; \varphi(t^*, y^*) \tag{42}$$

(resulting from a defining equation $f(t^*) = \ldots f(s^*) \ldots$), one may replace the non-induction variables y^* in the induction hypotheses by arbitrary terms r^*. Thus, instead of (42) one may construct an induction formula of the form

$$\varphi(s^*, r_1^*) \wedge \ldots \wedge \varphi(s^*, r_n^*) \Rightarrow \varphi(t^*, y^*).$$

Of course, a corresponding extension can also be used for structural inductions and for inductions w.r.t. algorithms with conditionals and with several recursive calls in a defining equation.

For example, when proving partial truth of $z \geq y \Rightarrow x - y \geq x - z$ by induction w.r.t. $x - y$ (where "$\geq$" abbreviates ge), in the induction hypothesis the non-induction variable z may be instantiated by $\mathsf{p}(z)$. Here, p computes the predecessor of a natural number, where $\mathsf{p}(\mathsf{0}) = \mathsf{0}$. Then a subsequent induction (resp. case analysis) w.r.t. p can be used to solve the remaining proof obligations. Thus, with this extension one can for instance model the heuristic of [6, 42] to *merge* the suggested induction relations of $x - y$ and $x - z$. Moreover, this extension also allows the use of other refined techniques for the generation of suitable induction relations, cf. e.g. [6, "measured subsets"], [75, "domain generalization"], and [42, "non-basic induction schemes"]. Note that an arbitrary instantiation of non-induction variables could not be done in our first calculus for partial functions from Section 3 and 4, since in this calculus an instantiation of non-induction variables y^* with undefined terms r^* would enable the proof of false facts.

To conclude, by explicit reasoning about definedness it is possible to perform more refined inference steps than those allowed by the calculus of Section 3 and 4. On the other hand, the new calculus 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 Section 3 and 4 instead.

The calculus of the present section represents a very powerful method for induction proofs about partial functions. 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 one of the rules $1''$, $2''$, $3''$, or $6''$ is applied. Hence, the existing induction provers can easily be extended to this calculus and in this way, these systems can be directly used to reason about definedness of partial functions. In particular, they may now perform an induction w.r.t. partial functions whenever the corresponding permissibility conjecture can be verified.

6. Computation of Domains

In the calculus of the previous section we introduced an explicit (object-level) notion of definedness using the function def. Hence, now definedness of a function (or in particular, its termination) can also be proved within the calculus. However, while termination of plus can indeed be verified, a similar conjecture for div, viz.

$$\mathsf{def}(x, y) \wedge \neg y = \mathsf{0} \;\Rightarrow\; \mathsf{def}(\mathsf{div}(x, y)), \tag{43}$$

is problematic for the calculus. The reason is that here we would need an induction w.r.t. the recursions of div. But this induction cannot be done using Rule $1''$, because we cannot prove the corresponding permissibility conjecture which states that (43) also holds under the condition $\neg\mathsf{def}(\mathsf{div}(x, y)) = \mathsf{true}$. In fact, the only induction possible for conjecture (43) is structural induction or an induction w.r.t. an algorithm whose termination is already verified for $y \neq \mathsf{0}$. Hence, we need a method to ensure well-foundedness of relations *outside* of our calculus. This is similar to the calculi in Section 2 and 3, which required totality of functions in the conditions of some rules, but where totality had to be verified outside of the calculus, too.

Thus, in this section we present a method to analyze the definedness of possibly 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 the evaluation of $f(t^*)$ is defined. Thus, θ_f is a total function specifying the domain of f. Our aim is to synthesize[7] domain predicates which return true as often as possible, but of course in general this goal cannot be reached as the domains of functions are undecidable. As domain predicates themselves are always total, we do not construct domain predicates θ_{θ_f} of domain predicates.

In our calculus we now introduce additional axioms Ax^{dom}. For every algorithm f with the domain predicate θ_f, Ax^{dom} contains the axiom

$$\theta_f(x^*) \;\Rightarrow\; \mathsf{def}(f(x^*)) \tag{44}$$

which states that the truth of θ_f is sufficient for definedness of f, i.e., the domain predicate θ_f is *partially correct*. Moreover, for each such θ_f, Ax^{dom} also contains the following axiom which states that domain predicates are *total* functions.

$$\mathsf{def}(x^*) \;\Rightarrow\; \mathsf{def}(\theta_f(x^*)) \tag{45}$$

[7] Strictly speaking, we synthesize *algorithms* which compute domain predicates. For the sake of brevity we also refer to these algorithms as "domain predicates".

To use these additional axioms, the fourth rule of our calculus is refined by allowing the inference $\frac{\psi_1,\ldots,\psi_n}{\varphi}$ whenever $Ax^{\mathrm{data}} \cup Ax^{\mathrm{def}} \cup Ax^{\mathrm{dom}} \vdash \psi_1 \wedge \ldots \wedge \psi_n \Rightarrow \varphi$ holds. Now conjecture (43) can easily be proved, provided that θ_{div} has the defining equations $\theta_{\mathsf{div}}(x, \mathsf{0}) = \mathsf{false}$ and $\theta_{\mathsf{div}}(x, \mathsf{s}(y)) = \mathsf{true}$. Then Rule $4''$ transforms (43) into

$$\mathsf{def}(x, y) \wedge \neg y = \mathsf{0} \;\Rightarrow\; \theta_{\mathsf{div}}(x, y).$$

Subsequent induction (resp. case analysis) w.r.t. the algorithm θ_{div} and symbolic evaluation of θ_{div} results in trivial theorems and consequences of Ax^{dom}. So in this way, the original conjecture (43) is verified, i.e., now the calculus is able to prove the well-foundedness of "new" relations.

Moreover, for algorithms f whose totality can be verified by existing automated methods, we would have $\theta_f(x^*) = \mathsf{true}$. Then the above refinement allows us to prove all definedness conditions about f immediately (since the axiom (44) would now be $\mathsf{true} \Rightarrow \mathsf{def}(f(x^*))$). Thus, if all algorithms are total and this totality is reflected in the corresponding domain predicates, then compared to the standard techniques for induction proofs, the calculus of Section 5 does not result in any significant extra proof obligations.

Of course, we still have to explain how domain predicate algorithms like θ_{div} can be generated automatically. For that purpose the termination behaviour of the algorithms under consideration is analyzed.

While most work on automated termination proofs has been done for *term rewriting systems* (for surveys see e.g. [20, 73]) and *logic programs* (e.g. [19, 64, 74]), only a few methods have been developed for *functional programs* (e.g. [6, 24, 25, 26, 76]). These approaches aim to prove that an algorithm terminates for *each* input. However, together with *J. Brauburger* we developed a technique to adapt these methods for automated termination analysis of *partial* functions, cf. [11, 12, 13, 28].

A first approach to generate a domain predicate θ_{div} would be to replace the results of div's non-recursive cases by true and to replace div's recursive call by a corresponding call of θ_{div}.

$$\begin{aligned} &\textit{function}\ \theta_{\mathsf{div}} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool} \\ &\quad \theta_{\mathsf{div}}(\mathsf{0}, \mathsf{s}(y)) \;=\; \mathsf{true} \\ &\quad \theta_{\mathsf{div}}(\mathsf{s}(x), y) \;=\; \mathsf{if}(\,\mathsf{ge}(\mathsf{s}(x), y),\ \theta_{\mathsf{div}}(\mathsf{minus}(\mathsf{s}(x), y), y),\ \mathsf{true}) \end{aligned}$$

This algorithm returns true whenever its evaluation is defined. As the patterns and recursions of θ_{div} correspond to those of div, the algorithm div is defined whenever θ_{div} yields true. Hence, the above algorithm for θ_{div} is in fact *partially correct*, cf. (44).

However, this algorithm does not compute a total function (i.e., (45) does not hold). First of all, θ_{div} is not completely defined. For

every set of (non-overlapping and linear) patterns $t_1^*, \ldots, t_n^*$ one can easily compute a set of (linear) *missing patterns* $t_1'^*, \ldots, t_m'^*$ such that the whole set of patterns $t_1^*, \ldots, t_n^*, t_1'^*, \ldots, t_m'^*$ is non-overlapping and complete, i.e., for every tuple of data objects q^* there exists a pattern t_i^* or $t_j'^*$ such that $q^* = \sigma(t_i^*)$ or $q^* = \sigma(t_j'^*)$ for some substitution σ, cf. e.g. [38].[8] Thus, the definition of a domain predicate θ_f should contain the equation $\theta_f(t'^*) = \mathsf{false}$ for all missing patterns t'^*. In our example, this results in the additional defining equation $\theta_{\mathsf{div}}(\mathsf{0}, \mathsf{0}) = \mathsf{false}$.

Second, we have to ensure that in the domain predicates, auxiliary algorithms like minus are only called with inputs from their domains. Of course, in the above algorithm the condition $\mathsf{ge}(\mathsf{s}(x), y)$ already ensures definedness of $\mathsf{minus}(\mathsf{s}(x), y)$, but in general the definedness of auxiliary algorithms has to be guaranteed separately. So in general, before calling $\mathsf{minus}(\mathsf{s}(x), y)$ we have to check whether $\theta_{\mathsf{minus}}(\mathsf{s}(x), y)$ yields true. Hence, the result of the last defining equation is changed to

$$\mathsf{if}(\,\mathsf{ge}(\mathsf{s}(x), y),\ \mathsf{if}(\theta_{\mathsf{minus}}(\mathsf{s}(x), y)), \theta_{\mathsf{div}}(\mathsf{minus}(\mathsf{s}(x), y), y), \mathsf{false}),\ \mathsf{true}).$$

But a third problem is that θ_{div} does not terminate. In other words, no well-founded ordering $\succ$ satisfies θ_{div}'s *termination hypothesis*

$$\mathsf{ge}(\mathsf{s}(x), y) = \mathsf{true} \ \Rightarrow\ \langle \mathsf{s}(x),\ y\rangle \ \succ\ \langle \mathsf{minus}(\mathsf{s}(x), y),\ y\rangle.$$

The central idea to transform the above algorithm into a terminating one is to choose some well-founded ordering $\succ$ and to enter the recursive call $\theta_{\mathsf{div}}(\mathsf{minus}(\mathsf{s}(x), y), y)$ only if $\langle \mathsf{s}(x),\ y\rangle \ \succ\ \langle \mathsf{minus}(\mathsf{s}(x), y),\ y\rangle$ holds and to return false otherwise.

A successful heuristic for the choice of $\succ$ is to use a well-founded ordering which satisfies at least the *non-strict* unconditional version of the termination hypothesis $\langle \mathsf{s}(x),\ y\rangle \ \succeq\ \langle \mathsf{minus}(\mathsf{s}(x), y),\ y\rangle$ and which satisfies the strict termination hypothesis "as often as possible". Such orderings can be determined automatically by existing approaches for termination analysis, cf. e.g. [25, 28]. For example, one may use an ordering $\succ_{\#}$ which compares pairs by the *size* of their first arguments (where the *size* of an object of type nat is the number it represents, i.e., the number of its s-occurrences). Hence, we demand that $\theta_{\mathsf{div}}(x, y)$ may only enter its recursive call if $\langle \mathsf{s}(x), y\rangle \ \succ_{\#}\ \langle \mathsf{minus}(\mathsf{s}(x), y), y\rangle$ holds.

Now the method of [25, 28] can also compute a sufficient and necessary requirement for $\langle \mathsf{s}(x), y\rangle \ \succ_{\#}\ \langle \mathsf{minus}(\mathsf{s}(x), y), y\rangle$, viz. a so-called *difference equivalent* $\Delta_{\succ_{\#}}(\langle \mathsf{s}(x), y\rangle, \langle \mathsf{minus}(\mathsf{s}(x), y), y\rangle)$, which is equivalent to $\mathsf{ge}(y, \mathsf{1})$ in our example. In this way we obtain the following domain predicate algorithm where "$t_1 \mathbin{\&} t_2$" abbreviates "$\mathsf{if}(t_1, t_2, \mathsf{false})$" and "$t_1 \mathbin{\&} t_2 \mathbin{\&} \ldots$" abbreviates "$t_1 \mathbin{\&} (t_2 \mathbin{\&} \ldots)$" to ease readability.

[8] Essentially, the reason is that the depth of the missing patterns is bounded by the maximum depths of function symbols in the patterns $t_1^*, \ldots, t_n^*$.

$$
\begin{array}{l}
\mathit{function}\ \theta_{\mathsf{div}} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool} \\
\quad \theta_{\mathsf{div}}(0,0) \ = \ \mathsf{false} \\
\quad \theta_{\mathsf{div}}(0,\mathsf{s}(y)) \ = \ \mathsf{true} \\
\quad \theta_{\mathsf{div}}(\mathsf{s}(x),y) \ = \ \mathsf{if}(\,\mathsf{ge}(\mathsf{s}(x),y), \\
\qquad\qquad \theta_{\mathsf{minus}}(\mathsf{s}(x),y) \mathbin{\mathbb{\wedge}} \mathsf{ge}(y,1) \mathbin{\mathbb{\wedge}} \theta_{\mathsf{div}}(\mathsf{minus}(\mathsf{s}(x),y),y), \\
\qquad\qquad \mathsf{true})
\end{array}
$$

Here, the difference equivalent $\mathsf{ge}(y,1)$ guarantees that evaluation of θ_{div} can only lead to a recursive call if the arguments of the recursive call are $\succ_{\#}$-smaller than the original input. In a similar way, we obtain a domain predicate for minus where we omitted the difference equivalent $\Delta_{\succ_{\#}}(\langle \mathsf{s}(x),\, \mathsf{s}(y)\rangle,\, \langle x,\, y\rangle)$ because it is equivalent to true.

$$
\begin{array}{l}
\mathit{function}\ \theta_{\mathsf{minus}} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool} \\
\quad \theta_{\mathsf{minus}}(x,0) \ = \ \mathsf{true} \\
\quad \theta_{\mathsf{minus}}(0,\mathsf{s}(y)) \ = \ \mathsf{false} \\
\quad \theta_{\mathsf{minus}}(\mathsf{s}(x),\mathsf{s}(y)) \ = \ \theta_{\mathsf{minus}}(x,y)
\end{array}
$$

The above algorithms indeed define domain predicates for div and minus, i.e., they compute *total* functions and their truth is sufficient for the definedness of div and minus, respectively. In fact, $\theta_{\mathsf{minus}}(x,y)$ is true iff x is greater than or equal to y and $\theta_{\mathsf{div}}(x,y)$ is true iff y is greater than 0. Hence, in these examples we have even generated the weakest possible domain predicates, as θ_{minus} and θ_{div} return true not only for a subset but for *all* elements of the domains of minus and div.

To automate the above construction of domain predicates we associate a boolean term $\Theta(t)$ with each term t (not containing the symbol def) such that evaluation of $\Theta(t)$ is always defined and evaluation of t is defined whenever $\Theta(t)$ is true.[9] $\Theta(t)$ is called a *domain condition* for t and its definition is based on the domain predicates:

$$
\begin{array}{lllll}
\text{(i)} & \Theta(x) & :\equiv \mathsf{true} & & \text{for variables } x, \\
\text{(ii)} & \Theta(\mathsf{if}(s_1,s_2,s_3)) & :\equiv \Theta(s_1) \mathbin{\mathbb{\wedge}} \mathsf{if}(s_1,\, \Theta(s_2),\, \Theta(s_3)) & & \\
\text{(iii)} & \Theta(c(s_1 \ldots s_n)) & :\equiv \Theta(s_1) \mathbin{\mathbb{\wedge}} \ldots \mathbin{\mathbb{\wedge}} \Theta(s_n) & & \text{for constructors } c, \\
\text{(iv)} & \Theta(g(s_1 \ldots s_n)) & :\equiv \Theta(s_1) \mathbin{\mathbb{\wedge}} \ldots \mathbin{\mathbb{\wedge}} \Theta(s_n) \mathbin{\mathbb{\wedge}} \theta_g(s_1 \ldots s_n) & & \text{for algorithms } g.
\end{array}
$$

Thus, if f and g are algorithms, then $\Theta(f(g(t)))$ is $\Theta(t) \mathbin{\mathbb{\wedge}} \theta_g(t) \mathbin{\mathbb{\wedge}} \theta_f(g(t))$. So indeed, domain conditions are always defined, because $\theta_g(t)$ is only evaluated if $\Theta(t)$ holds.

Now if $f(t^*) = r$ is a defining equation of an algorithm f, then our aim is to use $\theta_f(t^*) = \Theta(r)$ as a defining equation for the corresponding domain predicate. However, for termination of θ_f we also have to

[9] More precisely, this implication holds for each substitution σ of t's variables by data objects: For all such σ, evaluation of $\sigma(\,\Theta(t)\,)$ is defined and $\sigma(\,\Theta(t)\,) = \mathsf{true}$ implies that the evaluation of $\sigma(t)$ is also defined.

guarantee that the arguments of recursive calls are smaller than the corresponding inputs. Therefore, for recursive calls in a defining equation $f(t^*) = \ldots f(s^*) \ldots$ we add a requirement to the domain condition of $f(s^*)$ which ensures $t^* \succ s^*$. For that purpose, instead of (iv) we use

(v) $\quad \Theta(f(s_1 \ldots s_n)) \;:\equiv\; \Theta(s_1) \wedge \ldots \wedge \Theta(s_n) \wedge \Delta_{\succ}(t^*, s^*) \wedge \theta_f(s_1 \ldots s_n)$

when computing the domain conditions of f's result terms. So we obtain the following construction principle for domain predicates.

From each defining equation $f(t^*) = r$ of f construct a defining equation $\theta_f(t^*) = \Theta(r)$ for θ_f. For each missing pattern t'^* of f construct a defining equation $\theta_f(t'^*) = \mathsf{false}$ for θ_f.

This construction of domain predicates can be automated directly. For instance, the above algorithms θ_{div} and θ_{minus} were built according to this principle (where we omitted the domain predicate θ_{ge} for the total function ge and replaced terms like $\mathsf{if}(\mathsf{true}, t, \ldots)$ by t).

To ease subsequent reasoning, we also developed a procedure to *simplify* the generated domain predicates. For instance, by the technique of *recursion elimination* [76], the algorithm for θ_{div} can be simplified to

$$\begin{aligned} &\mathit{function}\ \theta_{\mathsf{div}} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool} \\ &\quad \theta_{\mathsf{div}}(x, \mathsf{0}) \;=\; \mathsf{false} \\ &\quad \theta_{\mathsf{div}}(x, \mathsf{s}(y)) \;=\; \mathsf{true}. \end{aligned}$$

See [11, 13] for a description of the simplification techniques and for a collection of examples which demonstrate that this approach is indeed able to generate sophisticated domain predicates (e.g., for quot our method synthesizes the domain predicate "divides", for a logarithm algorithm it generates a domain predicate which checks if one number is a power of another one, for a delete algorithm a domain predicate for list membership is synthesized, etc.). For all these examples the generated domain predicate is not only sufficient for definedness, but it even describes the *exact* domain of the function.

7. Applications of our Calculi

In this section we analyze areas for possible applications of our results, cf. Figure 1. At first sight, 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

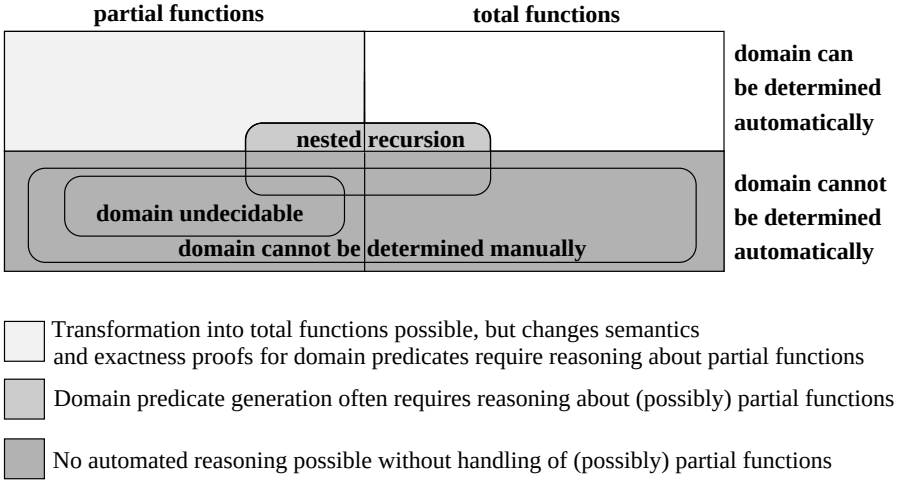

Figure 1. The use of our calculus for different classes of functions

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 changes the semantics of f. For example, the partial correctness statement (6) for minus does not hold any more if minus is replaced by a total extension minus'. On the other hand, conjectures like $\mathsf{minus}(0,1) = \mathsf{minus}(3,5)$ would now be considered as true (because minus' returns the same default value for all inputs where minus is undefined), although the algorithm for minus provides no evidence for this conjecture. Hence, many authors advocate that it is preferable to keep partial functions instead of their total extensions, cf. e.g. [1, 21, 46, 47, 51, 77].

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. The reason is that if evaluation of $f(t)$ leads to a nested recursive call $f(f(r))$, then one has check whether both arguments r and $f(r)$ are smaller than the corresponding input t. In other words, one has to prove $t \succ r$ and $t \succ f(r)$ for some well-founded ordering $\succ$ (resp. one has to generate corresponding difference equivalents). But the statement $t \succ f(r)$ contains the function f which may be partial (as we have not yet verified termination of its algorithm). For that reason previously developed methods for automated termination analysis of functional programs often failed for nested and mutual recursion [6, 25, 76]. How-

ever, our calculi allow us to prove termination of algorithms with nested or mutual recursion without having to verify their correctness simultaneously. In fact for these proof obligations $t \succ f(r)$, the calculus of Section 3 and 4 is often already sufficient, since for most algorithms, in their termination proof one only has to deal with conjectures containing at most one occurrence of a possibly partial function. Thus, by combining the existing techniques for automated termination proofs with this calculus, these techniques can be directly extended to nested and mutual recursion. This for instance 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 [26].[10]

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 f's exact domain, one may again apply our calculus from Section 5. For example, if θ_{div} is defined as in Section 6, then $\mathsf{def}(\mathsf{div}(x, y)) \Rightarrow \theta_{\mathsf{div}}(x, y)$ 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* [67] is total, but its termination is a "deep theorem" [61] 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 [27] we applied our calculi on a large benchmark of conjectures from the area of *term rewriting systems*. In this case study, we regard a data type termlist for lists of terms[11] and to represent substitutions, we use termlists of the form

[10] For termination proofs one only has to verify statements of a special form (viz. inequalities $t \succ s$). Therefore, to control the application of the rules $1'$– $6'$, here it is advantageous to use a very restricted version of our calculus which is especially suited for the verification of such inequalities, e.g., the approach suggested in [26].

[11] The reason for using just one data type of termlists (instead of two separate *mutually recursive* types for terms and termlists) is that this formalization simplifies the proofs considerably. Techniques for automated reasoning about mutually recursive data types and algorithms can for instance be found in [4, 14, 26, 43].

"⟨variable, term, variable, term, ...⟩". Then $\mathsf{is\_subst}(\sigma)$ checks whether σ is a termlist of this form, $\mathsf{apply}(\sigma, l)$ applies the substitution σ to all terms in l, and unifies is Robinson's unification algorithm. The following theorem states that $\mathsf{unifies}(l, k)$ returns true whenever the termlists l and k are unifiable.

$$\mathsf{is\_subst}(\sigma) \wedge \mathsf{apply}(\sigma, l) = \mathsf{apply}(\sigma, k) \;\Rightarrow\; \mathsf{unifies}(l, k) \qquad (46)$$

In contrast to previous correctness proofs of the unification algorithm (e.g. [55, 61]), our calculi can prove the partial truth of (46) by induction w.r.t. unifies without having to verify its termination, cf. [27]. Indeed, this proof is already possible with the calculus from Section 3 and 4 if the auxiliary function apply is total. Thus, the proof can even be done without any reasoning about definedness. So the ability of our calculi 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 (attributed to Collatz) [23] where totality is still an open question, i.e., algorithms whose domain has not even been determined *manually*.

Moreover, 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, regard the following algorithm $\mathsf{rewrites^*}(l, k, R)$ which returns true iff there exist $t \in l$ and $s \in k$ such that $t \to_R^* s$. Here, term rewriting systems $t_1 \to s_1,\ t_2 \to s_2, \ldots$ are represented as termlists $\langle t_1, s_1, t_2, s_2, \ldots \rangle$ and $\mathsf{rewrite}(l, R)$ is an auxiliary algorithm which generates a list of all those terms s which result from a term $t \in l$ by one rewrite step with R (i.e., $s \in \mathsf{rewrite}(l, R)$ iff $t \to_R s$ for some $t \in l$).

$$\begin{array}{l}\textit{function}\ \mathsf{rewrites^*} : \mathsf{termlist} \times \mathsf{termlist} \times \mathsf{termlist} \to \mathsf{bool}\\ \quad \mathsf{rewrites^*}(l, k, R) = \mathsf{if}(\mathsf{disjoint}(l, k),\\ \qquad\qquad\qquad \mathsf{if}(\mathsf{rewrite}(l, R) \subseteq l,\\ \qquad\qquad\qquad\quad \mathsf{false},\\ \qquad\qquad\qquad\quad \mathsf{rewrites^*}(\mathsf{rewrite}(l, R), k, R)),\\ \qquad\qquad\qquad \mathsf{true})\end{array}$$

The domain of this algorithm is obviously undecidable. Hence, if one wants to prove any conjecture about this algorithm, one definitely needs a method to deal with partial functions. For example, one might try to prove that rewrites* is stable under subsets.

$$\mathsf{rewrites^*}(l, k_1, R) = \mathsf{true} \wedge k_1 \subseteq k_2 \;\Rightarrow\; \mathsf{rewrites^*}(l, k_2, R) \qquad (47)$$

With the calculus of Section 5 one can prove the truth (not just the *partial* truth) of this conjecture by induction w.r.t. the partial function rewrites* using l, k_1, R as induction variables. The corresponding permissibility conjecture obviously holds (and it can be proved by Rule $4''$) as $\mathsf{rewrites^*}(l, k_1, R) = \mathsf{true}$ implies $\mathsf{def}(\mathsf{rewrites^*}(l, k_1, R)) = \mathsf{true}$. After generating the induction formulas, Rule $6''$ is used to perform suitable case analyses. If $\mathsf{disjoint}(l, k_1) = \mathsf{true}$ and $\mathsf{rewrite}(l, R) \not\subseteq l$, then symbolic evaluation of rewrites* transforms the induction conclusion into the induction hypothesis. The other cases can be proved using a straightforward lemma about disjoint and "$\subseteq$".

In a similar way one can define an algorithm $\mathsf{joinable}(l, k, R)$ which is true iff there exist terms $t \in l$, $s \in k$, and q such that $t \to_R^* q \leftarrow_R^* s$ and an algorithm $\mathsf{jcp}(R)$ which checks whether all critical pairs of R are joinable. Of course, the domains of joinable and jcp are again undecidable. Moreover, let $\mathsf{trs}(R)$ be a function which determines whether R represents a proper term rewriting system. Then with the calculus of Section 5 we also proved partial truth of a variant of *D. E. Knuth* and *P. B. Bendix*' critical pair lemma [49] which states that if all critical pairs of a (possibly non-terminating, cf. [34]) term rewriting system are joinable, then the system is locally confluent.

$$\mathsf{trs}(R) \wedge \mathsf{jcp}(R) \wedge \mathsf{length}(l) = 1 \wedge \mathsf{rewrites}(l, k_1, R) \wedge \mathsf{rewrites}(l, k_2, R) \Rightarrow \mathsf{joinable}(k_1, k_2, R)$$

The proof of this fundamental theorem used several inductions w.r.t. functions like rewrites* whose domains are undecidable.[12]

In our case study [27], inductions w.r.t. partial functions (or functions like unifies where automated termination proofs fail) were used for about 40 % of the conjectures considered. We experimented with mechanized support by the induction theorem prover INKA [36, 75], but we did not modify its implementation in a significant way. (Essentially, the only modification needed for the calculus of Section 5 was to trace all applications of induction, symbolic evaluation, and case analysis and to generate the corresponding permissibility conjectures, which had to be proved as well subsequently.) In particular, we did not develop any refinements to its general theorem proving capabilities, since our aim only was to demonstrate that the restrictions imposed by the rules of Section 5 are not "significant", i.e., they still allow the proof of (almost all) interesting theorems. Therefore, in our case study many proof steps (in particular, providing the lemmata required) were done by the human. For further details and for a collection of numerous other theorems about partial functions proved with our calculus see [27].

[12] Thus, our proof differs substantially from other case studies in related areas (e.g., the proofs of the Church-Rosser theorem for the λ-calculus in [59, 69]).

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.

As an example consider an imperative program which computes the truncated logarithm $\lfloor \log_2(x) \rfloor$ w.r.t. base 2 in its result variable r (where r is 0 for $x = 0$). It uses the auxiliary functions neq (for non-equality), odd, and the predecessor function p.

$r := 0;$
while $\mathsf{neq}(x, 0)$
do *if* $\mathsf{odd}(x)$ *then* $x := \mathsf{p}(x)$
else $r := \mathsf{s}(r);$
$z := 0;$
while $\mathsf{neq}(x, z)$
do $x := \mathsf{p}(x);$
$z := \mathsf{s}(z)$ *od*
fi *od*

} sets x to $\frac{x}{2}$

To translate this program into a functional one, every *while*-loop is transformed into a separate function, cf. [56]. So for the *inner while*-loop we obtain a function mean which takes the input values of x and z as arguments. If the loop-condition is satisfied (i.e., if $\mathsf{neq}(x, z)$ is true), then mean is called recursively with the new values of x and z. Otherwise, mean returns the value of x. (Of course, a similar function computing the output value of z could also be constructed.)

function $\mathsf{mean} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$
$\mathsf{mean}(x, z) \;=\; \mathsf{if}(\mathsf{neq}(x, z),\ \mathsf{mean}(\mathsf{p}(x), \mathsf{s}(z)),\ x)$

The algorithm $\mathsf{mean}(x, z)$ computes the arithmetical mean $\frac{x+z}{2}$ of two numbers provided that $x - z$ is a non-negative even number. Otherwise, the algorithm mean does not terminate. Using this function, the outer *while*-loop is translated into the function $\mathsf{while}(x, r)$ which computes $r + \lfloor \log_2(x) \rfloor$.

function $\mathsf{while} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$
$\mathsf{while}(x, r) \;=\; \mathsf{if}(\mathsf{neq}(x, 0),$
$\mathsf{if}(\mathsf{odd}(x),\ \mathsf{while}(\mathsf{p}(x), r),\ \mathsf{while}(\mathsf{mean}(x, 0), \mathsf{s}(r))),$
$r)$

Now the whole imperative program is transformed into a function log with the defining equation $\mathsf{log}(x) = \mathsf{while}(x, 0)$. Note that even if the original imperative program is terminating, in general the auxiliary functions resulting from such a translation are *partial*. The reason is that in imperative programs, termination of *while*-loops often depends on their contexts. For instance, in our example the inner *while*-loop is only entered with an *even* input x. However, this restriction on x is not present in the function mean. Therefore log and while are total, but their auxiliary function mean is partial, as $\mathsf{mean}(x, z)$ only terminates iff $x \geq z$ and if $x - z$ is even.

For the verification of an imperative program (resp. of the corresponding functional one) one has to prove lemmata about its auxiliary functions. For instance, let us verify partial truth of the conjecture

$$\mathsf{neq}(x, 0) \;\Rightarrow\; \mathsf{ge}(x, \mathsf{exp}(2, \mathsf{log}(x)))$$

which can be generalized to the following conjecture about while

$$\mathsf{neq}(x, 0) \;\Rightarrow\; \mathsf{ge}(\mathsf{times}(\mathsf{exp}(2, r), x), \mathsf{exp}(2, \mathsf{while}(x, r)))$$

(where $\mathsf{exp}(x, y)$ computes x^y). However, for this proof one needs the lemmata $\mathsf{ge}(\mathsf{mean}(x, z), z) = \mathsf{true}$ and $\mathsf{times}(2, \mathsf{mean}(x, z)) = \mathsf{plus}(x, z)$ about the partial function mean. Their partial truth (i.e., their truth under the premise $\mathsf{def}(\mathsf{mean}(x, z))$) can now be proved by induction w.r.t. mean using the calculus of Section 5.

Hence, to use induction theorem provers for the verification of imperative programs, these programs can be translated into functional ones. But as the resulting functional programs are usually partial, one again needs a method for induction proofs with partial functions.

Summing up, our calculi for reasoning about partial functions are needed for all algorithms which correspond to the shaded areas in Figure 1. These algorithms constitute an important class used in many areas and inductions w.r.t. partial functions are required for many fundamental theorems about such algorithms.

8. Related Work

In this section we give a detailed survey on related work. We first discuss alternative notions of "truth" for partial functions in Section 8.1. Then we comment on other techniques for automated reasoning with partial functions. Section 8.2 focuses on approaches based on denotational semantics, whereas in Section 8.3 we compare our approach with the treatment of partial functions in existing induction theorem provers.

8.1. 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. [58, 66] for an overview and alternatives). For example, our notion of *partial* truth corresponds to validity in the initial partial model of the specification with a "weak equality" semantics, cf. [50]. Here, an equation $t = s$ is considered to be valid if t or s is undefined or if both are interpreted the same.

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. [21]). But one may also use a formalization with a three-valued logic [48], where the truth value of formulas depending on undefined terms is "undefined". See [46, 47] for a mechanization of this last 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 by *C.-P. Wirth* and *B. Gramlich* in [77, "Type E"] and further elaborated in [51]. However, while the definitions are "equivalent", we used a different formalization, as we did not define the semantics of the definedness function def on the meta-level. Based on Wirth and Gramlich's analysis, inference systems were presented in [1] and [78]. These approaches can also deal with non-termination, but in contrast to our work, they do not focus on determining suitable induction relations automatically (by deriving them from the recursions of the algorithms).

An alternative notion of truth has been proposed by *D. Kapur* and *D. R. Musser* [39, 40] and a corresponding definition is also used by *C. Walther* in [75]. Here, an equation is considered to be inductively true if it holds in the intersection of all maximal congruence relations satisfying the specification (where a specification consists of a set of equations and a set of ground terms which must not be equal to each other). Thus, an incompletely specified function is interpreted as the set of all possible complete and consistent extensions, cf. also [77, "Type D'"]. This corresponds to the intuition that such a function is not really *partial*, but it is a total function with (partly) unknown behaviour. However, there is a problem when dealing with non-termination, because

a function like $\mathsf{f}(x) = \mathsf{s}(\mathsf{f}(x))$ may fail to have a complete consistent extension. In contrast, in our approach every specification is consistent (because the initial model of Definition 5 always exists). Hence, we can handle non-termination without any consistency checks. For a further discussion on the differences between the semantics see e.g. [1, 39, 77].

Finally, there are also suggestions to deal with partial functions in a total *higher order* logic, e.g. [22]. In this approach partial functions are regarded as total functions whose behaviour is only specified on the intended domain and which return an arbitrary value if applied to other arguments. To define the domains of functions a (non-constructive) notion of domain predicates is used. However, this approach cannot model proper call-by-value semantics. The reason is that if $\mathsf{quot}(1, 0)$ is equal to some (arbitrary) natural, then $\mathsf{times}(\mathsf{quot}(1, 0), 0)$ evaluates to 0 although it should be undefined in an eager language.

8.2. Proof Techniques Based on Denotational Semantics

In this paper, we gave an *algebraic* and an *operational* semantics for our programming language and presented a calculus for inductions on the objects of the data structures. Similar approaches are used in most induction provers (i.e., in systems with powerful heuristics especially designed for induction). However, many general purpose tools for reasoning about programs are based on *denotational* semantics instead.

Although call-by-value is not a fixpoint computation rule [54], the semantics of our programming language can also be defined in a denotational way by using an appropriate program transformation. The classical technique for proofs about denotational semantics is *computational induction* (e.g. *D. Scott*'s *fixpoint induction* [68]). However, an important problem is that computational induction is only sound for so-called *admissible* formulas. Hence, systems based on denotational semantics usually have complicated tests for admissibility (which still reject many admissible formulas) [62, p. 70]. Moreover, a full formalization of denotational semantics requires a *higher order* logic (as it is for instance used in LCF [31, 62] and its descendants HOL [32] and ISABELLE [63]). Of course, compared to provers working on a first order language, these systems are much more expressive, but in general, higher order logics often raise harder problems for automation [5].

For that reason, an alternative formalization has been suggested by *N. Shankar* who defined an LCF-like calculus using first order logic [70, 71]. Here, lambda and apply are binary function symbols, symbols like plus are nullary, and algorithms are defined via a least fixpoint operator fix. Hence, the definition of plus has the form $\mathsf{plus} = \mathsf{apply}(\mathsf{fix}, \mathsf{lambda}(f, \ldots))$. Shankar's induction rule is Scott's fixpoint

induction, which can now be stated as a rule on first order formulas. So the requirement of a higher order logic is not necessary for the development of an axiomatic framework based on denotational semantics.

Denotational semantics and the computational induction principle are a powerful logical framework for reasoning about programs. However, "experience shows that the use of this principle is not always natural" [52, p. 181]. Thus, for an *automation* of induction proofs, an induction on the *values* of the program variables is preferable, since "fixpoint induction definitely seems more painful and less intuitive to use" [7, p. 3] (cf. also the discussions in [6, 10, 57]). Thus, even in systems like LCF where fixpoint induction forms the basis of the underlying calculus, one immediately derives rules for structural induction, which are usually much better suited for subsequent verification tasks.

But while *structural* induction is easy to automate, it often is not powerful enough. Hence, one needs *Noetherian* induction to allow the use of arbitrary *well-founded* induction relations on the objects of the data structures. To find suitable induction relations automatically, a successful heuristic is to choose relations which correspond to the recursions of algorithms occurring in the conjecture [6, 15, 75, 79].

This approach is not only used in most (explicit) induction provers (e.g. NQTHM [6], RRL [41, 79], CLAM [16, 17], INKA [36, 75]), but it has also been implemented in systems like HOL, ISABELLE, and LAMBDA, cf. [5, 18, 72]. This may indicate that even in provers for higher order logics, Noetherian induction on the data structure seems to be better suitable for automation than computational induction (see also [61]).

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 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. Hence, the restriction only to derive induction relations from terminating algorithms is no longer necessary.

While we proved our claim using algebraic semantics, our observation could of course also be justified using denotational semantics. In fact, Rule $1''$ transfers the idea of computational induction to an induction on the data structure.[13] This is possible, as we regard an

[13] The connection between induction w.r.t. algorithms and computational induction was already investigated in [57]. However, [57] did not consider issues like

eager language. Hence, for every defining equation $f(t^*) = \ldots f(s^*) \ldots$, evaluation of $f(s^*)$ takes less steps than evaluation of $f(t^*)$ (provided the conditions of the conditionals governing $f(s^*)$ are true). Thus, the first premises of Rule $1''$ imply the step formula and the permissibility conjecture implies the base formula of computational induction.

Note that Rule $1''$ only operates on universally quantified first order formulas. This restriction has the advantage that such formulas are *always admissible*.[14] Moreover, the formulas resulting from application of Rule $1''$ are again universally quantified. Hence, our rule directly allows nested (multiple) inductions without admissibility checks.

This is in contrast to computational induction. Suppose that f is defined as the least fixpoint of the functional F. When using computational induction, the induction step formula resulting from the universal formula $\forall x \ldots f(x) \ldots$ has the form

$$(\forall x \ldots F^i[\bot](x) \ldots) \;\Rightarrow\; (\forall x \ldots F^{i+1}[\bot](x) \ldots),$$

i.e., it is not universal (and possibly not admissible) any more.[15] Hence, for computational induction, repeated admissibility checks are required before every induction.

So Rule $1''$ combines the advantages of structural and computational induction. It performs an induction on the data structure, which is less abstract (and therefore easier to automate) than fixpoint induction. On the other hand, our rule uses the recursion structure of the algorithms to "suggest" plausible induction relations, i.e., it performs inductions w.r.t. possibly non-terminating algorithms. Hence, induction proofs w.r.t. partial functions can now be automated without using proof techniques based on denotational semantics. Thus, the existing

definedness, evaluation strategies, or the semantics of undefined terms (which are required for an extension to partial functions). So in a sense, the present work can be regarded as a refinement of [57], because we examined in detail what restrictions have to be imposed in order to use inductions w.r.t. partial algorithms, too.

[14] Essentially, the reason is that our basic (i.e. non-function) data types are flat complete partial orderings. Thus, in a first order formula, all occurrences of functions are in terms of chain-finite type. A similar admissibility criterion is for instance used in LCF [62] and a slightly weaker criterion has also been suggested in [70, 71], where however the restriction to chain-finite terms is not mentioned.

[15] As an example, let f be defined by $f(x) = \mathsf{0}$ and let g have the defining equations $g(\mathsf{0}) = \mathsf{0}$, $g(\mathsf{s}(x)) = g(x)$. The conjecture $\forall x\ (\mathsf{def}(x) \Rightarrow \neg\mathsf{def}(f(x)) \wedge \mathsf{def}(g(x)))$ is flawed, but admissible. Performing fixpoint induction on f results in a valid base formula and the induction step formula

$$\begin{array}{l} \forall x(\mathsf{def}(x) \;\Rightarrow\; \neg\mathsf{def}(F^i\ \ [\bot](x)) \wedge \mathsf{def}(g(x))) \;\Rightarrow \\ \forall x(\mathsf{def}(x) \;\Rightarrow\; \neg\mathsf{def}(F^{i+1}[\bot](x)) \wedge \mathsf{def}(g(x))). \end{array}$$

This formula is flawed and not admissible. However, if one neglects the check for admissibility, then by another fixpoint induction on g, it can be falsely proved.

induction provers and their powerful heuristics can also be applied for partial functions without adapting them to a new logical framework.

8.3. Partial Functions in Existing Induction Provers

In this section, we discuss other approaches to handle partial functions in existing induction theorem provers. One of the first and most successful induction provers is the NQTHM system of *R. S. Boyer* and *J Moore* [6]. It has been used to prove a large collection of impressive theorems and has influenced most of the subsequent developments in this area. Although in their framework all functions have to be total it nevertheless allows certain reasoning about partial functions. One approach is to transform every partial function $f(x)$ into a total one $f'(x, n)$ which operates like f, but which decreases the bound n in each recursive call and terminates if $n = 0$, cf. e.g. [9]. However, a drawback is that for theorems containing several occurrences of f one now has to deal with existential quantifiers (or to guess suitable instantiations of the bound variable n). For example, instead of a conjecture $\forall x^*\ f(t) = f(s)$ one has to prove $\forall x^*\ \forall n_1\ \exists n_2\ f'(t, n_1) = f'(s, n_2)$.[16]

A different approach is used in [10], where Boyer and Moore treat partial functions as inputs to an interpreter. In other words, instead of reasoning about the partial functions themselves, they reason about the *program text* defining the partial functions. As all functions have to be total, this interpreter is a *total* (non-constructive) function, which is not given by an algorithm, but only specified by axioms.

For conjectures about partial functions (resp. about the interpreter), they have to use induction relations whose well-foundedness has already been guaranteed (e.g. structural inductions). Thus, our approach has the advantage that in contrast to Boyer and Moore, we can still derive induction relations from the recursions of the occurring functions, even if these functions are partial. Our experiments in [27] show that this choice of the induction relation is indeed required for many conjectures.

In [7, 8, 45], Boyer and *M. Kaufmann* propose an extension of the Boyer-Moore prover to a functional language with *lazy* evaluation strategy. Their approach can handle functions which are partial on *infinite* objects. However, in order to derive induction schemes from the algorithms, they still require the functions to be defined at least for all *finite* arguments. Thus, the topic of Boyer and Kaufmann's work is quite different to ours, since we focus on an eager language without infinite objects and in contrast to them we regard functions which are partial

[16] Considering f' to be partial leads to similar problems in general. Such an approach was used by *P. Padawitz* in [60]. Here, the verification of definedness conditions was an additional difficulty, since he did not use an eager evaluation strategy.

on *finite* objects. However, we developed a method which nevertheless allows to generate induction schemes from these partial functions.

Our approach has similarities to the technique of *cover set induction* [42, 79], because cover sets are also based on the observation that a conjecture about a function f can often be proved by an induction according to f's recursions. This technique has been implemented in the theorem prover RRL [41] and proved successful on many challenge problems in mathematics as well as in software and hardware analysis.

In the original method of *H. Zhang*, *D. Kapur*, and *M. S. Krishnamoorthy* [79], incompletely defined functions are allowed, but induction schemes can only be derived from completely defined functions. In [42], Kapur and *M. Subramaniam* relax these conditions. In the proof of $\varphi_1 \Rightarrow \varphi_2$ they allow an induction w.r.t. an incompletely defined function f, if f is defined at least for all data objects satisfying φ_1. So similar to our approach, in [42] the permissibility of an induction scheme also depends on the conjecture being proved.

But Kapur and Subramaniam only perform inductions w.r.t. *terminating* functions. Thus, they have to supplement non-terminating functions by a termination condition before using them for the derivation of induction schemes. However, in general these termination conditions are not *exact* and there are many interesting partial functions whose domain is not even decidable, cf. Section 7. Here, our approach has the advantage that we can perform inductions w.r.t. partial algorithms without knowing anything about their termination behaviour.

Our proposal for inductions w.r.t. non-terminating functions can be integrated into the cover set technique, but for that purpose such partial functions must be evaluated in an *eager* way (whereas in [42, 79] no evaluation strategy is fixed). The reason is that otherwise an induction w.r.t. partial functions would be unsound. For example, if f has the defining equation $\mathsf{f}(x) = \mathsf{times}(\mathsf{f}(x), \mathsf{0})$, then partial truth of $\mathsf{f}(x) = \mathsf{1}$ can be proved by induction w.r.t. f. For an eager language, this conjecture is indeed partially true, as f is undefined on all arguments. But if the semantics of the language were not eager, then f would be a total function and all f-terms would be equal to $\mathsf{0}$.

We demanded that all defining equations must be non-overlapping. (This is a restriction which is not present in [42, 79].) Otherwise the definedness of an evaluation would depend on the order of the defining equations applied (as can be seen by the confluent, but overlapping equations $\mathsf{f}(x) = \mathsf{0}$ and $\mathsf{f}(x) = \mathsf{f}(x)$). Then instead of eager evaluation, it would be more natural not to fix any evaluation strategy. This however leads to problems for inductions w.r.t. partial functions as sketched above. Moreover, to guarantee consistence of $Eq \cup Ax^{\mathrm{data}}$, we wanted to ensure confluence of Eq in spite of non-termination. For that

reason, we also required left-linearity and hence, orthogonality of the defining equations. Note that natural formulations of conditional equations and of equations with relations between constructors are often not orthogonal. Therefore, in contrast to [42, 79], we only considered freely constructed data types and we modelled conditions with the special function if.

In [44], Kapur introduces an interesting extension of the cover set technique to data types with partial *constructors* and he uses so-called applicability conditions to describe the intended domains of the functions. This has a similarity to our definedness conditions $\mathsf{def}(f(x^*))$, but the semantics of applicability conditions can be arbitrarily specified, whereas our definedness conditions have a fixed semantics (i.e., $\mathsf{def}(f(t^*))$ is true iff evaluation of $f(t^*)$ is defined).

In Section 5 we added definedness conditions explicitly as premises of conjectures. However as advocated in [44], to simplify the proofs, it is often advantageous to keep such conditions implicit. Of course, for partial correctness statements of the form $\mathsf{def}(t^*) \Rightarrow \varphi$, the calculus of Section 5 can easily be used in such a way.[17] But the "explicit" formulation has the advantage that the inference rules become more general, because in this way they can also be used for conjectures like (38) – (41), (43), and (47), which are not partial correctness statements.

To conclude, all approaches discussed in this section perform inductions w.r.t. algorithms. However, our method also permits inductions based on the recursions of partial functions, whereas in the other approaches this is only possible for *terminating* algorithms.

Most work mentioned up to now uses the *explicit* induction paradigm, i.e., the induction relation is explicitly given (e.g., by the recursions of a function) and using this relation base and step formulas are constructed. But there has also been a lot of work based on the *implicit* induction principle (also called "proof by consistency" or "inductionless induction"), cf. e.g. [2, 4, 35, 37, 38, 40, 65]. However, these approaches are also restricted to terminating functions (i.e., the specification has to be given by a terminating term rewriting system). Then the rewrite ordering of this system is implicitly used for induction proofs.

9. Conclusion

Partial functions are important in many areas, but the techniques implemented in most induction theorem provers rely on the termination

[17] Reasoning about definedness can be separated from ordinary reasoning about functions, if every proof obligation of the form $(\mathsf{def}(r^*) \Rightarrow \psi) \Rightarrow (\mathsf{def}(t^*) \Rightarrow \varphi)$ is transformed into $\mathsf{def}(t^*) \Rightarrow \mathsf{def}(r^*)$ and $\psi \Rightarrow \varphi$. Here, t^* and r^* are the top-level terms of φ and ψ, respectively.

of the occurring algorithms. In this paper we showed that by introducing a few appropriate restrictions, these techniques can be applied for partial functions, too.

We first presented a calculus which does not require any reasoning about definedness and which is already very successful for a certain class of conjectures (in particular, conjectures containing at most one occurrence of a partial function). But to increase the power of our approach, subsequently we developed a refined calculus where definedness was made explicit. To demonstrate its power, we tested our approach on numerous examples and used it to prove many theorems about partial functions with undecidable domains [27]. While we did not develop a new theorem proving system based on our inference rules, in our case study we used some mechanized support by an existing prover. The results of this study show that our rules indeed allow the proof of non-trivial theorems, i.e., they do not represent a significant restriction on the class of theorems that can be proved.

Our calculi correspond 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 Jürgen Brauburger, Deepak Kapur, Matt Kaufmann, Thomas Kolbe, Natarajan Shankar, Christoph Walther, and Claus-Peter Wirth for helpful comments and fruitful discussions. This work was supported by the DFG under grant no. GI 274/4-1.

References

1. 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).
2. Bachmair, L., 'Proof by Consistency in Equational Theories', in *Proc. 3rd IEEE Symp. Logic in Computer Science*, Edinburgh, Scotland, IEEE Press (1988).
3. Birkhoff, G., 'On the Structure of Abstract Algebras', *Proc. Cambridge Philos. Soc.* **31**, 433-454 (1934).
4. Bouhoula, A. and Rusinowitch, M., 'Implicit Induction in Conditional Theories', *Journal of Automated Reasoning* **14**, 189-235 (1995).
5. Boulton, R. J., 'Boyer-Moore Automation for the HOL System', in *Proc. 6th Int. Workshop HOL Theorem Pr. Appl.*, Vancouver, Canada, Elsevier (1993).
6. Boyer, R. S. and Moore, J S., *A Computational Logic*, Academic Press (1979).
7. Boyer, R. S. and Kaufmann, M., On the Feasibility of Mechanically Verifying SASL Programs, Tech. Rep. ARC 84-16, Burroughs Research Center (1984).

8. Boyer, R. S. and Kaufmann, M., A Prototype Theorem Prover for a Higher-Order Functional Language, Tech. Rep. ARC 84-17, Burroughs Cent. (1984).
9. Boyer, R. S. and Moore, J S., 'A Mechanical Proof of the Turing Completeness of Pure LISP', in *Automated Theorem Proving: After 25 Years* (eds. W. W. Bledsoe and D. W. Loveland), American Mathematical Society, RI (1984).
10. 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).
11. Brauburger, J. and Giesl, J., 'Termination Analysis for Partial Functions', in *Proc. 3rd Int. Static Analysis Symp.*, Aachen, Germany, Lect. Notes in CS 1145, Springer (1996). Extended version as Tech. Rep.[18] IBN 96/33, TU Darmstadt.
12. Brauburger, J. and Giesl, J., 'Termination Analysis by Inductive Evaluation', in *Proc. 15th CADE*, Lindau, Germany, LNAI 1421, Springer-Verlag (1998).
13. Brauburger, J. and Giesl, J., 'Approximating the Domains of Functional and Imperative Programs', *Science of Computer Programming* **35** (1999).
14. Bronsard, F., Reddy, U. S., and Hasker, R. W., 'Induction Using Term Orders', *Journal of Automated Reasoning* **16**, 3-37 (1996).
15. Bundy, A., 'A Rational Reconstruction and Extension of Recursion Analysis', in *Proc. 11th Int. Joint Conf. AI*, Detroit, MI, Morgan Kaufmann (1989).
16. Bundy, A., van Harmelen, F., Smaill, A., and Ireland, A., 'The OYSTER-CLAM system', in *Proc. 10th Int. Conf. Automated Deduction*, Kaiserslautern, Germany, Lecture Notes in Artificial Intelligence 449, Springer-Verlag (1990).
17. Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., and Smaill, A., 'Rippling: A Heuristic for Guiding Inductive Proofs', *Artif. Int.* **62**, 185-253 (1993).
18. Busch, H., 'Unification-Based Induction', in *Proc. 6th Int. Workshop Higher Order Logic Theorem Proving Appl.*, Vancouver, Canada, Elsevier (1993).
19. De Schreye, D. and Decorte, S., 'Termination of Logic Programs: The Never-Ending Story', *Journal of Logic Programming* **19, 20**, 199-260 (1994).
20. Dershowitz, N., 'Termination of Rewriting', *J. Symb. Comp.* **3**, 69-115 (1987).
21. Farmer, W. M., 'A Partial Function's Version of Church's Simple Theory of Types', *Journal of Symbolic Logic* **55**, 1269-1291 (1990).
22. Finn, S., Fourman, M. P., and Longley, J., 'Partial Functions in a Total Setting', *Journal of Automated Reasoning* **18**, 85-104 (1997).
23. Gardner, M., *Wheels, Life And Other Mathematical Amusements*, W. H. Freeman and Company (1983).
24. Giesl, J., 'Automated Termination Proofs with Measure Functions', in *Proc. 19th Ann. German Conf. AI*, Bielefeld, Germany, LNAI 981, Springer (1995).
25. Giesl, J., 'Termination Analysis for Functional Programs using Term Orderings', in *Proceedings of the 2nd International Static Analysis Symposium*, Glasgow, Scotland, Lecture Notes in Computer Science 983, Springer-Verlag (1995).
26. Giesl, J., 'Termination of Nested and Mutually Recursive Algorithms', *Journal of Automated Reasoning* **19**, 1-29 (1997).
27. Giesl, J., The Critical Pair Lemma: A Case Study for Induction Proofs with Partial Functions, Technical Report[18] IBN 98/49, TU Darmstadt (1998).
28. 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), Applied Logic Series 10, Kluwer (1998).
29. Giesl, J. and Middeldorp, A., 'Transforming Context-Sensitive Rewrite Systems', in *Proc. RTA-99*, Trento, Italy, Lecture Notes in CS, Springer (1999).
30. Goguen, J. A., Thatcher, J. W., and Wagner, E. G., 'An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types', in *Current Trends in Programming Methodology, Vol. 4* (ed. R. T. Yeh), Prentice-Hall (1978).

[18] Available from `http://www.inferenzsysteme.informatik.tu-darmstadt.de/~reports/notes/{ibn-96-33.ps,ibn-98-49.ps}`.

31. Gordon, M. J. C., Milner, R., and Wadsworth, C. P., *Edinburgh* LCF*: A Mechanised Logic of Computation*, Lect. Notes in Comp. Sc. 78, Springer (1979).
32. Gordon, M. J. C. and Melham, T. F., *Introduction to* HOL*: A Theorem-Proving Environment for Higher-Order Logic*, Cambridge University Press (1993).
33. Guttag, J. V., 'Abstract Data Types and the Development of Data Structures', *Comm. ACM* **20**, 396-404 (1977).
34. Huet, G., 'Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems', *Journal of the ACM* **27**, 797-821 (1980).
35. Huet, G. and Hullot, J.-M., 'Proofs by Induction in Equational Theories With Constructors', *J. Computer and System Sciences* **25**, 239-266 (1982).
36. Hutter, D. and Sengler, C., 'INKA: The next generation', in *Proc. 13th CADE*, New Brunswick, NJ, Lecture Notes in AI 1104, Springer (1996).
37. Jouannaud, J.-P. and Kounalis, E., 'Automatic Proofs by Induction in Theories Without Constructors', *Information and Computation* **82**, 1-33 (1989).
38. Kapur, D., Narendran, P., and Zhang, H., 'Proof by Induction Using Test Sets', in *Proceedings 8th International Conference on Automated Deduction*, Oxford, England, Lecture Notes in Computer Science 230, Springer-Verlag (1986).
39. Kapur, D. and Musser, D. R., 'Inductive Reasoning with Incomplete Specifications', in *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press (1986).
40. Kapur, D. and Musser, D. R., 'Proof by Consistency', *Artificial Intelligence* **31**, 125-157 (1987).
41. Kapur, D. and Zhang, H., 'An Overview of Rewrite Rule Laboratory (RRL)', *Journal of Computer and Mathematics with Applications* **29**, 91-114 (1995).
42. Kapur, D. and Subramaniam, M., 'New Uses of Linear Arithmetic in Automated Theorem Proving by Induction', *J. Aut. Reasoning* **16**, 39-78 (1996).
43. Kapur, D. and Subramaniam, M., 'Automating Induction over Mutually Recursive Functions', in *Proc. 5th Int. Conf. Algebraic Meth. and Software Technology*, Munich, Germany, Lect. Notes in Comp. Sc. 1101, Springer-Verlag (1996).
44. Kapur, D., 'Constructors can be Partial, too', in *Automated Reasoning and Its Applications – Essays in Honor of L. Wos* (ed. R. Veroff), MIT Press (1997).
45. Kaufmann, M., A Sound Theorem Prover for a Higher-Order Functional Language, Technical Report ARC 86-01, Burroughs Austin Research Center, 1986.
46. Kerber, M. and Kohlhase, M., 'A Mechanization of Strong Kleene Logic for Partial Functions', in *Proc. 12th Int. Conf. Automated Deduction*, Nancy, France, Lecture Notes in Artificial Intelligence 814, Springer-Verlag (1994).
47. Kerber, M. and Kohlhase, M., 'A Tableau Calculus for Partial Functions', *Collegium Logicum – Annals of the Kurt Gödel-Society* **2**, 21-49 (1996).
48. Kleene, S. C., *Introduction to Metamathematics*, Van Nostrand (1952).
49. Knuth, D. E. and Bendix, P. B., 'Simple Word Problems in Universal Algebras', *Computational Problems in Abstract Algebra* (ed. J. Leech), Pergamon (1970).
50. Kreowski, H.-J., 'Partial Algebras flow from Algebraic Specifications', in *Proc. 14th Int. Coll. on Automata, Languages, and Programming*, Karlsruhe, Germany, Lecture Notes in Computer Science 267, Springer-Verlag (1987).
51. Kühler, U. and Wirth, C.-P., 'Conditional Equational Specifications of Data Types with Partial Operations for Inductive Theorem Proving', in *Proceedings of the 8th International Conference on Rewriting Techniques and Applications*, Sitges, Spain, Lecture Notes in Computer Science 1232, Springer-Verlag (1997).
52. Loeckx, J. and Sieber, K., *The Foundations of Program Verification*, Wiley-Teubner (1987).
53. Lucas, S., 'Context-Sensitive Computations in Functional and Functional Logic Programs', *Journal of Functional and Logic Programming* **1**, 1-61 (1998).
54. Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill (1974).
55. Manna, Z. and Waldinger, R., 'Deductive Synthesis of the Unification Algorithm', *Science of Computer Programming* **1**, 5-48 (1981).

56. McCarthy, J., 'Recursive Functions of Symbolic Expressions and their Computation by Machine', *Communications of the ACM* **3**, (1960).
57. Morris, J. H. and Wegbreit, B., 'Subgoal Induction', *Communications of the ACM* **20**, 209-222, 1977.
58. Mosses, P. D., 'The Use of Sorts in Algebraic Specifications', in *Proceedings of the 7th Workshop on Specifications of Abstract Data Types*, Lecture Notes in Computer Science 655, Springer-Verlag (1991).
59. Nipkow, T., 'More Church-Rosser Proofs (in ISABELLE/HOL)', in *Proc. 13th Int. Conf. Aut. Ded.*, New Brunswick, NJ, Lect. Notes in AI 1104, Springer (1996).
60. Padawitz, P., Inductive Expansion, Internal Report MIP-8907, Universität Passau, Germany, 1989.
61. Paulson, L. C., 'Verifying the Unification Algorithm in LCF', *Science of Computer Programming* **5**, 143-169 (1985).
62. Paulson, L. C., *Logic and Computation*, Cambridge University Press (1987).
63. Paulson, L. C., ISABELLE: *A Generic Theorem Prover*, Lecture Notes in Computer Science 828, Springer-Verlag (1994).
64. Plümer, L., *Termination Proofs for Logic Programs*, Lecture Notes in Artificial Intelligence 446, Springer-Verlag (1990).
65. Reddy, U. S., 'Term Rewriting Induction', in *Proceedings of the 10th International Conference on Automated Deduction*, Kaiserslautern, Germany, Lecture Notes in Computer Science 449, Springer-Verlag (1990).
66. Reichel, H., *Initial Computability, Algebraic Specifications and Partial Algebras*, Oxford University Press (1987).
67. Robinson, J. A., 'A Machine Oriented Logic Based on the Resolution Principle', *Journal of the ACM* **12**, 23-41 (1965).
68. Scott, D. S., A Type-Theoretic Alternative to CUCH, ISWIM, PWHY, Notes, Oxford (1969). Annotated version in *Theor. Comp. Sc.* **121**, 411-440 (1993).
69. Shankar, N., 'A Mechanical Proof of the Church-Rosser Theorem', *Journal of the ACM* **35**, 475-522 (1988).
70. Shankar, N., A Logical Basis for Functional Programming, Draft, Stanford University (1989).
71. Shankar, N., Recursive Programming and Proving, Course Notes CS 306, SRI International (1990).
72. Slind, K., 'Derivation and Use of Induction Schemes in Higher-Order Logic', in *Proc. 10th Int. Conf. on Theorem Proving in Higher Order Logics*, Murray Hill, NJ, Lecture Notes in Computer Science 1275, Springer-Verlag (1997).
73. Steinbach, J., 'Simplification Orderings: History of Results', *Fundamenta Informaticae* **24**, 47-87 (1995).
74. Ullman, J. D. and van Gelder, A., 'Efficient Tests for Top-Down Termination of Logical Rules', *Journal of the ACM* **35**, 345-373 (1988).
75. 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).
76. Walther, C., 'On Proving the Termination of Algorithms by Machine', *Artificial Intelligence* **71**, 101-157 (1994).
77. Wirth, C.-P. and Gramlich, B., 'On Notions of Inductive Validity for First-Order Equational Clauses', in *Proc. 12th Int. Conference on Automated Deduction*, Nancy, France, Lecture Notes in AI 814, Springer-Verlag (1994).
78. Wirth, C.-P. and Kühler, U., Inductive Theorem Proving in Theories Specified by Positive/Negative-Conditional Equations, SEKI-Report SR-95-15, Universität Kaiserslautern, Germany (1995).
79. Zhang, H., Kapur, D., and Krishnamoorthy, M. S., 'A Mechanizable Induction Principle for Equational Specifications', in *Proc. 9th Int. Conf. Automated Deduction*, Argonne, IL, Lect. Notes in Comp. Science 310, Springer (1988).