Closure Induction in a Z-like Language[⋆] [⋆⋆]

David A. Duffy[1] and Jürgen Giesl[2]

[1] Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK, dad@cs.york.ac.uk
[2] Computer Science Department, University of New Mexico,
Albuquerque, NM 87131, USA, giesl@cs.unm.edu

Abstract. Simply-typed set-theoretic languages such as Z and B are widely used for program and system specifications. The main technique for reasoning about such specifications is *induction*. However, while partiality is an important concept in these languages, many standard approaches to automating induction proofs rely on the totality of all occurring functions. Reinterpreting the second author's recently proposed induction technique for partial functional programs, we introduce in this paper the new principle of "closure induction" for reasoning about the inductive properties of partial functions in simply-typed set-theoretic languages. In particular, closure induction allows us to prove partial correctness, that is, to prove those instances of conjectures for which designated partial functions are explicitly defined.

1 Motivation

Partial functions are endemic in specifications written in languages such as Z and B. To reason about their inductive properties a method amenable to mechanical support by automated theorem provers is inevitable. In [13], Giesl has shown that, under certain conditions, many of the reasoning processes used to prove inductive properties of total functions (e.g., those in [5, 9, 19, 25, 27]) may be transposed to partial functions. The inference rules proposed by Giesl allow us to prove conjectures involving partial functions for all instances of the conjecture for which designated partial functions are explicitly defined.

However, Giesl's technique has been designed for a first-order functional language with an eager (call-by-value) evaluation strategy. In this paper, we examine thoroughly which interpretation of partiality and which restrictions on the allowed theories are required in order to extend Giesl's induction principle from the original functional programming framework to a simply-typed set-theoretic language closely related to Z and B.

We refer to our new principle as "closure induction", since instances of it may be described within our set-theoretic language itself, and these instances may be viewed as "closure axioms" for a function definition, asserting that the function

⋆ *Proceedings of the International Conference of Z and B Users (ZB2000)*, York, UK, Lecture Notes in Computer Science 1878, pages 471-490, Springer-Verlag, 2000.

⋆⋆ D. Duffy was supported by the EPSRC under grant no. GR/L31104, J. Giesl was supported by the DFG under grant no. GI 274/4-1.

is defined in only those cases explicitly specified. For the soundness of closure induction we must make certain assumptions about the semantics of types (i.e., the carrier of a type must include "undefined" values that can be used as the value of a partial function when applied outside of its domain). We describe an appropriate semantics for our language in Section 2.

Our approach to induction is applicable to languages such as Z and B if they too assume our semantics. This semantics is, we claim, not very restrictive; we would argue that it imposes the minimal requirements needed in order to distinguish between defined and undefined expressions. A commonplace interpretation of partial-function application in the Z community [1, 24] is that any such application always returns a value in the function's range type; we refer to this as the "classical" semantics. In such a framework we cannot distinguish between defined and undefined function applications. However, there is some debate as to whether this is the appropriate interpretation of function application [17], and our alternative semantics has already gained some interest within the Z community via its earlier presentation in a more general set-theoretic framework [11]. Apparently, no particular semantics is fixed by the standard definition of Z [20, 21]. Moreover, our semantics may be simulated within the classical semantics in a straightforward way [11]; this allows us to simulate our approach to induction in the CADiZ system [22], a tool for reasoning about Z specifications, which currently supports the classical semantics.

In Section 3 we formalize our concept of inductive validity in the context of partial functions and in Section 4 we introduce the technique of closure induction in order to perform induction proofs automatically. We then discuss conditions under which closure induction is sound. We formalize these conditions in terms of rewriting, and it may thus come as no surprise that a confluence property forms part of the conditions. In particular, we show that the applicability of closure induction extends beyond the "orthogonal" equational theories considered previously by Giesl [13]. Finally, in Section 5 we present some further rules that are needed in addition to closure induction to verify definedness conditions that arise in most proofs about partial functions.

The closure-induction approach described in this paper has been simulated within the CADiZ system [22]; simulations of the *diff* and *quot* examples we describe may be found on the web at ftp://ftp.cs.york.ac.uk/pub/aig/examples.

2 A Typed Language and its Semantics

Elsewhere [11], Duffy has described a quite general set-theoretic language (essentially a subset of Z) and its associated semantics. Since, in the present paper, we are concerned with inductive reasoning in the context of free types and equational theories, we are able to consider a much restricted subset of this higher-order language, which we will refer to as $\mathcal{F}$ (signifying "free types").

2.1 The Syntax of Expressions

We refer to all allowed syntactic objects as "expressions". We separate expressions into "types", "terms", and "formulae", distinguishing types from terms, for simplicity, since we do not allow types as subterms.

$$Type ::= TypeName \mid \mathbb{P}\,Type \mid Type \times \cdots \times Type$$

Here, $TypeName$ denotes *given sets* [20] which are introduced in a so-called declaration part of specifications. Intuitively, $\mathbb{P}$ is the powerset operator and $\times$ denotes cross product.

$$Term ::= Const \mid Var \mid Tuple \mid Application$$

$Const$ is used for function names — as for $TypeName$s they are introduced in declaration parts of specifications. Variable names Var are introduced by the quantification of a formula (as in, e.g., $\forall x : \mathbb{N} \bullet P$).

$$Tuple ::= (Term, \ldots, Term)$$

An n-tuple of terms $(t_1, \ldots, t_n)$, where $n \geq 1$, is often abbreviated $\mathbf{t}$; the type of $(t_1, \ldots, t_n)$ is $T_1 \times \cdots \times T_n$, where T_i is the type of t_i.

$$Application ::= Term\ Term$$

where the first $Term$ is of type $\mathbb{P}\,(T_1 \times T_2)$ and the second $Term$ has type T_1; the type of the application is T_2. We often write $f(t)$ instead of "$f\,t$".

$$\begin{aligned} Form ::= {} & Term = Term \mid Term \in Type \mid Term \in Term \mid \\ & \neg Form \mid Form \wedge Form \mid Form \vee Form \mid Form \Rightarrow Form \mid \\ & Q\,Var : Type \bullet Form \end{aligned}$$

where $Q \in \{\forall, \exists, \exists_1\}$ ($\exists_1$ denoting unique existence). We also allow the formula $Qx_1 : T_1; \ldots; x_n : T_n \bullet P$ as an abbreviation for $Qx_1 : T_1 \bullet \ldots Qx_n : T_n \bullet P$, and if $T_1 = \ldots = T_n = T$, we also write $Qx_1, \ldots, x_n : T \bullet P$. Moreover, we always demand that all terms and all formulae must be well typed. So for example, for any formula $t_1 = t_2$, both terms t_1 and t_2 must have the same type.

A *specification* consists of a declaration and an axiom part, where the declaration part introduces all given sets (i.e., all $TypeName$s) and constants used, and the axiom part is a set of formulae.

2.2 The Semantics of Expressions

In the "classical semantics" described by Arthan [1], every expression is a member of its type. In our semantics, we include "undefined" expressions that are not members of their type, thus allowing function applications to "return a value" not a member of the function's range type. For this purpose, we distinguish "having type T" from "being a member of T". We formalize this as follows.

Let Σ be a specification involving a type T. In an interpretation for Σ we assign a set T^* to T, constructed according to the form of T:

– If T is a given set, then T^* is the union of two disjoint sets $T^+ \cup T^-$, where T^+ is assumed to be non-empty.
– If T is a product $T_1 \times \cdots \times T_n$, then $T^+ = T_1^+ \times \cdots \times T_n^+$ and $T^* = T_1^* \times \cdots \times T_n^*$.
– If $T = \mathbb{P}(T_1)$, then $T^+ = \mathbb{P}(T_1^+)$ and $T^* = \mathbb{P}(T_1^*)$.

Informally, T^+ may be interpreted as the defined values of type T. The assumption that T^+ is non-empty ensures that there is at least one possible value for any application, and allows us to avoid treating the special case of an empty type. In the language of our models we use the same symbols $\mathbb{P}$, $\times$, etc. as in $\mathcal{F}$, since no confusion should arise. The symbol $=$ is our metalogical equality.

We now define the total function *App*, which will be assigned to function applications. Let r be a subset of $\mathbb{P}(T_1^* \times T_2^*)$, and x be an element of T_1^*.

$$App(r,x) = \begin{cases} \text{the unique } y \text{ such that } (x,y) \in r \text{ if such a } y \text{ exists} \\ \text{some } y \text{ in } T_2^* \text{ otherwise} \end{cases}$$

App is defined so that it is consistent with the usual Z interpretation of application [20]. Note that $App(r,x) = y \not\Rightarrow (x,y) \in r$.

We are now able to define the meaning of $\mathcal{F}$ expressions in an interpretation I, under an assignment a to any occurring free variables. In the following, let T denote a type, P, Q denote formulae, x denote a variable, c denote a constant, s, t, t_i denote terms, and f denote a term of type $\mathbb{P}(T \times T')$ for some T, T'. As the relationship between the symbol $\in$ of $\mathcal{F}$ and membership in the models is not straightforward, we use ϵ for membership in the model language.

The interpretation of a term of type T is some value of T^*. Only function application is given special treatment; the meaning of other terms is standard.

$I(c)[a] = c_I$, an element of T^*, where T is the type of c
$I(x)[a] = a(x)$, the value assigned to x by the function a
$I((t_1, \ldots, t_n))[a] = (I(t_1)[a], \ldots, I(t_n)[a])$
$I(f\ t)[a] = App(I(f)[a], I(t)[a])$

For a formula P, we always have $I(P)[a] = True$ or $I(P)[a] = False$. The interpretation of equality and the propositional connectives is standard; only membership and quantification are given special treatment.

$I(s = t)[a] = True$	*iff* $I(s)[a] = I(t)[a]$
$I(s \in t)[a] = True$	*iff* $I(s)[a] \ \epsilon\ I(t)[a]$
$I(t \in T)[a] = True$	*iff* $I(t)[a] \ \epsilon\ T^+$
$I(\neg P)[a] = True$	*iff* $I(P)[a] = False$
$I(P \wedge Q)[a] = True$	*iff* $I(P)[a] = True$ *and* $I(Q)[a] = True$
$I(P \vee Q)[a] = True$	*iff* $I(P)[a] = True$ *or* $I(Q)[a] = True$
$I(P \Rightarrow Q)[a] = False$	*iff* $I(P)[a] = True$ *and* $I(Q)[a] = False$
$I(\forall x : T \bullet P)[a] = True$	*iff* $I(P)[a^{e/x}] = True$ for all $e \ \epsilon\ T^+$
$I(\exists x : T \bullet P)[a] = True$	*iff* $I(P)[a^{e/x}] = True$ for some $e \ \epsilon\ T^+$
$I(\exists_1 x : T \bullet P)[a] = True$	*iff* $I(P)[a^{e/x}] = True$ for one unique $e \ \epsilon\ T^+$

In the last three equations, e is assigned to any occurrences of x in P (i.e., $a^{e/x}(x) = e$ and $a^{e/x}(y) = a(y)$ for all $y \neq x$).

Note, in particular, that, under our semantics, the symbol "$\in$" does not represent true membership, but only membership of the "defined part" of any type. Similarly, the quantifiers only range over the defined parts of the respective types.

Example 1. If o is a constant of a type *nats*, and f is a function from *nats* to *nats*, then

$$I(f(o) \in nats) = App(f_I, o_I) \,\epsilon\, nats^{+}. \qquad \square$$

We may simulate our semantics in the classical semantics in the following way [11]. Let Σ be a specification with exactly the given sets $T_1, \ldots, T_n$. Then the declaration of each T_i is replaced by the declaration of a new given set T_i^*. Subsequently, a declaration for each T_i is added asserting it to be a subset of T_i^*. The rest of Σ remains unchanged. Now, under the classical semantics, every expression will return a value of its type T_i^*; the "undefined" expressions are those that do not return a value of the subset T_i of their type.

We may now define models in the usual way.

Definition 1 (Model). *An interpretation I is a* model *of a specification Σ if all axioms in Σ are satisfied by I under all variable assignments a.*

For example, let Σ be a specification involving the type *nats*, a member o of *nats*, two functions s and f from *nats* to *nats*, and the axioms

$$\{\forall x : nats \bullet \neg\, x = s(x),\ f(o) = s(f(o))\}.$$

Then $f(o)$ is of type *nats*, but the value of $App(f_I, o_I)$ in any model of Σ will not be in $nats^{+}$ in order to avoid violating the first axiom. Having defined which interpretations are models of a specification, we can now define *consequence*.

Definition 2 (Consequence). *A formula P is a* consequence *of a specification Σ (or "*valid*"), denoted $\Sigma \models P$, if every model of Σ satisfies P under all variable assignments.*

In this paper, we are concerned not so much with the consequences as with the "inductive consequences" of specifications — though these two terms become synonymous if we include the appropriate "induction formulae" within a specification. Our goal is to present an induction principle that allows us to prove such inductive consequences. First, we clarify what we mean by this term in the context of specifications that may involve partial functions.

3 Inductive Reasoning

For our purposes, a *free type* is a *given set* whose elements are freely generated by a set of constructors [20]. For example, the elements of a type *nats*, representing the natural numbers, can be generated from the nullary constructor o

and the unary constructor s. In Z, the free type *nats* would be introduced into a specification by the abbreviation

$$nats ::= o \mid s \langle\langle nats \rangle\rangle.$$

Such a statement would then be expanded into a declaration and a set of axioms. The declaration introduces the *given set nats* and the constants o of type *nats* and s of type $\mathbb{P}\,(nats \times nats)$. The axioms assert that s is a total injection, that $\{o\}$ and the range of s are disjoint, and that any subset of *nats* that includes o and is closed under s is the whole of *nats*. The latter axiom corresponds to a structural induction principle for *nats*. Sufficient conditions for the consistency of an arbitrary free type are outlined by Spivey [20]; the presentation of *nats* above satisfies these conditions.

The details of the expansion for any free type may be found in [23]. For illustration, the axioms for *nats* are (equivalent to) the following formulae:

1. Membership	$o \in nats, \quad s \in \mathbb{P}\,(nats \times nats)$
2. Total Function	$\forall x : nats \bullet \exists_1 y : nats \bullet (x, y) \in s$
3. Injectivity	$\forall x, y : nats \bullet s(x) = s(y) \Rightarrow x = y$
4. Disjointness	$\forall x : nats \bullet \neg\, o = s(x)$
5. Induction	$\forall nats_0 : \mathbb{P}\, nats \bullet\ o \in nats_0 \wedge$ $(\forall x : nats \bullet x \in nats_0 \Rightarrow s(x) \in nats_0) \Rightarrow$ $\forall x : nats \bullet x \in nats_0$

Under our semantics, the meaning of the declaration and axioms associated with *nats* is that, in every model of the specification, $nats^+$ must be isomorphic to the constructor ground term algebra generated by the constructors o and s. In other words, $nats^+$ may contain only objects which occur as interpretations of constructor ground terms and, moreover, different constructor ground terms must be interpreted as different objects. This corresponds to the notion of initial algebras usually applied in inductive theorem proving, cf. e.g. [4, 13, 15, 25–27].

The structural induction principle associated with any free type allows us to prove conjectures that hold for every element of the type. However, typically we wish to prove properties of a partial function on its defined cases only, as illustrated by the following example from [13].

Example 2.

$$
\begin{array}{l}
nats ::= o \mid s \langle\langle nats \rangle\rangle \\
\hline
\left| \begin{array}{l}
diff, quot : nats \times nats \rightarrow\!\!\!\!\!\!+\ nats \\
\hline
\forall x : nats \bullet diff(x, o) = x \\
\forall x, y : nats \bullet diff(s(x), s(y)) = diff(x, y) \\
\forall y : nats \bullet quot(o, y) = o \\
\forall x, y : nats \bullet quot(s(x), y) = s(quot(diff(s(x), y), y))
\end{array} \right.
\end{array}
$$

We use the usual Z bar notation to separate the declaration part of a specification from the axiom part. For types T and T', we use the expression $f : T \rightarrow\!\!\!\!\!\!+\ T'$ to introduce a new constant f in the declaration of a specification and to denote the assumption that f is a "partial function" from T to T'. More precisely, the expansion of $f \in T \nrightarrow T'$ is

$$f \in \mathbb{P}(T \times T') \;\wedge\; \forall x : T; y, z : T' \bullet (x, y) \in f \wedge (x, z) \in f \Rightarrow y = z.$$

Clearly, *diff* is explicitly defined only for $x \geq y$ and $quot(x, y)$ is explicitly defined only if y is a divisor of x. Note that in the "classical" semantics there is no model of the *quot* specification respecting the semantics of free types, because $quot(s(o), o)$ must be equal to $s(quot(s(o), o))$. However, our semantics solves this problem, because the interpretation of $quot(s(o), o)$ is now a member of the carrier set $nats^* \setminus nats^+$. □

Note that we have not explicitly specified the domains of the functions *diff* and *quot* in the above example. Our approach to partiality thus differs from the more conventional one in which the equations defining a function are usually conditional on predicates that ensure that the function is assigned explicit values only for arguments within its domain. In this conventional approach, the value of a function application is always a member of its type, this value simply being left unspecified for arguments outside of the function's domain. This approach thus models *underspecified* rather than *partial* functions. In contrast, our approach allows a function application to be *undefined* for arguments outside of the function's domain. This makes our approach significantly more expressive, allowing a more general class of consistent specifications, and providing several other advantages for specification and reasoning.

In particular, there are many important and practically relevant algorithms with *undecidable* domains. Typical examples are interpreters for programming languages and sound and complete calculi for first-order logic. For these algorithms, there do not exist any (recursive) predicates describing their domains. The conventional approach for modelling partial functions cannot handle such "real" partial functions. In our framework, on the other hand, such algorithms can be expressed without difficulty, and, moreover, the proof technique described in this paper supports their verification [12, 13]. More generally, our framework has the advantage that specifications can be formulated much more easily, since one does not have to determine the domains of functions. Consequently, our approach is well-suited to the early "loose" stages of specification when the function domains may be still unknown. Finally, our representation allows proofs which do not have to deal with definedness conditions, which makes (automated) reasoning much more efficient, cf. [18].

For those cases where *diff* and *quot* are (explicitly) defined it can be shown that the following conjectures follow from the above specification (if the specification is extended by appropriate definitions for $+$ and $*$):

$$\forall x, y : nats \bullet diff(x, y) + y = x \tag{1}$$
$$\forall x, y : nats \bullet quot(x, y) * y = x \tag{2}$$

The problem in trying to prove these conjectures is that the equations for *diff* and *quot* provide us with only sufficient conditions for these functions to be defined; we cannot infer that they are defined in *only* those cases. We may overcome this problem by adding suitable "closure axioms". Whenever there is a model of the specification where a function application is undefined, these closure axioms eliminate all models where this function application would be defined. Examples of such closure axioms are the following:

$$\forall x, y : nats \bullet \mathit{diff}(x,y) \in nats \;\Rightarrow\; y = o \vee \exists u, v : nats \bullet x = s(u) \wedge y = s(v)$$

$$\begin{aligned}\forall x, y : nats \bullet \mathit{quot}(x,y) \in nats \Rightarrow (x = o \vee \exists u : nats \;\bullet x = s(u) \wedge \\ \mathit{quot}(\mathit{diff}(s(u),y),y) \in nats \wedge \\ \mathit{diff}(s(u),y) \in nats).\end{aligned}$$

These closure axioms, the equations for *diff* and *quot*, and the free type axioms imply for $m, n \in nats$ that $\mathit{diff}(m,n)$ is not in *nats* if m is "smaller" than n, and that $\mathit{quot}(m,n)$ is not in *nats* if m is not "divisible" by n. Most importantly, now the axioms imply our original conjectures in the forms

$$\forall x, y : nats \bullet \mathit{diff}(x,y) \in nats \Rightarrow \mathit{diff}(x,y) + y = x \tag{3}$$

$$\forall x, y : nats \bullet \mathit{quot}(x,y) \in nats \Rightarrow \mathit{quot}(x,y) * y = x. \tag{4}$$

We refer to specifications that consist only of free types, function declarations, and equations as *equational.* For such specifications Σ, the desired properties of closure axioms are given by the following definition.

Definition 3 (Closure Axioms). *A set of* closure axioms *for an equational specification Σ is a set of formulae C consistent with Σ such that*

$$\Sigma \not\models f(q_1, \ldots, q_n) \in T \;\text{ implies }\; \Sigma \cup C \models \neg\,(f(q_1, \ldots, q_n) \in T),$$

for each n-ary function f (whose application has type T) and each n-tuple of appropriately-typed constructor ground terms $(q_1, \ldots, q_n)$. The addition of a set of closure axioms to a specification is referred to as the closure *of the specification. In those cases where we assume that a specification includes all the relevant closure axioms, we will say that the specification is a* closed system.

For *diff* and *quot*, their above closure axioms may be derived automatically from their equations, but this is not so straightforward in general. For example, consider a function $f : nats \nrightarrow nats$ "defined" by only the equation

$$\forall x : nats \bullet f(x) = f(x).$$

Since this equation tells us nothing about the values returned by f, we infer that f is undefined for all m in *nats*, and the corresponding closure axiom must support this inference. An appropriate closure axiom is thus

$$\forall x : nats \bullet \neg\, f(x) \in nats.$$

However, it is not obvious how we may derive this closure axiom automatically from the given equation. Giesl *et al.* [6, 7, 14] have developed techniques for termination analysis of partial functions, which would easily find out the domains of such simple functions as f (and also *quot* and *diff*) automatically, but, in general, this is an undecidable problem. In fact, we will only use the (non-constructive) closure axioms to define our notion of partial validity. To prove partial validity in practice, we will introduce the proof technique of closure induction, which allows us to verify properties of partial functions without knowing their domains and without having to compute closure axioms explicitly.

Definition 4 (Partial Validity). *For an equational specification Σ we say that a conjecture P is* partially valid *if $\Sigma \cup C \models P$ holds for any set of closure axioms C.*

In practice, the verification of partial validity of a conjecture is accomplished in two separate steps. The first is a proof of the $f(\mathbf{x})$-validity of a conjecture, which means that the conjecture is valid for all those instantiations of $\mathbf{x}$ where $f(\mathbf{x})$ is defined. These proofs are supported by the principle of closure induction.

Definition 5 ($f(\mathbf{x})$-Validity). *Let Σ be a specification involving the free types $T_1, \dots, T_n, T$ and the function $f : T_1 \times \dots \times T_n \nrightarrow T$. Let $x_1, \dots, x_n$ be variables of types $T_1, \dots, T_n$, respectively, and let P be a quantifier-free formula.*[1] *We say that the conjecture $\forall x_1 : T_1; \dots; x_n : T_n \bullet P$ is* $f(\mathbf{x})$-valid, *where $\mathbf{x}$ represents $x_1, \dots, x_n$, if*[2] *$\Sigma \models P(q_1, \dots, q_n)$ holds for every sequence $q_1, \dots, q_n$ of constructor ground terms such that $\Sigma \models f(q_1, \dots, q_n) \in T$.*

The conjectures (1)-(4) are respectively *diff*(x, y)*-valid* and *quot*(x, y)*-valid*. For a closed system Σ, P is $f(\mathbf{x})$-valid iff

$$\Sigma \models \forall x_1 : T_1; \dots; x_n : T_n \bullet (\, f(\mathbf{x}) \in T \;\Rightarrow P\,).$$

It is clear that this notion of $f(\mathbf{x})$-validity does not make any sense for the classical semantics of "$\in$": $f(q_1, \dots, q_n) \in T$ holds automatically in that case, and thus $f(\mathbf{x})$-validity collapses to general (inductive) validity.

The second step in proving partial validity of a conjecture P is a proof of

$$\Sigma \models \forall x_1 : T_1; \dots; x_n : T_n \bullet \neg f(\mathbf{x}) \in T \Rightarrow P. \tag{5}$$

If (5) can be verified, then $f(\mathbf{x})$-validity of P implies that P is a consequence of each closure of Σ, and thus partially valid. To see this, let I be an interpretation that is a model of $\Sigma \cup C$ and let $q_1, \dots, q_n$ be arbitrary constructor ground terms. We have to show that I is a model of $P(q_1, \dots, q_n)$. If $\Sigma \models f(q_1, \dots, q_n) \in T$, then the claim follows from $f(\mathbf{x})$-validity of P. Otherwise, $\Sigma \not\models f(q_1, \dots, q_n) \in T$ and hence, $\Sigma \cup C \models \neg(f(q_1, \dots, q_n) \in T)$. As I is a model of $\Sigma \cup C$, I satisfies $\neg(f(q_1, \dots, q_n) \in T)$ and by (5) we have that I is a model of $P(q_1, \dots, q_n)$.

We refer to Requirement (5) as the *permissibility condition* [13]. Note that if Σ is not a closed system, then proving (5) is, of course, not the same as proving for all constructor ground terms $q_1, \dots, q_n$

$$\Sigma \not\models f(q_1, \dots, q_n) \in T \text{ implies } \Sigma \models P(q_1, \dots, q_n). \tag{6}$$

[1] It does not matter if the x_i do not occur in P, or if other variables do occur in P.

[2] We denote by $P(q_1, \dots, q_n)$ the formula P with each variable x_i replaced by q_i.

(In fact, (6) implies (5), but not vice versa.) A proof of $f(\mathbf{x})$-validity and (6) would constitute a proof of the inductive validity of P (instead of just partial validity). Proving the permissibility condition becomes trivial if suitable hypotheses are included in the conjecture, as in the conjectures (3) and (4) and the conjecture of the following example.

Example 3. Suppose we have the free type $A ::= a \mid b$, the function $f : A \nrightarrow A$, and the single axiom $f(a) = a$. To prove that

$$\forall x : A \bullet f(x) \in A \Rightarrow f(x) = x \tag{7}$$

is partially valid we first prove its $f(x)$-validity. Since f is (explicitly) defined only for a, we have to show $f(a) \in A \Rightarrow f(a) = a$, which is clearly valid by the given axiom. We now prove the permissibility condition

$$\forall x : A \bullet \neg f(x) \in A \Rightarrow (f(x) \in A \Rightarrow f(x) = x),$$

which is also clearly valid. This completes the proof of (7)'s partial validity. □

Note that our logic is non-monotonic w.r.t. extensions of the specification. For example, $f(b) \in A \Rightarrow f(b) = b$ is an instance of (7) and hence, it is partially valid. But adding $f(b) = a$ subsequently to our specification would make $f(b) \in A \Rightarrow f(b) = b$ and (7) false. (But note also that the non-monotonicity of our logic has the advantage that we never need any consistency checks, which are required in monotonic frameworks for partiality and which are difficult to automate for non-terminating functions.) We discuss this problem further in the next section.

4 Closure Induction

In principle, for $f(\mathbf{x})$-validity we have to consider infinitely many instantiations. To perform such proofs (automatically), we introduce the principle of closure induction. We restrict ourselves to equational specifications whose equations E are universally quantified over the (defined parts) of the respective types — we will frequently omit their quantifiers in the rest of this discussion.

Definition 6 (Equations Defining Functions). *A subset E' of E* defines *the function f if E' consists of all equations from E of the form $f(t_1, \dots, t_n) = r$.*

Definition 7 (Closure Induction). *Suppose that $f : T_1 \times \cdots \times T_n \nrightarrow T$ (for free types $T_1, \dots, T_n$ and T) is a declared function symbol defined by a set of equations of the form*

$$f(t_{11}, \dots, t_{1n}) = r_1,\ \dots,\ f(t_{m1}, \dots, t_{mn}) = r_m$$

such that each r_i has a (possibly empty) set of subterms of the form $\{f(\mathbf{s}_{i1}), \dots, f(\mathbf{s}_{ik_i})\}$. Let P be a quantifier-free formula, let Γ_i be the (possibly empty) conjunction of the formulae $P(\mathbf{s}_{ij})$ for $j = 1 \dots k_i$, and let $\forall \bullet F$ denote the universal *closure of any formula F. The principle of* closure induction *is the following: "from the $f(\mathbf{x})$-validity of*

$$\forall \bullet (\Gamma_1 \Rightarrow P(t_{11}, \ldots, t_{1n})) \ \wedge \ \ldots \ \wedge \ \forall \bullet (\Gamma_m \Rightarrow P(t_{m1}, \ldots, t_{mn}))$$

infer the $f(\mathbf{x})$-validity of $\forall x_1 : T_1; \ldots; x_n : T_n \bullet P$."

Note that closure induction directly corresponds to the techniques commonly used in inductive theorem proving (such as *cover set induction* or *recursion analysis*), cf. e.g. [5, 9, 19, 25, 27]. However, the important differences are that our induction principle also works for non-terminating partial functions (like the induction principle of [13]) and that it can be used in the framework of a simply-typed set-theoretic language (*unlike* the induction principle of [13]).

As closure induction proves only $f(\mathbf{x})$-validity (and it can also be applied if f is partial), to verify that P is partially valid w.r.t. the specification, we must also prove the permissibility condition (5). If we consider a specific function, say *quot*, we may express the induction principle and the associated permissibility condition in the language $\mathcal{F}$ *itself*:

$$\begin{aligned}\forall p : \mathbb{P}\,(nats \times nats) \bullet\ & (\forall y : nats \bullet (o, y) \in p \\ & \wedge\ \forall x, y : nats \bullet ((\mathit{diff}(s(x), y), y) \in p \Rightarrow (s(x), y) \in p) \\ & \wedge\ \forall x, y : nats \bullet (\neg quot(x, y) \in nats \Rightarrow (x, y) \in p)) \\ & \Rightarrow (\forall m : nats \times nats \bullet m \in p).\end{aligned}$$

Thus, we show that a conjecture p holds if *quot* is explicitly defined, and that p also holds when *quot* is not defined. Since we have expressed the principle as an $\mathcal{F}$ formula, we may add it as an axiom to the specification Σ.

This possibility of stating the induction rule on the object level is due to the expressiveness of our set-theoretic language (this was not possible in the first-order language of [13]). Not only does this demonstrate that closure induction may be simulated within $\mathcal{F}$, thus allowing a quite straightforward simulation of closure induction in CADiZ, without the need to implement the inference rule (at least for initial experimental purposes), but it also provides a partial solution to the problem of non-monotonicity. The problem is that while a new axiom may be consistent with the initially given axioms, it may not be consistent with some proven conjectures. Representing closure induction as an additional axiom eliminates this possibility, and makes more transparent to the specifier what properties are being assigned to each function.

We now describe sufficient conditions under which closure induction is sound. Firstly, the arguments to each function definition must be "constructor terms", and, secondly, if $f(q_1, \ldots, q_n)$ is equal to a type element q, then it must be "reducible" to q. For the formal expression of these conditions, we reinterpret a set of $\mathcal{F}$ equations as a set of rewrite rules. The next three definitions restate the required notions from the theory of term rewriting. For a detailed introduction to term rewriting see e.g. [2, 10].

Definition 8 (Cbv-Rewriting). *A* rewrite rule *has the form $l \rightarrow r$, where l is a non-variable term, r is a term, and every variable in r also occurs in l.*

We use the following restriction of rewriting to model a call-by-value (or "cbv") evaluation strategy. For a set of rules R, let $\Rightarrow_R$ be the smallest relation such that $s \Rightarrow_R t$ holds if there is a rule $l \to r$ in R such that some subterm s' of s is an instance $l\theta$ of l, for each variable x in l there is some constructor ground term q such that $x\theta \Rightarrow_R^* q$, and t is s with (some occurrence of) s' replaced by $r\theta$. In this case, we say that the term s cbv-rewrites in one step *to a term t via a set of rules R. A term s* cbv-rewrites *(or "*cbv-reduces*") in zero or more steps to t if $s \Rightarrow_R^* t$, the notation $\Rightarrow_R^*$ denoting the reflexive and transitive closure of $\Rightarrow_R$.*

Definition 9 (Constructor System). *Let E be the equations defining a set of functions F. Provided that orienting E from left to right yields rewrite rules, we call these rules the rewrite system* corresponding *to E. A set of rules R is a* constructor system *if the proper subterms of the left-hand sides of R-rules are built from free-type function symbols (that is, "constructors") and variables only.*

Now we introduce a localized confluence (and termination) property depending on E.

Definition 10 (Type Convergence). *Suppose that a specification Σ consists of a set of free types, a set of function declarations F, and a set of equations E defining the functions in F. If R is the set of rewrite rules corresponding to E, then we say that R is* type convergent for the function $f : T_1 \times \cdots \times T_n \nrightarrow T$ *if, whenever $\Sigma \models f(q_1, \ldots, q_n) = q$ for any constructor ground terms $q_1, \ldots, q_n, q$, then we have $f(q_1, \ldots, q_n) \Rightarrow_R^* q$; if this holds for all functions in F, then we say that R is* type convergent.

Finally, we are able to present our main result.

Theorem 1 (Soundness of Closure Induction). *Let R and f be as above. Then closure induction proves $f(\mathbf{x})$-validity if R is a constructor system that is type convergent for f.*

A proof may be found in the Appendix. Informally, the argument is as follows. If R is a type convergent constructor system for f, then, for each application $f(\mathbf{q})$ that is equal to a constructor ground term, we can find a rule $l \to r$ such that l matches $f(\mathbf{q})$ and the corresponding instances of any applications of f in r are smaller than $f(\mathbf{q})$ with respect to some particular well-founded ordering. Consequently, in the application of closure induction to a formula P, we generate all the cases for which f is defined, and, for each such case, we assume instances of P that are smaller according to this well-founded ordering. Thus, if the hypotheses of closure induction are $f(\mathbf{x})$-valid, then so is the conclusion.

That closure induction is unsound when the associated rewrite system is not type convergent is illustrated by the following.

Example 4. Let E be

$$\{f(o) = o,\ \forall x : nats \bullet f(x) = f(s(x))\}.$$

We may prove via structural induction that $\forall x : \mathit{nats} \bullet f(x) = o$ follows from E. However, by closure induction we are also able to prove the clearly false conjecture $\forall x : \mathit{nats} \bullet f(x) \geq x$ (for the usual definition of $\geq$), giving us $\forall x : \mathit{nats} \bullet o \geq x$. The proof proceeds as follows. The "base case" is $f(o) \geq o$, which is obviously valid. In the "step case" we prove

$$\forall x : \mathit{nats} \bullet f(s(x)) \geq s(x) \;\Rightarrow\; f(x) \geq x.$$

This may be reduced to

$$\forall x : \mathit{nats} \bullet f(x) \geq s(x) \Rightarrow f(x) \geq x$$

by the second defining equation of f. But this is clearly valid by the usual properties of $\geq$ (since $\forall x : \mathit{nats} \bullet f(x) = o$ and thus, $\forall x : \mathit{nats} \bullet f(x) \in \mathit{nats}$).

We are left to prove the permissibility condition, which in this case is

$$\forall x : \mathit{nats} \bullet \neg\, f(x) \in \mathit{nats} \Rightarrow f(x) \geq x.$$

But we know that $\forall x : \mathit{nats} \bullet f(x) = o$ holds, which is inconsistent with the hypothesis of this permissibility condition; thus, the condition holds trivially, and the conjecture is "proven". $\square$

The problem in this example is that, for each $n > 0$, $f(s^n(o))$ is equal to a constructor ground term, but not reducible to one via the rewrite system corresponding to the given axioms; this rewrite system is thus not type convergent. For the sound application of closure induction, whenever $f(\mathbf{q})$ is defined, the attempted proof that the conjecture holds for $\mathbf{q}$ must rely on induction hypotheses that are smaller w.r.t. a well-founded relation; for a constructor system, type convergence ensures that this condition is satisfied.

That type convergence alone is insufficient for the soundness of closure induction is illustrated by the next example. Thus, one really needs both conditions, i.e., being a constructor system and type convergence.

Example 5. Let E be

$$\{\forall x : \mathit{nats} \bullet f(x) = g(f(x)),\ \forall x : \mathit{nats} \bullet g(f(x)) = o,\ \forall x : \mathit{nats} \bullet g(x) = x\}.$$

Obviously, the rewrite system R corresponding to E is type convergent, but it is not a constructor system. By closure induction, we can prove the false conjecture

$$\forall x : \mathit{nats} \bullet f(x) \in \mathit{nats} \Rightarrow f(x) = s(o).$$

The induction formula is trivial (the induction hypothesis is equal to the induction conclusion) and the permissibility conjecture is also a tautology. $\square$

The problem in this example is that while $f(q)$ reduces to o for each constructor ground term q, the only possible such reduction in the given system is via an "expansion" step. Consequently, we again cannot construct a well-founded ordering that justifies the assumed induction hypothesis in the proposed proof, and we are not saved by a separate induction case for which the induction hypothesis can be so justified.

Finally, we give an example of the successful application of closure induction.

Example 6. Let *nats* and *diff* be as before. Suppose we wish to prove $\mathit{diff}(x,y)$-validity of

$$\forall x,y : \mathit{nats} \bullet \mathit{diff}(x,y) \in \mathit{nats} \Rightarrow \mathit{diff}(x,y) + y = x. \tag{3}$$

The rules represent a constructor system that is type convergent; we may thus apply closure induction. This involves proving

$$\forall x : \mathit{nats} \bullet \mathit{diff}(x,o) \in \mathit{nats} \Rightarrow \mathit{diff}(x,o) + o = x,$$

which reduces to the reflexivity axiom $\forall x : \mathit{nats} \bullet x = x$, and proving

$$\forall x,y : \mathit{nats} \bullet P(x,y) \Rightarrow P(s(x),s(y)),$$

where $P(r,t)$ denotes $\mathit{diff}(r,t) \in \mathit{nats} \Rightarrow \mathit{diff}(r,t) + t = r$. The proof of this second subgoal is also straightforward. To prove the partial validity of the original conjecture (3), we also need to prove the permissibility conjecture

$$\forall x,y : \mathit{nats} \bullet \neg\, \mathit{diff}(x,y) \in \mathit{nats} \Rightarrow (\mathit{diff}(x,y) \in \mathit{nats} \Rightarrow \mathit{diff}(x,y) + y = x),$$

but this is a tautology.

If we were to add the axiom $\forall x,y : \mathit{nats} \bullet \mathit{diff}(x,y) = \mathit{diff}(x,y)$ to our specification, then the associated rewrite system would still be a type convergent constructor system, and thus closure induction would still be applicable. Now an extra case would be included in which we assume the conjecture holds for (x,y) in the proof that it holds for (x,y); this clearly does not correspond to a well-founded ordering, but the $\mathit{diff}(x,y)$-validity of the conjecture will have been proven already by the other cases in the application of the closure induction.

Thus, compared to the induction principle of [13], the present principle of closure induction has the advantage that it can also deal with overlapping equations. (Another advantage over that previous principle is that the requirement of type convergence is localized to the function under consideration, i.e., the rules need not be type convergent for *other* functions.) □

This example illustrates the fact that closure induction does not involve the construction of merely a "cover set" of cases in the sense of Bronsard *et al.* [8]. Instead it constructs all cases suggested by a function definition. Utilizing only sufficient rules to cover all cases would, in fact, be unsound. For example, using just the rule $\mathit{diff}(x,y) \rightarrow \mathit{diff}(x,y)$ to generate the induction cases would allow us to prove any conjecture, as (x,y) covers all possible pairs of type elements.

5 Definedness Rules

In general, closure induction is not always sufficient to prove $f(x)$-validity. In our example, to prove the $\mathit{quot}(x,y)$-validity of

$$\forall x,y : \mathit{nats} \bullet \mathit{quot}(x,y) \in \mathit{nats} \Rightarrow \mathit{quot}(x,y) * y = x \tag{4}$$

we need to be able to make inferences about the definedness of function applications. For this, Giesl [13] has proposed definedness rules for functions; in the present context these take the form

$$\textit{from } f(\mathbf{t}) \in T \textit{ infer } t_1 \in T_1 \textit{ and } \ldots \textit{ and } t_n \in T_n$$

for any tuple of terms $\mathbf{t}$ and each n-ary function symbol.

The condition that a set of rules is both a constructor system and type convergent is not sufficient to ensure that the above definedness rules may be applied soundly. For example, consider the type convergent constructor system

$$\{f(o) \to o,\ f(o) \to f(g(o))\},$$

where $o \in \mathit{nats}$ is given and f and g are partial functions from nats to nats. The formula $f(g(o)) \in \mathit{nats}$ follows from this system, but $g(o) \in \mathit{nats}$ does not. To characterize a class of rewrite systems where the definedness rules are sound, we propose a strengthening of the notion of type convergence.

Definition 11 (Complete Type Convergence). *Let Σ be a specification which consists of a set of free types, a set of function declarations F, and a set of equations E defining the functions in F. If R is the set of rewrite rules corresponding to E, then we say that R is* completely type convergent *iff $\Sigma \models t = q$ implies $t \Rightarrow_R^* q$ for all ground terms t and all constructor ground terms q.*

For example, the specification of *diff* and *quot* is completely type convergent. Note that here the semantics of the universal quantifier $\forall$ is crucial. Since it quantifies over only the objects of nats^+, the specification does not imply equations like $\mathit{quot}(o, \mathit{quot}(s(o), o)) = o$.

The definedness rules are justified for completely type convergent constructor systems; the argument is as follows. Let C be a set of closure axioms for Σ, let $\mathbf{x}$ be the variables in $\mathbf{t}$ (of type $\mathbf{T_X}$), let $\mathbf{q}$ be a constructor ground term tuple, and let $[\mathbf{q}/\mathbf{x}]$ denote the substitution of $\mathbf{x}$ by $\mathbf{q}$. If $\Sigma \models f(\mathbf{t})[\mathbf{q}/\mathbf{x}] \in T$, then we have $\Sigma \models f(\mathbf{t})[\mathbf{q}/\mathbf{x}] = q$ for some constructor ground term q and thus, $f(\mathbf{t})[\mathbf{q}/\mathbf{x}] \Rightarrow_R^* q$ due to the *complete* type convergence of R. Consequently, the terms $t_1[\mathbf{q}/\mathbf{x}], \ldots, t_n[\mathbf{q}/\mathbf{x}]$ also cbv-rewrite to constructor ground terms (see Lemma 1 in the Appendix). It follows that $\Sigma \models t_i[\mathbf{q}/\mathbf{x}] \in T_i$ for each i, and thus

$$\Sigma \cup C \ \models\ f(\mathbf{t})[\mathbf{q}/\mathbf{x}] \in T \ \Rightarrow\ t_1[\mathbf{q}/\mathbf{x}] \in T_1 \ \wedge\ \ldots\ \wedge\ t_n[\mathbf{q}/\mathbf{x}] \in T_n \qquad (8)$$

holds. If, on the other hand, $\Sigma \not\models f(\mathbf{t})[\mathbf{q}/\mathbf{x}] \in T$, then (8) holds again, since $\Sigma \cup C \models \neg f(\mathbf{t})[\mathbf{q}/\mathbf{x}] \in T$ by the definition of closure axioms. As (8) holds for all constructor ground term tuples $\mathbf{q}$, we finally obtain the desired result $\Sigma \cup C \models \forall \mathbf{x} : \mathbf{T_X} \bullet f(\mathbf{t}) \in T \Rightarrow t_1 \in T_1 \wedge \ldots \wedge t_n \in T_n$.

We may "simulate" these definedness rules too in $\mathcal{F}$, in the following way. For every defining equation $f(\mathbf{t}) = r$ we add to our specification the implication

$$\forall \mathbf{x} : \mathbf{T_X} \bullet f(\mathbf{t}) \in T \ \Rightarrow\ r' \in T'$$

for every subterm r' of r (of type T').

Example 7. For the *quot* system we obtain (besides others) the implications

$$\forall x, y : nats \bullet quot(s(x), y) \in nats \Rightarrow quot(diff(s(x), y), y) \in nats$$
$$\forall x, y : nats \bullet quot(s(x), y) \in nats \Rightarrow diff(s(x), y) \in nats.$$

Now, using these definedness formulae, we can indeed prove the conjecture (4) by closure induction. For instance, the first implication above is used in the following way. The proof of $quot(x, y)$-validity of (4) involves the proof of

$$\ldots \Rightarrow (\, quot(s(x), y) \in nats \Rightarrow quot(s(x), y) * y = s(x)\,).$$

By the definition of *quot*, this may be reduced to

$$\ldots \Rightarrow (\, quot(s(x), y) \in nats \Rightarrow s(quot(diff(s(x), y), y)) * y = s(x)\,).$$

We now wish to apply the definition of "$*$" to the left-hand side of the equality; but for this to be possible, the property $quot(diff(s(x), y), y) \in nats$ must hold. Fortunately, since we have the hypothesis $quot(s(x), y) \in nats$, the desired property does hold by the definedness formulae for *quot*. □

Of course, we need a method to ensure (complete) type convergence automatically. Let R be the set of rewrite rules corresponding to the equations E, where R is a constructor system. Moreover, let $R' = \{l\sigma \to r\sigma \,|\, l \to r \in R, \sigma$ replaces all variables of l by constructor ground terms $\}$. Then *confluence* of R' implies complete type convergence of R. The reason is that

$\Sigma \models t = q$
iff $E' \models t = q$ where $E' = \{s_1\sigma = s_2\sigma \mid \forall \mathbf{x} \bullet s_1 = s_2 \in E, \sigma$ replaces $\mathbf{x}$ by constructor ground terms $\}$
iff $t \Leftrightarrow^*_{R'} q$ by Birkhoff's theorem [3]
iff $t \Rightarrow^*_{R'} q$ due to R''s confluence and as R' is a constructor system.

Finally, $t \Rightarrow^*_{R'} q$ of course implies $t \Rightarrow^*_R q$.

A sufficient condition for confluence of R' is the requirement that the rules in R (i.e., the defining equations E of the specification) should be non-overlapping. In other words, for two different equations $s_1 = t_1$ and $s_2 = t_2$, the terms s_1 and s_2 must not unify. For example, the equations for *diff* and *quot* are non-overlapping. This sufficient criterion can easily be checked automatically. The reason for this requirement being sufficient for R''s confluence is that $\Rightarrow_{R'}$ is equal to the innermost rewrite relation $\Rightarrow^{\mathrm{i}}_{R'}$ for ground constructor systems R' and having non-overlapping rules implies confluence of innermost reductions [16]. So compared to [13], the requirement of orthogonality is not needed due to the definition of cbv-rewriting.

6 Conclusion

We have introduced a new "closure induction" principle in order to reason about specifications in a simply-typed Z-like set-theoretic language that includes partial functions. For this purpose, we adapted Giesl's induction principle for partial functions [13]. While Giesl's induction principle was tailored to functional programs with an eager evaluation strategy, in the present paper we adapted it to equational specifications of our set-theoretic language, and exhibited sufficient conditions in order to render this induction principle correct.

In this process, we relaxed some of the assumptions Giesl made about his programs, showing that a sufficient condition for the soundness of our principle is that the rewrite system corresponding to the equations is a constructor system that is type convergent for the function under consideration. In order to employ the frequently necessary further rules for reasoning about definedness, we also have to demand complete type convergence. A sufficient syntactic criterion for complete type convergence (and thus type convergence) is that the rewrite rules corresponding to the equational definitions of a specification are a non-overlapping constructor system.

Note that the use of a much more powerful language than Giesl's partial functional programs enables us to express the induction principle within the language itself. This allows for an easy implementation of our principle and solves the non-monotonicity problem w.r.t. extensions of specifications.

For future work, we intend to find criteria for allowing non-equations in specifications, and we aim at relaxing the restriction to constructor systems. Moreover, while we do not impose the constraint that our equations are left-linear as in [13], at the moment we still have to restrict ourselves to non-overlapping equations to ensure complete type convergence; weaker criteria are needed to increase the applicability of our approach to a wider class of specifications. We plan also to consider extensions to cover *conditional* equations and to develop more specific techniques for nested or mutually recursive definitions.

A Proof of the Soundness Theorem for Closure Induction

Lemma 1. *Let R be a constructor system. For all ground terms t and all constructor ground terms q, if $t \Rightarrow_R^* q$ then each subterm of t can also be cbv-reduced to a constructor ground term.*

Proof. Suppose $t \Rightarrow_R^* q$. We proceed by induction on the structure of t. If t is a constant then the lemma is obvious. Otherwise, t has the form $f(\mathbf{t})$. If f is a constructor, then the lemma directly follows from the induction hypothesis. Otherwise, the reduction of t is as follows:

$$f(\mathbf{t}) \Rightarrow^* f(\mathbf{s}) \Rightarrow r\theta \Rightarrow^* q,$$

where $t_i \Rightarrow^* s_i$ for all i and $f(\mathbf{s}) = l\theta$ for a rule $l \to r$. Here, l has the form $f(\mathbf{u})$. By the definition of cbv-rewriting, $x\theta$ reduces to a constructor ground term for all variables x in $\mathbf{u}$. As R is a constructor system, all u_i are constructor terms and hence, each $u_i\theta$ also reduces to a constructor ground term. Thus, as $t_i \Rightarrow^* s_i = u_i\theta$, each t_i reduces to a constructor ground term. For proper subterms of the t_i, reducibility to a constructor ground term follows from the induction hypothesis. □

Note that this result does not hold for usual rewriting (instead of cbv-rewriting). For example, via the constructor system $f(x) \to o$ the term $f(g(o))$ rewrites to o, though $g(o)$ is irreducible. But when using cbv-rewriting, $g(o)$ must be reducible to a constructor ground term in order to reduce $f(g(o))$ to o.

Definition 12 (Full Reduction in n Steps). *Let R be a set of rewrite rules and let s, t be terms. We say that s* cbv-reduces to t in n steps via R, *denoted $s \Rightarrow_{R,n} t$, if there is a rule $l \to r$ in R such that t is s with the subterm $l\theta$ replaced by $r\theta$ and if, for each variable x_i $(1 \leq i \leq j)$ occurring in l, $x_i\theta$ "fully reduces" to some constructor ground term q_i in k_i steps via R, and if $k_1 + \cdots + k_j = n$. We say that s* fully reduces to t in n steps via R, *denoted $s \Rightarrow^n_R t$, if there is some t' such that $s \Rightarrow_{R,i} t' \Rightarrow^j_R t$, and $i + j + 1 = n$.*

Thus, "full reduction" counts all the rule applications involved in the rewriting.

Lemma 2. *Let R be a constructor system involving the function $f : T_1 \times \ldots \times T_n \nrightarrow T$. We define the relation $>_f$ over n-tuples of ground terms as follows: $\mathbf{s} >_f \mathbf{t}$ iff there exists a constructor ground term q such that $f\,\mathbf{s} \Rightarrow^i_R C[f\,\mathbf{t}] \Rightarrow^j_R q$ (where C denotes some context), $i > 0$, and there is no $k < i + j$ and no constructor ground term p such that $f\,\mathbf{s} \Rightarrow^k_R p$. Then $>_f$ is well founded.*

Proof. Suppose $\mathbf{s}_1 >_f \mathbf{s}_2 >_f \mathbf{s}_3 >_f \ldots$; then

$$f\,\mathbf{s}_1 \Rightarrow^{i_1}_R C_1[f\,\mathbf{s}_2], f\,\mathbf{s}_2 \Rightarrow^{i_2}_R C_2[f\,\mathbf{s}_3], \ldots,$$

and $C_1[f\,\mathbf{s}_2], C_2[f\,\mathbf{s}_3], \ldots$ all reduce to constructor ground terms. Since $f\,\mathbf{s}_1$ fully reduces to a constructor ground term in a minimum of $i_1 + j_1$ steps, the minimum number of steps for the full reduction of $C_1[f\,\mathbf{s}_2]$ is j_1. But in that case $f\,\mathbf{s}_2$ fully reduces to some constructor ground term p in at most j_1 steps, by Lemma 1. Thus, we have

$$i_1 + j_1 > j_1 \geq i_2 + j_2 > j_2 \geq i_3 + j_3 > j_3 \geq \ldots$$

But this is impossible. □

As a simple counterexample for the well-foundedness of the same relation without the minimality condition (i.e., without the requirement that $f\,\mathbf{s} \Rightarrow^k_R p$ does not hold for $k < i + j$), consider $R = \{f(o) \to o, f(o) \to f(o)\}$. This set is a constructor system and $f(o) \Rightarrow^1_R f(o) \Rightarrow^1_R o$, but $>_f$ would not be well founded, as we would have $o >_f o$.

Theorem 1 (Soundness of Closure Induction). *Let Σ be a specification with free types and a set of (universally quantified) equations and let R be the corresponding rewrite system. Then closure induction proves $f(\mathbf{x})$-validity in Σ if R is a constructor system that is type convergent for f.*

Proof. We wish to show that if the hypotheses of closure induction are $f(\mathbf{x})$-valid then so is the conclusion. Note that due to Lemma 1, a conjecture P is

$f(\mathbf{x})$-valid iff $\Sigma \models P(s_1, \ldots, s_n)$ holds for all those ground (rather than just *constructor* ground) terms $\mathbf{s}$ such that $\Sigma \models f(\mathbf{s}) \in T$. If R is type convergent, then $\Sigma \models f(\mathbf{s}) \in T$ is equivalent to the existence of a constructor ground term q with $f(\mathbf{s}) \Rightarrow_R^* q$.

Now suppose that the conclusion is false, that is, there is a term $f(s_1, \ldots, s_n)$, where $f(s_1, \ldots, s_n) \Rightarrow_R^* q$ for some ground constructor term q, such that the formula $P(s_1, \ldots, s_n)$ is false, and that $(s_1, \ldots, s_n)$ is minimal with respect to $>_f$ among such n-tuples. Without loss of generality, let $f(s_1, \ldots, s_n) \Rightarrow_R^* q$ be the minimal reduction of $f(s_1, \ldots, s_n)$ to a constructor ground term.

Since f is not a constructor, the reduction $f(s_1, \ldots, s_n) \Rightarrow_R^* q$ must involve the application of a rule $l \to r \in R$ such that $f(s'_1, \ldots, s'_n)$ is an instance $l\theta$ of l, where $s_i \Rightarrow_R^* s'_i$ for each s_i. Consequently, for any subterm $f(t_1, \ldots, t_n)$ of $r\theta$, we have that

$$(s_1, \ldots, s_n) >_f (t_1, \ldots, t_n).$$

But if all the $P(t_1, \ldots, t_n)$ were valid, then so would be $P(s'_1, \ldots, s'_n)$, by the hypotheses of closure induction, and hence $P(s_1, \ldots, s_n)$ would be valid as well, since $s_i \Rightarrow_R^* s'_i$. Thus, if $l \to r$ is a non-recursive rule, then we directly obtain a contradiction. Otherwise, one of the $P(t_1, \ldots, t_n)$ must also be false, which contradicts the $>_f$-minimality of $(s_1, \ldots, s_n)$. $\square$

Acknowledgement. We would like to thank the anonymous referees for many helpful comments.

References

1. R. D. Arthan. Undefinedness in Z: Issues for specification and proof. In *CADE-13 Workshop on Mechanisation of Partial Functions*. New Brunswick, New Jersey, USA, 1996.
2. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.
3. G. Birkhoff. On the structure of abstract algebras. *Proc. Cambridge Philos. Soc.*, 31:433–454, 1934.
4. A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14:189–235, 1995.
5. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.
6. J. Brauburger and J. Giesl. Termination analysis by inductive evaluation. In *Proc. CADE-15*, LNAI 1421, pages 254–269. Springer, 1998.
7. J. Brauburger and J. Giesl. Approximating the domains of functional and imperative programs. *Science of Computer Programming*, 35:113–136, 1999.
8. F. Bronsard, U. S. Reddy, and R. W. Hasker. Induction using term orders. *Journal of Automated Reasoning*, 16:3–37, 1996.
9. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.
10. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990.

11. D. A. Duffy. On partial-function application in Z. In *3rd Northern Formal Methods Workshop*, Ilkley, UK, 1998. Springer. http://www.ewic.org.uk/ewic/.
12. J. Giesl. The critical pair lemma: A case study for induction proofs with partial functions. Technical Report IBN 98/49, TU Darmstadt, 1998. http://www.inferenzsysteme.informatik.tu-darmstadt.de/~reports/notes/ibn-98-49.ps.
13. J. Giesl. Induction proofs with partial functions. *Journal of Automated Reasoning*, 2000. To appear. Preliminary version appeared as Technical Report IBN 98/48, TU Darmstadt, Germany. Available from http://www.inferenzsysteme.informatik.tu-darmstadt.de/~giesl/ibn-98-48.ps.
14. J. Giesl, C. Walther, and J. Brauburger. Termination analysis for functional programs. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications, Vol. III*, Applied Logic Series 10, pages 135–164. Kluwer, 1998.
15. J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4. Prentice-Hall, 1978.
16. B. Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, 34:3–23, 1995.
17. C. B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54:65–67, 1995.
18. D. Kapur. Constructors can be partial, too. In R. Veroff, editor, *Automated Reasoning and its Applications – Essays in Honor of Larry Wos*, pages 177–210. MIT Press, 1997.
19. D. Kapur and M. Subramaniam. New uses of linear arithmetic in automated theorem proving by induction. *Journal of Automated Reasoning*, 16:39–78, 1996.
20. J. M. Spivey. *The Z Notation: A Reference Manual, Second Edition.* Prentice Hall, 1992.
21. I. Toyn. Z standard (draft). Available from the Department of Computer Science, University of York at http://www.cs.york.ac.uk/~ian/zstan, 1999.
22. I. Toyn. CADiZ. Available from the Department of Computer Science, University of York at the web address http://www.cs.york.ac.uk/~ian/cadiz/home.html, 2000.
23. I. Toyn, S. H. Valentine, and D. A. Duffy. On mutually recursive free types in Z. In *Proceedings International Conference of Z and B Users, ZB2000*, LNCS. Springer, 2000. To appear.
24. S. Valentine. Inconsistency and undefinedness in Z – a practical guide. In *Proceedings 11th International Conference of Z Users, ZUM'98*, LNCS 1493, pages 233–249. Springer, 1998.
25. C. Walther. Mathematical induction. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2. Oxford University Press, 1994.
26. C.-P. Wirth and B. Gramlich. On notions of inductive validity for first-order equational clauses. In *Proc. CADE-12*, LNAI 814. Springer, 1994.
27. H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable principle of induction for equational specifications. In *Proc. CADE-9*, LNAI 310, pages 162–181. Springer, 1988.