Deductive Search for Errors in Free Data Type Specifications using Model Generation

Wolfgang Ahrendt

Department of Computing Science,
Chalmers University of Technology, Göteborg, Sweden
ahrendt@cs.chalmers.se

Abstract. The presented approach aims at identifying false conjectures about free data types. Given a specification and a conjecture, the method performs a search for a model of an according *counter specification*. The model search is tailor-made for the semantical setting of free data types, where the fixed domain allows to describe models just in terms of *interpretations*. For sake of interpretation construction, a theory specific calculus is provided. The concrete rules are 'executed' by a procedure known as *model generation*. As most free data types have infinite domains, the ability of automatically solving the non-consequence problem is necessarily limited. That problem is addressed by limiting the *instantiation* of the axioms. This approximation leads to a restricted notion of model correctness, which is discussed. At the same time, it enables model completeness for free data types, unlike approaches based on limiting the domain size.

1 Introduction

The main approaches to abstract data type (ADT) specification have in common that, unlike in pure first order logic, only certain models are considered. In the *initial semantics* approach, the domain is identified with one particular quotient over the set of terms, where the size of the single equivalence classes is 'minimal'. In the *loose semantics* approach, the signature is split up into *constructors* and (other) *function symbols*. Here, the semantical domain is identified with any quotient over the set of *constructor terms*. The function symbols are interpreted as mappings over such domains. The term 'loose' refers the possibility of one specification having a 'polymorphic' meaning, i.e. owning different models, varying particularly in the *interpretation* of the (non-constructor) function symbols. In contrast to that, initial semantics is always 'monomorphic'. This paper is concerned with *free data type* specifications, which are an important special case of loose specifications. Free data types own the additional property that different constructor terms denote different elements. The domain is therefore fixed to be the set of constructor terms. The only thing which is left open is the interpretation of the function symbols. Given an ADT specification and a

conjecture φ, we call it an error if φ is *not* a consequence of the axioms AX, regardless of whether the error intuitively lies in the axioms or the conjecture. The issue of non-consequence translates to the existence of certain models. A formula φ is *not* a consequence of a set AX of axioms, if there exists a model of AX which violates φ. Our method performs the construction of such models, which in the case of free data types reduces to the *construction of interpretations*. The advantage of having fixed domains is opposed by the disadvantage of domain infinity, caused by recursive constructors. As interpretations over infinite domains are not even countable, an automated procedure can hardly solve the issue of non-consequence in a total way. Instead, the issue is approached by solving the non-consequence problem for an *approximation of the specification*. The method generates models for *finitely many, ground instantiated axioms*. To decide if the model found is extendible to the original axioms, i.e. if the model actually reveals an error, the user can vary the number of ground instances. In spite of these restrictions, the method is complete with respect to error detection. This means that the output will complain about a conjecture whenever the conjecture is faulty.

2 Free Data Types

In the described approach, the distinction between constructors and other function symbols is such important that we completely separate both. We simply call the non-constructor function symbols 'functions'. In the following, if $\mathcal{X}$ is a family of sets, $\overline{\mathcal{X}}$ denotes the union of all sets in $\mathcal{X}$.

Signature. An *ADT signature* Σ is a tuple $(S, \mathcal{C}, \mathcal{F}, \alpha)$, where S is a finite set of sort symbols, $\mathcal{C} = \{C_s | \ s \in S\}$ is a finite family of disjoint, S-indexed sets of constructor symbols, $\mathcal{F} = \{F_s | \ s \in S\}$ is a finite family of disjoint, S-indexed sets of functions ($\overline{\mathcal{C}} \cap \overline{\mathcal{F}} = \emptyset$), and $\alpha : \overline{\mathcal{C}} \cup \overline{\mathcal{F}} \to S^*$ gives the argument sorts for every constructor or function symbol.

Example 1. We consider the following signature $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ for stacks of natural numbers (the constructors are written sans serif):

$$S = \{Nat, Stack\}$$
$$\mathcal{C} = \{\{\mathsf{zero}, \mathsf{succ}\}_{Nat}, \{\mathsf{nil}, \mathsf{push}\}_{Stack}\} \quad \mathcal{F} = \{\{pred, top\}_{Nat}, \{pop, del\}_{Stack}\}$$
$$\alpha(\mathsf{succ}) = Nat \quad \alpha(\mathsf{push}) = [Nat\ Stack] \quad \alpha(\mathsf{zero}) = \alpha(\mathsf{nil}) = \lambda \text{ (no arguments)}$$
$$\alpha(pred) = Nat \quad \alpha(top) = \alpha(pop) = Stack \quad \alpha(del) = [Nat\ Stack]$$

A concrete syntax for Σ can look like:

sorts

Nat ::= zero | succ(Nat);
$Stack$ ::= nil | push(Nat; $Stack$);

functions

$pred$: $Nat \to Nat$;
top : $Stack \to Nat$;
pop : $Stack \to Stack$;
del : $Nat \times Stack \to Stack$;

Terms. A signature induces terms in general, and constructor terms in particular. T_Σ is the set of all terms, T_s is the set of terms of sort s. V_s is the set of variables of sort s. CT_Σ is the set of all constructor terms, CT_s is the set of constructor terms of sort s, and $\mathcal{CT}_\Sigma = \{CT_s | \ s \in S\}$. We only consider signatures

where $CT_s \neq \emptyset$ for all $s \in S$. For a term $t \in T_\Sigma$ with at least i arguments, $t{\downarrow_i}$ denotes the *i-th argument* of t, such that $l(t_1, \ldots, t_i, \ldots, t_n){\downarrow_i} = t_i$.

Semantics (of functions and terms). An *$\mathcal{F}$-interpretation* $\mathcal{I}$ assigns to each function symbol f, with $f \in F_s$ and $\alpha(f) = s_1 \ldots s_n$, a mapping $\mathcal{I}(f) : CT_{s_1} \times \ldots \times CT_{s_n} \to CT_s$. If $\mathcal{I}$ is an $\mathcal{F}$-interpretation, then the pair $(\mathcal{CT}_\Sigma, \mathcal{I})$ is a *freely generated Σ-algebra.* A *variable assignment* $\beta : V_\Sigma \to CT_\Sigma$ is a mapping, such that, for $x \in V_s$, $\beta(x) \in CT_s$. For every $\mathcal{F}$-interpretation $\mathcal{I}$ and variable assignment β, the *valuation* $val_{\mathcal{I},\beta} : T_\Sigma \to CT_\Sigma$ *of terms* is defined by:
– $val_{\mathcal{I},\beta}(x) = \beta(x)$, for $x \in V_\Sigma$.
– $val_{\mathcal{I},\beta}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(val_{\mathcal{I},\beta}(t_1), \ldots, val_{\mathcal{I},\beta}(t_n))$, for $f \in \overline{\mathcal{F}}$.
– $val_{\mathcal{I},\beta}(c(t_1, \ldots, t_n)) = c(val_{\mathcal{I},\beta}(t_1), \ldots, val_{\mathcal{I},\beta}(t_n))$, for $c \in \overline{\mathcal{C}}$.
We discuss some particular features of these definitions: (a) Only function, not the constructors are interpreted by $\mathcal{I}$. (b) For a given Σ, all freely generated Σ-algebras have the same domain, which is $\mathcal{CT}_\Sigma$, the sorted partitioning of the set of constructor terms. (Therefore, *val* is not indexed by the domain.) (c) The valuation of terms can be seen as a combination of standard valuations, see "$val_{\mathcal{I},\beta}(f(\cdot,\ldots)) = \mathcal{I}(f)(val_{\mathcal{I},\beta}(\cdot),\ldots)$", and Herbrand structure valuations, see "$val_{\mathcal{I},\beta}(c(\cdot,\ldots)) = c(val_{\mathcal{I},\beta}(\cdot),\ldots)$".

Equalities are the only atoms in our logic. For_Σ is the set of first order equality formulae, built from atoms by $\neg$, $\wedge$, $\vee$, $\to$, $\forall$ and $\exists$. Literals ($\in Lit_\Sigma$) are equalities and negated equalities. Clauses ($\in Cl_\Sigma$) are disjunctions of literals. The *contrary* of a formula φ, $Contr(\varphi)$, the free variables of which are $x_1, \ldots, x_n$, is defined by $Contr(\varphi) = \exists x_1 \ldots \exists x_n.\ \neg\varphi$. The valuation $val_{\mathcal{I},\beta}$ of terms and formulae is defined as usual. It is not indexed over some domain, as the domain is fixed. We just point out that in $\forall x.\varphi$ and $\exists x.\varphi$, the x is semantically quantified over CT_s, if $x \in V_s$. Given an $\mathcal{F}$-interpretation $\mathcal{I}$, a formula $\varphi \in For_\Sigma$ is *valid in* $\mathcal{I}$, abbreviated '$\mathcal{I} \models \varphi$', if for all variable assignments β it holds that $val_{\mathcal{I},\beta}(\varphi) = true$. A freely generated Σ-algebra $(\mathcal{CT}_\Sigma, \mathcal{I})$ is a *model* of $\varphi \in For_\Sigma$ resp. $\Phi \subseteq For_\Sigma$, if $\mathcal{I} \models \varphi$ resp. $\mathcal{I} \models \psi$ for all $\psi \in \Phi$. Given $\Phi \subseteq For_\Sigma$ and $\varphi \in For_\Sigma$, then *φ is a consequence of Φ*, abbreviated '$\Phi \models_\Sigma \varphi$', if every model of Φ is a model of φ. $\models_\Sigma \varphi$ abbreviates $\emptyset \models_\Sigma \varphi$.

Example 2. Let $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ be an ADT signature with
$S = \{Nat, Bool\}$ $\mathcal{C} = \{\{\mathsf{zero}, \mathsf{succ}\}_{Nat}, \{\mathsf{tt}, \mathsf{ff}\}_{Bool}\}$ $\mathcal{F} = \{\{\}_{Nat}, \{p\}_{Bool}\}$
$\alpha(\mathsf{succ}) = \alpha(p) = Nat$ $\alpha(\mathsf{zero}) = \alpha(\mathsf{tt}) = \alpha(\mathsf{ff}) = \lambda$ Then:

– $\models_\Sigma \ \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))) \not\doteq \mathsf{succ}(\mathsf{zero})$
– $\{p(\mathsf{zero}) \doteq \mathsf{tt},\ p(x) \doteq \mathsf{tt} \to p(\mathsf{succ}(x)) \doteq \mathsf{tt}\} \ \models_\Sigma \ p(x) \doteq \mathsf{tt}$

('$\doteq$' is the equality symbol of the object logic. '$\not\doteq$' abbreviates negated equality.) An *ADT specification* is a pair $\langle \Sigma, AX \rangle$, where Σ is an ADT signature and $AX \subseteq For_\Sigma$. AX is the set of *axioms*. The notions of model and consequence are extended to specifications (while overloading '$\models$' a bit): $(\mathcal{CT}_\Sigma, \mathcal{I})$ is a *model* of $\langle \Sigma, AX \rangle$ if it is a model of AX. φ is a *consequence* of $\langle \Sigma, AX \rangle$, abbreviated '$\langle \Sigma, AX \rangle \models \varphi$', if $AX \models_\Sigma \varphi$.
'$\langle \Sigma, AX \rangle \not\models \varphi$' abbreviates that φ is *not* a consequence of $\langle \Sigma, AX \rangle$.

Example 3. The specification NatStack is given by $\langle \Sigma, AX \rangle$, where Σ is taken from *Example 1* and AX is given by

$$AX = \{\ pred(\mathsf{succ}(n)) \doteq n\ ,\ top(\mathsf{push}(n, st)) \doteq n\ ,\ pop(\mathsf{push}(n, st)) \doteq st,$$
$$del(n, \mathsf{nil}) \doteq \mathsf{nil}\ ,\ del(n, \mathsf{push}(n, st)) \doteq st\ ,$$
$$n \not\doteq n' \rightarrow del(n, \mathsf{push}(n', st)) \doteq \mathsf{push}(n', del(n, st))\ \}$$

Given a specification SPEC and a formula φ, it may be that neither φ nor the opposite, $Contr(\varphi)$, is a consequence of a specification SPEC. For instance, neither (a) NatStack $\models pred(n) \not\doteq n$ nor (b) NatStack $\models \exists n.\ pred(n) \doteq n$ holds. This is due to the underspecification of *pred*. In one model of NatStack, $\mathcal{I}(pred)(\mathsf{zero})$ is zero, falsifying (a). In another model of NatStack, $\mathcal{I}(pred)(\mathsf{zero})$ is succ(zero), falsifying (b).

Proposition 1. *Let $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ be an ADT signature and $\langle \Sigma, AX \rangle$ an ADT specification. Then:*

$$\langle \Sigma, AX \rangle \not\models \varphi$$
$$\Longleftrightarrow$$
there exists an $\mathcal{F}$-interpretation with $\mathcal{I} \models AX \cup Contr(\varphi)$

In that context, we call $\langle \Sigma, AX \cup Contr(\varphi) \rangle$ a 'counter specification', and $\mathcal{I}$ the 'counter interpretation'. Our method mainly constructs such counter interpretations.

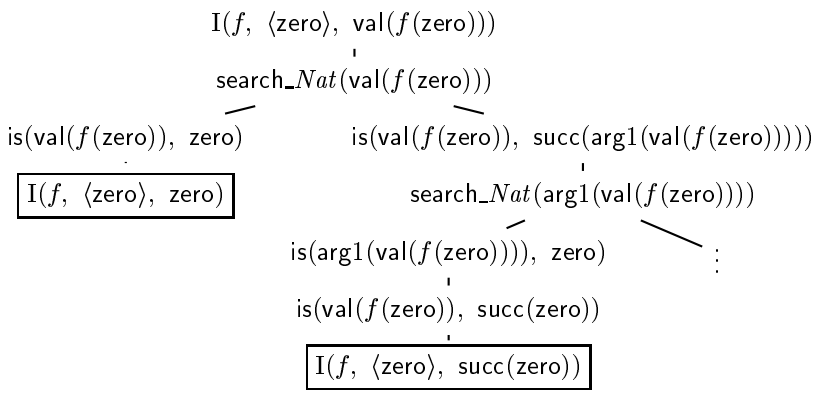
Given a specification $\langle \Sigma, AX \rangle$ and a conjecture φ, the method consists in three steps. The first is to construct *and normalise* the counter specification $\langle \Sigma, AX \cup Contr(\varphi) \rangle$. A specification is normalised if its axioms are clauses ($\in Cl_\Sigma$). Particularly, the existential quantifiers introduced by *Contr* are Skolemized away. The Skolemization handling has to respect our particular semantical setting, by adding the Skolem symbols to the *functions*, not to the constructors! The second and main step is the search for, and construction of, an according interpretation, see the following section. In the case of success, the last step consists in some post-processing, for sake of giving the user feedback in terms of the original specification and conjecture. We give examples for such output in section 7.

3 Explicit Reasoning about Interpretations

The core of the proposed method constructs $\mathcal{F}$-interpretations for normalised ADT specifications. We can think of an $\mathcal{F}$-interpretation $\mathcal{I}$ being a set of (in general infinite) tables, one for each function $f \in \overline{\mathcal{F}}$. The basic idea of our approach is to *perform reasoning about $\mathcal{F}$-interpretations using a representation that immediately describes individual lines of interpretation tables.* In particular, we represent lines of these tables as *atoms*, using the three argument predicate 'I'. $\mathrm{I}(f, \langle ct_1, \ldots, ct_n \rangle, ct)$ represents the information that $\mathcal{I}(f)(ct_1, \ldots, ct_n)$ is ct. It is important to note that such atoms are *not* part of the object logic, used in formulae $\in For_\Sigma$, if only because the object signatures we consider do not contain predicate symbols. Instead, I-atoms are formulae *on the meta level.* These (and others) will be used for a kind of 'meta reasoning'. Beside 'I-atoms', we will

use some others, calling them all 'meta atoms'. Each set of I-atoms represents a part of some $\mathcal{F}$-interpretation, if the set is *functional* in the last arguments, i.e. if there are no two I-atoms $\mathrm{I}(g, \langle ct_1, \ldots, ct_n \rangle, ct_0)$ and $\mathrm{I}(g, \langle ct_1, \ldots, ct_n \rangle, ct'_0)$ with *different* constructor terms ct_0 and ct'_0. An arbitrary set of I-atoms, not necessarily being functional, describes an *interpretation candidate*. The search for proper interpretations consists mainly in the construction of interpretation candidates, by inferring new I-atoms using proof rules. Other proof rules reject candidates, e.g. as soon as they turn out not to be functional.

The inferred I-atoms do not have the pure form sketched above, in general. Some constructor terms may be unknown, initially, and must be searched for. They are represented by *place holders*, which are replaced later. Consider a function $f : Nat \to Nat$, and suppose we are searching for the value of $\mathcal{I}(f)(\mathsf{zero})$. The following discussion is supported by the tree depicted here.



The search is initialised by creating the I-atom $\mathrm{I}(f, \langle \mathsf{zero} \rangle, \mathsf{val}(f(\mathsf{zero})))$. Its last argument, $\mathsf{val}(f(\mathsf{zero}))$, acts as a placeholder for the constructor term which we search for, and its syntax tells that it replaces the constructor term which equals '$\mathsf{val}(f(\mathsf{zero}))$'. As such, this atom does not contain much information. However, it is only meant to be a starting point. The search for a more informative last argument is initialised by adding another meta-atom, $\mathsf{search\_}Nat(\mathsf{val}(f(\mathsf{zero})))$, to the model candidate. This atom causes a branching of the candidate, where each branch corresponds to one constructor of the sort Nat. On the first branch, we assume $val_{\mathcal{I}}(f(\mathsf{zero}))$ to equal zero, by inferring $\mathsf{is}(\mathsf{val}(f(\mathsf{zero})), \mathsf{zero})$. On the second branch, we assume $val_{\mathcal{I}}(f(\mathsf{zero}))$ to equal a constructor term starting with succ, by inferring an atom of the form $\mathsf{is}(\mathsf{val}(f(\mathsf{zero})), \mathsf{succ}(\ldots))$. The left out argument of $\mathsf{succ}(\ldots)$ is explained now. On *this* branch $val_{\mathcal{I}}(f(\mathsf{zero}))$ equals $\mathsf{succ}(t)$ for some $t \in CT_{Nat}$. What we know about t is (a) that it is the *first argument* of $\mathsf{succ}(t)$, i.e. $t = \mathsf{succ}(t)\!\downarrow_1 = val_{\mathcal{I}}(f(\mathsf{zero}))\!\downarrow_1$, and therefore represent t using the syntax $\mathsf{arg1}(\mathsf{val}(f(\mathsf{zero})))$, such that we actually have $\mathsf{is}(\mathsf{val}(f(\mathsf{zero})), \mathsf{succ}(\mathsf{arg1}(\mathsf{val}(f(\mathsf{zero})))))$. What we also know about t is (b) that t is a constructor term $\in CT_{Nat}$ which we have to search for further. Corresponding to the above discussion, we also add $\mathsf{search\_}Nat(\mathsf{arg1}(\mathsf{val}(f(\mathsf{zero}))))$ to the second branch. This search-atom causes another split, which is sketched in tree.

Coming back to our first branch, it remains to propagate the information from $\mathsf{is}(\mathsf{val}(f(\mathsf{zero})),\ \mathsf{zero})$ to the initial atom $\mathrm{I}(f,\ \langle\mathsf{zero}\rangle,\ \mathsf{val}(f(\mathsf{zero})))$, by inferring $\mathrm{I}(f,\ \langle\mathsf{zero}\rangle,\ \mathsf{zero})$. A similar propagation happens twice on the first subbranch of the second branch, leading to the atom $\mathrm{I}(f,\ \langle\mathsf{zero}\rangle,\ \mathsf{succ}(\mathsf{zero}))$.

Looking at the leaves of the tree, we see that the different possible values of $\mathcal{I}(f)(\mathsf{zero})$ are enumerated. If this was everything we wanted, we should not have chosen a deductive treatment. But at first, this mechanism will interfere with others explained below. And at second, the stepwise construction of constructor terms from the outer to the inner allows to *reject* a model candidate earlier, in some cases it enables rejection in finite time at all. In our example, the term $\mathsf{succ}(\mathsf{arg1}(\mathsf{val}(f(\mathsf{zero}))))$ represents all (i.e. infinitely many) terms starting with succ.

After this demonstration, we introduce the *rules* we used, denoting them in a tableaux style. x, y, z, fv and tv are rule variables.

$$\frac{\mathsf{search\_}Nat(x)}{\mathsf{is}(x,\ \mathsf{zero}) \qquad \mathsf{is}(x,\ \mathsf{succ}(\mathsf{arg1}(x)))\ \ \mathsf{search\_}Nat(\mathsf{arg1}(x))} \qquad \frac{\mathrm{I}(fv,\ tv,\ x) \qquad \mathsf{is}(x,\ z)}{\mathrm{I}(fv,\ tv,\ z)} \qquad \frac{\mathsf{is}(x,\ \mathsf{succ}(y)) \qquad \mathsf{is}(y,\ z)}{\mathsf{is}(x,\ \mathsf{succ}(z))}$$

In the following, we turn over to use a linear notation for such rules, using the general pattern:

$$\underbrace{at_1, \ldots, at_n}_{\text{premise}} \rightarrow \overbrace{\underbrace{at_{11}, \ldots, at_{1n_1}}_{\text{1.extension}}\ ;\ \ldots\ ;\ \underbrace{at_{m_1}, \ldots, at_{mn_m}}_{\text{m.extension}}}^{\text{conclusion}}\ .$$

This simplifies the task of defining the transformation of specifications into rules. Moreover, this notation of rules is very close to the input notation of the tool we later use for rule execution.

4 Transforming the Signature

The linear notation of the above $\mathsf{search\_}Nat$-rule is:

$\mathsf{search\_}Nat(x)\ \rightarrow\ \mathsf{is}(x, \mathsf{zero})\ ;\ \mathsf{is}(x, \mathsf{succ}(\mathsf{arg1}(x)))),\ \mathsf{search\_}Nat(\mathsf{arg1}(x))\ .$

We now define the general case.

Definition 1. *Let $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ be an ADT signature, with $s \in S$ and $C_s = \{c_1, \ldots, c_n\}$, where $|\alpha(c_i)| \leq |\alpha(c_j)|$ for $i \leq j$. Then*

$$\begin{array}{c} \mathit{TransSort}_\Sigma(s) \\ = \\ \mathsf{search\_}s(x) \rightarrow\ \mathit{TransConstr}_\Sigma(x, c_1)\ ;\ \ldots\ ;\ \mathit{TransConstr}_\Sigma(x, c_n)\ . \end{array}$$

Note the semi-colon between the different extensions of the rule. The condition $|\alpha(c_i)| \leq |\alpha(c_j)|$ ensures that we order the extensions after the number of the constructor's arguments. The individual extensions are defined as follows.

Definition 2. *Let Σ be an ADT signature, with $c \in C_\Sigma$.*
– if $|\alpha(c)| = 0$, then: $\mathit{TransConstr}_\Sigma(x, c)\ =\ \{\ \mathsf{is}(x, c)\ \}$

– *if* $\alpha(c) = s_1 \ldots s_n$, *then:*

$$TransConstr_\Sigma(x, c) = \{\ \mathsf{is}(x, c(\mathsf{arg1}(x), \ldots, \mathsf{arg}n(x)))\ ,\ \mathsf{search\_}s_1(\mathsf{arg1}(x))\ ,\ \ldots,\ \mathsf{search\_}s_n(\mathsf{arg}n(x))\ \}$$

In concrete rules resulting from the transformation, we skip the set braces. Here is the result of $TransSort_\Sigma(Stack)$:

$$\mathsf{search\_}Stack(x) \rightarrow \mathsf{is}(x, \mathsf{nil})\ ;\ \mathsf{is}(x, \mathsf{push}(\mathsf{arg1}(x), \mathsf{arg2}(x)))\ ,\ \mathsf{search\_}Nat(\mathsf{arg1}(x))\ ,\ \mathsf{search\_}Stack(\mathsf{arg2}(x))\ .$$

We now introduce the handling of (in)equality, discussing concrete rules, for *Nat* and *Stack* at first.

$$\mathsf{same}(\mathsf{succ}(x),\ \mathsf{zero}) \rightarrow .$$
$$\mathsf{same}(\mathsf{push}(x_1, x_2),\ \mathsf{push}(y_1, y_2)) \rightarrow \mathsf{same}(x_1, y_1)\ ,\ \mathsf{same}(x_2, y_2)\ .$$
$$\mathsf{different}(\mathsf{zero},\ \mathsf{zero}) \rightarrow .$$
$$\mathsf{different}(\mathsf{push}(x_1, x_2),\ \mathsf{push}(y_1, y_2)) \rightarrow \mathsf{different}(x_1, y_1)\ ;\ \mathsf{different}(x_2, y_2)\ .$$

The first and the third rule cause the proof procedure to *reject* a model candidate. Note that the last rule is a branching rule. We define the general case now:

Definition 3. *Let* $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ *be an ADT signature. The rules reflecting the 'freely generatedness', are contained in* $FreeGen_\Sigma = TestSame_\Sigma \cup TestDiff_\Sigma$.

– $TestSame_\Sigma$ *is the smallest set fulfilling:*
  - *for each two different constructors* c_1, c_2 *of the same sort* ($\{c_1, c_2\} \subseteq C_s$), *where* $|\alpha(c_1)| = n$ *and* $|\alpha(c_2)| = m$,
    $\mathsf{same}(c_1(x_1, \ldots, x_n),\ c_2(y_1, \ldots, y_m)) \rightarrow . \in TestSame_\Sigma$
  - *for each constructor* $c \in C_s$, *where* $|\alpha(c)| = n \neq 0$
    $\left.\begin{array}{l}\mathsf{same}(c(x_1, \ldots, x_n),\ c(y_1, \ldots, y_n)) \\ \quad \rightarrow \mathsf{same}(x_1,\ y_1)\ ,\ \ldots\ ,\ \mathsf{same}(x_n,\ y_n)\ .\end{array}\right\} \in TestSame_\Sigma$

– $TestDiff_\Sigma$ *is the smallest set fulfilling:*
  - *for each constructor* $c \in C_s$,
    * *if* $\alpha(c) = \lambda$, *then*
      $\mathsf{different}(c,\ c) \rightarrow . \in TestDiff_\Sigma$
    * *if* $|\alpha(c)| = n \neq 0$, *then:*
      $\left.\begin{array}{l}\mathsf{different}(c(x_1, \ldots, x_n),\ c(y_1, \ldots, y_n)) \\ \quad \rightarrow \mathsf{different}(x_1,\ y_1)\ ;\ \ldots\ ;\ \mathsf{different}(x_n,\ y_n)\ .\end{array}\right\} \in TestDiff_\Sigma$

The same- and different-atoms are introduced either by transformed axioms (see below), or by the following rule which 'checking' for *functionality.*

$$\mathrm{I}(fv, tv, z)\ ,\ \mathrm{I}(fv, tv, z')\ \rightarrow\ \mathsf{same}(z, z')\ .$$

In the end of section 3, we encountered the two rules:

$$\mathsf{is}(x, \mathsf{succ}(y))\ ,\ \mathsf{is}(y, z)\ \rightarrow\ \mathsf{is}(x, \mathsf{succ}(z))\ .$$
$$\mathrm{I}(fv, tv, x)\ ,\ \mathsf{is}(x, z)\ \rightarrow\ \mathrm{I}(fv, tv, z)\ .$$

The general case of such 'replacement' rules is described here only informally. The first rule must be provided for each constructor, and for each of a constructor's argument positions. The second rule is general enough. In addition, we need similar rules to replace each position in the tuples of I-atoms, as well as rules for replacing arguments of same() and different().

5 Transforming the Axioms

The rules discussed so far only consider the signature. But we actually are searching for a *model* of a specification, i.e. for a model of its *axioms*. In our approach, also (or particularly) the axioms are transformed to rules. We now explain this transformation, using very simple examples at the beginning. We start with ground equalities. Let $f_1(ct_1) \doteq f_2(ct_2)$ be an axiom, where f_1, f_2 are functions of some sort s, and ct_1, ct_2 are constructor terms. This equality can be represented by the rule:

$$\rightarrow \mathrm{I}(f_1, \langle ct_1 \rangle, \mathsf{val}(f_1(ct_1))) \ , \ \mathsf{search\_}s(\mathsf{val}(f_1(ct_1))) \ , \ \mathrm{I}(f_2, \langle ct_2 \rangle, \mathsf{val}(f_2(ct_2))) \ , \ \mathsf{search\_}s(\mathsf{val}(f_2(ct_2))) \ , \ \mathsf{same}(\mathsf{val}(f_1(ct_1)), \mathsf{val}(f_2(ct_2))) \ .$$

The rule intuitively says that we have to search for the two last arguments of the I-atoms, but with the constraint that they have to be the same. The empty premise means that the extension atoms can be added to *any* model candidate. In practice, the rule will be applied towards the beginning, before the initial model candidate branches. A transformation of $f_1(ct_1) \not\doteq f_2(ct_2)$ results in almost the same rule, just that we have '$\mathsf{different}$' instead of 'same'.

The rule for $f_1(ct_1) \doteq f_2(ct_2)$ can be optimised, by loss of its symmetry. Instead of *twice* searching for something that should finally be the 'same' thing, it suffices to search for one of both:

$$\rightarrow \mathrm{I}(f_1, \langle ct_1 \rangle, \mathsf{val}(f_2(ct_2))) \ , \ \mathrm{I}(f_2, \langle ct_2 \rangle, \mathsf{val}(f_2(ct_2))) \ , \ \mathsf{search\_}s(\mathsf{val}(f_2(ct_2))) \ .$$

This is of course more efficient. Moreover, the examples are easier to understand when the resulting rules are as short as possible. On the other hand, the definition of the transformation is much simpler in a version that is not optimised and therefore more regular. In this paper, only define the unoptimised transformation formally. However, in the example transformations, we also show the optimised versions, which are more readable. A formal definition of the optimised transformation is given in [Ahr01, Sect. 3.2.3]. (Note that the rule for $f_1(ct_1) \not\doteq f_2(ct_2)$ cannot be optimised similarly.)

The next example shows that, in general, we have to transform function terms in a recursive manner. The (again quite artificial) axiom $f_1(f_2(ct_1)) \doteq ct_2$ translates to the rule:

$$\rightarrow \ \mathrm{I}(f_2, \langle ct_1 \rangle, \mathsf{val}(f_2(ct_1))) \ , \ \mathsf{search\_}s(\mathsf{val}(f_2(ct_1))) \ , \ \mathrm{I}(f_1, \langle \mathsf{val}(f_2(ct_1)) \rangle, ct_2) \ .$$

Intuitively, this says that the last argument of $\mathrm{I}(f_2, \langle ct_1 \rangle, \mathsf{val}(f_2(ct_1)))$ is a not yet known constructor term, which has to be searched for. What *is* known about $\mathsf{val}(f_2(ct_1))$ is represented by the I-atom $\mathrm{I}(f_1, \langle \mathsf{val}(f_2(ct_1)) \rangle, ct_2)$.

So far, we discussed ground axioms, for simplicity. We now consider the axiom $f(x) \doteq ct$. The resulting rule is:

$$s(x) \ \rightarrow \ \mathrm{I}(f, \langle x \rangle, ct) \ .$$

'Binding' variables by *sort predicates* is a technique widely used. The operational meaning for the above rule is that, whenever we have $s(ct')$ on the current branch, for some ct', then we can infer $\mathrm{I}(f, \langle ct' \rangle, ct)$. In general, we can have functions on both side, as well as nested functions. The transformation then follows the same patterns as sketched above for the ground case, but finally 'binding ' all

variables by providing sort atoms in the rule premise. We demonstrate this in another example: the commutativity axiom $f(x,y) \doteq f(y,x)$ becomes

$$s(x),\ s(y)\ \rightarrow$$
$$\mathrm{I}(f, \langle x, y\rangle, \mathsf{val}(f(y,x)))\ ,\ \mathrm{I}(f, \langle y, x\rangle, \mathsf{val}(f(y,x)))\ ,\ \mathsf{search\_}s(\mathsf{val}(f(y,x)))\ .$$

Note again that x and y are 'rule variables'. They do not appear on the branches, which are always ground. The application of this rule generates a *new* place holder $\mathsf{val}(f(ct, ct'))$ for every pair ct and ct' for which s-predicates are provided. The difference between our usage of val *terms*, and computing new place holder *symbols* in each rule application, is that we can possibly *reuse* the val terms, even when applying other rules. Therefore, the usage of val terms has similarities to the usage of ϵ-terms in free variable tableaux, described in [GA99]. In our context, the less place holders we produce, the less searches we start.

In the general case, axioms of normalised specifications are clauses, i.e. disjunctions of equalities and inequalities. The according rules can directly reflect the disjunction in the 'branching', by transforming each literal to a distinct extension of the rule.

Definition 4. *Be $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ an ADT signature and $ax \in Kl_\Sigma$, with $ax = lit_1 \vee \ldots \vee lit_n$, $Var(ax) = \{x_1, \ldots, x_m\}$ and $sort(x_i) = s_i$. Then:*

$$TransAxiom_\Sigma(ax)\ =$$
$$s_1(x_1)\ ,\ \ldots\ ,\ s_m(x_m)\ \rightarrow\ TransLit_\Sigma(lit_1)\ ;\ \ldots\ ;\ TransLit_\Sigma(lit_n)\ .$$

In the following definition of $TransLit_\Sigma$ and its subparts, we use '$Rep(t)$' to factor out a certain case distinction. The 'representation' of a term $t \in T_\Sigma$, $Rep(t)$, is defined to be t itself, if t contains only constructors and variables, or $\mathsf{val}(t)$, if t contains any functions. In particular, Rep is the identity for variables as well as for constructor terms.

Definition 5. *Let $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ be an ADT signature.*

- *Let $t_1, t_2 \in T_s$ for some $s \in S$. Then:*

$$TransLit_\Sigma(t_1 \doteq t_2) =$$
$$\{\ \mathsf{same}(Rep(t_1), Rep(t_2))\ ,\ TransTerm_\Sigma(t_1)\ ,\ TransTerm_\Sigma(t_2)\ \}$$
$$TransLit_\Sigma(t_1 \not\doteq t_2) =$$
$$\{\ \mathsf{different}(Rep(t_1), Rep(t_2))\ ,\ TransTerm_\Sigma(t_1)\ ,\ TransTerm_\Sigma(t_2)\ \}$$

- *Let $t \in T_\Sigma$. Then:*
  - *if t contains no functions, then:* $\quad TransTerm_\Sigma(t) \quad = \quad \emptyset$
  - *if $t = a$ with $a \in F_s$, $\alpha(a) = \lambda$, then:*

    $$TransTerm_\Sigma(t) = \{\ \mathrm{I}(a, \langle\rangle, \mathsf{val}(a)),\ \mathsf{search\_}s(\mathsf{val}(a))\ \}$$

  - *if $t = f(t_1, \ldots, t_n)$ with $f \in F_s$, $\alpha(f) = s_1 \ldots s_n$, then:*

$$TransTerm_\Sigma(t) =$$
$$\{\ \mathrm{I}(f, \langle Rep(t_1), \ldots, Rep(t_n)\rangle, \mathsf{val}(t))\ ,\ \mathsf{search\_}s(\mathsf{val}(t))\ \}$$
$$\cup$$
$$\bigcup_{i=1}^{n} TransTerm_\Sigma(t_i)$$

- *if* $t = c(t_1, \ldots, t_n)$ *with* $c \in C_s$, $\alpha(c) = s_1 \ldots s_n$, *and if* t *contains functions, then:*

$$\begin{gathered} \mathit{TransTerm}_\Sigma(t) = \\ \{\ \mathsf{is}(\mathsf{val}(t), c(\mathit{Rep}(t_1), \ldots, \mathit{Rep}(t_n)))\ \} \ \cup \ \bigcup_{i=1}^{n} \mathit{TransTerm}_\Sigma(t_i) \end{gathered}$$

Note that $\mathit{TransTerm}_\Sigma(t)$ is empty exactly when $\mathit{Rep}(t)$ is t. This means that the recursion stops at terms that can be represented by themselves in the resulting rules. In contrast, function terms can only appear nested in val terms, i.e. place holders. Also note that we took the liberty to take over the variables as they are, even if in the rules they act as 'rule variables', to be matched/instantiated by rule application.

As an example, we show the result of transforming the (normalised) last axiom of NatStack (see Example 3).

$$\begin{gathered} \mathit{TransAxiom}_\Sigma(\ n \doteq m \ \vee \ \mathit{del}(n, \mathsf{push}(m, st)) \doteq \mathsf{push}(m, \mathit{del}(n, st))\ ) \\ = \\ \mathit{Nat}(n),\ \mathit{Nat}(m),\ \mathit{Stack}(st) \\ \rightarrow \\ \mathsf{same}(n, m)\ ; \\ \mathsf{same}(\mathsf{val}(\mathit{del}(n, \mathsf{push}(m, st))),\ \mathsf{val}(\mathsf{push}(m, \mathit{del}(n, st)))), \\ \mathrm{I}(\mathit{del}, \langle n, \mathsf{push}(m, st)\rangle, \mathsf{val}(\mathit{del}(n, \mathsf{push}(m, st)))), \\ \mathsf{search\_}\mathit{Stack}(\mathsf{val}(\mathit{del}(n, \mathsf{push}(m, st)))), \\ \mathsf{is}(\mathsf{val}(\mathsf{push}(m, \mathit{del}(n, st))), \mathsf{push}(m, \mathsf{val}(\mathit{del}(n, st)))), \\ \mathrm{I}(\mathit{del}, \langle n, st\rangle, \mathsf{val}(\mathit{del}(n, st))), \\ \mathsf{search\_}\mathit{Stack}(\mathsf{val}(\mathit{del}(n, st)))\ . \end{gathered}$$

The optimised version results in a shorter rule:

$$\begin{gathered} \mathit{Nat}(n),\ \mathit{Nat}(m),\ \mathit{Stack}(st) \\ \rightarrow \\ \mathsf{same}(n, m)\ ; \\ \mathrm{I}(\mathit{del}, \langle n, \mathsf{push}(m, st)\rangle, \mathsf{val}(\mathsf{push}(m, \mathit{del}(n, st)))), \\ \mathsf{is}(\mathsf{val}(\mathsf{push}(m, \mathit{del}(n, st))), \mathsf{push}(m, \mathsf{val}(\mathit{del}(n, st)))), \\ \mathrm{I}(\mathit{del}, \langle n, st\rangle, \mathsf{val}(\mathit{del}(n, st))), \\ \mathsf{search\_}\mathit{Stack}(\mathsf{val}(\mathit{del}(n, st)))\ . \end{gathered}$$

We also show the transformation of another, simpler axiom:

$$\begin{gathered} \mathit{TransAxiom}_\Sigma(\ \mathit{pop}(\mathsf{push}(n, st)) \doteq st\ ) \\ = \\ \mathit{nat}(n), \mathit{stack}(st) \\ \rightarrow \\ \mathsf{same}(\mathsf{val}(\mathit{pop}(\mathsf{push}(n, st))), st), \\ \mathrm{I}(\mathit{pop}, \langle \mathsf{push}(n, st)\rangle, \mathsf{val}(\mathit{pop}(\mathsf{push}(n, st)))), \\ \mathsf{search\_}\mathit{Stack}(\mathsf{val}(\mathit{pop}(\mathsf{push}(n, st))))\ . \end{gathered}$$

In this quite typical axiom pattern, the optimised transformation gains an enormous simplification:

$$nat(n)\ ,\ stack(st)\ \rightarrow\ \mathrm{I}(pop, \langle \mathsf{push}(n, st)\rangle, st)\ .$$

6 Model Generation for Approximated Specifications

The rules transformed from the axioms can only be applied if the current model candidate contains appropriate sort atoms. Ideally, we would need to have $s(ct)$ for *every* constructor term ct of sort s, and that for each sort. This cannot be realized in finite time. But what we want is a method which *terminates in case there is a model*. This makes the real difference to the traditional (refutational) methods for *proving* conjectures. Our approach to the issue of model construction for free data type specifications does not solve the problem completely. An automated method hardly can. Instead we construct a model of an *approximation* of the specification.

Let us assume now that the constructors are recursive (which is mostly the case) and, therefore, determine an *infinite* domain. The set of (quantifier free) axioms is then equivalent to an *infinite set of ground axioms*, which results from instantiating the variables by all constructor terms. We now approximate the specification by considering a *finite subset* of these ground axioms, which results from instantiating the variables by a *finitely many constructor terms*. Particularly, we limit the number of instances by limiting their 'size', which is simply defined to be the number of constructors. '$\langle \Sigma, AX_{\leq n}\rangle$' denotes such an '$n$-restricted specification', where the axioms are instantiated by all constructor terms of maximal size n. The instantiation of axioms is reflected by applying the rules transformed from the axioms, where the arguments of the matched sort atoms are the instances. Therefore, to make the rules search for a model of '$\langle \Sigma, AX_{\leq n}\rangle$', we just initialise the first model candidate to be the set of sort atoms for all constructor terms up to size n. In the theorems below, we call this 'n-initialisation'. In practice, the n has to be rather small. But, depending on the signature, the *number* of terms is significantly bigger than their maximal size.

On this initial 'model candidate' (at first only containing sort atoms), the rules, transformed from a (counter) specification, are 'executed' by some procedure. We use a procedure known as *model generation* ([MB88], [FH91]), which can be seen positive, regular hyper tableaux. The regularity ensures termination in case every matching rule has *at least one extension* not adding anything new to the branch. If one branch cannot be extended further, model generation stops. In the theorems below, we call this 'termination by saturation' (in contrast to 'termination by rejection'). Our realization uses the tool 'MGTP' (model generation theorem prover, [FH91]) for executing the described rules. The input of a model generation procedure is a set of what they called 'clauses', which corresponds to what we called 'rules'. These 'clauses' have the general form depicted in the end of section 3. In addition, the rules must be 'range restricted', which

means that each variable must also occur on the left side of '$\rightarrow$'. Our rules fulfil that restriction.

Taking the basic model generation procedure which is implemented in MGTP as an execution model for the transformed rules, we state the following

Theorem 1. *(n-restricted model correctness)*
Let $\langle \Sigma, AX \rangle$ be a normalised ADT spec., $n \in \mathbb{N}$ and $R = TransSpec(\langle \Sigma, AX \rangle)$. If the n-initialised model generation procedure with input R terminates by saturation, then (a) $\langle \Sigma, AX_{\leq n} \rangle$ has a model, and (b) for every $\mathcal{F}$-interpretation $\mathcal{I}$ which corresponds to the I*-atoms on the saturated branch, it holds that $\mathcal{I} \models \langle \Sigma, AX_{\leq n} \rangle$.*

Theorem 2. *(model completeness)*
Let $\langle \Sigma, AX \rangle$ be a normalised ADT spec., $n \in \mathbb{N}$ and $R = TransSpec(\langle \Sigma, AX \rangle)$. If $\langle \Sigma, AX \rangle$ has a model, then an n-initialised, fair *model generation procedure with input R terminates by saturation, and for every $\mathcal{F}$-interpretation $\mathcal{I}$ which corresponds to the* I*-atoms on the saturated branch, it holds that $\mathcal{I} \models \langle \Sigma, AX_{\leq n} \rangle$.*

The fairness in Theorem 2 is a requirement not implemented in MGTP. In practice, this is less important than in theory, as the search for constructor terms builds small terms first, and as small terms usually suffice to find a validating interpretation. However, the rules as such are complete, and this independent of n! Note that the theorem says "If $\langle \Sigma, AX \rangle$ has a model" instead of "If $\langle \Sigma, AX_{\leq n} \rangle$ has a model". We translate the completeness result to the non-consequence problem we are originally interested in. If it holds that '$\langle \Sigma, AX \rangle \not\models \varphi$, then model generation applied to the transformation of '$\langle \Sigma, AX \cup Contr(\varphi) \rangle$ terminates by saturation.

Both proofs for these theorems are nontrivial, particularly the completeness argument, which requires a termination argument, to be inferred from the model which is assumed to exist. The detailed proofs are given in [Ahr01].

7 Implementation and Examples

The method is implemented as a JAVA program, which, given a specification $\langle \Sigma, AX \rangle$ and a conjecture φ, (a) computes the transformation of the normalisation of $\langle \Sigma, AX \cup Contr(\varphi) \rangle$, (b) calls MGTP, and in case of saturation (c) analyses the saturated branch, producing an output both to the prompt and to a LaTeX file, telling why φ might not be a consequence of $\langle \Sigma, AX \rangle$.

For instance, given NatStack and the conjecture $del(top(st), st) \doteq pop(st)$, the (abbreviated) LaTeX output is:
the conjecture `del( top( ST ), ST ) = pop( ST )`
is violated by the following variable assignment: `ST : nil`
and by the following evaluation of conjecture subterms:
`del(top(ST),ST) : nil`
`top(ST) : zero`
`pop(ST) : push(zero,nil)`

The interpretation found by the system satisfies the axioms,
if instantiated by constructor terms with less than 4 constructors!
(*end of output*)
The warning reminds the user on what we called *n-restricted* correctness. Nevertheless, the system tells that the specification allows $pop(\mathsf{nil})$ being evaluated to $\mathsf{push}(\mathsf{zero}, \mathsf{nil})$, in which case $del(top(\mathsf{nil}), \mathsf{nil}) \doteq pop(\mathsf{nil})$ is false, and therefore the conjecture is false. This shows that either the conjecture or the specification has to be changed. Another example for a false conjecture on NatStack which the system complains about is $\mathsf{push}(top(st), pop(st)) \doteq st$.

Due to the *n-restricted* correctness, the system possibly can complain about a conjecture that actually *is* a consequence of the axioms. This happens for instance when we ask if $p(x) \doteq \mathsf{tt}$ is a consequence of (see Example 2, page 3):

$$\{p(\mathsf{zero}) \doteq \mathsf{tt},\ \ p(x) \doteq \mathsf{tt} \to p(\mathsf{succ}(x)) \doteq \mathsf{tt}\}$$

The system complains about this conjecture, because it can always construct $\mathrm{I}(p, \langle \mathsf{succ}(ct) \rangle, \mathsf{ff})$ for a ct which is slightly bigger than the size restriction n.

The last example we mention here is based on a specification taken from [Thu98]. Even if [Thu98] also investigates errors in specifications, this error is neither discussed nor detected nor even intended there. We refer to that revealed error not to blame the author, but to demonstrate how easily such errors happen, even in a context where one is very aware of the possibility of errors. (In general, an more open exchange of errors that really happen would be of great benefit to the development of error revealing techniques.) The cited specification is intended to describe a 'merge sort' algorithm. The two main axioms are: $sort(\mathsf{empty}) \doteq \mathsf{empty}$ and $sort(append(l, l')) \doteq merge(sort(l), sort(l'))$.

Our system, when being asked if the singleton list is stable under *sort*, i.e. $sort(\mathsf{cons}(n, \mathsf{empty})) \doteq \mathsf{cons}(n, \mathsf{empty})$, complains and suggests to evaluate $sort(\mathsf{cons}(n, \mathsf{empty}))$ to empty (!), as this is consistent with the specification, which does not specify at all how to sort a singleton. (To comprehend this, it suffices to know two more axioms: $merge(\mathsf{empty}, l') \doteq l'$ and $merge(l, \mathsf{empty}) \doteq l$.) As any other sorting reduces to sorting the singleton, the specification does not specify the sorting of *any* (but the empty) list.

8 Related Work and Conclusion

The works related to our task and approach can be divided in two (overlapping) fields: (1.) model construction and (2.) detecting faulty conjectures. In the first area, there are several methods searching for finite domain models. The methods described in [Sla94] and [ZZ96] search for models of a fixed size, whereas [BT98] dynamically extends the finite domain. As free data types usually have infinite domains, these finite domain methods cannot directly be applied to our setting. (A further discussion follows bellow.) Other methods in the first area are more syntax oriented, describing models by (extensions of) formulae ([FL96], [CP00]). These approaches construct models for *first order* formulae, usually not containing equalities. Our object logic, however, is completely equality based and, because of constructor generatedness, beyond first order. In the second area, a

lot of work is done in the context of initial (or rewrite) semantics, where due to monomorphicity, the notions of proof and consistency are very close ([Bac88]). Also where monomorphicity is imposed by purely sysntactical means, the detection of faulty conjectures reduces to proving their opposite ([Pro92]). In that context, even the *correction* of faulty conjectures is examined ([Pro96], [MBI94]).

To the best knowledge of the author, the only work that is similarly dedicated to the detection of faulty conjectures in *loose* specifications (not even restricted to *free* data types), is [Thu98,RST01]. There, a 'counter example' is essentially a falsifying variable assignment, rather than a falsifying model. Unsurprisingly, that method, as well as ours, cannot totally solve the issue of non-consequence. During the construction of falsifying variable assignments, the method produces side condition, the consistence of which left to be judged by the user. Like in our approach, the user has to take the final decision. The assignment of values to axiom variables is included in our method (see the example outputs). Moreover, our method analyses possible valuation of function terms.

We conclude by stressing the main features of the presented approach and its implementation. We provide a fully automated method which is tailor-made for *detecting non-consequence* between a free data type specification and a conjecture. It searches for a counter model, basically by constructing an interpretation table and searching for its entries. The user receives feedback in form of variable assignments and subterm evaluations which falsify the conjecture. To enable termination, the property of a falsifying interpretation to actually *be* a model of the specification is approximated only. This is done by instantiating the axioms with terms of a limited size only. This size is a parameter of the method and its implementation. The price of the limited term size is a restricted model correctness: a model of a limited instantiation is not necessarily a model of the full specification. The user must attack this problem by (a) examining the proposed term evaluations and (b) varying the term size limit.

It is important not to confuse our limited instantiation of the axioms with a limited domain size. In our case, by increasing the limit we can only lose models, and by decreasing the limit, we can only gain models. This is the very reason why our model completeness result is *not* restricted by the chosen limit. Such a monotonous behaviour would not hold if we varied domain sizes. We could not gain model completeness by following a similar approach like [BT98] (see above). Model construction is the means rather that the purpose of our method. We finally want to detect faulty conjectures. From this pint of view, having model completeness is worth to pay a price for. The system indeed detects all non-consequences, even if it detects to many. At the same time, the restrictions are kept transparent to the user (see the example output above). In case the error is real, it is usually not difficulty to comprehend once one is pointed to. Providing unexpected valuations of function terms then helps to identify underspecified properties which are the source of errors.

Acknowledgements I am grateful to Reiner Hähnle for his general support as well as for many, many, fruitful discussions, and for carefully checking the proofs in [Ahr01]. I am also grateful to Sonja Pieper for implementing the presented method.

References

[Ahr01] Wolfgang Ahrendt. *Deduktive Fehlersuche in Abstrakten Datentypen.* 2001. Dissertation (preversion, in German), University of Karlsruhe, available under `http://www.cs.chalmers.se/~ahrendt/cade02/diss.ps.gz`.

[Bac88] Leo Bachmair. Proof by consistency in equational theories. In *Proc. Third Annual Symposium on Logic in Computer Science, Edinburgh, Scotland*, pages 228–233. IEEE Press, 1988.

[BT98] François Bry and Sunna Torge. A deduction method complete for refutation and finite satisfiability. In *Proc. 6th European Workshop on Logics in AI (JELIA)*, volume 1489 of *LNAI*, pages 122–136. Springer-Verlag, 1998.

[CP00] Ricardo Caferra and Nicolas Peltier. Combining enumeration and deductive techniques in order to increase the class of constructible infinite models. *Journal of Symbolic Computation*, 29:177–211, 2000.

[FH91] Hiroshi Fujita and Ryuzo Hasegawa. A model generation theorem prover in KL1 using a ramified-stack algorithm. In Koichi Furukawa, editor, *Proceedings 8th International Conference on Logic Programming, Paris/France*, pages 535–548. MIT Press, 1991.

[FL96] Christian Fermüller and Alexander Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 6(2), 1996.

[GA99] Martin Giese and Wolfgang Ahrendt. Hilbert's ϵ-terms in Automated Theorem Proving. In Neil V. Murray, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, Saratoga Springs, USA*, volume 1617 of *LNAI*, pages 171–185. Springer-Verlag, 1999.

[MB88] Rainer Manthey and François Bry. SATCHMO: A theorem prover implemented in Prolog. In *Proceedings 9th Conference on Automated Deduction*, volume 310 of *LNCS*, pages 415–434. Springer-Verlag, 1988.

[MBI94] Raul Monroy, Alan Bundy, and Andrew Ireland. Proof plans for the correction of false conjectures. In Frank Pfenning, editor, *Proc. 5th International Conference on Logic Programming and Automated Reasoning, Kiev, Ukraine*, volume 822 of *LNAI*, pages 54–68. Springer-Verlag, 1994.

[Pro92] Martin Protzen. Disproving conjectures. In D. Kapur, editor, *Proc. 11th CADE, Albany/NY, USA*, volume 607 of *LNAI*, pages 340–354. Springer-Verlag, 1992.

[Pro96] Martin Protzen. Patching faulty conjectures. In Michael McRobbie and John Slaney, editors, *Proc. 13th CADE, New Brunswick/NJ, USA*, volume 1104 of *LNCS*, pages 77–91. Springer-Verlag, 1996.

[RST01] Wolfgang Reif, Gerhard Schellhorn, and Andreas Thums. Flaw detection in formal specifications. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning, IJCAR 2001 Siena, Italy, June 18-23, 2001 Proceedings*, volume 2083 of *LNAI*. Springer-Verlag, 2001.

[Sla94] John Slaney. FINDER: finite domain enumerator. In Alan Bundy, editor, *Proc. 12th CADE, Nancy/France*, volume 814 of *LNCS*, pages 798–801. Springer-Verlag, 1994.

[Thu98] Andreas Thums. Fehlersuche in Formalen Spezifikationen. diploma thesis, Fakultät für Informatik, Universität Ulm, 1998.

[ZZ96] Jian Zhang and Hantao Zhang. Generating models by SEM. In Michael McRobbie and John Slaney, editors, *Proc. 13th CADE, New Brunswick/NJ, USA*, volume 1104 of *LNCS*, pages 309–327. Springer-Verlag, 1996.