A Basis for Model Computation in Free Data Types

Wolfgang Ahrendt

Institut für Logik, Komplexität und Deduktionssysteme,
Universtät Karlsruhe, Germany
ahrendt@ira.uka.de

Abstract. Abstract data types, specified by some equality logic under the assumption of term generatedness, are called 'free', if terms, built only by constructors, are semantically unique. This paper presents a calculus, intended to search for models of free data type specifications. A semantical view is discussed, where the uniqueness of constructor terms is 'hard wired'. This suggests an explicit reasoning about interpretations instead of performing real equality reasoning. The rules, which depend on signature, constructor definitions and axioms, are formulated as range restricted clauses. This allows to 'perform' the calculus simply by calling a model generation prover, in particular the MGTP system.
This approach is a 'basis' only, because one of the core problems in model construction, the *terminating* detection of satisfying models, is not yet solved for the described frame. Perspectives in this issue are briefly discussed against the background of using the method for *disproving conjectures* about *consistent* data types.

1 Free Data Types

Abstract Data Types (ADTs) are frequently used to model the structure and the manipulation of data in computers. In this regard, the *structure* of data is reflected by so called *constructors* (e.g., 'nil' and 'push' in the case of the ADT 'stack'). Consequently, all (potential) data are covered by the set of *constructor terms*, exclusively built by constructors. An ADT may have different *sorts*, each characterized by a separate set of constructors.

The *manipulation* of data, on the other hand, is reflected by *function symbols*[1] (e.g., 'pop' and 'del' on 'stacks'). These symbols denote mappings over the elements of the data type. The intended properties of such mappings are specified by *axioms*, usually written in some equality logic. Variants are, in increasing expressiveness, pure equality (no negation, no disjunction), Horn equality, quantifier free equality (negation, disjunction, implicit universal closure), and full first-order equality. The approach presented here aims at full first-order equality. However, the paper mainly deals with the quantifier free case.

[1] Throughout the paper, we systematically distinguish between constructors and function symbols.

As a simple example, a specification for stacks of natural numbers is given in Fig. 1. (To distinguish the constructors, they are written sans serif.) Intuitively, the axioms in NatStack describe how function terms 'reduce' to constructor terms of sort *nat* or *stack*. Please note that, even though the formulae are quite primitive, one of them is not Horn.

sorts

nat generated by
$\mathsf{0}$; $\mathsf{succ}(nat)$
$stack$ generated by
nil ; $\mathsf{push}(nat, stack)$

functions

$pred : nat \rightarrow nat$
$top : stack \rightarrow nat$
$pop : stack \rightarrow stack$
$del : nat, stack \rightarrow stack$

axioms

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

Fig. 1. ADT NatStack

So far, nothing is said about if and when different constructor terms are considered to be equal. In general, this can be specified by so called extensionality axioms. For example in an ADT Set with constructors $\emptyset$ and insert, some axiom is needed to infer equalities between constructor terms, e.g., $\mathsf{insert}(a, \mathsf{insert}(a, \emptyset)) \doteq \mathsf{insert}(a, \emptyset)$ or $\mathsf{insert}(a, \mathsf{insert}(b, \emptyset)) \doteq \mathsf{insert}(b, \mathsf{insert}(a, \emptyset))$. In contrast, it is clear that in NatStack two stacks with a different number of pushed *nat*s should be different, as well as stacks with different *nat*s at certain positions. Here, each element should be uniquely represented by a separate constructor term. This is not made explicit in Fig. 1, so far. In the field of algebraic specification, data types with such a domain property are called *free*. This property, which is typical for many frequently used data types can be expressed by formulae, or built into the semantics.

The present work is exclusively concerned with free data types. Therefore, it is natural to choose the semantics in such a way that the domain really *is* the set of constructor terms itself, not any set of equivalence classes. The domain, as well as the meaning of constructors, is predefined, not depending on any axiom. The only model property specified by the axioms is how function symbols must be interpreted.

Notation. If $\mathcal{X}$ is a family of sets, $\overline{\mathcal{X}}$ denotes the union of all sets in $\mathcal{X}$.

Signature. An *abstract data type (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 disjoint family of S-indexed sets of constructor symbols, $\mathcal{F} = \{F_s\}_{s \in S}$ is a disjoint family of S-indexed sets of function symbols ($\overline{\mathcal{C}} \cap \overline{\mathcal{F}} = \emptyset$), and $\alpha : \overline{\mathcal{C}} \cup \overline{\mathcal{F}} \rightarrow S^*$ gives the argument sorts for every constructor or function symbol.

Syntax. Let $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ be an adt-signature.

- Let $\{V_s\}_{s \in S}$ be a disjoint family of S-indexed, infinite sets of variables. Then $Var(\Sigma) = \overline{\{V_s\}_{s \in S}}$.
- $\mathcal{T}_\Sigma = \{T_s\}_{s \in S}$ is the family of S-indexed sets, minimally defined by:
  - if $x \in V_s$, then $x \in T_s$,
  - if $l \in C_s \cup F_s$, $\alpha(l) = s_1 \dots s_n$ and $\langle t_1, \dots, t_n \rangle \in T_{s_1} \times \dots \times T_{s_n}$, then $l(t_1, \dots, t_n) \in T_s$.

  T_s is the set of *terms of sort s*.
  $T_\Sigma = \overline{\mathcal{T}_\Sigma}$ is the set of *Σ-terms*.
- $\mathcal{T}_\mathcal{C} = \{CT_s\}_{s \in S}$ is the family of S-indexed sets, minimally defined by:
  - if $c \in C_s$, $\alpha(c) = s_1 \dots s_n$ and $\langle t_1, \dots, t_n \rangle \in CT_{s_1} \times \dots \times CT_{s_n}$, then $c(t_1, \dots, t_n) \in CT_s$.

  CT_s is the set of *constructor terms of sort s*.
  $T_\mathcal{C} = \overline{\mathcal{T}_\mathcal{C}}$ is the set of *constructor Σ-terms*.
- An *atomic formula* is an equality $t \doteq t'$, where $t, t' \in T_s$, for some $s \in S$.

Non-atomic formulae are defined as usual. $\mathcal{L}_\Sigma$ is the set of all Σ-formulae.

Semantics. Let $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ be an adt-signature, fulfilling the condition that, for all $s \in S$, $CT_s \neq \emptyset$.

- An *$\frac{\mathcal{F}}{\mathcal{C}}$-interpretation* $\mathcal{I}$ assigns to each function symbol f, with $f \in F_s$ and $\alpha(f) = s_1 \dots s_n$, a mapping: $\mathcal{I}(f) : CT_{s_1}, \dots, CT_{s_n} \to CT_s$.
  $(\mathcal{T}_\mathcal{C}, \mathcal{I})$ then is a *freely generated Σ-structure (fg$_\Sigma$-structure)*.
- A *variable assignment* $\beta : Var(\Sigma) \to T_\mathcal{C}$ is a mapping, such that: for $x \in V_s$, $\beta(x) \in CT_s$.
- For any $\frac{\mathcal{F}}{\mathcal{C}}$-interpretation $\mathcal{I}$ and variable assignment β, a *$\frac{\Sigma}{\mathcal{C}}$-valuation* of terms $val_{\mathcal{I},\beta} : T_\Sigma \to T_\mathcal{C}$ is defined by:
  - $val_{\mathcal{I},\beta}(x) = \beta(x)$, for $x \in Var(\Sigma)$.
  - $val_{\mathcal{I},\beta}(f(t_1, \dots, t_n)) = \mathcal{I}(f)(val_{\mathcal{I},\beta}(t_1), \dots, val_{\mathcal{I},\beta}(t_n))$, for $f \in \overline{\mathcal{F}}$, $f(t_1, \dots, t_n) \in T_\Sigma$.
  - $val_{\mathcal{I},\beta}(c(t_1, \dots, t_n)) = c(val_{\mathcal{I},\beta}(t_1), \dots, val_{\mathcal{I},\beta}(t_n))$, for $c \in \overline{\mathcal{C}}$, $c(t_1, \dots, t_n) \in T_\Sigma$.

To clarify the characteristics of these definitions, some particular features, differing from usual first-order logic, are pointed out here:

- A freely generated Σ-structure is really 'generated' because all domain elements can be denoted by a term (i.e. by themselves). And it is really 'free' because the domain contains each constructor term as a separate element.
- For a given Σ, all fg$_\Sigma$-structures have the same domain, i.e. the sorted partition of the constructor terms. (Therefore, *val* is not indexed by the domain.)
- The interpretation $\mathcal{I}$ is not defined for constructors.
- The valuation of terms can be seen as a combination of standard valuations, cf. "$val_{\mathcal{I},\beta}(f(\cdot, \dots)) = \mathcal{I}(f)(val_{\mathcal{I},\beta}(\cdot), \dots)$", and Herbrand structure valuations, cf. "$val_{\mathcal{I},\beta}(c(\cdot, \dots)) = c(val_{\mathcal{I},\beta}(\cdot), \dots)$".

The valuation of formulae is defined as usual. In particular, $val_{\mathcal{I},\beta}(t \doteq t') = \top$ iff $val_{\mathcal{I},\beta}(t) = val_{\mathcal{I},\beta}(t')$.

With these definitions, *term generatedness* and *'freeness'* are built into the semantics. This is necessary for generatedness, which cannot be expressed in first-order formulae. 'Freeness', on the other hand, could alternatively be expressed by the axioms of a data type. The advantage of 'hard wiring' this property too lies in its fundamental consequences to the search for models.

Model, Satisfying Interpretation, Consistency, Consequence.
Let $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ be an adt-signature.

- An fg$_\Sigma$-structure $(\mathcal{T}_\mathcal{C}, \mathcal{I})$ is a *freely generated Σ-model (fg$_\Sigma$-model)* of a set of formulae $\Phi \subseteq \mathcal{L}_\Sigma$, if $val_{\mathcal{I},\beta}(\varphi) = \top$ for all $\varphi \in \Phi$ and variable assignments β.
  $\mathcal{I}$ then is called a *satisfying $\frac{\mathcal{F}}{\mathcal{C}}$-interpretation* of Φ.
- φ is a *fg$_\Sigma$-consequence* of Φ, denoted $\Phi \models^{\Sigma}_{fg} \varphi$, if each fg$_\Sigma$-model of Φ is also an (fg$_\Sigma$-)model of $\{\varphi\}$.
- Φ is *fg$_\Sigma$-consistent* if it has an fg$_\Sigma$-model. φ is *fg$_\Sigma$-consistent relative to Φ* if the fg$_\Sigma$-consistency of Φ implies the fg$_\Sigma$-consistency of $\Phi \cup \{\varphi\}$.

Example 1. Let $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ be an adt-signature with
$S = \{nat, bool\}$, $\mathcal{C} = \langle \{0, succ\}_{nat}, \{tt, ff\}_{bool} \rangle$, $\mathcal{F} = \langle \{\}_{nat}, \{p\}_{bool} \rangle$,
$\alpha(s) = \alpha(p) = nat$, $\alpha(0) = \alpha(tt) = \alpha(ff) = \lambda$ (the empty word).
Let $\Phi = \{p(0) \doteq tt,\ p(x) \doteq tt \to p(succ(x)) \doteq tt\}$. Then the following holds:

- $\models^{\Sigma}_{fg} succ(succ(succ(0))) \neq succ(0)$ (freely)
- $\Phi \models^{\Sigma}_{fg} p(x) \doteq tt$ (generated)

On the other hand, with the usual definition of $\models^{\Sigma}$ (and a signature not distinguishing $\mathcal{C}$ and $\mathcal{F}$), it holds that

- $\not\models^{\Sigma} succ(succ(succ(0))) \neq succ(0)$ and
- $\Phi \not\models^{\Sigma} p(x) \doteq tt$.

2 Explicit Reasoning about Interpretations

As discussed above, two freely generated structures for the same Σ can only differ in their respective $\frac{\mathcal{F}}{\mathcal{C}}$-interpretation $\mathcal{I}$, which assigns to function symbols $f \in \overline{\mathcal{F}}$ mappings over constructor terms. Therefore, the construction of structures resp. models reduces to the construction of $\frac{\mathcal{F}}{\mathcal{C}}$-interpretations.

We can think of $\mathcal{I}$ being an infinite table, in which each line relates one function symbol and an appropriate tuple of domain elements with a resulting domain element. This is visualized in Fig. 2. Here, the domain elements are written as 'ct_{ij}' to emphasize that they are constructor terms, and *nothing else.*

One main suggestion of this paper is, to *perform reasoning about (resp. search for) $\frac{\mathcal{F}}{\mathcal{C}}$-interpretations on a representation that immediately describes (parts of) such interpretation tables.* In particular, we will represent lines of $\mathcal{I}$-tables, called

f	$ct_{11},\ldots,ct_{1n}$	ct_{10}
f	$ct_{21},\ldots,ct_{2n}$	ct_{20}
$\vdots$	$\vdots$	$\vdots$
f'	$ct'_{11},\ldots,ct'_{1m}$	ct'_{10}
f'	$ct'_{21},\ldots,ct'_{2m}$	ct'_{20}
$\vdots$	$\vdots$	$\vdots$

Fig. 2. $\mathcal{I}$ as a table

$\mathcal{I}$-lines, as *atoms* in our mechanism. Describing interpretations by their $\mathcal{I}$-lines in a sense is an *atomic representation* of models. But instead of using equality atoms, which are the only atoms allowed in the object logic, we use an extra predicate I to build atoms like $I(f, ct_1, \ldots, ct_n, ct_0)$, representing single $\mathcal{I}$-lines like $\boxed{f \mid ct_1,\ldots,ct_n \mid ct_0}$. A collection of I-atoms serves as a (partial) *interpretation candidate*. Building an interpretation then consists of expanding candidates by newly inferred I-atoms. The presence of disjunctive problem structure (the origin of disjunctions is discussed below) causes splittings of interpretation candidates by inferring alternative expansions. We obtain the picture of a tableau style tree, where the nodes are I-atoms and the branches are interpretation candidates.

Provided we comply with the respective arity and sorts, any collection of I-atoms partly describes a legal $\frac{\mathcal{F}}{\mathcal{C}}$-interpretation, *with the only condition* that the *functionality* of interpretations must be respected, i.e. the resulting I-predicate must be functional in the last argument. In consequence, inferring two I-atoms, which violate the functionality property, is *the* reason for rejecting a branch.

The inference rules for expansion, splitting and rejection of interpretation candidates will be represented by (slightly generalized) clauses. That means, instead of tableau style rules like $\dfrac{\begin{array}{c}A\\B\end{array}}{C \mid D}$, we write $A \wedge B \rightarrow C \vee D$. The advantage of this representation is, that we can give such clauses as input to an existing system building *clausal tableaux*. In particular, *positive hyper tableau*[2] provers implement the operational semantics of clauses in such a way that the above clause 'behaves' like the above rule. Moreover, we can use a restricted version of hyper tableaux. As long as we infer *ground* atoms only, we can describe the rules by clauses that are *range restricted*, which means that all variables appearing on the right side of the implication also appear on the left side. The range restricted variant of positive hyper tableau calculi is known as *model generation* [7]. For the understanding of this paper, it suffices to know that a) clauses are applied as sketched above, b) model generation manage with matching, not with unification, c) deriving $\perp$ (false) on a branch means rejecting it, d) branches are kept regular (i.e. clauses cannot be applied if any atom would be doubled on any branch), and e) a branch is saturated if no clause can be applied anymore. In

[2] For hyper tableaux in general see, e.g., [1].

the following, we abbreviate model generation by "MG", to prevent confusion with our notion of 'model construction'. (The first serves as a tool to perform the second.)

The fastest implementation of MG currently is the system MGTP (Model Generation Theorem Prover) [4], as far as we know. MGTP allows to use a slightly more general form of clauses. The right side of the implication may consist in a disjunction of conjunctions, e.g., '$A \wedge B \rightarrow (C \wedge D) \vee E$'. This allows to express inference rules which add more than one atom to the same branch. In a sense, the language of generalized, range restricted clauses serves, in the described approach, as a programming language for implementing a machinery performing 'interpretation inference'.

The first clause we give here is the, quite simple, *functionality rule*[3]:

$$I(F, CT, CT_1) \wedge I(F, CT, CT_2) \rightarrow \mathit{SameElems}(CT_1, CT_2).$$

This rule is only applicable for I-atoms whose function symbols have exactly one argument. Variants are provided for each arity appearing in $\mathcal{F}$. *SameElems* is one of the *extra predicates* we need besides I to be present in interpretation candidates. These predicates are dedicated for additionally controlling the expansion, splitting and rejection of branches. More of them are discussed below.

Intuitively, $\mathit{SameElems}(CT_1, CT_2)$ means that CT_1 and CT_2 must be the same domain element, i.e. the same constructor term, in the current branch. This is implemented by Σ-dependent *freeness rules*. Here we give just two examples:

$$\begin{array}{rcl} \mathit{SameElems}(\mathit{push}(N, S), \mathit{nil}) & \rightarrow & \bot. \\ \mathit{SameElems}(\mathit{push}(N_1, S_1), \mathit{push}(N_2, S_2)) & \rightarrow & \left(\begin{array}{c} \mathit{SameElems}(N_1, N_2) \\ \wedge\ \mathit{SameElems}(S_1, S_2) \end{array} \right). \end{array}$$

The first kind of clause is needed for each pair of different constructors (of the same sort). The second kind is needed for each constructor with more than zero arguments. At first glance, checking the syntactical identity by recursively applying such rules seems to be an overkill. The reason why we really need this recursive analysis is the following: in addition to pure constructor terms, we have to handle *place holders* for constructor terms not yet known (see next section). In presence of such place holders in constructor terms, we cannot just syntactically check the constraint of 'being the same element'.

3 Exploiting Domain Knowledge: Constructor Splitting

The explicit representation of $\mathcal{I}$-lines reflects the information, resulting from object equalities, in a more structured way than the equalities themselves do. For example, it makes a big difference if (non-constructor) functions appear

[3] In this as in following rules, arguments of atoms, if they start with a capital letter, are considered to be the variables for the clause.

on both sides of an equality or not, resp. if they are nested or not. This is exemplified in Table 1 (where again the f_i denote function symbols and the ct_j denote constructor terms).

object equality	I-atoms
ground case:	
$f(ct_1) \doteq ct_2$	$I(f, ct_1, ct_2)$
(functions on both sides:) $f_1(ct_1) \doteq f_2(ct_2)$	*There exists* a constructor term k, such that $I(f_1, ct_1, k)$ and $I(f_2, ct_2, k)$ holds.
(nested functions:) $f_1(f_2(ct_1)) \doteq ct_2$	*There exists* a constructor term k, such that $I(f_2, ct_1, k)$ and $I(f_1, k, ct_2)$ holds.
(mix:) $f_1(f_2(ct_1)) \doteq f_3(ct_2)$	*There exist* constructor terms k_1, k_2, such that $I(f_2, ct_1, k_1)$, $I(f_1, k_1, k_2)$ and $I(f_3, ct_2, k_2)$ holds.
universal case:	
$f(x) \doteq ct$	*For all* constructor terms X, $I(f, X, ct)$ holds.
(functions on both sides:) $f_1(x) \doteq f_2(x)$	*For all* c.-terms X, *there exists* a c.-term k, s.t. $I(f_1, X, k)$ and $I(f_2, X, k)$ holds.
$\vdots$	$\vdots$

Table 1. Examples for equalities and corresponding I-atoms

On the right side of this table, constructor terms appear to be quantified on a meta level ('*for all k*' and '*there exists k*'). The restriction to constructor generated domains has significant consequences for the deductive handling of (explicit or implicit) existential quantification. In the general first-order case, existential quantifiers are removed by Skolemization, i.e. introduction of new constants. Semantically, the current Herbrand model is considered to be *extended* by the new constant.

However, in the case of predefined domains, usual Skolemization is not complete, with respect to refutation. Even worse, this exactly means that, *in predefined domains, standard Skolemization is not sound with respect to model construction*! For example, from '$\exists\, x_{nat}.\ \varphi(x_{nat})$' we cannot infer the satisfiability of '$\varphi(sko)$' (where sko is a new constant), because x_{nat} here must be a natural number, whereas sko might be anything. Instead, we can infer that either $\varphi(0)$ or $\exists\, x_{nat}.\ \varphi(succ(x_{nat}))$ must hold. Written as a tableau rule, this is:

$$\frac{\exists\, x_{nat}.\ \varphi(x_{nat})}{\varphi(0) \,\big|\, \exists\, x_{nat}.\ \varphi(succ(x_{nat}))}$$

In case of stacks, the rule is:

$$\frac{\exists\, x_{stack}.\ \varphi(x_{stack})}{\varphi(nil) \,\big|\, \exists\, x_{nat}, y_{stack}.\ \varphi(push(x_{nat}, y_{stack}))}$$

We call this *constructor splitting*. The general form is:

$$\frac{\exists\ x_s.\ \varphi(x_s)}{\varphi(c_1)\big|\cdots\big|\varphi(c_i)\big|\exists\ \overline{y}.\ \varphi(c_{i+1}(\overline{y}))\big|\cdots\big|\exists\ \overline{z}.\ \varphi(c_n(\overline{z}))}$$

where $x_s \in V_s$ is a variable of sort s, $C_s = \{c_1, \ldots, c_i, c_{i+1}, \ldots, c_n\}$ are the constructors of sort s, c_1 to c_i have no arguments, and the variable vectors $\overline{y}$ to $\overline{z}$ match the respective arity of c_{i+1} to c_n.

Branching δ-rules are also used in the field of (domain-)finite model construction, see [5] (minimization rule) and [2] (δ^*-rule in the EP tableaux calculus). In both cases, the branching δ-rules *disjunctively enumerate finite domains*. Constructor splitting, as demonstrated above, in the general case performs an approximation of *disjunctively enumerating infinite domains*.

Even if existential quantifiers cannot appear explicitly in trees built by MG, Table 1 gives reasons for the necessity of, at least implicit, existential quantification. This is done by introducing place-markers for constructor terms not yet fixed. But in contrast to standard Skolemization, we have to search disjunctively for suitable instances of place-markers. This is done by the *constructor splitting rules*, which are Σ-dependent clauses, controlled by an extra *search*-predicate. We give two examples here:

$$search\_nat(X)\ \rightarrow\ X\ is\ 0\ \vee\ (X\ is\ succ(\texttt{new(1)})\ \wedge\ search\_nat(\texttt{new(1)})).$$

$$search\_stack(X)\ \rightarrow\ X\ is\ nil\ \vee\ \begin{pmatrix} X\ is\ push(\texttt{new(1)}, \texttt{new(2)}) \\ \wedge\ search\_nat(\texttt{new(1)}) \\ \wedge\ search\_stack(\texttt{new(2)}) \end{pmatrix}.$$

When these rules are applied, then, for every i, the first appearance of `new(`i`)` is meant to create a new symbol (relative to the current branch) that replaces all occurrences of `new(`i`)` in the atoms of the extension. These 'place-markers' are treated as constants by MG (though they do not belong to the object signature). The `new`-construct is the only feature we need that is not standard in MG systems.

Whenever we have a *search_s*-atom on the current branch, this initiates a search for a constructor term of sort s. Applying the *constructor splitting rule* then creates new place-markers, for which again a search is initiated. The extra predicate *is* determines partly the result of the search. Later, '*is*'-atoms are treated by primitive *rewriting rules*, e.g.:

$$\begin{array}{l} I(F,X,Y)\ \wedge\ X\ is\ CT\ \rightarrow\ I(F,CT,Y). \\ I(F,X,Y)\ \wedge\ Y\ is\ CT\ \rightarrow\ I(F,X,CT). \end{array}$$

Such rules are needed also for I-atoms with varying arity and moreover for all extra predicates, even for *is* itself.

4 Representation of Axioms

The search for interpretations must respect the axioms which originally specify the data type. Therefore, the axioms, written in equality logic, are transformed into (generalized) clauses. We recall the primary restriction to quantifier free formulae (with implicit universal closure). Essentially, a CNF of the axioms is computed. Every disjunction of (in)equalities then becomes a single clause in the following way. Each equality is turned in a conjunction of I-atoms as informally indicated by Table 1. (To express inequalities, we moreover need an extra predicate *DifferentElems*, having the sole effect that it contradicts *SameElems* when applied to the same arguments. We omit the respective rule.) If necessary, existential quantification is simulated by adding suitable *search*-atoms to the conjunction. Universal quantification is represented just by putting variables at respective positions. The resulting disjunction of conjunctions constitutes the right side of clauses only. The left side now is used to, in a sense, 'bind' the variables by sort predicates. (The last step is similar to the general transformation of arbitrary clauses into range restricted clauses found in [7]. Here, however, we have to distinguish between different sorts.) Fig. 3 shows the result of transforming three axioms found in Fig. 1.

$del(n, nil) \doteq nil$	$\blacktriangleright$	$nat(N) \;\rightarrow\; I(del, N, nil, nil).$
$del(n, push(n, st)) \doteq st$	$\blacktriangleright$	$nat(N) \;\wedge\; stack(ST) \;\rightarrow\; I(del, N, push(N, ST), ST).$

$$n \neq n' \;\rightarrow\; del(n, push(n', st)) \doteq push(n', del(n, st))$$
$$\blacktriangledown$$
$$\begin{gathered} nat(N) \;\wedge\; nat(N') \;\wedge\; stack(ST) \;\rightarrow \\ SameElems(N, N') \vee \\ \begin{pmatrix} I(del, N, push(N', ST), push(N', \texttt{new(1)})) \;\wedge \\ I(del, N, ST, \texttt{new(1)}) \;\wedge\; search\_stack(\texttt{new(1)}) \end{pmatrix}. \end{gathered}$$

Fig. 3. Transforming axioms to clauses

These clauses are guarded by sort atoms, i.e. sort predicates applied to arguments. Transformed axioms are therefore only applied to instances for which a sort atom is present on the current branch. To make the axioms applicable to the whole domain, we could add *naive domain generation rules*, e.g.:

$$\begin{array}{ll} \top \;\rightarrow nat(0). & nat(N) \;\rightarrow\; nat(succ(N)). \\ \top \;\rightarrow stack(nil). & nat(N) \;\wedge\; stack(ST) \;\rightarrow\; stack(push(N, ST)). \end{array}$$

(This again is similar to [7], where a single domain predicate is propagated over function applications. Here, however, we respect sorts and constructors.) Additionally, we have to make axioms applicable to (terms containing) placemarkers. This is done by adding clauses like:

$$search\_nat(X) \;\rightarrow\; nat(X). \qquad search\_stack(X) \;\rightarrow\; stack(X).$$

The transformation of axioms can be extended to full first-order equality. Then, elimination of existential quantifiers creates Skolem function symbols, which are considered as being added to $\mathcal{F}$, *not to* $\mathcal{C}$. Consequently, the mechanism searches for the interpretation of Skolem functions also.

5 This is just the Basis

The described approach to interpretation reasoning is part of ongoing work which aims at *disproving* conjectures about freely generated data types. Given an adt-signature Σ and a set Φ of Σ-axioms, disproving a formula φ means to show that $\Phi \not\models^{\Sigma}_{fg} \varphi$ holds. This is equivalent to finding a satisfying $\frac{\mathcal{F}}{C}$-interpretation for $\Phi \cup \{\exists\overline{x}.\neg\varphi\}$ (where $\overline{x}$ are the free variables in φ). Therefore, Φ is transformed into clauses as discussed in Sect. 4. $\exists\overline{x}.\neg\varphi$ is transformed in the same manner, where additionally the variables $\overline{x}$ are replaced by place-markers, for which suitable *search*-atoms are added. Indeed, MG applied to the resulting rule system constructs a tree in which the I-predicate defined by any saturated, unrejected branch *is* a satisfying $\frac{\mathcal{F}}{C}$-interpretation of $\Phi \cup \{\exists\overline{x}.\neg\varphi\}$.[4]

Unfortunately, but not surprisingly, the rule system is not able to construct satisfying interpretations in finite time! This is due to the, intentionally called *naive*, *domain generation rules*. They perform a conjunctive enumeration of the (usually infinite) domains. This leads to non-terminating saturation of unrejectable branches, provided the signature has recursive constructors. Essentially the same problem appears in [7], where, in presence of non-constant function symbols, the range restricted transformation of consistent formulae results in non-terminating MG. In general, tableaux systems usually run forever on consistent input. The *terminating* saturation of consistent branches in tableau-like frames requires additional concepts (e.g., constraints [3, 9], or terms with exponents [8, 6]).

The approach described here is based on an atomic representation of interpretations. To stop their construction, we need criteria telling that the I-predicate defined by the current branch is extendible to a (total) satisfying interpretation of the respective formulae. This will not be possible without a pragmatic simplification of the problem to be solved: if we *assume a specification to be fg$_\Sigma$-consistent*, then $\Phi \not\models^{\Sigma}_{fg} \varphi$ holds if $\exists\overline{x}.\neg\varphi$ is fg$_\Sigma$-consistent *relative to Φ*. Then, the mechanism may not 'take care' of potential contradictions between axioms. The naive domain generation rules must somehow be replaced by rules, depending on φ and Φ, which either generate *enough* instances to ensure relative consistency, or, where this is not possible, result in non-termination. This guarantees soundness of each terminating answer.

Currently, the characteristics of 'typical' (free) data type specifications and conjectures are investigated, to enable a terminating detection of relative consistency in many cases.

Acknowledgments

I am grateful to Reiner Hähnle for his support and many, many, fruitful discussions.

[4] To be precise, this is only the case for branches with a finite number of constructor splitting rule applications. Other branches constitute non-standard models, possessing 'terms' with unfounded chains of constructor applications.

References

1. Peter Baumgartner. Hyper Tableaux — The Next Generation. In Harry de Swaart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 1397 of *LNCS*, pages 60–76. Springer-Verlag, 1998.
2. 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 *LNCS*, pages 122–136. Springer-Verlag, 1998.
3. Ricardo Caferra and Nicolas Zabel. A tableaux method for systematic simultaneous search for refutations and models using equational problems. *Journal of Logic and Computation*, 3(1):3–26, 1993.
4. 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.
5. Jaakko Hintikka. Model minimization – an alternative to circumscription. *Journal of Automated Reasoning*, 4(1):1–13, 1988.
6. Stefan Klingenbeck. *Counter Examples in Semantic Tableaux*. PhD thesis, University of Karlsruhe, 1997. Diski 51, infix Verlag.
7. 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.
8. Nicolas Peltier. Increasing the capabilities of model building by constraint solving with terms with integer exponents. *Journal of Symbolic Computation*, 24(1):59–101, 1997.
9. Nicolas Peltier. Simplifying and generalizing formulae in tableaux: pruning the search space and building models. In Didier Galmiche, editor, *Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Pont-à-Mousson, France*, volume 1227 of *LNCS*, pages 313–327. Springer-Verlag, 1997.