Noname manuscript No.
(will be inserted by the editor)

Relational Analysis of (Co)inductive Predicates, (Co)algebraic Datatypes, and (Co)recursive Functions

Jasmin Christian Blanchette

the date of receipt and acceptance should be inserted later

Abstract We present techniques for applying a finite relational model finder to logical specifications that involve high-level definitional principles such as (co)inductive predicates, (co)algebraic datatypes, and (co)recursive functions. In contrast to previous work, which focused on algebraic datatypes and restricted occurrences of unbounded quantifiers in formulas, we can handle arbitrary formulas by means of a three-valued Kleene logic. The techniques form the basis of the counterexample generator Nitpick for Isabelle/HOL. As case studies, we consider formulas about an inductively defined context-free grammar, a functional implementation of AA trees, and a coalgebraic list datatype.

Keywords Model finding · Higher-order logic · First-order relational logic

1 Introduction

SAT and SMT solvers, model checkers, model finders, and other lightweight formal methods are today available to test or verify specifications written in various languages. These tools are often integrated in more powerful systems, such as interactive theorem provers, to discharge proof obligations or generate (counter)models.

For testing logical specifications, a particularly attractive approach is to express these in first-order relational logic (FORL) and use a model finder such as Kodkod [35] to find counterexamples. FORL extends traditional first-order logic (FOL) with relational calculus operators and the transitive closure, and offers a good compromise between automation and expressiveness. Kodkod relies on a SAT solver and forms the basis of Alloy [19]. It features several important optimizations, notably symmetry breaking, that make it preferable to a direct reduction to SAT. In a case study, the Alloy Analyzer checked a mechanized version of the paper proof of the Mondex protocol and revealed several bugs in the proof [32].

However, FORL lacks the high-level definitional principles usually provided in interactive theorem provers, namely (co)inductive predicates, (co)algebraic datatypes, and (co)recursive functions (Sect. 3). Solutions have been proposed by Kuncak and Jackson [25], who

This work is supported by the Deutsche Forschungsgemeinschaft grant Ni 491/11-1.

J. C. Blanchette
Institut für Informatik, Technische Universität München
Tel.: +49 89 289 17332 · E-mail: blanchette@in.tum.de

modeled lists and trees in Alloy, and Dunets et al. [13], who showed how to translate algebraic datatypes and recursive functions in the context of the first-order theorem prover KIV [3]. In both cases, the translation is restricted to formulas whose prenex normal forms contain no unbounded universal quantifiers ranging over datatypes.

This article generalizes previous work in several directions: First, we lift the unbounded quantifier restriction by using a three-valued logic coded in terms of the binary logic FORL (Sect. 4.2). Second, we show how to translate (co)inductive predicates, coalgebraic datatypes, and corecursive functions (Sect. 5). Third, in our treatment of algebraic datatypes, we show how to handle mutually recursive datatypes (Sect. 5.2).

The use of a three-valued Kleene logic makes it possible to analyze formulas such as $\mathit{True} \vee \forall n^{\mathit{nat}}.\, P(n)$, which are rejected by Kuncak and Jackson's syntactic criterion. Unbounded universal quantification remains problematic in general, but suitable definitional principles and their proper handling, in conjunction with various optimizations (Sect. 6), mitigate this problem.

The ideas presented here form the basis of the higher-order counterexample generator Nitpick [8], which is included with recent versions of Isabelle/HOL [29]. Nitpick can be run on putative theorems or on specific subgoals in a proof to spare users the Sisyphean task of trying to prove non-theorems. As a case study, we employ Nitpick on an inductively defined context-free grammar, a functional implementation of AA trees, and a coalgebraic (or "lazy") list datatype (Sect. 7).

To simplify the presentation, we use FOL as our specification language. Issues specific to higher-order logic (HOL) are mostly orthogonal and explained in the paper on Nitpick [8].

2 Logics

2.1 First-Order Logic (FOL)

The first-order logic that will serve as our specification language is essentially the first-order fragment of HOL [10, 15]. The types and terms are given below.

Types:		*Terms:*	
$\sigma ::= \alpha$	(type variable)	$t ::= x^{\sigma}$	(variable)
$\quad\mid (\sigma,\ldots,\sigma)\,\kappa$	(type constructor)	$\quad\mid c^{\tau}(t,\ldots,t)$	(function term)
$\tau ::= (\sigma,\ldots,\sigma) \to \sigma$	(function type)	$\quad\mid \forall x^{\sigma}.\, t$	(universal quantification)

The standard semantics interprets the Boolean type o and the constants $\mathit{False}^{\mathrm{o}}$, $\mathit{True}^{\mathrm{o}}$, $\longrightarrow^{(\mathrm{o},\mathrm{o})\to\mathrm{o}}$ (implication), $\simeq^{(\sigma,\sigma)\to\mathrm{o}}$ (equality on basic type σ), and $\mathit{if\ then\ else}^{(\mathrm{o},\sigma,\sigma)\to\sigma}$. Formulas are terms of type o. We assume throughout this article that terms are well-typed using the standard typing rules and usually omit the type superscripts. In conformity with first-order practice, application of x and y on f is written $f(x,y)$, the function type $() \to \sigma$ is identified with σ, and the parentheses in the function term $c()$ are optional. We also assume that the connectives $\neg$, $\wedge$, $\vee$ and existential quantification are available.

In contrast to HOL, our logic requires variables to range over basic types, and it forbids partial function application and λ-abstractions. On the other hand, it supports the limited form of polymorphism provided by proof assistants for HOL [18, 29, 34], with the restriction that type variables may only be instantiated by atomic types (or left uninstantiated in a polymorphic formula).

Types and terms are interpreted in the standard set-theoretic way, relative to a scope that fixes the interpretation of basic types. A *scope* S is a function from basic types to nonempty sets (domains), which need not be finite.[1] We require $S(\mathrm{o}) = \{\mathit{ff}, \mathit{tt}\}$.

The standard interpretation $[\![\tau]\!]_S$ of a type τ is given by $S(\tau)$ for basic types and

$$[\![(\sigma_1,\ldots,\sigma_n) \to \sigma]\!]_S = [\![\sigma_1]\!]_S \times \cdots \times [\![\sigma_n]\!]_S \to [\![\sigma]\!]_S,$$

where $A \to B$ denotes the set of (total) functions from A to B. In contexts where S is clear or irrelevant, the cardinality of $[\![\tau]\!]_S$ is written $|\tau|$.

2.2 First-Order Relational Logic (FORL)

Our analysis logic, first-order relational logic, combines elements from FOL and relational calculus extended with the transitive closure [19, 35]. Formulas involve variables and terms ranging over relations (sets of tuples drawn from a universe of uninterpreted atoms) of arbitrary arities. The logic is unsorted, but each term denotes a relation of a fixed arity. Our translation from FOL relies on the following FORL fragment.

Formulas:		*Terms:*	
$\varphi ::= \mathsf{false}$	(falsity)	$r ::= \mathsf{none}$	(empty set)
$\mid \mathsf{true}$	(truth)	$\mid \mathsf{iden}$	(identity relation)
$\mid m\ r$	(multiplicity constraint)	$\mid \mathsf{a}_i$	(atom)
$\mid r \simeq r$	(equality)	$\mid x$	(variable)
$\mid r \subseteq r$	(inclusion)	$\mid r^+$	(transitive closure)
$\mid \neg\varphi$	(negation)	$\mid r.r$	(dot-join)
$\mid \varphi \wedge \varphi$	(conjunction)	$\mid r \times r$	(Cartesian product)
$\mid \forall x{\in}r\colon \varphi$	(universal quantification)	$\mid r \cup r$	(union)
		$\mid r - r$	(difference)
$m ::= \mathsf{no} \mid \mathsf{lone} \mid \mathsf{one} \mid \mathsf{some}$		$\mid \mathsf{if}\ \varphi\ \mathsf{then}\ r\ \mathsf{else}\ r$	(conditional)

FORL syntactically distinguishes between terms and formulas. The universe of discourse is $\mathscr{A} = \{\mathsf{a}_1,\ldots,\mathsf{a}_k\}$, where each a_i is an uninterpreted atom. Atoms and n-tuples are identified with singleton sets and singleton n-ary relations, respectively. Bound variables in quantifications range over the tuples in a relation; thus, $\forall x \in (\mathsf{a}_1 \cup \mathsf{a}_2) \times \mathsf{a}_3\colon \varphi[x]$ is equivalent to $\varphi[\mathsf{a}_1 \times \mathsf{a}_3] \wedge \varphi[\mathsf{a}_2 \times \mathsf{a}_3]$.

Although they are not listed above, we will sometimes make use of $\vee$ (disjunction), $\longrightarrow$ (implication), * (reflexive transitive closure), and $\cap$ (intersection) as well. The constraint $\mathsf{no}\ r$ expresses that r is the empty relation, $\mathsf{one}\ r$ expresses that r is a singleton, $\mathsf{lone}\ r \Longleftrightarrow \mathsf{no}\ r \vee \mathsf{one}\ r$, and $\mathsf{some}\ r \Longleftrightarrow \neg\,\mathsf{no}\ r$.

The dot-join operator is unconventional; its semantics is given by the equation

$$[\![r.s]\!] = \{\langle r_1,\ldots,r_{m-1},s_2,\ldots,s_n\rangle \mid \exists t.\ \langle r_1,\ldots,r_{m-1},t\rangle \in [\![r]\!] \wedge \langle t,s_2,\ldots,s_n\rangle \in [\![s]\!]\}.$$

The operator admits three important special cases. Let s be unary and r, r' be binary relations. The expression $s.r$ gives the direct image of the set s under r; if s is a singleton and r a function, it coincides with the function application $r(s)$. Analogously, $r.s$ gives the inverse image of s under r. Finally, $r.r'$ expresses the relational composition $r \circ r'$.

[1] The use of the word "scope" for a domain specification is consistent with Jackson [19].

The relational operators often make it possible to express first-order problems concisely. For example, the following FORL specification attempts to fit 30 pigeons in 29 holes:

$$\begin{array}{l}\mathsf{var}\ \mathit{pigeons} = \{\mathsf{a}_1,\ldots,\mathsf{a}_{30}\}\\ \mathsf{var}\ \mathit{holes} = \{\mathsf{a}_{31},\ldots,\mathsf{a}_{59}\}\\ \mathsf{var}\ \emptyset \subseteq \mathit{nest} \subseteq \{\mathsf{a}_1,\ldots,\mathsf{a}_{30}\} \times \{\mathsf{a}_{31},\ldots,\mathsf{a}_{59}\}\\ \mathsf{solve}\ (\forall p \in \mathit{pigeons}\colon \mathsf{one}\ p\boldsymbol{.}\mathit{nest}) \land (\forall h \in \mathit{holes}\colon \mathsf{lone}\ \mathit{nest}\boldsymbol{.}h)\end{array}$$

The variables *pigeons* and *holes* are given fixed values, whereas *nest* is specified with a lower and an upper bound. Variable declarations are an extralogical way of specifying sort constraints and partial solutions. They also indirectly specify the variables' arities, which in turn dictate the arities of all the terms in the formula to solve.

The constraint one $p\boldsymbol{.}\mathit{nest}$ states that pigeon p is in relation with exactly one hole, and lone $\mathit{nest}\boldsymbol{.}h$ that hole h is in relation with at most one pigeon. Taken as a whole, the formula states that *nest* is a one-to-one function. It is, of course, not satisfiable, a fact that Kodkod can establish in less than a second.

When reducing FORL to SAT, each n-ary relational variable y is in principle translated to an $|\mathscr{A}|^n$ array of propositional variables $V[i_1,\ldots,i_n]$, with $V[i_1,\ldots,i_n] \iff \langle \mathsf{a}_{i_1},\ldots,\mathsf{a}_{i_n}\rangle \in y$. Most relational operations can be coded efficiently; for example, $\cup$ is simply $\vee$. The quantified formula $\forall r \in s\colon \varphi[r]$ is treated as $\bigwedge_{j=1}^{n} t_j \subseteq s \longrightarrow \varphi[t_j]$, where the t_j's are the tuples that may belong to s. Transitive closure is unrolled to saturation.

3 Definitional Principles

3.1 Simple Definitions

We extend our specification logic FOL with several definitional principles to introduce new constants and types. The first principle defines a constant as equal to another term:

$$\mathsf{definition}\ c^{\tau}\ \mathsf{where}\ c(\bar{x}) \simeq t$$

Logically, the above definition is equivalent to the axiom $\forall\bar{x}.\ c(\bar{x}) \simeq t$.

Provisos: The constant c is fresh, the variables $\bar{x}$ are distinct, and the right-hand side t does not refer to any other free variables than $\bar{x}$, to any undefined constants or c, or to any type variables not occurring in τ. These restrictions ensure consistency [37].

An example definition follows:

$$\mathsf{definition}\ \mathit{snd}^{(\alpha,\beta)\to\beta}\ \mathsf{where}\ \mathit{snd}(x,y) \simeq y$$

3.2 (Co)inductive Predicates

The inductive and coinductive commands define inductive and coinductive predicates specified by their introduction rules:

$$\begin{array}{l}[\mathsf{co}]\mathsf{inductive}\ p^{\tau}\ \mathsf{where}\\ p(\bar{t}_{11}) \land \cdots \land p(\bar{t}_{1\ell_1}) \land Q_1 \longrightarrow p(\bar{u}_1)\\ \quad\vdots\\ p(\bar{t}_{n1}) \land \cdots \land p(\bar{t}_{n\ell_n}) \land Q_n \longrightarrow p(\bar{u}_n)\end{array}$$

Provisos: The constant p is fresh, and the arguments to p and the side conditions Q_i do not refer to p, undeclared constants, or any type variables not occurring in τ.

The introduction rules may involve any number of free variables $\bar{y}$. The syntactic restrictions on the rules ensure monotonicity; by the Knaster–Tarski theorem, the fixed-point equation

$$p(\bar{x}) \;\simeq\; \big(\exists \bar{y}.\, \bigvee_{j=1}^{n} \bar{x} \simeq \bar{u}_j \wedge p(\bar{t}_{j1}) \wedge \cdots \wedge p(\bar{t}_{j\ell_j}) \wedge Q_j\big)$$

admits a least and a greatest solution [17, 30]. Inductive definitions provide the least fixed point, and coinductive definitions provide the greatest fixed point.

As an example, assuming a type *nat* of natural numbers generated freely by 0^{nat} and $Suc^{nat \to nat}$, the following definition introduces the predicate *even* of even numbers:

inductive $even^{nat \to \mathrm{o}}$ where
$even(0)$
$even(n) \longrightarrow even(Suc(Suc(n)))$

The associated fixed-point equation is

$$even(x) \;\simeq\; \big(\exists n.\; x \simeq 0 \vee (x \simeq Suc(Suc(n)) \wedge even(n))\big).$$

The syntax can be generalized to support mutual definitions, as in the next example:

inductive $even^{nat \to \mathrm{o}}$ and $odd^{nat \to \mathrm{o}}$ where
$even(0)$
$even(n) \longrightarrow odd(Suc(n))$
$odd(n) \longrightarrow even(Suc(n))$

Mutual definitions for $p_1, \ldots, p_m$ can be reduced to a single predicate q whose domain is the disjoint sum of the domains of each p_i [30]. Assuming *Inl* and *Inr* are the disjoint sum constructors, the definition of *even* and *odd* is replaced by

inductive $even\_or\_odd^{(nat,nat)\,sum \to \mathrm{o}}$ where
$even\_or\_odd(Inl(0))$
$even\_or\_odd(Inl(n)) \longrightarrow even\_or\_odd(Inr(Suc(n)))$
$even\_or\_odd(Inr(n)) \longrightarrow even\_or\_odd(Inl(Suc(n)))$

definition $even^{nat \to \mathrm{o}}$ where $even(n) \simeq even\_or\_odd(Inl(n))$
definition $odd^{nat \to \mathrm{o}}$where $odd(n) \simeq even\_or\_odd(Inr(n))$

3.3 (Co)algebraic Datatypes

The datatype and codatatype commands define mutually recursive (co)algebraic datatypes specified by their constructors:

[co]datatype $(\bar{\alpha})\,\kappa_1 = C_{11}\ \big[\text{of } \bar{\sigma}_{11}\big] \mid \cdots \mid C_{1\ell_1}\ \big[\text{of } \bar{\sigma}_{1\ell_1}\big]$
and …
and $(\bar{\alpha})\,\kappa_n = C_{n1}\ \big[\text{of } \bar{\sigma}_{n1}\big] \mid \cdots \mid C_{n\ell_n}\ \big[\text{of } \bar{\sigma}_{n\ell_n}\big]$

The defined types $(\bar{\alpha})\,\kappa_i$ are parameterized by a list of distinct type variables $\bar{\alpha}$, providing type polymorphism. Each constructor C_{ij} has type $\bar{\sigma}_{ij} \to (\bar{\alpha})\,\kappa_i$. If the optional syntax "of $\bar{\sigma}_{ij}$"

is omitted, the constructor has type $(\bar{\alpha})\,\kappa_i$. The arguments $\bar{\alpha}$ are required to be the same for all the type constructors κ_i.

Provisos: The type names κ_i and the constructor constants C_{ij} are fresh and distinct, the type parameters $\bar{\alpha}$ are distinct, and the argument types $\bar{\sigma}_{ij}$ do not refer to any other type variables than $\bar{\alpha}$ (but may refer to the types $(\bar{\alpha})\,\kappa_i$ being defined).

The datatype command corresponds roughly to Standard ML datatypes [5, 16], whereas codatatype behaves like Haskell data in that it allows infinite objects [31].

The commands can be used to define natural numbers, pairs, finite lists, and possibly infinite lazy lists as follows:

$$\begin{array}{l}
\textsf{datatype}\ \mathit{nat} = 0 \mid \mathit{Suc}\ \textsf{of}\ \mathit{nat}\\
\textsf{datatype}\ (\alpha,\beta)\,\mathit{pair} = \mathit{Pair}\ \textsf{of}\ (\alpha,\beta)\\
\textsf{datatype}\ \alpha\,\mathit{list} = \mathit{Nil} \mid \mathit{Cons}\ \textsf{of}\ (\alpha, \alpha\,\mathit{list})\\
\textsf{codatatype}\ \alpha\,\mathit{llist} = \mathit{LNil} \mid \mathit{LCons}\ \textsf{of}\ (\alpha, \alpha\,\mathit{llist})
\end{array}$$

Mutually recursive trees and forests can be defined just as easily:

$$\begin{array}{l}
\textsf{datatype}\ \alpha\ \mathit{tree} = \mathit{Empty} \mid \mathit{Node}\ \textsf{of}\ (\alpha, \alpha\,\mathit{forest})\\
\quad\ \textsf{and}\ \alpha\,\mathit{forest} = \mathit{FNil} \mid \mathit{FCons}\ \textsf{of}\ (\alpha\ \mathit{tree}, \alpha\,\mathit{forest})
\end{array}$$

Defining a (co)datatype introduces the appropriate axioms for the constructors [30]. It also introduces the syntax

$$\mathit{case}\ t\ \mathit{of}\ C_{i1}(\bar{x}_1) \Rightarrow u_1 \mid \ldots \mid C_{i\ell_i}(\bar{x}_{\ell_i}) \Rightarrow u_{\ell_i},$$

characterized by the axioms

$$\forall \bar{x}_j.\ (\mathit{case}\ C_{ij}(\bar{x}_j)\ \mathit{of}\ C_{i1}(\bar{x}_1) \Rightarrow u_1 \mid \ldots \mid C_{i\ell_i}(\bar{x}_{\ell_i}) \Rightarrow u_{\ell_i}) \simeq u_j$$

for $j \in \{1, \ldots, \ell_i\}$.

3.4 (Co)recursive Functions

The primrec command defines primitive recursive functions on algebraic datatypes:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textsf{primrec}\ f_1^{\tau_1}\ \textsf{and}\ \ldots\ \textsf{and}\ f_n^{\tau_n}\ \textsf{where}}\\
f_1(C_{11}(\bar{x}_{11}), \bar{z}_{11}) \simeq t_{11} & \ldots & f_1(C_{1\ell_1}(\bar{x}_{1\ell_1}), \bar{z}_{1\ell_1}) \simeq t_{1\ell_1}\\
\quad\vdots & & \\
f_n(C_{n1}(\bar{x}_{n1}), \bar{z}_{n1}) \simeq t_{n1} & \ldots & f_n(C_{n\ell_n}(\bar{x}_{n\ell_n}), \bar{z}_{n\ell_n}) \simeq t_{n\ell_n}
\end{array}$$

Provisos: The constants f_i are fresh and distinct, the variables $\bar{x}_{ij}$ and $\bar{z}_{ij}$ are distinct for any given i and j, the right-hand sides t_{ij} involve no other variables than $\bar{x}_{ij}$ and $\bar{z}_{ij}$ and no type variables that do not occur in τ_i, and the first argument of any recursive call must be one of the $\bar{x}_{ij}$'s. The last condition ensures that each recursive call peels off one constructor from the first argument and hence that the recursion is well-founded, guaranteeing consistency.

Corecursive function definitions follow a rather different syntactic schema, with a single equation per function f_i that must return type $(\bar{\alpha})\,\kappa_i$:

$$\begin{array}{l}
\textsf{coprimrec}\ f_1^{\tau_1}\ \textsf{and}\ \ldots\ \textsf{and}\ f_n^{\tau_n}\ \textsf{where}\\
f_1(\bar{y}_1) \simeq \mathit{if}\ Q_{11}\ \mathit{then}\ t_{11}\ \mathit{else}\ \mathit{if}\ Q_{12}\ \mathit{then}\ \ldots\ \mathit{else}\ t_{1\ell_1}\\
\quad\vdots\\
f_n(\bar{y}_n) \simeq \mathit{if}\ Q_{n1}\ \mathit{then}\ t_{n1}\ \mathit{else}\ \mathit{if}\ Q_{n2}\ \mathit{then}\ \ldots\ \mathit{else}\ t_{n\ell_n}
\end{array}$$

Provisos: The constants f_i are fresh and distinct, the variables $\bar{y}_i$ are distinct, the right-hand sides involve no other variables than $\bar{y}_i$, no corecursive calls occur in the conditions Q_{ij}, and either t_{ij} does not involve any corecursive calls or it has the form $C_{ij}(\bar{u}_{ij})$.[2]

The syntax can be relaxed to allow a *case* expression instead of a sequence of conditionals. What matters is that corecursive calls are protected by constructors, to ensure that the functions f_i are productive and hence well-defined.

The following examples define concatenation for the algebraic and coalgebraic list datatypes from Sect. 3.3:

$$\begin{aligned}
&\textsf{primrec}\ \mathit{cat}^{(\alpha\,\mathit{list},\,\alpha\,\mathit{list})\to\alpha\,\mathit{list}}\ \textsf{where}\\
&\mathit{cat}(\mathit{Nil}, \mathit{zs}) \simeq \mathit{zs}\\
&\mathit{cat}(\mathit{Cons}(y, \mathit{ys}), \mathit{zs}) \simeq \mathit{Cons}(y, \mathit{cat}(\mathit{ys}, \mathit{zs}))
\end{aligned}$$

$$\begin{aligned}
&\textsf{coprimrec}\ \mathit{lcat}^{(\alpha\,\mathit{llist},\,\alpha\,\mathit{llist})\to\alpha\,\mathit{llist}}\ \textsf{where}\\
&\mathit{lcat}(\mathit{ys}, \mathit{zs}) \simeq \mathit{case}\ \mathit{ys}\ \mathit{of}\ \mathit{LNil} \Rightarrow \mathit{zs} \mid \mathit{LCons}(y, \mathit{ys}') \Rightarrow \mathit{LCons}(y, \mathit{lcat}(\mathit{ys}', \mathit{zs}))
\end{aligned}$$

4 Basic Translations

4.1 A Sound and Complete Translation

This section presents the translation of FOL to FORL, excluding the definitional principles from Sect. 3. We consider only finite domains; for these the translation is both sound and complete. This simple (and well-known [35]) translation serves as a stepping stone toward the more sophisticated translations of Sects. 4.2 and 5.

We start by mapping FOL types τ to sets of FORL atom tuples $\langle\!\langle \tau \rangle\!\rangle$:

$$\langle\!\langle \sigma \rangle\!\rangle = \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\sigma|}\} \qquad \langle\!\langle (\sigma_1, \ldots, \sigma_n) \to \sigma \rangle\!\rangle = \langle\!\langle \sigma_1 \rangle\!\rangle \times \cdots \times \langle\!\langle \sigma_n \rangle\!\rangle \times \langle\!\langle \sigma \rangle\!\rangle.$$

For simplicity, we reuse the same atoms for distinct basic types. A real implementation can benefit from using distinct atoms because it produces more symmetric problems amenable to symmetry breaking [12, 35].[3]

Since FORL's syntax distinguishes between formulas and terms, the translation to FORL is performed by two mutually recursive functions, $\mathsf{F}\langle\!\langle t \rangle\!\rangle$ and $\mathsf{T}\langle\!\langle t \rangle\!\rangle$:[4]

$$\begin{aligned}
\mathsf{F}\langle\!\langle \mathit{False} \rangle\!\rangle &= \mathsf{false} & \mathsf{T}\langle\!\langle x \rangle\!\rangle &= x\\
\mathsf{F}\langle\!\langle \mathit{True} \rangle\!\rangle &= \mathsf{true} & \mathsf{T}\langle\!\langle \mathit{False} \rangle\!\rangle &= \mathsf{a}_1\\
\mathsf{F}\langle\!\langle t \simeq u \rangle\!\rangle &= \mathsf{T}\langle\!\langle t \rangle\!\rangle \simeq \mathsf{T}\langle\!\langle u \rangle\!\rangle & \mathsf{T}\langle\!\langle \mathit{True} \rangle\!\rangle &= \mathsf{a}_2\\
\mathsf{F}\langle\!\langle t \longrightarrow u \rangle\!\rangle &= \mathsf{F}\langle\!\langle t \rangle\!\rangle \longrightarrow \mathsf{F}\langle\!\langle u \rangle\!\rangle & \mathsf{T}\langle\!\langle \mathit{if}\ t\ \mathit{then}\ u_1\ \mathit{else}\ u_2 \rangle\!\rangle &= \mathsf{if}\ \mathsf{F}\langle\!\langle t \rangle\!\rangle\ \mathsf{then}\ \mathsf{T}\langle\!\langle u_1 \rangle\!\rangle\ \mathsf{else}\ \mathsf{T}\langle\!\langle u_2 \rangle\!\rangle\\
\mathsf{F}\langle\!\langle \forall x^{\sigma}.\, t \rangle\!\rangle &= \forall x \in \langle\!\langle \sigma \rangle\!\rangle \colon \mathsf{F}\langle\!\langle t \rangle\!\rangle & \mathsf{T}\langle\!\langle c(t_1, \ldots, t_n) \rangle\!\rangle &= \mathsf{T}\langle\!\langle t_n \rangle\!\rangle.(\ldots.(\mathsf{T}\langle\!\langle t_1 \rangle\!\rangle.c)\ldots)\\
\mathsf{F}\langle\!\langle t \rangle\!\rangle &= \mathsf{T}\langle\!\langle t \rangle\!\rangle \simeq \mathsf{T}\langle\!\langle \mathit{True} \rangle\!\rangle & \mathsf{T}\langle\!\langle t^{\mathrm{o}} \rangle\!\rangle &= \mathsf{T}\langle\!\langle \mathit{if}\ t\ \mathit{then}\ \mathit{True}\ \mathit{else}\ \mathit{False} \rangle\!\rangle.
\end{aligned}$$

[2] Other authors formulate corecursion in terms of selectors instead of constructors [20].

[3] Because of bound declarations, which refer to atoms by name, FORL atoms are generally not interchangeable. Kodkod's symmetry breaker infers symmetries (classes of atoms that can be permuted with each other) from the bound declarations and generates additional constraints to rule out needless permutations [35]. This usually speeds up model finding, especially for higher cardinalities.

[4] Metatheoretic functions here and elsewhere are defined using sequential pattern matching.

The metavariable c ranges over nonstandard constants, so that the $\mathsf{T}\langle\!\langle t^{\mathrm{o}}\rangle\!\rangle$ equation is used for $\simeq$ and $\longrightarrow$ (as well as for $\forall$). The Boolean values false and true are arbitrarily coded as a_1 and a_2 when they appear as FORL terms.

For each free variable or nonstandard constant u^{τ}, we must also generate the declaration $\mathsf{var}\ \emptyset \subseteq u \subseteq \langle\!\langle\tau\rangle\!\rangle$ as well as a constraint $\Phi(u)$ to ensure that single values are singletons and functions are functions:

$$\Phi(u^{\sigma}) = \mathsf{one}\ u$$
$$\Phi(u^{(\sigma_1,\ldots,\sigma_n)\to\sigma}) = \forall x_1 \in \langle\!\langle\sigma_1\rangle\!\rangle, \ldots, x_n \in \langle\!\langle\sigma_n\rangle\!\rangle\colon \mathsf{one}\ x_n.(\ldots.(x_1.u)\ldots).$$

The variables $x_1, \ldots, x_n$ must be fresh.

Theorem 4.1 *The FOL formula P with free variables and nonstandard constants $u_1^{\tau_1}, \ldots, u_n^{\tau_n}$ is satisfiable for a given finite scope iff the FORL formula $\mathsf{F}\langle\!\langle P\rangle\!\rangle \wedge \bigwedge_{j=1}^{n} \Phi(u_j)$ with bounds $\emptyset \subseteq u_j \subseteq \langle\!\langle\tau_j\rangle\!\rangle$ is satisfiable for the same scope.*

Proof Let $[\![t]\!]_M$ denote the set-theoretic semantics of the FOL term t w.r.t. a model M and the given scope S, let $[\![\varphi]\!]_V$ denote the truth value of the FORL formula φ w.r.t. a variable valuation V and the scope S, and let $[\![r]\!]_V$ denote the set-theoretic semantics of the FORL term r w.r.t. V and S. Furthermore, for $v \in [\![\sigma]\!]_S$, let $\lfloor v\rfloor$ denote the corresponding value in $\langle\!\langle\sigma\rangle\!\rangle$, with $\lfloor f\!f\rfloor = \mathsf{a}_1$ and $\lfloor tt\rfloor = \mathsf{a}_2$. Using recursion induction, it is straightforward to prove that $[\![\mathsf{F}\langle\!\langle t^{\mathrm{o}}\rangle\!\rangle]\!]_V \iff [\![t]\!]_M = tt$ and $[\![\mathsf{T}\langle\!\langle t\rangle\!\rangle]\!]_V = \lfloor[\![t]\!]_M\rfloor$ if $V(u_j) = \lfloor M(u_j)\rfloor$ for all u_j's. Moreover, from a satisfying valuation V of the u_j's, we can construct a FOL model M such that $\lfloor M(u_j)\rfloor = V(u_j)$; the Φ constraints and the bounds ensure that such a model exists. Hence, $[\![\mathsf{F}\langle\!\langle P\rangle\!\rangle]\!]_V \iff [\![P]\!]_M = tt$. $\square$

The translation is parameterized by a scope, which specifies the exact cardinalities of the basic types occurring in the formula. To exhaust all models up to a cardinality bound k for n basic types, a model finder must a priori iterate through k^n combinations of cardinalities and must consider all models for each of these combinations. This can be made more efficient if the problem is scope-monotonic [7, 25]. Another option is to avoid hard-coding the exact cardinalities in the translation and let the SAT solver try all cardinalities up to a given bound; this is Alloy's normal mode of operation [19, p. 129].

Example 4.1 Given a cardinality k of α, the FOL formula $P(x^{\alpha}) \longrightarrow P(y)$ is translated into the FORL specification

$$\begin{array}{l}
\mathsf{var}\ \emptyset \subseteq x \subseteq \{\mathsf{a}_1,\ldots,\mathsf{a}_k\}\\
\mathsf{var}\ \emptyset \subseteq y \subseteq \{\mathsf{a}_1,\ldots,\mathsf{a}_k\}\\
\mathsf{var}\ \emptyset \subseteq P \subseteq \{\mathsf{a}_1,\ldots,\mathsf{a}_k\} \times \{\mathsf{a}_1,\mathsf{a}_2\}\\
\mathsf{solve}\ \mathsf{one}\ x \wedge \mathsf{one}\ y \wedge (\forall x_1 \in \{\mathsf{a}_1,\ldots,\mathsf{a}_k\}\colon \mathsf{one}\ x_1.P)\\
\quad \wedge\ (x.P \simeq \mathsf{a}_2 \longrightarrow y.P \simeq \mathsf{a}_2)
\end{array}$$

The first three conjuncts ensure that x and y are scalars and that P is a function. The last conjunct is the translation of the FOL formula. A solution exists already for $k = 1$, namely $x = y = \mathsf{a}_1$ and $P = \{\langle\mathsf{a}_1,\mathsf{a}_1\rangle\}$. $\blacksquare$

4.2 Approximation of Infinite Types and Partiality

Besides its lack of support for the definitional principles, the above translation suffers from a serious limitation: It disregards infinite types such as natural numbers, lists, and trees, which are ubiquitous in real-world specifications. Fortunately, it is not hard to adapt the translation to take these into account in a sound (but incomplete) way.

Given an infinite atomic type κ, we consider a finite subset of $[\![\kappa]\!]_S$ and map every element not in this subset to a special undefined value $\star$. For the type *nat* of natural numbers, an obvious choice is to consider prefixes $\{0,\ldots,K\}$ of $\mathbb{N}$ and map numbers $> K$ to $\star$. Observe that the successor function *Suc* becomes partial, with $\mathit{Suc}(K) = \star$. The technique can also be used to speed up the analysis of finite types with a high cardinality: We can approximate a 256-value *byte* type by a subset of, say, 5 values.

Leaving out some elements of atomic types means that we must cope with partiality. Not only may functions be partial, but any term or formula can evaluate to $\star$. The logic becomes a three-valued Kleene logic [21]. Universal quantifiers whose bound variable ranges over an approximated type, such as $\forall n^{\mathit{nat}}.\ P[n]$, will evaluate to either *False* (if $P[n]$ gives *False* for some $n \le K$) or $\star$, but never to *True*, since we cannot ascertain whether $P[K+1]$, $P[K+2]$, …, are true.

Partiality can be encoded in FORL as follows. Inside terms, we let none (the empty set) stand for $\star$. This choice is convenient because none is an absorbing element for the dot-join operator, which models function application; thus, $f(\star) = \star$ irrespective of f. Inside a formula, we keep track of the polarity of the subformulas: In positive contexts (i.e., under an even number of negations), true codes *True* and false codes *False* or $\star$; in negative contexts, false codes *False* and true codes *True* or $\star$.

The translation of FOL terms is performed by two functions, $\mathsf{F}^s\langle\!\langle t\rangle\!\rangle$ and $\mathsf{T}\langle\!\langle t\rangle\!\rangle$, where s indicates the polarity ($+$ or $-$):

$$
\begin{aligned}
\mathsf{F}^s\langle\!\langle \mathit{False}\rangle\!\rangle &= \mathsf{false} \\
\mathsf{F}^s\langle\!\langle \mathit{True}\rangle\!\rangle &= \mathsf{true} \\
\mathsf{F}^+\langle\!\langle t \simeq u\rangle\!\rangle &= \mathsf{some}\ (\mathsf{T}\langle\!\langle t\rangle\!\rangle \cap \mathsf{T}\langle\!\langle u\rangle\!\rangle) \\
\mathsf{F}^-\langle\!\langle t \simeq u\rangle\!\rangle &= \mathsf{lone}\ (\mathsf{T}\langle\!\langle t\rangle\!\rangle \cup \mathsf{T}\langle\!\langle u\rangle\!\rangle) \\
\mathsf{F}^s\langle\!\langle t \longrightarrow u\rangle\!\rangle &= \mathsf{F}^{-s}\langle\!\langle t\rangle\!\rangle \longrightarrow \mathsf{F}^s\langle\!\langle u\rangle\!\rangle \\
\mathsf{F}^+\langle\!\langle \forall x^\sigma\!.\, t\rangle\!\rangle &= \mathsf{false} \quad \text{if } |\langle\!\langle \sigma\rangle\!\rangle| < |\sigma| \\
\mathsf{F}^s\langle\!\langle \forall x^\sigma\!.\, t\rangle\!\rangle &= \forall x \in \langle\!\langle \sigma\rangle\!\rangle\colon \mathsf{F}^s\langle\!\langle t\rangle\!\rangle \\
\mathsf{F}^+\langle\!\langle t\rangle\!\rangle &= \mathsf{T}\langle\!\langle t\rangle\!\rangle \simeq \mathsf{T}\langle\!\langle \mathit{True}\rangle\!\rangle \\
\mathsf{F}^-\langle\!\langle t\rangle\!\rangle &= \mathsf{T}\langle\!\langle t\rangle\!\rangle \not\simeq \mathsf{T}\langle\!\langle \mathit{False}\rangle\!\rangle
\end{aligned}
\qquad
\begin{aligned}
\mathsf{T}\langle\!\langle x\rangle\!\rangle &= x \\
\mathsf{T}\langle\!\langle \mathit{False}\rangle\!\rangle &= \mathsf{a}_1 \\
\mathsf{T}\langle\!\langle \mathit{True}\rangle\!\rangle &= \mathsf{a}_2 \\
\mathsf{T}\langle\!\langle \mathit{if}\ t\ \mathit{then}\ u_1\ \mathit{else}\ u_2\rangle\!\rangle &= \mathsf{if}\ \mathsf{F}^+\langle\!\langle t\rangle\!\rangle\ \mathsf{then}\ \mathsf{T}\langle\!\langle u_1\rangle\!\rangle \\
&\ \mathsf{else\ if}\ \neg\,\mathsf{F}^-\langle\!\langle t\rangle\!\rangle\ \mathsf{then}\ \mathsf{T}\langle\!\langle u_2\rangle\!\rangle \\
&\ \mathsf{else\ none} \\
\mathsf{T}\langle\!\langle c(t_1,\ldots,t_n)\rangle\!\rangle &= \mathsf{T}\langle\!\langle t_n\rangle\!\rangle.(\ldots.(\mathsf{T}\langle\!\langle t_1\rangle\!\rangle.c)\ldots) \\
\mathsf{T}\langle\!\langle t^{\mathrm{o}}\rangle\!\rangle &= \mathsf{T}\langle\!\langle \mathit{if}\ t\ \mathit{then}\ \mathit{True} \\
&\ \ \mathit{else}\ \mathit{False}\rangle\!\rangle.
\end{aligned}
$$

In the equation for implication, $-s$ denotes $-$ if s is $+$ and $+$ if s is $-$. Taken together, the Boolean values $\mathsf{F}^+\langle\!\langle t\rangle\!\rangle$ and $\mathsf{F}^-\langle\!\langle t\rangle\!\rangle$ encode a three-valued logic, with $\langle\mathsf{true}, \mathsf{true}\rangle$ denoting *True*, $\langle\mathsf{false}, \mathsf{true}\rangle$ denoting $\star$, and $\langle\mathsf{false}, \mathsf{false}\rangle$ denoting *False*. The remaining case, $\langle\mathsf{true}, \mathsf{false}\rangle$, is impossible by construction.

When mapping FOL types to sets of FORL atom tuples, basic types σ are now allowed to take any finite cardinality $|\langle\!\langle \sigma\rangle\!\rangle| \le |\sigma|$. We also need to relax the definition of $\Phi(u)$ to allow empty sets, by substituting lone for one:

$$\Phi(u^\sigma) = \mathsf{lone}\ u \qquad \Phi(u^{(\sigma_1,\ldots,\sigma_n)\to\sigma}) = \forall x_1 \in \langle\!\langle \sigma_1\rangle\!\rangle, \ldots, x_n \in \langle\!\langle \sigma_n\rangle\!\rangle\colon \mathsf{lone}\ x_n.(\ldots.(x_1.u)\ldots).$$

In the face of partiality, the new encoding is sound but no longer complete.

Theorem 4.2 *Given a FOL formula P with free variables and nonstandard constants $u_1^{\tau_1}$, $\ldots$, $u_n^{\tau_n}$ and a scope S, the FORL formula $\mathsf{F}^+\langle\!\langle P\rangle\!\rangle \wedge \bigwedge_{j=1}^{n} \Phi(u_j)$ with bounds $\emptyset \subseteq u_j \subseteq \langle\!\langle \tau_j\rangle\!\rangle$ is satisfiable for S only if P is satisfiable for S.*

Proof The proof is similar to that of Theorem 4.1, but partiality requires us to compare the actual value of a FORL expression with its expected value using $\subseteq$ rather than $=$. Using recursion induction, we can prove that $[\![\mathsf{F}^+\langle\!\langle t^{\mathrm{o}}\rangle\!\rangle]\!]_V \Longrightarrow [\![t]\!]_M = tt$, $\neg\,[\![\mathsf{F}^-\langle\!\langle t^{\mathrm{o}}\rangle\!\rangle]\!]_V \Longrightarrow [\![t]\!]_M = f\!f$, and $[\![\mathsf{T}\langle\!\langle t\rangle\!\rangle]\!]_V \subseteq \lfloor [\![t]\!]_M \rfloor$ if $V(u) \subseteq \lfloor M(u) \rfloor$ for all free variables and nonstandard constants u occurring in t. Some of the cases deserve more justification:

- The $\mathsf{F}^+\langle\!\langle t \simeq u\rangle\!\rangle$ equation is sound because if the intersection of $\mathsf{T}\langle\!\langle t\rangle\!\rangle$ and $\mathsf{T}\langle\!\langle u\rangle\!\rangle$ is nonempty, then t and u must be equal (since they are singletons).
- The $\mathsf{F}^-\langle\!\langle t \simeq u\rangle\!\rangle$ equation is dual: If the union of $\mathsf{T}\langle\!\langle t\rangle\!\rangle$ and $\mathsf{T}\langle\!\langle u\rangle\!\rangle$ has more than one element, then t and u must be unequal.
- Universal quantification occurring positively can never yield true if the bound variable ranges over an approximated type. (In negative contexts, approximation compromises the encoding's completeness but not its soundness.)
- The *if then else* equation carefully distinguishes between the cases where the condition is *True*, *False*, and $\star$. In the *True* case, it returns the *then* value. In the *False* case, it returns the *else* value. In the $\star$ case, it returns $\star$ (none).[5]
- The $\mathsf{T}\langle\!\langle c(t_1,\ldots,t_n)\rangle\!\rangle$ equation is as before. If any of the arguments t_j evaluates to none, the entire dot-join expression yields none.

Moreover, from a satisfying valuation V of the u_j's, we can construct a FOL model M such that $V(u_j) \subseteq \lfloor M(u_j) \rfloor$ for all u_j's, by defining $M(u_j)$ arbitrarily if $V(u_j) = \emptyset$ or at points where the partial function $V(u_j)$ is undefined. Hence, $[\![\mathsf{F}^+\langle\!\langle P\rangle\!\rangle]\!]_V$ implies $[\![P]\!]_M = tt$. $\square$

Although our translation is sound, a lot of precision is lost for $\simeq$ and $\forall$. Fortunately, by handling high-level definitional principles specially (as opposed to directly translating their FOL axiomatization), we can bypass the imprecise translation and increase the precision. This is covered in the next section.

5 Translation of Definitional Principles

5.1 Axiomatization of Simple Definitions

Once we extend the specification logic with simple definitions, we must also encode these in the FORL formula. More precisely, if c^τ is defined and an instance $c^{\tau'}$ occurs in a formula, we must conjoin c's definition with the formula, instantiating τ with τ'. This process must be repeated for any defined constants occurring in c's definition. It will eventually terminate, since cyclic definitions are disallowed. If several type instances of the same constant are needed, they must be given distinct names in the translation.

Given the command

$$\textsf{definition } c^\tau \textsf{ where } c(\bar{x}) \simeq t$$

the naive approach would be to conjoin $\mathsf{F}^+\langle\!\langle \forall \bar{x}.\ c(\bar{x}) \simeq t\rangle\!\rangle$ with the FORL formula to satisfy and recursively do the same for any defined constants in t. However, there are two problems with this approach:

[5] We could gain some precision by returning if $\mathsf{T}\langle\!\langle u_1\rangle\!\rangle \simeq \mathsf{T}\langle\!\langle u_2\rangle\!\rangle$ then $\mathsf{T}\langle\!\langle u_1\rangle\!\rangle$ else none instead.

- If any of the variables $\bar{x}$ is of an approximated type, the equation $\mathsf{F}^{+}\langle\!\langle \forall \bar{x}.\, t\rangle\!\rangle = \mathsf{false}$ applies, and the axiom becomes unsatisfiable. This is sound but extremely imprecise, as it prevents the discovery of any model.
- Otherwise, the body of $\forall \bar{x}.\, c(\bar{x}) \simeq t$ is translated to $\mathsf{some}\ (\mathsf{T}\langle\!\langle c(\bar{x})\rangle\!\rangle \cap \mathsf{T}\langle\!\langle t\rangle\!\rangle)$, which evaluates to false whenever $\mathsf{T}\langle\!\langle t\rangle\!\rangle$ is none for some values of $\bar{x}$.

Fortunately, we can take a shortcut and translate the definition directly to the following FORL axiom, bypassing F^{+} altogether (cf. Weber [36, p. 66]):

$$\forall x_1 \in \langle\!\langle \sigma_1\rangle\!\rangle, \ldots, x_n \in \langle\!\langle \sigma_n\rangle\!\rangle\colon \mathsf{T}\langle\!\langle c(x_1,\ldots,x_n)\rangle\!\rangle \simeq \mathsf{T}\langle\!\langle t\rangle\!\rangle.$$

We must also define the variable using appropriate bounds for the constant's type.

Example 5.1 The formula $snd(a, a) \simeq a^{\alpha}$, where snd is defined as

$$\mathsf{definition}\ snd^{(\beta,\gamma)\to\gamma}\ \mathsf{where}\ snd(x, y) \simeq y$$

is translated to

$$\begin{array}{l}
\mathsf{var}\ \emptyset \subseteq a \subseteq \langle\!\langle \alpha\rangle\!\rangle \\
\mathsf{var}\ \emptyset \subseteq snd \subseteq \langle\!\langle \alpha\rangle\!\rangle \times \langle\!\langle \alpha\rangle\!\rangle \times \langle\!\langle \alpha\rangle\!\rangle \\
\mathsf{solve}\ \mathsf{lone}\ a \\
\quad \wedge\ \big(\forall x_1, x_2 \in \langle\!\langle \alpha\rangle\!\rangle\colon \mathsf{lone}\ x_2.(x_1.snd)\big) \\
\quad \wedge\ \big(\forall x, y \in \langle\!\langle \alpha\rangle\!\rangle\colon y.(x.snd) \simeq y\big) \\
\quad \wedge\ \mathsf{some}\ (a.(a.snd) \cap a)
\end{array}$$

Notice that *snd*'s type variables β and γ are instantiated with α in the translation, which happens to be a type variable itself. ■

Theorem 5.1 *The encoding of Sect. 4.2 extended with simple definitions is sound.*

Proof Any FORL valuation V that satisfies the FORL axiom for a constant c can be extended into a FOL model M that satisfies the corresponding FOL axiom, by setting $M(c)(\bar{v}) = [\![t]\!]_M(\bar{v})$ for any values $\bar{v}$ at which $V(c)$ is not defined (either because $\bar{v}$ is not representable in FORL or because the partial function $V(c)$ is not defined at that point). The apparent circularity in $M(c)(\bar{v}) = [\![t]\!]_M(\bar{v})$ is harmless, because simple definitions are required to be acyclic and so we can construct M one constant at a time. □

Incidentally, we obtain a simpler (and still sound) SAT encoding by replacing the $\simeq$ operator with $\subseteq$ in the encoding of simple definition. Any entry of a defined constant's relation table that is not needed to construct the model can then be $\star$, even if the right-hand side of the definition is representable.

5.2 Axiomatization of Algebraic Datatypes and Recursive Functions

The FORL axiomatization of algebraic datatypes follows the lines of Kuncak and Jackson [25]. Let

$$\kappa = C_1\ \mathsf{of}\ (\sigma_{11},\ldots,\sigma_{1n_1})\ |\ \cdots\ |\ C_\ell\ \mathsf{of}\ (\sigma_{\ell 1},\ldots,\sigma_{\ell n_\ell})$$

be a datatype instance. With each constructor C_i, we associate a discriminator $D_i^{\kappa\to o}$ and n_i selectors $S_{ik}^{\kappa\to\sigma_{ik}}$ obeying the laws

$$D_j(C_i(\bar{x})) \simeq (i \simeq j) \qquad\qquad S_{ik}(C_i(x_1,\ldots,x_n)) \simeq x_k.$$

For example, the type α *list* is assigned the discriminators *nilp* and *consp* and the selectors *head* and *tail*:[6]

$$\begin{array}{ccc} nilp(Nil) \simeq True & nilp(Cons(x, xs)) \simeq False & head(Cons(x, xs)) \simeq x \\ consp(Nil) \simeq False & consp(Cons(x, xs)) \simeq True & tail(Cons(x, xs)) \simeq xs. \end{array}$$

The discriminator and selector view almost always results in a more efficient SAT encoding than the constructor view because it breaks high-arity constructors into several low-arity discriminators and selectors, declared as follows (for all possible i, k):

$$\mathsf{var}\ \emptyset \subseteq D_i \subseteq \langle\!\langle \kappa \rangle\!\rangle \qquad\qquad \mathsf{var}\ \emptyset \subseteq S_{ik} \subseteq \langle\!\langle \kappa \to \sigma_{ik} \rangle\!\rangle$$

The predicate D_i is directly coded as a set of atoms, rather than as a function to $\{\mathsf{a}_1, \mathsf{a}_2\}$.

Let $C_i\langle r_1, \ldots, r_n\rangle$ stand for $S_{i1}.r_1 \cap \cdots \cap S_{in}.r_n$ if $n \geq 1$, and $C_i\langle\rangle = D_i$ for parameterless constructors. Intuitively, $C_i\langle r_1, \ldots, r_n\rangle$ represents the constructor C_i with arguments $r_1, \ldots, r_n$ at the FORL level [13]. A faithful axiomatization of datatypes in terms of D_i and S_{ik} involves the following axioms (for all possible i, j, k):

$$\begin{array}{rl} \textsc{Disjoint}_{ij}\text{:} & \mathsf{no}\ D_i \cap D_j \quad \text{for } i < j \\ \textsc{Exhaustive}\text{:} & D_1 \cup \cdots \cup D_\ell \simeq \langle\!\langle \kappa \rangle\!\rangle \\ \textsc{Selector}_{ik}\text{:} & \forall y \in \langle\!\langle \kappa \rangle\!\rangle\text{: } \mathsf{if}\ y \subseteq D_i\ \mathsf{then}\ \mathsf{one}\ y.S_{ik}\ \mathsf{else}\ \mathsf{no}\ y.S_{ik} \\ \textsc{Unique}_i\text{:} & \forall x_1 \in \langle\!\langle \sigma_1 \rangle\!\rangle, \ldots, x_{n_i} \in \langle\!\langle \sigma_{n_i} \rangle\!\rangle\text{: } \mathsf{lone}\ C_i\langle x_1, \ldots, x_{n_i}\rangle \\ \textsc{Generator}_i\text{:} & \forall x_1 \in \langle\!\langle \sigma_1 \rangle\!\rangle, \ldots, x_{n_i} \in \langle\!\langle \sigma_{n_i} \rangle\!\rangle\text{: } \mathsf{some}\ C_i\langle x_1, \ldots, x_{n_i}\rangle \\ \textsc{Acyclic}\text{:} & \mathsf{no}\ sup_\kappa \cap \mathsf{iden}. \end{array}$$

In the last axiom, sup_κ denotes the proper superterm relation for κ. For example, we have $sup_\kappa(Cons(x, xs), xs)$ for any x and xs because the second argument's value is a proper subterm of the first argument's value. We will see shortly how to derive sup_κ.

DISJOINT and EXHAUSTIVE ensure that the discriminators partition $\langle\!\langle \kappa \rangle\!\rangle$. The four remaining axioms, sometimes called the SUGA axioms (after the first letter of each axiom name), ensure that selectors are functions whose domain is given by the corresponding discriminator (SELECTOR), that constructors are total functions (UNIQUE and GENERATOR), and that datatype values cannot be proper subterms or superterms of themselves (ACYCLIC). The injectivity of constructors follows from the functionality of selectors.

With this axiomatization, occurrences of $C_i(u_1, \ldots, u_n)$ in FOL are simply mapped to $C_i\langle \mathsf{T}\langle\!\langle u_1 \rangle\!\rangle, \ldots, \mathsf{T}\langle\!\langle u_n \rangle\!\rangle\rangle$, whereas *case* t *of* $C_1(\bar{x}_1) \Rightarrow u_1 \mid \ldots \mid C_\ell(\bar{x}_\ell) \Rightarrow u_\ell$ is coded as

$$\mathsf{if}\ \mathsf{T}\langle\!\langle t \rangle\!\rangle \subseteq D_1\ \mathsf{then}\ \mathsf{T}\langle\!\langle u_1^\bullet \rangle\!\rangle\ \mathsf{else\ if} \ldots \mathsf{else\ if}\ \mathsf{T}\langle\!\langle t \rangle\!\rangle \subseteq D_\ell\ \mathsf{then}\ \mathsf{T}\langle\!\langle u_\ell^\bullet \rangle\!\rangle\ \mathsf{else\ none},$$

where $u_i^\bullet$ denotes the term u_i in which all occurrences of the variables $\bar{x}_i = x_{i1}, \ldots, x_{in_i}$ are replaced with the corresponding selector expressions $S_{i1}(t), \ldots, S_{in_i}(t)$.

Unfortunately, the SUGA axioms admit no finite models if the type κ is recursive (and hence infinite), because they force the existence of infinitely many values. The solution is to leave GENERATOR out, yielding SUA. The SUA axioms characterize precisely the subterm-closed finite substructures of an algebraic datatype. In a two-valued logic, this is generally unsound, but Kuncak and Jackson [25] showed that omitting GENERATOR is sound for *existential–bounded-universal* (EBU) sentences—namely, the formulas whose prenex normal forms contain no unbounded universal quantifiers ranging over datatypes.

[6] These names were chosen for readability; any fresh names would do.

In our three-valued setting, omitting the GENERATOR axiom is always sound. The construct $C_i\langle r_1, \ldots, r_{n_i}\rangle$ sometimes returns none for non-none arguments, but this is not a problem since our translation of Sect. 4.2 is designed to cope with partiality. Non-EBU formulas such as $\mathit{True} \vee \forall n^{\mathit{nat}}.\ P(n)$ become analyzable when moving to a three-valued logic. This is especially important for complex specifications, because they are likely to contain non-EBU parts that are not needed for finding a model.

Example 5.2 The *nat list* instance of *α list* would be axiomatized as follows:

DISJOINT:	$\textsf{no}\ \mathit{nilp} \cap \mathit{consp}$
EXHAUSTIVE:	$\mathit{nilp} \cup \mathit{consp} \simeq \langle\!\langle \mathit{nat\ list} \rangle\!\rangle$
SELECTOR$_{\mathit{head}}$:	$\forall ys \in \langle\!\langle \mathit{nat\ list} \rangle\!\rangle$: $\textsf{if}\ ys \subseteq \mathit{consp}\ \textsf{then one}\ ys.\mathit{head}\ \textsf{else no}\ ys.\mathit{head}$
SELECTOR$_{\mathit{tail}}$:	$\forall ys \in \langle\!\langle \mathit{nat\ list} \rangle\!\rangle$: $\textsf{if}\ ys \subseteq \mathit{consp}\ \textsf{then one}\ ys.\mathit{tail}\ \textsf{else no}\ ys.\mathit{tail}$
UNIQUE$_{\mathit{Nil}}$:	$\textsf{lone}\ \mathit{Nil}\langle\rangle$
UNIQUE$_{\mathit{Cons}}$:	$\forall x \in \langle\!\langle \mathit{nat} \rangle\!\rangle, xs \in \langle\!\langle \mathit{nat\ list} \rangle\!\rangle$: $\textsf{lone}\ \mathit{Cons}\langle x, xs\rangle$
ACYCLIC:	$\textsf{no}\ \mathit{sup}_{\mathit{nat\ list}} \cap \textsf{iden}$ with $\mathit{sup}_{\mathit{nat\ list}} = \mathit{tail}^+$.

Examples of subterm-closed list substructures using traditional notation are $\{[], [0], [1]\}$ and $\{[], [1], [2,1], [0,2,1]\}$. In contrast, the set $L = \{[], [1,1]\}$ is not subterm-closed, because $\mathit{tail}([1,1]) = [1] \notin L$. Given a cardinality, Kodkod systematically enumerates all corresponding subterm-closed list substructures. ■

To generate the proper superterm relation needed for ACYCLIC, we must consider the general case of mutually recursive datatypes. We start by computing the datatype dependency graph, in which vertices are labeled with datatypes and arcs with selectors. For each selector $S^{\kappa \to \kappa'}$, we add an arc from κ to κ' labeled S. Next, we compute for each datatype a regular expression capturing the nontrivial paths in the graph from the datatype to itself. This can be done using Kleene's construction [22; 23, pp. 51–53]. The proper superterm relation is obtained from the regular expression by replacing concatenation with relational composition, alternative with set union, and repetition with transitive closure.

Example 5.3 Let *sym* be an atomic type, and consider the definitions

$$\begin{array}{l}\textsf{datatype}\ \alpha\,\mathit{list} = \mathit{Nil} \mid \mathit{Cons}\ \textsf{of}\ (\alpha, \alpha\,\mathit{list}) \\ \textsf{datatype}\ \mathit{tree} = \mathit{Leaf}\ \textsf{of}\ \mathit{sym} \mid \mathit{Node}\ \textsf{of}\ \mathit{tree\ list}\end{array}$$

Their dependency graph is

$$\begin{tikzcd}\mathit{tree} \arrow[r, bend left, "\mathit{children}"] & \mathit{tree\ list} \arrow[l, bend left, "\mathit{head}"] \arrow[loop right, "\mathit{tail}"]\end{tikzcd}$$

where *children* is the selector associated with *Node*. The superterm relations are

$$\mathit{sup}_{\mathit{tree}} = (\mathit{children}.\mathit{tail}^*.\mathit{head})^+ \qquad \mathit{sup}_{\mathit{tree\ list}} = (\mathit{tail} \cup \mathit{head}.\mathit{children})^+.$$

Notice that in the presence of polymorphism, instances of sequentially declared datatypes can be mutually recursive. ■

With a suitable axiomatization of datatypes as subterm-closed substructures, it is easy to encode primrec definitions. A recursive equation $f(C_i(x_1^{\sigma_1},\ldots,x_m^{\sigma_m}),\, z_1^{\sigma'_1},\,\ldots,\,z_n^{\sigma'_n}) \simeq t$ is translated to

$$\forall y \in D_i,\, z_1 \in \langle\!\langle \sigma'_1 \rangle\!\rangle, \ldots, z_n \in \langle\!\langle \sigma'_n \rangle\!\rangle\colon\ \mathsf{T}\langle\!\langle f(y,\, z_1, \ldots, z_n) \rangle\!\rangle \simeq \mathsf{T}\langle\!\langle t^{\bullet} \rangle\!\rangle,$$

where $t^{\bullet}$ is obtained from t by replacing the variables x_i with the selector expressions $S_i(y)$. By quantifying over the constructed values y rather than over the arguments to the constructors, we reduce the number of copies of the quantified body by a factor of $|\langle\!\langle \sigma_1 \rangle\!\rangle| \cdot \ldots \cdot |\langle\!\langle \sigma_n \rangle\!\rangle| \,/\, |\langle\!\langle \kappa \rangle\!\rangle|$ in the SAT problem. Although we focus here on primitive recursion, general well-founded recursion with non-overlapping pattern matching (as defined using Isabelle's function package [24]) can be handled in essentially the same way.

Example 5.4 The recursive function *cat* from Sect. 3.4 is translated to

$$\begin{aligned}
&\forall ys \in nilp,\, zs \in \langle\!\langle \alpha\ list \rangle\!\rangle\colon\ zs.(ys.cat) \simeq zs \\
&\forall ys \in consp,\, zs \in \langle\!\langle \alpha\ list \rangle\!\rangle\colon\ zs.(ys.cat) \simeq Cons\langle ys.head,\, zs.((ys.tail).cat)\rangle. \quad \blacksquare
\end{aligned}$$

Theorem 5.2 *The encoding of Sect. 5.1 extended with algebraic datatypes and primitive recursion is sound.*

Proof Kuncak and Jackson [25] proved that SUA axioms precisely describe subterm-closed finite substructures of an algebraic datatype, and showed how to generalize this result to mutually recursive datatypes. This means that we can always extend the valuation of the SUA-specified descriptors and selectors to obtain a model of the entire datatype. For recursion, we can prove $[\![\mathsf{T}\langle\!\langle f(C(x_1,\ldots,x_m),z_1,\ldots,z_n)\rangle\!\rangle]\!]_V \subseteq \lfloor [\![f(C(x_1,\ldots,x_m),z_1,\ldots,z_n)]\!]_M \rfloor$ by structural induction on the value of the first argument to f and extend f's model as in the proof of Theorem 5.1, exploiting the injectivity of constructors. $\square$

5.3 Axiomatization of (Co)inductive Predicates

With datatypes and recursion in place, we are ready to consider (co)inductive predicates. Recall from Sect. 3.2 that an inductive predicate is the least fixed point p of the equation $p(\bar{x}) \simeq t[p]$ (where $t[p]$ is some formula involving p) and a coinductive predicate is the greatest fixed point. A first intuition would be to take $p(\bar{x}) \simeq t[p]$ as p's definition. In general, this is unsound since it underspecifies p, but there are two important cases for which this method is sound.

First, if the recursion in $p(\bar{x}) \simeq t[p]$ is well-founded, the equation admits exactly one solution [17]; we can safely use it as p's specification, and encode it the same way as a recursive function (Sect. 5.2). To ascertain wellfoundedness, we can perform a simple syntactic check to ensure that each recursive call peels off at least one constructor. Alternatively, we can invoke an off-the-shelf termination prover such as AProVE [14] or Isabelle's *lexicographic_order* tactic [9]. Given introduction rules of the form

$$p(\bar{t}_{i1}) \wedge \cdots \wedge p(\bar{t}_{i\ell_i}) \wedge Q_i \longrightarrow p(\bar{u}_i)$$

for $i \in \{1,\ldots,n\}$, the prover attempts to exhibit a well-founded relation R such that

$$\textstyle\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\ell_i} \big(Q_i \longrightarrow \langle \bar{t}_{ij},\, \bar{u}_i \rangle \in R\big).$$

This is the approach implemented in Nitpick.

Second, if p is inductive and occurs negatively in the formula, we can replace these occurrences by a fresh constant q satisfying $q(\bar{x}) \simeq t[q]$. The resulting formula is equisatisfiable to the original formula: Since p is a least fixed point, q overapproximates p and thus $\neg\, q(\bar{x}) \Longrightarrow \neg\, p(\bar{x})$. Dually, this method can also handle positive occurrences of coinductive predicates.

To deal with positive occurrences of inductive predicates, we adapt a technique from bounded model checking [6]: We replace these occurrences of p by a fresh predicate r_k defined by the FOL equations

$$r_0(\bar{x}) \simeq \mathit{False} \qquad\qquad r_{\mathit{Suc}(m)}(\bar{x}) \simeq t[r_m],$$

which corresponds to p unrolled k times. In essence, we have made the predicate well-founded by introducing a counter that decreases by one with each recursive call. The above equations are primitive recursive over the datatype *nat* and can be translated using the approach shown in Sect. 5.2. The unrolling comes at a price: The function table for r_k is k times larger than that of p directly encoded as $p(\bar{x}) \simeq t[p]$.

The situation is mirrored for coinductive predicates: Negative occurrences are replaced by the overapproximation r_k defined by

$$r_0(\bar{x}) \simeq \mathit{True} \qquad\qquad r_{\mathit{Suc}(m)}(\bar{x}) \simeq t[r_m].$$

Example 5.5 The *even* predicate defined by

$$\begin{array}{l}\textsf{inductive } \mathit{even}^{\mathit{nat}\to o} \textsf{ where}\\ \mathit{even}(0)\\ \mathit{even}(n) \longrightarrow \mathit{even}(n)\\ \mathit{even}(n) \longrightarrow \mathit{even}(\mathit{Suc}(\mathit{Suc}(n)))\end{array}$$

is not well-founded because of the (needless) cyclic rule $\mathit{even}(n) \longrightarrow \mathit{even}(n)$. We can use the fixed-point equation

$$\mathit{even}(x) \simeq \big(\exists n.\ x \simeq 0 \vee (x \simeq n \wedge \mathit{even}(n)) \vee (x \simeq \mathit{Suc}(\mathit{Suc}(n)) \wedge \mathit{even}(n))\big)$$

as an overapproximation of *even* in negative contexts. In positive contexts, we must unroll the predicate:

$$\begin{aligned}\mathit{even}_0(x) &\simeq \mathit{False}\\ \mathit{even}_{\mathit{Suc}(m)}(x) &\simeq \big(\exists n.\ x \simeq 0 \vee (x \simeq n \wedge \mathit{even}_m(n)) \vee (x \simeq \mathit{Suc}(\mathit{Suc}(n)) \wedge \mathit{even}_m(n))\big). \quad \blacksquare\end{aligned}$$

Theorem 5.3 *The encoding of Sect. 5.2 extended with (co)inductive predicates is sound.*

Proof We consider only inductive predicates; coinduction is dual. If p is well-founded, the fixed-point equation fully characterizes p [17], and the proof is identical to that of primitive recursion in Theorem 5.2 but with recursion induction instead of structural induction. If p is not well-founded, $q \simeq t[q]$ is satisfied by several q's, and by Knaster–Tarski $p \sqsubseteq q$. Substituting q for p's negative occurrences in the FORL formula strengthens it, which is sound. For the positive occurrences, we have $r_0 \sqsubseteq \cdots \sqsubseteq r_k \sqsubseteq p$ by monotonicity of the definition; substituting r_k for p's positive occurrences strengthens the formula. $\square$

As an alternative to the explicit unrolling, we can mobilize FORL's transitive closure for an important class of inductive predicates, *linear inductive predicates*, whose introduction rules are of the form $Q \longrightarrow p(\bar{u})$ (the *base rules*) or $p(\bar{t}) \wedge Q \longrightarrow p(\bar{u})$ (the *step rules*). Informally, the idea is to replace positive occurrences of $p(\bar{x})$ with

$$\exists \bar{x}_0.\; p_{\text{base}}(\bar{x}_0) \wedge p^*_{\text{step}}(\bar{x}_0, \bar{x}),$$

where $p_{\text{base}}(\bar{x}_0)$ iff $p(\bar{x}_0)$ can be deduced from a base rule, $p_{\text{step}}(\bar{x}_0, \bar{x})$ iff $p(\bar{x})$ can be deduced by applying one step rule assuming $p(\bar{x}_0)$, and p^*_{step} is the reflexive transitive closure of p_{step}. For example, a reachability predicate $\textit{reach}(s)$ defined inductively would be coded as a set of initial states $\textit{reach}_{\text{base}}$ and the small-step transition relation $\textit{reach}_{\text{step}}$. The approach is not so different from explicit unrolling, since Kodkod internally unrolls the transitive closure to saturation. Nonetheless, on some problems the transitive closure approach is several times faster, presumably because Kodkod unfolds the relation inline instead of introducing an explicit counter.

5.4 Axiomatization of Coalgebraic Datatypes and Corecursive Functions

Coalgebraic datatypes are similar to algebraic datatypes, but they allow infinite values. For example, the infinite lists $[0,0,\ldots]$ and $[0,1,2,3,\ldots]$ are possible values of the type *nat llist* of coalgebraic (lazy) lists over natural numbers.

In principle, we could use the same SUA axiomatization for codatatypes as for datatypes (Sect. 5.2). This would exclude all infinite values but nonetheless be sound. However, in practice, infinite values often behave in surprising ways; excluding them would also exclude many interesting models.

It turns out we can modify the SUA axiomatization to support an important class of infinite values, namely those that are ω-regular. For lazy lists, this means lasso-shaped objects such as $[0,0,\ldots]$ and $[8,1,2,1,2,\ldots]$ (where the cycle $1,2$ is repeated infinitely).

The first step is to leave out the ACYCLIC axiom. However, doing only this is unsound, because we might obtain several atoms encoding the same value; for example, $\mathsf{a}_1 = \textit{LCons}(0, \mathsf{a}_1)$, $\mathsf{a}_2 = \textit{LCons}(0, \mathsf{a}_3)$, and $\mathsf{a}_3 = \textit{LCons}(0, \mathsf{a}_2)$ all encode the infinite list $[0,0,\ldots]$. This violates the bisimilarity principle, according to which two values are equal unless they lead to different observations (the observations being $0,0,\ldots$).

For lazy lists, we add the definition

$$\begin{array}{l}\textsf{coinductive} \sim^{(\alpha\ \textit{llist},\, \alpha\ \textit{llist}) \to o} \textsf{ where} \\ \textit{LNil} \sim \textit{LNil} \\ x \simeq x' \wedge xs \sim xs' \longrightarrow \textit{LCons}(x, xs) \sim \textit{LCons}(x', xs')\end{array}$$

and we require that $\simeq$ coincides with $\sim$ on α *llist* values. More generally, we generate mutual coinductive definitions of $\sim$ for all the codatatypes. For each constructor $C^{(\sigma_1,\ldots,\sigma_n)\to\sigma}$, we add an introduction rule

$$x_1 \approx_1 x'_1 \wedge \cdots \wedge x_n \approx_n x'_n \longrightarrow C(x_1,\ldots,x_n) \sim C(x'_1,\ldots,x'_n),$$

where $\approx_i$ is $\sim^{(\sigma_i,\sigma_i)\to o}$ if σ_i is a codatatype and $\simeq$ otherwise. Finally, for each codatatype κ, we add the axiom

$$\text{BISIMILAR:}\quad \forall y, y' \in \langle\!\langle \kappa \rangle\!\rangle\colon\ y \sim y' \longrightarrow y \simeq y'.$$

With the SUB (SU plus BISIMILAR) axiomatization in place, it is easy to encode coprimrec definitions. A corecursive equation $f(y_1^{\sigma_1},\ldots,y_n^{\sigma_1}) \simeq t$ is translated to

$$\forall y_1 \in \langle\!\langle \sigma_1 \rangle\!\rangle,\ldots,y_n \in \langle\!\langle \sigma_n \rangle\!\rangle\colon\ \mathsf{T}\langle\!\langle f(y_1,\ldots,y_n) \rangle\!\rangle \simeq \mathsf{T}\langle\!\langle t \rangle\!\rangle.$$

Theorem 5.4 *The encoding of Sect. 5.3 extended with coalgebraic datatypes and primitive corecursion is sound.*

Proof Codatatypes are characterized by selectors, which are axiomatized by the SU axioms, and by finality, which is equivalent to the bisimilarity principle [20, 30]. Our finite axiomatization gives a subterm-closed substructure of the coalgebraic datatype, which can be extended to yield a FOL model of the complete codatatype, as we did for algebraic datatypes in the proof of Theorem 5.2.

The soundness of the encoding of primitive corecursion is proved by coinduction. Given the equation $f(\bar{y}) \simeq t$, assuming that for each corecursive call $f(\bar{x})$ we have $[\![\mathsf{T}\langle\!\langle f(\bar{x}) \rangle\!\rangle]\!]_V \subseteq \lfloor [\![f(\bar{x})]\!]_M \rfloor$, we must show that $[\![\mathsf{T}\langle\!\langle f(\bar{y}) \rangle\!\rangle]\!]_V \subseteq \lfloor [\![f(\bar{y})]\!]_M \rfloor$. This follows from the soundness of the encoding of the constructs occurring in t and from the hypotheses. $\square$

6 Techniques for Improving Precision and Efficiency

6.1 Constructor Elimination

Since datatype constructors may return $\star$ in our encoding, we can increase precision by eliminating them. A formula such as $Cons(x, Cons(y, Nil)) \simeq Cons(a, Cons(b, Nil))$ can easily be rewritten into $x \simeq a \wedge y \simeq b$, which evaluates to either *True* or *False* if x, y, a, and b are representable, even if $Cons(x, Cons(y, Nil))$ or $Cons(a, Cons(b, Nil))$ would yield $\star$. By introducing discriminators and selectors, we can also rewrite equalities where only one side is expressed using constructors; for example, $xs \simeq Cons(a, Nil)$ would become $consp(xs) \wedge head(xs) \simeq a \wedge nilp(tail(xs))$.

6.2 Quantifier Massaging

The equations that encode (co)inductive predicates are marred by existential quantifiers, which blow up the size of the resulting propositional formula. The following steps, described in the Nitpick paper [8] but repeated here for completeness, can be applied to eliminate quantifiers or reduce their binding range:

1. Replace quantifications of the forms $\forall x.\ x \simeq t \longrightarrow P[x]$ and $\exists x.\ x \simeq t \wedge P[x]$ by $P[t]$ if x does not occur free in t.
2. Skolemize.
3. Distribute quantifiers over congenial connectives ($\forall$ over $\wedge$, $\exists$ over $\vee$ and $\longrightarrow$).
4. For any remaining subformula $Qx_1 \ldots x_n.\ p_1 \otimes \cdots \otimes p_m$, where Q is a quantifier and $\otimes$ is a connective, move the p_i's out of as many quantifiers as possible by rebuilding the formula using $qfy(\{x_1,\ldots,x_n\}, \{p_1,\ldots,p_m\})$, defined as

   $$qfy(\emptyset, P) = \bigotimes P \qquad qfy(x \uplus X, P) = qfy(X, P - P_x \cup \{Qx.\ \bigotimes P_x\}),$$

   where $P_x = \{p \in P \mid x \text{ occurs free in } p\}$.

The order in which individual variables x are removed from the first argument in step 4 is crucial because it affects which p_i's can be moved out. For clusters of up to 7 quantifiers, Nitpick considers all permutations of the bound variables and chooses the one that minimizes the sum $\sum_{i=1}^{m} |\langle\!\langle \sigma_{i1} \rangle\!\rangle| \cdot \ldots \cdot |\langle\!\langle \sigma_{ik_i} \rangle\!\rangle| \cdot \mathit{size}(p_i)$, where $\sigma_{i1}, \ldots, \sigma_{ik_i}$ are the types of the variables that have p_i in their binding range, and $\mathit{size}(p_i)$ is a rough syntactic measure of p_i's size; for larger clusters, it falls back on a greedy heuristic inspired by Paradox's clause splitting procedure [12]. Thus, the formula $\exists x^\alpha\, y^\alpha.\ p(x) \wedge q(x,y) \wedge r(y, f(y,y))$ is transformed into $\exists y^\alpha.\ r(y, f(y,y)) \wedge (\exists x^\alpha.\ p(x) \wedge q(x,y))$. Processing y before x in qfy would instead give $\exists x^\alpha.\ p(x) \wedge (\exists y^\alpha.\ q(x,y) \wedge r(y, f(y,y)))$, which is more expensive because $r(y, f(y,y))$, the most complex conjunct, is doubly quantified and hence $|\langle\!\langle \alpha \rangle\!\rangle|^2$ copies of it are needed in the resulting propositional formula. It could be argued that this optimization really belongs in Kodkod, but until it is implemented there we must do it ourselves.

6.3 Tabulation

FORL relations can be assigned fixed values, in which case no propositional variables are generated for them. We can use this facility to store tables that precompute the value of basic operations on natural numbers, such as *Suc*, $+$, $-$, $*$, *div*, *mod*, $<$, *gcd*, and *lcm*. This is possible for natural numbers because for any cardinality k there exists exactly one subterm-closed substructure $\{0, 1, \ldots, k-1\}$.

Example 6.1 If $\langle\!\langle \mathit{nat} \rangle\!\rangle = \{0,1,2,3,4\}$, we encode each representable number n as a_{n+1}, effectively performing our own symmetry breaking. We can then declare the successor function as follows:

$$\mathsf{var}\ \mathit{Suc} = \{\langle \mathsf{a}_1, \mathsf{a}_2 \rangle, \langle \mathsf{a}_2, \mathsf{a}_3 \rangle, \langle \mathsf{a}_3, \mathsf{a}_4 \rangle, \langle \mathsf{a}_4, \mathsf{a}_5 \rangle\} \quad \blacksquare$$

7 Case Studies

7.1 A Context-Free Grammar

Our first case study is taken from the Isabelle/HOL tutorial [29]. The following context-free grammar, originally due to Hopcroft and Ullman, produces all strings with an equal number of a's and b's:

$$S ::= \epsilon \mid bA \mid aB \qquad A ::= aS \mid bAA \qquad B ::= bS \mid aBB$$

The intuition behind the grammar is that A generates all string with one more a than b's and B generates all strings with one more b than a's.

Context-free grammars can easily be expressed as inductive predicates. The following FOL specification attempts to capture the above grammar, but a few errors were introduced to make it interesting.

$\mathsf{datatype}\ \Sigma = a \mid b$

$\mathsf{inductive}\ S^{\Sigma\ list \to bool}\ \mathsf{and}\ A^{\Sigma\ list \to bool}\ \mathsf{and}\ B^{\Sigma\ list \to bool}\ \mathsf{where}$

$$\begin{array}{ll}
S(\mathit{Nil}) & S(w) \longrightarrow A(\mathit{Cons}(a,w)) \\
A(w) \longrightarrow S(\mathit{Cons}(b,w)) & S(w) \longrightarrow S(\mathit{Cons}(b,w)) \\
B(w) \longrightarrow S(\mathit{Cons}(a,w)) & B(v) \wedge B(v) \longrightarrow B(\mathit{cat}(\mathit{Cons}(a,v),w))
\end{array}$$

Debugging faulty specifications is at the heart of Nitpick's *raison d'être*. A good approach is to state desirable properties of the specification—here, that the predicate S corresponds to the set (or predicate) of strings over $\{a,b\}$ with as many a's as b's—and check them with Nitpick. If the properties are correctly stated, counterexamples will point to bugs in the specification. For our grammar example, we will proceed in two steps, separating the soundness and the completeness of the set S:

$$\begin{aligned}\textsc{Sound}\colon&\ \ S(w) \longrightarrow \mathit{count}(a,w) \simeq \mathit{count}(b,w)\\ \textsc{Complete}\colon&\ \ \mathit{count}(a,w) \simeq \mathit{count}(b,w) \longrightarrow S(w).\end{aligned}$$

The auxiliary function *count* is defined as follows:

$$\begin{array}{l}\textsf{primrec}\ \mathit{count}^{(\alpha,\alpha\ \mathit{list})\to \mathit{nat}}\ \textsf{where}\\ \mathit{count}(x,\mathit{Nil}) \simeq 0\\ \mathit{count}(x,\ \mathit{Cons}(y,ys)) \simeq (\mathit{if}\ x \simeq y\ \mathit{then}\ 1\ \mathit{else}\ 0) + \mathit{count}(x,\ ys)\end{array}$$

We first focus on soundness. The predicate S occurs negatively in SOUND, but positively in the negated conjecture $\neg$ SOUND. Wellfoundedness is easy to establish because the words in the conclusions are always at least one symbol longer than the corresponding words in the assumptions. As a result, Nitpick can use the fixed-point equations

$$\begin{aligned}S(x) &\simeq \big(x \simeq \mathit{Nil} \lor (\exists w.\ x \simeq \mathit{Cons}(b,w) \land A(w)) \lor (\exists w.\ x \simeq \mathit{Cons}(a,w) \land B(w))\\ &\qquad \lor (\exists w.\ x \simeq \mathit{Cons}(b,w) \land S(w))\big)\\ A(x) &\simeq \big(\exists w.\ x \simeq \mathit{Cons}(a,w) \land S(w)\big)\\ B(x) &\simeq \big(\exists v\,w.\ x \simeq \mathit{cat}(\mathit{Cons}(a,v),w) \land B(v) \land B(v)\big),\end{aligned}$$

which can be syntactically derived from the introduction rules.

When invoked on SOUND with the default settings, Nitpick produces 10 FORL problems corresponding to the scopes $|\langle\!\langle \mathit{nat}\rangle\!\rangle| = |\langle\!\langle \Sigma\ \mathit{list}\rangle\!\rangle| = k$ and $|\langle\!\langle \Sigma\rangle\!\rangle| = \min\,\{k,2\}$ for $k \in \{1,\ldots,10\}$ and passes them on to Kodkod. Datatypes approximated by subterm-closed substructures are always scope-monotonic, so it would be sufficient to try only the largest scope ($k = 10$), but in practice it is usually more efficient to start with smaller scopes. The models obtained this way also tend to be simpler.

Nitpick almost instantly finds the counterexample $w = [b]$ built using the substructures

$$\langle\!\langle \mathit{nat}\rangle\!\rangle = \{0,\,1\} \qquad\qquad \langle\!\langle \Sigma\ \mathit{list}\rangle\!\rangle = \{[],\,[b]\} \qquad\qquad \langle\!\langle \Sigma\rangle\!\rangle = \{a,\,b\}$$

and the constant interpretations

$$\begin{array}{rrrrr}\mathit{cat}([],[]) = [] & \mathit{count}(a,[]) = 0 & S([]) = \mathit{True} & A([]) = \mathit{False} & B([]) = \star\\ \mathit{cat}([b],[]) = \star & \mathit{count}(b,[]) = 0 & S([b]) = \mathit{True} & A([b]) = \mathit{False} & B([b]) = \star.\\ \mathit{cat}([],[b]) = [b] & \mathit{count}(a,[b]) = 0 & & & \\ \mathit{cat}([b],[b]) = \star & \mathit{count}(b,[b]) = 1 & & & \end{array}$$

It would seem that $S([b])$. How could this be? An inspection of the introduction rules reveals that the only rule with a right-hand side of the form $S(\mathit{Cons}(b,\ldots))$ that could have introduced $[b]$ into S is

$$S(w) \longrightarrow S(\mathit{Cons}(b,w)).$$

This rule is clearly wrong: To match the production $B ::= bS$, the second S should be a B. If we fix the typo and run Nitpick again, we now obtain the counterexample $w = [a,a,b]$, which requires $k = 4$. This takes about 1.5 seconds on the author's laptop.

Some detective work is necessary to find out what went wrong here. To get $S([a,a,b])$, we need $B([a,b])$, which in turn can only originate from

$$B(v) \wedge B(v) \longrightarrow B(cat(Cons(a,v),w)).$$

This introduction rule is highly suspicious: The same assumption occurs twice, and the variable w is unconstrained. Indeed, one of the two occurrences of v in the assumptions should have been a w.

With the correction made, we do not get any counterexample from Nitpick, which exhausts all scopes up to cardinality 10 well within the 30 second time limit. Let us move on and check completeness. Since the predicate S occurs negatively in the negated conjecture $\neg$ COMPLETE, Nitpick can safely use the fixed-point equations for S, A, and B as their specifications. This time we get the counterexample $w = [b,b,a,a]$, with $k = 5$.

Apparently, $[b,b,a,a]$ is not in S even though it has the same numbers of a's and b's. But since our inductive definition passed the soundness check, our introduction rules are likely to be correct. Perhaps we simply lack a rule. Comparing the grammar with the inductive definition, our suspicion is confirmed: There is no introduction rule corresponding to the production $A ::= bAA$, without which the grammar cannot generate two or more b's in a row. So we add the rule

$$A(v) \wedge A(w) \longrightarrow A(cat(Cons(b,v),w)).$$

With this last change, we do not get any counterexamples from Nitpick for either soundness or completeness. We can even generalize our result to cover A and B as well:

$$\begin{aligned}
S(w) &\simeq (count(a,w) \simeq count(b,w)) \\
A(w) &\simeq (count(a,w) \simeq count(b,w)+1) \\
B(w) &\simeq (count(a,w)+1 \simeq count(b,w)).
\end{aligned}$$

Nitpick can test these formulas up to cardinality 10 within 30 seconds on the author's laptop.

With some manual setup, the latest version of Quickcheck [4], a counterexample generator based on random testing, can find the same counterexamples as Nitpick. On the other hand, the SAT-based Refute [36] fails here, mostly because of its very rudimentary support for inductive predicates.

7.2 AA Trees

AA trees are a variety of balanced trees discovered by Arne Andersson that provide similar performance to red-black trees but are easier to implement [2]. They can be used to store sets of elements of type α equipped with a total order $<$. We start by defining the datatype and some basic extractor functions:

$$\textsf{datatype}\ \alpha\ aa\_tree\ =\ \Lambda \mid N\ \textsf{of}\ (\alpha,\ nat,\ \alpha\ aa\_tree,\ \alpha\ aa\_tree)$$

$\textsf{primrec}\ level^{\alpha\ aa\_tree \to nat}\ \textsf{where}$
$level(\Lambda) \simeq 0 \mid$
$level(N(\_,k,\_,\_)) \simeq k$

$\textsf{primrec}\ data^{\alpha\ aa\_tree \to \alpha}\ \textsf{where}$
$data(N(x,\_,\_,\_)) \simeq x$

$\textsf{primrec}\ is\_in^{(\alpha,\ \alpha\ aa\_tree) \to o}\ \textsf{where}$
$is\_in(\_,\Lambda) \simeq False \mid$
$is\_in(a, N(x,\_,t,u)) \simeq \big(a \simeq x \vee is\_in(a,t) \vee is\_in(a,u)\big)$

$\textsf{primrec}\ lf^{\alpha\ aa\_tree \to \alpha\ aa\_tree}\ \textsf{where}$
$lf(\Lambda) \simeq \Lambda \mid$
$lf(N(\_,\_,t,\_)) \simeq t$

$\textsf{primrec}\ rt^{\alpha\ aa\_tree \to \alpha\ aa\_tree}\ \textsf{where}$
$rt(\Lambda) \simeq \Lambda \mid$
$rt(N(\_,\_,\_,u)) \simeq u$

The wellformedness criterion for AA trees is fairly complex. Each node is equipped with a "level" field, which must satisfy the following constraints:

- Nil trees (Λ) have level 0.
- Leaf nodes (i.e., nodes of the form $N(\_,\_,\Lambda,\Lambda)$) have level 1.
- A node's level must be at least as large as its right child's, and greater than its left child's and its grandchildren's.
- Every node of level greater than 1 must have two children.

The *wf* predicate formalizes this description:

```
primrec $wf^{\alpha\ aa\_tree \to o}$ where
$wf(\Lambda) \simeq True \mid$
$wf(N(\_,k,t,u)) \simeq$
      (if $t \simeq \Lambda$ then
          $k \simeq 1 \wedge (u \simeq \Lambda \vee (level(u) \simeq 1 \wedge lf(u) \simeq \Lambda \wedge rt(u) \simeq \Lambda))$
       else
          $wf(t) \wedge wf(u) \wedge u \not\simeq \Lambda \wedge level(t) < k \wedge level(u) \le k \wedge level(rt(u)) < k)$
```

Rebalancing the tree upon insertion and removal of elements is performed by two auxiliary functions called *skew* and *split*, defined below:

```
primrec $skew^{\alpha\ aa\_tree \to \alpha\ aa\_tree}$ where
$skew(\Lambda) = \Lambda$
$skew(N(x,k,t,u)) =$
(if $t \neq \Lambda \wedge k = level(t)$ then
    $N(data(t),k,lf(t),N(x,k,rt(t),u))$
 else
    $N(x,k,t,u)$)
```

```
primrec $split^{\alpha\ aa\_tree \to \alpha\ aa\_tree}$ where
$split(\Lambda) = \Lambda$
$split(N(x,k,t,u)) =$
(if $u \neq \Lambda \wedge k = level(rt(u))$ then
    $N(data(u),Suc(k),N(x,k,t,lf(u)),rt(u))$
 else
    $N(x,k,t,u)$)
```

Performing a *skew* or a *split* should have no impact on the set of elements stored in the tree:

$$is\_in(a,skew(t)) \simeq is\_in(a,t) \qquad\qquad is\_in(a,split(t)) \simeq is\_in(a,t).$$

Furthermore, applying *skew* or *split* on a well-formed tree should not alter the tree:

$$wf(t) \longrightarrow skew(t) \simeq t \qquad\qquad wf(t) \longrightarrow split(t) \simeq t.$$

All these properties can be checked up to cardinality 7 or 8 by Nitpick, within the default time limit of 30 seconds.

Insertion is implemented recursively. It preserves the sort order:

```
primrec $insort^{(\alpha\ aa\_tree, \alpha) \to \alpha\ aa\_tree}$ where
$insort(\Lambda, x) \simeq N(x, 1, \Lambda, \Lambda)$
$insort(N(y, k, t, u), x) \simeq$
  $\text{~~split~~}(\text{~~skew~~}(N(y,\ k,\ \text{if } x < y \text{ then } insort(t, x) \text{ else } t,\ \text{if } x > y \text{ then } insort(u, x) \text{ else } u)))$
```

If we test the property

$$wf(t) \longrightarrow wf(insort(t, x)),$$

with the applications of *skew* and *split* commented out (as suggested by the notations ~~*split*~~ and ~~*skew*~~), we get the counterexample $t = N(a_1, 1, \Lambda, \Lambda)$ and $x = a_2$. It's hard to see why this is a counterexample without looking up the definition of $<$ on type α. To improve readability, we will restrict the theorem to *nat* and tell Nitpick to display the value of $insort(t, x)$. The counterexample is now $t = N(1, 1, \Lambda, \Lambda)$ and $x = 0$, with $insort(t, x) = N(1, 1, N(0, 1, \Lambda, \Lambda), \Lambda)$. The output reveals that the element 0 was added as a left child of 1, where both nodes have a level of 1. This violates the AA tree invariant, which requires that a left child's level must be less than its parent's. This should not come as a surprise, considering that we commented out the tree rebalancing code. If we reintroduce the code, Nitpick finds no counterexample up to cardinality 7.

As in the context-free grammar case study, Quickcheck can find the same counterexamples as Nitpick, whereas Refute fails.

7.3 Lazy Lists

The codatatype α *llist* of lazy lists [30] is generated by the constructors $LNil^{\alpha\ llist}$ and $LCons^{(\alpha,\ \alpha\ llist) \to \alpha\ llist}$. It is of particular interest to countermodel finding because many basic properties of finite lists do not carry over to infinite lists, often in baffling ways. To illustrate this, we conjecture that appending *ys* to *xs* yields *xs* iff *ys* is *LNil*:

$$(lcat(xs, ys) \simeq xs) \simeq (ys \simeq LNil).$$

The function *lcat* is defined corecursively in Sect. 3.4. Nitpick immediately finds the countermodel $xs = ys = [0, 0, \ldots]$, in which a cardinality of 1 is sufficient for α and α *llist*, and the bisimilarity predicate $\sim$ is unrolled only once. Indeed, appending $[0, 0, \ldots] \neq []$ to $[0, 0, \ldots]$ leaves $[0, 0, \ldots]$ unchanged. Many other counterexamples are possible—for example, $xs = [0, 0, \ldots]$ and $ys = [1]$—but Nitpick tends to reuse the objects that are part of its subterm-closed substructures and keep cardinalities low. Although very simple, the counterexample is outside Quickcheck's and Refute's reach, since they do not support codatatypes.

The next example requires the following lexicographic order predicate:

```
coinductive $\preceq^{(nat\ llist,\ nat\ llist) \to o}$ where
$LNil \preceq xs$
$x \leq y \longrightarrow LCons(x, xs) \preceq LCons(y, ys)$
$xs \preceq ys \longrightarrow LCons(x, xs) \preceq LCons(x, ys)$
```

The intention of this definition is to define a linear order on lazy lists of natural numbers, and hence the following properties should hold:

$$\begin{array}{rlrl} \text{REFL:} & xs \preceq xs & \text{ANTISYM:} & xs \preceq ys \wedge ys \preceq xs \longrightarrow xs \simeq ys \\ \text{LINEAR:} & xs \preceq ys \vee ys \preceq xs & \text{TRANS:} & xs \preceq ys \wedge ys \preceq zs \longrightarrow xs \preceq zs. \end{array}$$

However, Nitpick finds a counterexample for ANTISYM: $xs = [1,1]$ and $ys = [1]$. On closer inspection, the assumption $x \leq y$ of the second introduction rule for $\preceq$ should have been $x < y$; otherwise, any two lists xs, ys with the same head satisfy $xs \preceq ys$. Once we repair the specification, no more counterexamples are found for the four properties up to cardinality 6 for *nat* and *nat llist*, within the time limit of 30 seconds. This is a strong indication that the properties hold. Andreas Lochbihler used Isabelle to prove all four properties [27].

We could continue like this and sketch a complete theory of lazy lists. Once the definitions and main theorems are in place and have been thoroughly tested using Nitpick, we could start working on the proofs. Developing theories this way usually saves time, because faulty theorems and definitions are discovered much earlier in the process.

8 Related Work

The encoding of algebraic datatypes in FORL has been studied by Kuncak and Jackson [25] and Dunets et al. [13]. Kuncak and Jackson focused on lists and trees. Dunets et al. showed how to handle primitive recursion; their approach to recursion is similar to ours, but the use of a two-valued logic compelled them to generate additional definedness guards. The unrolling of inductive predicates was inspired by bounded model checking [6] and by the Alloy idiom for state transition systems [19, pp. 172–175].

Another inspiration has been Weber's higher-order model finder Refute [36]. It uses a three-valued logic, but sacrifices soundness for precision. Datatypes are approximated by subterm-closed substructures [36, pp. 58–64] that contain *all* datatype values built using up to k nested constructors. This scheme proved disadvantageous in practice, because it generally requires higher cardinalities to obtain the same models as with Kuncak and Jackson's approach. Weber handled (co)inductive predicates by expanding their HOL definition, which in practice does not scale beyond a cardinality of 3 for the predicate's domain because of the higher-order quantifier.

The Nitpick tool, which implements the techniques presented here, is described in a separate paper [8] that covers the handling of higher-order quantification and functions. The paper also presents an evaluation of the tool on various Isabelle/HOL theories, where it competes against Refute and Quickcheck [4], as well as two case studies.

9 Conclusion

Despite recent advances in lightweight formal methods, there remains a wide gap between specification languages that lend themselves to automatic analysis and those that are used in actual formalizations. As an example, infinite types are ubiquitous, yet most model finders either spin forever [12, 28], give up immediately [11], or are unsound [1; 33, p. 164; 36] on finitely unsatisfiable formulas.

We identified several commonly used definitional principles and showed how to encode them in first-order relational logic (FORL), the logic supported by the Kodkod model finder and the Alloy Analyzer. Our main contribution has been to develop three ways to translate (co)inductive predicates to FORL, based on wellfoundedness, polarity, and linearity. Other contributions have been to formulate an axiomatization of coalgebraic datatypes that caters for infinite (ω-regular) values and to devise a procedure that computes the acyclicity axiom for mutually recursive datatypes.

Counterexample generators encourages a style of theory development where users start by stating their definitions and main theorems and testing these thoroughly before working on the proofs. Developing theories this way usually saves time, because faulty conjectures and definitions are discovered much earlier in the process. Our experience with the counterexample generator Nitpick has shown that the techniques scale to handle real-world specifications, including a security type system and a hotel key card system [8]. A user reported saving several hours of failed proof attempts thanks to Nitpick's support for codatatypes and coinductive predicates while developing a formal theory of infinite process traces [26].

Acknowledgment I want to thank Sascha Böhme, Lukas Bulwahn, Andreas Lochbihler, Tobias Nipkow, Mark Summerfield, and the anonymous reviewers for suggesting many improvements to this article, and Alexander Krauss for helping to structure it.

References

1. W. Ahrendt. Deductive search for errors in free data type specifications using model generation. In A. Voronkov, editor, *CADE-18*, volume 2392 of *LNAI*, pages 211–225. Springer, 2002.
2. A. Andersson. Balanced search trees made simple. In F. K. H. A. Dehne, N. Santoro, and S. Whitesides, editors, *WADS 1993*, volume 709 of *LNCS*, pages 61–70. Springer, 1993.
3. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *FASE 2000*, number 1783 in LNCS. Springer, 2000.
4. S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, editors, *SEFM 2004*, pages 230–239. IEEE C.S., 2004.
5. S. Berghofer and M. Wenzel. Inductive datatypes in HOL—lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *TPHOLs '99*, volume 1690 of *LNCS*, pages 19–36, 1999.
6. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *TACAS '99*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
7. J. C. Blanchette and A. Krauss. Monotonicity inference for higher-order formulas. In J. Giesl and R. Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNCS*, pages 91–106. Springer, 2010.
8. J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. Paulson, editors, *ITP-10*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.
9. L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 38–53. Springer, 2007.
10. A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5:56–68, 1940.
11. K. Claessen and A. Lillieström. Automated inference of finite unsatisfiability. In R. A. Schmidt, editor, *CADE-22*, volume 5663 of *LNAI*, pages 388–403. Springer, 2009.
12. K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *MODEL*, 2003.
13. A. Dunets, G. Schellhorn, and W. Reif. Bounded relational analysis of free datatypes. In B. Beckert and R. Hähnle, editors, *TAP 2008*, volume 4966 of *LNCS*, pages 99–115. Springer, 2008.
14. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *IJCAR 2006*, volume 4130 of *LNAI*, pages 281–286, 2006.
15. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
16. E. L. Gunter. Why we can't have SML-style datatype declarations in HOL. In L. J. M. Claesen and M. J. C. Gordon, editors, *TPHOLs 1992*, IFIP Transactions, pages 561–568. North-Holland/Elsevier, 1993.

17. J. Harrison. Inductive definitions: Automation and application. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *TPHOLs 1995*, volume 971 of *LNCS*, pages 200–213. Springer, 1995.
18. J. Harrison. HOL Light: A tutorial introduction. In *FMCAD '96*, volume 1166 of *LNCS*, pages 265–269. Springer, 1996.
19. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
20. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bull. EATCS*, 62:222–259, 1997.
21. S. C. Kleene. On notation for ordinal numbers. *J. Symb. Log.*, 3(4):150–155, 1938.
22. S. C. Kleene. Representation of events in nerve nets and finite automata. In J. McCarthy and C. Shannon, editors, *Automata Studies*, pages 3–42. Princeton University Press, 1956.
23. D. C. Kozen. *Automata and Computability*. Undergrad. Texts in C.S. Springer, 1997.
24. A. Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Auto. Reas.*, 44(4):303–336, 2009.
25. V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. In H. C. Gall, editor, *ESEC/FSE 2005*, 2005.
26. A. Lochbihler. Private communication, 2009.
27. A. Lochbihler. Coinduction. In G. Klein, T. Nipkow, and L. C. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sourceforge.net/entries/Coinductive.shtml`, Feb. 2010.
28. W. McCune. A Davis–Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, ANL, 1994.
29. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
30. L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *CADE-12*, volume 814 of *LNAI*, pages 148–161. Springer, 1994.
31. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
32. T. Ramananandro. Mondex, an electronic purse: Specification and refinement checks with the Alloy model-finding method. *Formal Asp. Comput.*, 20(1):21–39, 2008.
33. J. M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, 2001.
34. K. Slind and M. Norrish. A brief overview of HOL4. In O. A. Mohamed, C. M. noz, and S. Tahar, editors, *TPHOLs 2008*, volume 5170 of *LNCS*, pages 28–32, 2008.
35. E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS 2007*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.
36. T. Weber. *SAT-Based Finite Model Generation for Higher-Order Logic*. Ph.D. thesis, Dept. of Informatics, T.U. München, 2008.
37. M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *TPHOLs 1997*, volume 1275 of *LNCS*, pages 307–322. Springer, 1997.