Turning inductive into equational specifications

Stefan Berghofer* and Lukas Bulwahn and Florian Haftmann**

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany
http://www.in.tum.de/~berghofe/
http://www.in.tum.de/~bulwahn/
http://www.in.tum.de/~haftmann/

Abstract. Inductively defined predicates are frequently used in formal specifications. Using the theorem prover *Isabelle*, we describe an approach to turn a class of systems of inductively defined predicates into a system of equations using data flow analysis; the translation is carried out *inside* the logic and resulting equations can be turned into functional program code in *SML*, *OCaml* or *Haskell* using the existing code generator of *Isabelle*. Thus we extend the scope of code generation in *Isabelle* from functional to functional-logic programs while leaving the trusted foundations of code generation itself intact.

1 Introduction

Inductively defined predicates (for short, *(inductive) predicates*) are a popular specification device in the theorem proving community. Major theory developments in the proof assistant *Isabelle/HOL* [8] make pervasive use of them, e.g. formal semantics of realistic programming language fragments [11]. From such large applications naturally the desire arises to generate *executable* prototypes from the abstract specifications. It is well-known how systems of predicates can be transformed to functional programs using *mode analysis*. The approach described in [1] for *Isabelle/HOL* works but has turned out unsatisfactory:

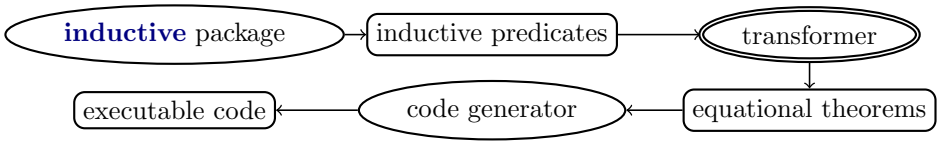
- The applied transformations are not trivial but are carried out outside the *LCF* inference kernel, thus relying on a large code base to be trusted.
- Recently a lot of code generation facilities in *Isabelle/HOL* have been generalized to cover type classes and more languages than *ML*, but this has not yet been undertaken for predicates.

In our view it is high time to tackle execution of predicates again; we present a transformation from predicates to function-like equations that is not a mere re-implementation, but brings substantial improvements:

* Supported by BMBF in the VerisoftXT project under grant 01 IS 07008 F
** Supported by DFG project NI 491/10-1.

– The transformation is carried out *inside* the logic; thus the transformation is guarded by *LCF* inferences and does not increase the trusted code base.
– The code generator itself can be fed with the function-like equations and does not need to be extended; also other tools involving equational reasoning could benefit from the transformation.
– Proposed extensions can also work *inside* the logic and do not endanger trustability.

The role of our transformation in this scenario is shown in the following picture:



The remainder of this paper is structured as follows: we briefly review existing work in §2 and explain the preliminaries in §3. The main section (§4) explains how the translation works, followed by a discussion of further extensions (§5). Our conclusion (§6) will deal with future work.

In our presentation we use fairly standard notation, plus little *Isabelle/HOL*-specific concrete syntax.

2 Related work

From the technical point of view, the execution of predicates has been extensively studied in the context of the programming languages *Curry* [4] and *Mercury* [10]. The central concept for executing predicates are *modes*, which describe dataflow by partitioning arguments into *input* and *output*.

We already mentioned the state-of-the-art implementation of code generation for predicates in *Isabelle/HOL* [1] which turns inductive predicates into *ML* programs extralogically using mode analysis.

Delahaye et al. provide a similar direct extraction for the *Coq* proof assistant [2]; however at most one solution is computed, multiple solutions are not enumerated.

For each of these approaches, correctness is ensured by pen-and-paper proofs. Our approach instead *animates* the correctness proof by applying it to each single predicate using the proof assistant itself; thus correctness is guaranteed by construction.

3 Preliminaries

3.1 Inductive predicates

An inductive predicate is characterized by a collection of *introduction rules* (or *clauses*), each of which has a *conclusion* and an arbitrary number of *premises*.

It corresponds to the *smallest* set closed under these clauses. As an example, consider the following predicate describing the concatenation of two lists, which can be defined in in *Isabelle/HOL* using the **inductive** command:

$$\begin{array}{l}\textbf{inductive}\ \mathit{append} :: \alpha\ \mathit{list} \Rightarrow \alpha\ \mathit{list} \Rightarrow \alpha\ \mathit{list} \Rightarrow \mathit{bool}\ \textbf{where}\\ \quad \mathit{append}\ []\ \mathit{ys}\ \mathit{ys}\\ \mid \mathit{append}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs} \Longrightarrow \mathit{append}\ (x \cdot \mathit{xs})\ \mathit{ys}\ (x \cdot \mathit{zs})\end{array}$$

For each predicate, an *elimination* (or *case analysis*) rule is provided, which for *append* has the form

$$\begin{array}{l}\mathit{append}\ \mathit{Xs}\ \mathit{Ys}\ \mathit{Zs} \Longrightarrow\\ (\bigwedge \mathit{ys}.\ \mathit{Xs} = [] \Longrightarrow \mathit{Ys} = \mathit{ys} \Longrightarrow \mathit{Zs} = \mathit{ys} \Longrightarrow P) \Longrightarrow\\ (\bigwedge \mathit{xs}\ \mathit{ys}\ \mathit{zs}\ x.\\ \quad \mathit{Xs} = x \cdot \mathit{xs} \Longrightarrow\\ \quad \mathit{Ys} = \mathit{ys} \Longrightarrow \mathit{Zs} = x \cdot \mathit{zs} \Longrightarrow \mathit{append}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs} \Longrightarrow P) \Longrightarrow\\ P\end{array}$$

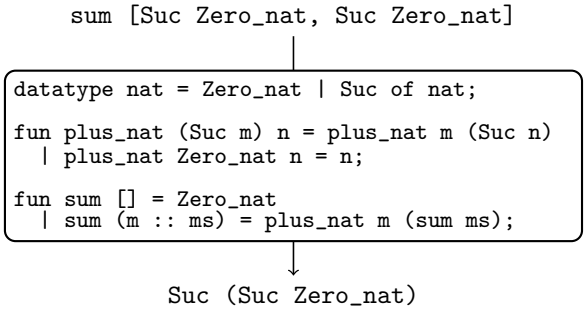
There is also an *induction* rule, which however is not relevant in our scenario. In introduction rules, we distinguish between premises of the form $Q\ u_1 \ldots u_k$, where Q is an inductive predicate, and premises of other shapes, which we call *side conditions*. Without loss of generality, we only consider clauses without side conditions in most parts of our presentation. The general form of a clause is

$$C_i:\ Q_{i,1}\ \overline{u}_{i,1} \Longrightarrow \cdots \Longrightarrow Q_{i,n_i}\ \overline{u}_{i,n_i} \Longrightarrow P\ \overline{t}_i$$

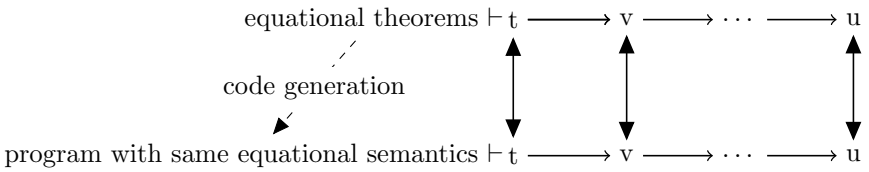
We use $k_{i,j}$ and l to denote the *arities* of the predicates $Q_{i,j}$ and P, i.e. the length of the argument lists $\overline{u}_{i,j}$ and $\overline{t}_i$, respectively.

3.2 Code generation

The *Isabelle* code generator views generated programs as an implementation of an equational rewrite system, e.g. the following program normalizes a list of natural numbers to its sum by equational rewriting:



The code generator turns a set of equational theorems into a program inducing the *same* equational rewrite system. This means that any sequence of reduction steps the generated program performs on a term can be simulated in the logic:



This guarantees partial correctness [3]. As a further consequence only program statements which contribute to a program's equational semantics (e.g. `fun` in ML) are correctness-critical, whereas others are not. For example, the constructors of a `datatype` in ML need only meet the syntactic characteristics of a datatype, but *not* the usual logical properties of a HOL datatype such as injectivity. This gives us some freedom in choosing datatype constructors which we will employ in §4.2.

4 Transforming clauses to equations

4.1 Mode analysis

In order to execute a predicate P, its arguments are classified as *input* or *output*. For example, all three arguments of *append* could be input, meaning that the predicate just checks whether the third list is the concatenation of the first and second list. Another possibility would be to consider only the first and second argument as input, while the third one is output. In this case, the predicate actually *computes* the concatenation of the two input lists. Yet another way of using *append* would be to consider the third argument as input, while the first two arguments are output. This means that the predicate enumerates all possible ways of splitting the input list into two parts. This notion of *dataflow* is made explicit by means of *modes* [6].

Modes. For a predicate P with k arguments, we denote a particular dataflow assignment by a *mode* which is a set $M \subseteq \{1, \ldots, k\}$ such that M is exactly the set of all parameter position numbers denoting *input* parameters. A *mode assignment* for a given clause

$$Q_{i,1}\ \overline{u}_{i,1} \Longrightarrow \cdots \Longrightarrow Q_{i,n_i}\ \overline{u}_{i,n_i} \Longrightarrow P\ \overline{t}_i$$

is a list of modes $M, M_{i,1}, \ldots M_{i,n_i}$ for the predicates $P, Q_{i,1}, \ldots, Q_{i,n_i}$, where $1 \leq i \leq m$, $M \subseteq \{1, \ldots, l\}$ and $Q_{i,j} \subseteq \{1, \ldots, k_{i,j}\}$. Let $FV(t)$ denote the set of free variables in a term t. Given a vector of arguments $\overline{t}$ and a mode M, the projection expression $\overline{t}\langle M \rangle$ denotes the list of all arguments in $\overline{t}$ (in the order of their occurrence) whose index is in M.

Mode consistency. Given a clause

$$Q_{i,1}\ \overline{u}_{i,1} \Longrightarrow \cdots \Longrightarrow Q_{i,n_i}\ \overline{u}_{i,n_i} \Longrightarrow P\ \overline{t}_i$$

a corresponding mode assignment $M, M_{i,1}, \ldots M_{i,n_i}$ is *consistent* if there exists a chain of sets $v_0 \subseteq \cdots \subseteq v_n$ of variables generated by

1. $v_0 = FV(\overline{t}_i\langle M\rangle)$
2. $v_j = v_{j-1} \cup FV(\overline{u}_{i,j})$

such that

3. $FV(\overline{u}_{i,j}\langle M_{i,j}\rangle) \subseteq v_{j-1}$
4. $FV(\overline{t}_i) \subseteq v_n$

Consistency models the possibility of a sequential evaluation of premises in a given order, where v_j represents the known variables after the evaluation of the j-th premise:

1. initially, all variables in input arguments of P are known
2. after evaluation of the j-th premise, the set of known variables is extended by all variables in the arguments of $Q_{i,j}$,
3. when evaluating the j-th premise, all variables in the arguments of $Q_{i,j}$ have to be known,
4. finally, all variables in the arguments of P must be contained in the set of known variables.

Without loss of generality we can examine clauses under mode inference modulo reordering of premises. For side conditions R, condition 3 has to be replaced by $FV(R) \subseteq v_{j-1}$, i.e. all variables in R must be known when evaluating it. This definition yields a check whether a given clause is consistent with a particular mode assignment.

4.2 Enumerating output arguments of predicates

A predicate of type $\alpha \Rightarrow \mathit{bool}$ is isomorphic to a set over type α; executing inductive predicates means to *enumerate* elements of the corresponding set. For this purpose we use an abstract algebra of primitive operations on such predicate enumerations. To establish an abstraction, we first define an explicit type to represent predicates:

datatype $\alpha\ \mathit{pred} = \mathit{pred}\ (\alpha \Rightarrow \mathit{bool})$

with a projection operator $\mathit{eval} :: \alpha\ \mathit{pred} \Rightarrow \alpha \Rightarrow \mathit{bool}$ satisfying

$$\mathit{eval}\ (\mathit{pred}\ f) = f$$

We provide four further abstract operations on $\alpha\ \mathit{pred}$:

– $\bot :: \alpha\ pred$ is the empty enumeration.
– $single :: \alpha \Rightarrow \alpha\ pred$ is the singleton enumeration.
– $(\gg\!=) :: \alpha\ pred \Rightarrow (\alpha \Rightarrow \beta\ pred) \Rightarrow \beta\ pred$ applies a function to every element of an enumeration which itself returns an enumeration and flattens all resulting enumerations.
– $(\sqcup) :: \alpha\ pred \Rightarrow \alpha\ pred \Rightarrow \alpha\ pred$ forms the union of two enumerations.

These abstract operations, which form a *plus monad*, are used to build up the code equations of predicates (§4.3). Table 1 contains their definitions and relates them to their counterparts on sets. In order to equip these abstract operations with an executable model, we introduce an auxiliary datatype:

datatype $\alpha\ seq = Empty \mid Insert\ \alpha\ (\alpha\ pred) \mid Union\ (\alpha\ pred\ list)$

Values of type $\alpha\ seq$ are embedded into type $\alpha\ pred$ by defining:

$Seq :: (unit \Rightarrow \alpha\ seq) \Rightarrow \alpha\ pred$
$Seq\ f =$
$(case\ f\ ()\ of\ Empty \Rightarrow \bot \mid Insert\ x\ xq \Rightarrow single\ x \sqcup xq$
$\mid Union\ xqs \Rightarrow \bigsqcup xqs)$

where $\bigsqcup :: \alpha\ pred\ list \Rightarrow \alpha\ pred$ flattens a list of predicates into one predicate. *Seq* will serve as datatype constructor for type $\alpha\ pred$; on top of this, we prove the following code equations for our $\alpha\ pred$ algebra:

$\bot = Seq\ (\lambda u.\ Empty)$

$single\ x = Seq\ (\lambda u.\ Insert\ x\ \bot)$

$Seq\ g \gg\!= f =$
$Seq\ (\lambda u.\ case\ g\ ()\ of\ Empty \Rightarrow Empty$
$\quad \mid Insert\ x\ xq \Rightarrow Union\ [f\ x,\ xq \gg\!= f]$
$\quad \mid Union\ xqs \Rightarrow Union\ (map\ (\lambda x.\ x \gg\!= f)\ xqs))$

$Seq\ f \sqcup Seq\ g =$
$Seq\ (\lambda u.\ case\ f\ ()\ of\ Empty \Rightarrow g\ ()$
$\quad \mid Insert\ x\ xq \Rightarrow Insert\ x\ (xq \sqcup Seq\ g)$
$\quad \mid Union\ xqs \Rightarrow Union\ (xqs\ @\ [Seq\ g]))$

eval	$eval\ (pred\ P)\ x \longleftrightarrow P\ x$	$x \in P$
$\bot$	$\bot = pred\ (\lambda x.\ False)$	$\{\}$
$single\ x$	$single\ x = pred\ (\lambda y.\ y = x)$	$\{x\}$
$P \gg\!= f$	$P \gg\!= f = pred\ (\lambda x.\ \exists y.\ eval\ P\ y \wedge eval\ (f\ y)\ x)$	$\bigcup f\ `\ P$
$P \sqcup Q$	$P \sqcup Q = pred\ (\lambda x.\ eval\ P\ x \vee eval\ Q\ x)$	$P \cup Q$

Here $(`) :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow bool) \Rightarrow \beta \Rightarrow bool$ is the image operator on sets satisfying $f\ `\ A = \{y.\ \exists x{\in}A.\ y = f\ x\}$.

Table 1. Abstract operations for predicate enumerations

For membership tests we define a further auxiliary constant:

$$\begin{array}{l} \mathit{member} :: \alpha\ \mathit{seq} \Rightarrow \alpha \Rightarrow \mathit{bool} \\ \mathit{member}\ \mathit{Empty}\ x \longleftrightarrow \mathit{False} \\ \mathit{member}\ (\mathit{Insert}\ y\ \mathit{yq})\ x \longleftrightarrow x = y \vee \mathit{eval}\ \mathit{yq}\ x \\ \mathit{member}\ (\mathit{Union}\ \mathit{xqs})\ x \longleftrightarrow \mathit{list\text{-}ex}\ (\lambda \mathit{xq}.\ \mathit{eval}\ \mathit{xq}\ x)\ \mathit{xqs} \end{array}$$

where $\mathit{list\text{-}ex} :: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha\ \mathit{list} \Rightarrow \mathit{bool}$ is existential quantification on lists, and use it to prove the code equation

$$\mathit{eval}\ (\mathit{Seq}\ f) = \mathit{member}\ (f\ ())$$

From the point of view of the logic, this characterization of the α *pred* algebra in terms of unit abstractions might seem odd; their purpose comes to surface when translating these equations to executable code, e.g. in ML:

```
datatype 'a pred = Seq of (unit -> 'a seq)
and 'a seq = Empty | Insert of 'a * 'a pred | Union of 'a pred list;

val bot_pred : 'a pred = Seq (fn u => Empty)

fun single x = Seq (fn u => Insert (x, bot_pred));

fun bind (Seq g) f =
  Seq (fn u =>
        (case g () of Empty => Empty
           | Insert (x, xq) => Union [f x, bind xq f]
           | Union xqs => Union (map (fn x => bind x f) xqs)));

fun sup_pred (Seq f) (Seq g) =
  Seq (fn u =>
        (case f () of Empty => g ()
           | Insert (x, xq) => Insert (x, sup_pred xq (Seq g))
           | Union xqs => Union (append xqs [Seq g])));

fun eval A_ (Seq f) = member A_ (f ())
and member A_ Empty x = false
  | member A_ (Insert (y, yq)) x = eq A_ x y orelse eval A_ yq x
  | member A_ (Union xqs) x = list_ex (fn xq => eval A_ xq x) xqs;
```

In the function definitions for *eval* and *member*, the expression `A_` is the dictionary for the *eq* class allowing for explicit equality checks using the overloaded constant *eq*.

In shape this follows a well-known ML technique for lazy lists: each inspection of a lazy list by means of an application `f ()` is protected by a constructor `Seq`. Thus we enforce a lazy evaluation strategy for predicate enumerations even for eager languages.

4.3 Compilation scheme for clauses

The central idea underlying the compilation of a predicate P is to generate a function P^M for each mode M of P that, given a list of input arguments, enumerates all tuples of output arguments. The clauses of an inductive predicate

can be viewed as a logic program. However, in contrast to logic programming languages like PROLOG, the execution of the functional program generated from the clauses uses *pattern matching* instead of *unification*. A precondition for the applicability of pattern matching is that the input arguments in the conclusions of the clauses, as well as the output arguments in the premises of the clauses are built up using only *datatype constructors* and variables. In the following description of the translation scheme, we will treat the pattern matching mechanism as a black box. However, our implementation uses a pattern translation algorithm due to Slind [9, §3.3], which closely resembles the techniques used in compilers for functional programming languages. The following notation will be used in our description of the translation mechanism:

$$\begin{array}{ll} \overline{x} & = x_1 \ldots x_l \\ \overline{\tau} & = \tau_1 \ldots \tau_l \\ \prod \overline{\tau} & = \tau_1 \times \cdots \times \tau_l \end{array} \qquad \begin{array}{ll} (\overline{x}) & = (x_1, \ldots, x_l) \\ \overline{\tau} \Rightarrow \sigma & = \tau_1 \Rightarrow \cdots \Rightarrow \tau_l \Rightarrow \sigma \\ M^- & = \{1, \ldots, l\} \backslash M \end{array}$$

Let $P :: \overline{\tau} \Rightarrow \mathit{bool}$ be a predicate and $M, M_{i,1}, \ldots M_{i,n_i}$ be a consistent mode assignment for the clauses C_i of P. The function P^M corresponding to mode M of P is defined as follows:

$$\begin{array}{l} P^M :: \overline{\tau}\langle M\rangle \Rightarrow (\prod \overline{\tau}\langle M^-\rangle)\ \mathit{pred} \\ P^M \overline{x}\langle M\rangle \equiv \mathit{pred}\ (\lambda(\overline{x}\langle M^-\rangle).\ P\ \overline{x}) \end{array}$$

Given the input arguments $\overline{x}\langle M\rangle :: \overline{\tau}\langle M\rangle$, function P^M returns a set of tuples of output arguments $(\overline{x}\langle M^-\rangle)$ for which $P\ \overline{x}$ holds. For modes $\{1,2\}$ and $\{3\}$ of the introductory *append* example, the corresponding definitions are as follows:

$$\begin{array}{l} \mathit{append}^{\{1,2\}} :: \alpha\ \mathit{list} \Rightarrow \alpha\ \mathit{list} \Rightarrow \alpha\ \mathit{list}\ \mathit{pred} \\ \mathit{append}^{\{1,2\}}\ \mathit{xs}\ \mathit{ys} = \mathit{pred}\ (\lambda \mathit{zs}.\ \mathit{append}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs}) \\ \mathit{append}^{\{3\}} :: \alpha\ \mathit{list} \Rightarrow (\alpha\ \mathit{list} \times \alpha\ \mathit{list})\ \mathit{pred} \\ \mathit{append}^{\{3\}}\ \mathit{zs} = \mathit{pred}\ (\lambda(\mathit{xs}, \mathit{ys}).\ \mathit{append}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs}) \end{array}$$

The recursion equation for P^M can be obtained from the clauses characterizing P in a canonical way:

$$P^M \overline{x}\langle M\rangle = \mathcal{C}_1\ \overline{x}\langle M\rangle \sqcup \cdots \sqcup \mathcal{C}_m\ \overline{x}\langle M\rangle$$

Intuitively, this means that the set of output values generated by P^M is the union of the output values generated by the clauses C_i. In order for pattern matching to work, all patterns occurring in the program must be *linear*, i.e. no variable may occur more than once. This can be achieved by renaming the free variables occurring in the terms $\overline{t}_i$, $\overline{u}_{i,1}$, …, $\overline{u}_{i,n_i}$, and by adding suitable equality checks to the generated program. Let $\overline{t}'_i$, $\overline{u}'_{i,1}$, …, $\overline{u}'_{i,n_i}$ denote these linear terms obtained by renaming the aforementioned ones, and let $\theta_i = \{\overline{y}_i \mapsto \overline{z}_i\}$, $\theta_{i,1} = \{\overline{y}_{i,1} \mapsto \overline{z}_{i,1}\}$, …, $\theta_{i,n_i} = \{\overline{y}_{i,n_i} \mapsto \overline{z}_{i,n_i}\}$ be substitutions such that $\theta_i(\overline{t}'_i) = t_i$, $\theta_{i,1}(\overline{u}'_{i,1}) = \overline{u}_{i,1}$, …, $\theta_{i,n_i}(\overline{u}'_{i,n_i}) = \overline{u}_{i,n_i}$, and $(\mathit{dom}(\theta_i) \cup \mathit{dom}(\theta_{i,1}) \cup$

$\cdots \cup dom(\theta_{i,n_i})) \cap FV(C_i) = \emptyset$. The expressions $\mathcal{C}_i$ corresponding to the clauses can then be defined by

$$
\begin{array}{l}
\mathcal{C}_i\ \overline{x}\langle M\rangle \equiv \\
\quad \mathit{single}\ (\overline{x}\langle M\rangle) \gg\!= (\lambda a_0.\ \mathit{case}\ a_0\ \mathit{of} \\
\qquad (\overline{t}_i'\langle M\rangle) \Rightarrow \mathit{if}\ \overline{y}_i \neq \overline{z}_i\ \mathit{then}\ \bot\ \mathit{else} \\
\qquad\quad Q_{i,1}^{M_{i,1}}\ (\overline{u}_{i,1}\langle M_{i,1}\rangle) \gg\!= (\lambda a_1.\ \mathit{case}\ a_1\ \mathit{of} \\
\qquad\qquad (\overline{u}_{i,1}'\langle M_{i,1}^-\rangle) \Rightarrow \mathit{if}\ \overline{y}_{i,1} \neq \overline{z}_{i,1}\ \mathit{then}\ \bot\ \mathit{else} \\
\qquad\qquad\quad \ddots \\
\qquad\qquad\qquad Q_{i,n_i}^{M_{i,n_i}}\ (\overline{u}_{i,n_i}\langle M_{i,n_i}\rangle) \gg\!= (\lambda a_{n_i}.\ \mathit{case}\ a_{n_i}\ \mathit{of} \\
\qquad\qquad\qquad\quad (\overline{u}_{i,n_i}'\langle M_{i,n_i}^-\rangle) \Rightarrow \mathit{if}\ \overline{y}_{i,n_i} \neq \overline{z}_{i,n_i}\ \mathit{then}\ \bot\ \mathit{else} \\
\qquad\qquad\qquad\qquad \mathit{single}\ (\overline{t}_i\langle M^-\rangle) \\
\qquad\qquad\qquad\quad \mid \_ \Rightarrow \bot) \\
\qquad\qquad \mid \_ \Rightarrow \bot) \\
\qquad \mid \_ \Rightarrow \bot)
\end{array}
$$

Here, $M_{i,1}^- = \{1, \ldots, k_{i,1}\} \backslash M_{i,1}$, …, $M_{i,n_i}^- = \{1, \ldots, k_{i,n_i}\} \backslash M_{i,n_i}$ denote the sets of indices of *output arguments* corresponding to the respective modes. As an example, we give the recursive equations for *append* on modes $\{1,2\}$ and $\{3\}$:

$$
\begin{array}{l}
\mathit{append}^{\{1,2\}}\ \mathit{xs}\ \mathit{ys} = \\
\quad \mathit{single}\ (\mathit{xs},\ \mathit{ys}) \gg\!= (\lambda a.\ \mathit{case}\ a\ \mathit{of} \\
\qquad ([],\ \mathit{zs}) \Rightarrow \mathit{single}\ \mathit{zs} \\
\qquad \mid (z \cdot \mathit{zs},\ \mathit{ws}) \Rightarrow \bot) \sqcup \\
\quad \mathit{single}\ (\mathit{xs},\ \mathit{ys}) \gg\!= (\lambda b.\ \mathit{case}\ b\ \mathit{of} \\
\qquad ([],\ \mathit{zs}) \Rightarrow \bot \\
\qquad \mid (z \cdot \mathit{zs},\ \mathit{ws}) \Rightarrow \mathit{append}^{\{1,2\}}\ \mathit{zs}\ \mathit{ws} \gg\!= (\lambda \mathit{vs}.\ \mathit{single}\ (z \cdot \mathit{vs})))
\end{array}
$$

$$
\begin{array}{l}
\mathit{append}^{\{3\}}\ \mathit{xs} = \\
\quad \mathit{single}\ \mathit{xs} \gg\!= (\lambda \mathit{ys}.\ \mathit{single}\ ([],\ \mathit{ys})) \sqcup \\
\quad \mathit{single}\ \mathit{xs} \gg\!= (\lambda a.\ \mathit{case}\ a\ \mathit{of} \\
\qquad [] \Rightarrow \bot \\
\qquad \mid z \cdot \mathit{zs} \Rightarrow \mathit{append}^{\{3\}}\ \mathit{zs} \gg\!= (\lambda b.\ \mathit{case}\ b\ \mathit{of} \\
\qquad\quad (\mathit{ws},\ \mathit{vs}) \Rightarrow \mathit{single}\ (z \cdot \mathit{ws},\ \mathit{vs})))
\end{array}
$$

Side conditions can be embedded into this translation scheme using the function

$$
\begin{array}{l}
\mathit{if\text{-}pred} :: \alpha \\
\mathit{ifpred}\ b = (\mathit{if}\ b\ \mathit{then}\ \mathit{single}\ ()\ \mathit{else}\ \bot)
\end{array}
$$

that maps *False* and *True* to the empty sequence and the singleton sequence containing only the unit element, respectively.

4.4 Proof of Recursion Equations

We will now describe how to prove the recursion equation for P^M given in the previous section using the definition of P^M, as well as the introduction and

elimination rules for P. We will also need introduction and elimination rules for the operators on type *pred*, which we show in Table 2. From the definition of P^M, we can easily derive the introduction rule

$$P\ \overline{x} \Longrightarrow \mathit{eval}\ (P^M\ \overline{x}\langle M\rangle)\ (\overline{x}\langle M^-\rangle)$$

and the elimination rule

$$\mathit{eval}\ (P^M\ \overline{x}\langle M\rangle)\ (\overline{x}\langle M^-\rangle) \Longrightarrow P\ \overline{x}$$

By extensionality (rule $=_I$), proving

$$P^M\overline{x}\langle M\rangle = \mathcal{C}_1\ \overline{x}\langle M\rangle \sqcup \cdots \sqcup \mathcal{C}_m\ \overline{x}\langle M\rangle$$

amounts to showing that

$$(1)\ \bigwedge x.\ \mathit{eval}\ (P^M\overline{x}\langle M\rangle)\ x \Longrightarrow \mathit{eval}\ (\mathcal{C}_1\ \overline{x}\langle M\rangle \sqcup \cdots \sqcup \mathcal{C}_m\ \overline{x}\langle M\rangle)\ x$$
$$(2)\ \bigwedge x.\ \mathit{eval}\ (\mathcal{C}_1\ \overline{x}\langle M\rangle \sqcup \cdots \sqcup \mathcal{C}_m\ \overline{x}\langle M\rangle)\ x \Longrightarrow \mathit{eval}\ (P^M\overline{x}\langle M\rangle)\ x$$

where $x :: \prod \overline{\tau}\langle M^-\rangle$. The variable x can be expanded to a tuple of variables:

$$(1)\ \bigwedge \overline{x}\langle M^-\rangle.\ \mathit{eval}\ (P^M\overline{x}\langle M\rangle)\ (\overline{x}\langle M^-\rangle) \Longrightarrow \mathit{eval}\ (\mathcal{C}_1\ \overline{x}\langle M\rangle \sqcup \cdots \sqcup \mathcal{C}_m\ \overline{x}\langle M\rangle)\ (\overline{x}\langle M^-\rangle)$$
$$(2)\ \bigwedge \overline{x}\langle M^-\rangle.\ \mathit{eval}\ (\mathcal{C}_1\ \overline{x}\langle M\rangle \sqcup \cdots \sqcup \mathcal{C}_m\ \overline{x}\langle M\rangle)\ (\overline{x}\langle M^-\rangle) \Longrightarrow \mathit{eval}\ (P^M\overline{x}\langle M\rangle)\ (\overline{x}\langle M^-\rangle)$$

Proof of (1). From $\mathit{eval}\ (P^M\overline{x}\langle M\rangle)\ (\overline{x}\langle M^-\rangle)$, we get $P\ \overline{x}$ using the elimination rule for P^M. Applying the elimination rule for P

$$P\ \overline{x} \Longrightarrow \mathcal{E}_1\ \overline{x} \Longrightarrow \cdots \Longrightarrow \mathcal{E}_m\ \overline{x} \Longrightarrow R$$
$$\mathcal{E}_i\ \overline{x} \equiv \bigwedge \overline{b_i}.\ \overline{x} = \overline{t}_i \Longrightarrow Q_{i,1}\ \overline{u}_{i,1} \Longrightarrow \cdots \Longrightarrow Q_{i,n_i}\ \overline{u}_{i,n_i} \Longrightarrow R$$

yields m proof obligations, each of which corresponds to an introduction rule. Note that $\overline{b_i}$ consists of the free variables of $\overline{u}_{i,j}$ and $\overline{t}_i$. For the ith introduction

$\bot_E$	$\mathit{eval}\ \bot\ x \Longrightarrow R$
single_I	$\mathit{eval}\ (\mathit{single}\ x)\ x$
single_E	$\mathit{eval}\ (\mathit{single}\ x)\ y \Longrightarrow (y = x \Longrightarrow R) \Longrightarrow R$
$\gg\!=_I$	$\mathit{eval}\ P\ x \Longrightarrow \mathit{eval}\ (Q\ x)\ y \Longrightarrow \mathit{eval}\ (P \gg\!= Q)\ y$
$\gg\!=_E$	$\mathit{eval}\ (P \gg\!= Q)\ y \Longrightarrow (\bigwedge x.\ \mathit{eval}\ P\ x \Longrightarrow \mathit{eval}\ (Q\ x)\ y \Longrightarrow R) \Longrightarrow R$
$\sqcup_{I1}$	$\mathit{eval}\ A\ x \Longrightarrow \mathit{eval}\ (A \sqcup B)\ x$
$\sqcup_{I2}$	$\mathit{eval}\ B\ x \Longrightarrow \mathit{eval}\ (A \sqcup B)\ x$
$\sqcup_E$	$\mathit{eval}\ (A \sqcup B)\ x \Longrightarrow (\mathit{eval}\ A\ x \Longrightarrow R) \Longrightarrow (\mathit{eval}\ B\ x \Longrightarrow R) \Longrightarrow R$
ifpred_I	$P \Longrightarrow \mathit{eval}\ (\mathit{ifpred}\ P)\ ()$
ifpred_E	$\mathit{eval}\ (\mathit{ifpred}\ b)\ x \Longrightarrow (b \Longrightarrow x = () \Longrightarrow R) \Longrightarrow R$
$=_I$	$(\bigwedge x.\ \mathit{eval}\ A\ x \Longrightarrow \mathit{eval}\ B\ x) \Longrightarrow (\bigwedge x.\ \mathit{eval}\ B\ x \Longrightarrow \mathit{eval}\ A\ x) \Longrightarrow A = B$

Table 2. Introduction and elimination rules for operators on *pred*

rule, we have to prove $\mathit{eval}\ (\mathcal{C}_1\ \overline{x}\langle M\rangle \sqcup \cdots \sqcup \mathcal{C}_m\ \overline{x}\langle M\rangle)\ (\overline{x}\langle M^-\rangle)$ from the assumptions $\overline{x} = \overline{t}_i$ and $Q_{i,1}\ \overline{u}_{i,1}$, …, $Q_{i,n_i}\ \overline{u}_{i,n_i}$. By applying the rules $\sqcup_{I1}$ and $\sqcup_{I2}$ in a suitable order, we select the $\mathcal{C}_i$ corresponding to the ith introduction rule, which leaves us with the proof obligation $\mathit{eval}\ (\mathcal{C}_i\ \overline{t}_i\langle M\rangle)\ (\overline{t}_i\langle M^-\rangle)$. By the definition of $\mathcal{C}_i$ and the rule $\gg\!=_I$, this gives rise to the two proof obligations

$$
\begin{array}{ll}
(1.i) & \mathit{eval}\ (\mathit{single}\ (\overline{t}_i\langle M\rangle))\ (\overline{t}_i\langle M\rangle) \\
(1.ii) & \mathit{eval}\ (\mathit{case}\ \overline{t}_i\langle M\rangle\ \mathit{of} \\
& \qquad (\overline{t}'_i\langle M\rangle) \Rightarrow \mathit{if}\ \overline{y}_i \neq \overline{z}_i\ \mathit{then}\ \bot\ \mathit{else} \\
& \qquad\quad Q_{i,1}^{M_{i,1}}\ (\overline{u}_{i,1}\langle M_{i,1}\rangle) \gg\!= (\lambda a_1.\ \mathit{case}\ a_1\ \mathit{of}\ \ldots) \\
& \quad \mid \_ \Rightarrow \bot)\ \overline{t}_i\langle M^-\rangle
\end{array}
$$

Goal $(1.i)$ is easily proved using single_I. Concerning goal $(1.ii)$, note that $(\overline{t}'_i\langle M\rangle)$ matches $(\overline{t}_i\langle M\rangle)$, so we have to consider the first branch of the *case* expression. Due to the definition of $\overline{t}'_i$, we also know that $\overline{y}_i = \overline{z}_i$, which means that we have to consider the *else* branch of the *if* clause. This leads to the new goal

$$\mathit{eval}\ (Q_{i,1}^{M_{i,1}}\ (\overline{u}_{i,1}\langle M_{i,1}\rangle) \gg\!= (\lambda a_1.\ \mathit{case}\ a_1\ \mathit{of}\ \ldots))\ \overline{t}_i\langle M^-\rangle$$

that, by applying rule $\gg\!=_I$, can be split up into the two goals

$$
\begin{array}{ll}
(1.iii) & \mathit{eval}\ (Q_{i,1}^{M_{i,1}}\ (\overline{u}_{i,1}\langle M_{i,1}\rangle))\ (\overline{u}_{i,1}\langle M^-_{i,1}\rangle) \\
(1.iv) & \mathit{eval}\ (\mathit{case}\ \overline{u}_{i,1}\langle M^-_{i,1}\rangle\ \mathit{of} \\
& \qquad (\overline{u}'_{i,1}\langle M^-_{i,1}\rangle) \Rightarrow \mathit{if}\ \overline{y}_{i,1} \neq \overline{z}_{i,1}\ \mathit{then}\ \bot\ \mathit{else}\ \ldots \\
& \quad \mid \_ \Rightarrow \bot)\ \overline{t}_i\langle M^-\rangle
\end{array}
$$

Goal $(1.iii)$ follows from the assumption $Q_{i,1}\ \overline{u}_{i,1}$ using the introduction rule for $Q_{i,1}^{M_{i,1}}$, while goal $(1.iv)$ can be solved in a similar way as goal $(1.ii)$. Repeating this proof scheme for $Q_{i,2}^{M_{i,2}}$, …, $Q_{i,n_i}^{M_{i,n_i}}$ finally leads us to a goal of the form

$$\mathit{eval}\ (\mathit{single}\ (\overline{t}_i\langle M^-\rangle))\ (\overline{t}_i\langle M^-\rangle)$$

which is trivially solvable using single_I.

Proof of (2). The proof of this direction is dual to the previous one: rather than splitting up the conclusion into simpler formulae, we now perform forward inferences that transform complex premises into simpler ones. Eliminating $\mathit{eval}\ (\mathcal{C}_1\ \overline{x}\langle M\rangle \sqcup \cdots \sqcup \mathcal{C}_m\ \overline{x}\langle M\rangle)\ (\overline{x}\langle M^-\rangle)$ using rule $\sqcup_E$ leaves us with m proof obligations of the form

$$\mathit{eval}\ (\mathcal{C}_i\ \overline{x}\langle M\rangle)\ (\overline{x}\langle M^-\rangle) \Longrightarrow \mathit{eval}\ (P^M \overline{x}\langle M\rangle)\ (\overline{x}\langle M^-\rangle)$$

By unfolding the definition of $\mathcal{C}_i$ and applying rule $\gg\!=_E$ to the premise of the above implication, we obtain a_0 such that

$$
\begin{array}{ll}
(2.i) & \mathit{eval}\ (\mathit{single}\ (\overline{x}\langle M\rangle))\ a_0 \\
(2.ii) & \mathit{eval}\ (\mathit{case}\ a_0\ \mathit{of} \\
& \qquad (\overline{t}'_i\langle M\rangle) \Rightarrow \mathit{if}\ \overline{y}_i \neq \overline{z}_i\ \mathit{then}\ \bot\ \mathit{else} \\
& \qquad\quad Q_{i,1}^{M_{i,1}}\ (\overline{u}_{i,1}\langle M_{i,1}\rangle) \gg\!= (\lambda a_1.\ \mathit{case}\ a_1\ \mathit{of}\ \ldots) \\
& \quad \mid \_ \Rightarrow \bot)\ \overline{x}\langle M^-\rangle
\end{array}
$$

From $(2.i)$, we get $\overline{x}\langle M\rangle = a_0$ by rule $single_E$. Since a_0 must be an element of a datatype, we can analyze its shape by applying suitable case splitting rules. Of the generated cases only one case is non-trivial. In the trivial cases, a_0 does not match $(\overline{t}'_i\langle M\rangle)$, so the *case* expression evaluates to $\bot$, and the goal can be solved using $\bot_E$. In the non-trivial case, we have that $a_0 = (\overline{t}'_i\langle M\rangle)$. Splitting up the *if* expression yields two cases. In the *then* case, the whole expression evaluates to $\bot$, so the goal is again provable using $\bot_E$. In the *else* branch, we have that $\overline{y}_i = \overline{z}_i$, and hence $a_0 = (\overline{t}_i\langle M\rangle)$ by definition of $\overline{t}'_i$, which also implies $\overline{x}\langle M\rangle = \overline{t}_i\langle M\rangle$. Assumption $(2.ii)$ can thus be rewritten to

$$eval\ (Q_{i,1}^{M_{i,1}}\ (\overline{u}_{i,1}\langle M_{i,1}\rangle) \gg\!= (\lambda a_1.\ case\ a_1\ of\ \ldots))\ \overline{x}\langle M^-\rangle$$

By another application of $\gg\!=_E$, we obtain a_1 such that

$$\begin{array}{ll}(2.iii) & eval\ (Q_{i,1}^{M_{i,1}}\ (\overline{u}_{i,1}\langle M_{i,1}\rangle))\ a_1\\ (2.iv) & eval\ (case\ a_1\ of\\ & \qquad (\overline{u}'_{i,1}\langle M^-_{i,1}\rangle) \Rightarrow if\ \overline{y}_{i,1} \neq \overline{z}_{i,1}\ then\ \bot\ else\ \ldots\\ & \quad \mid \_ \Rightarrow \bot)\ \overline{x}\langle M^-\rangle\end{array}$$

The assumption $(2.iv)$ is treated in a similar way as $(2.ii)$. A case analysis over a_1 reveals that the only non-trivial case is the one where $a_1 = (\overline{u}'_{i,1}\langle M^-_{i,1}\rangle)$. The only non-trivial branch of the *if* expression is the *else* branch, where $\overline{y}_{i,1} = \overline{z}_{i,1}$. Hence, by definition of $\overline{u}'_{i,1}$, it follows that $a_1 = (\overline{u}_{i,1}\langle M^-_{i,1}\rangle)$, which entitles us to rewrite $(2.iii)$ to $eval\ (Q_{i,1}^{M_{i,1}}\ (\overline{u}_{i,1}\langle M_{i,1}\rangle))\ (\overline{u}_{i,1}\langle M^-_{i,1}\rangle)$, from which we can deduce $Q_{i,1}\ \overline{u}_{i,1}$ by applying the elimination rule for $Q_{i,1}^{M_{i,1}}$. By repeating this kind of reasoning for $Q_{i,2}^{M_{i,2}}$, …, $Q_{i,n_i}^{M_{i,n_i}}$, we also obtain that $Q_{i,2}\ \overline{u}_{i,2}$, …, $Q_{i,n_i}\ \overline{u}_{i,n_i}$ holds. Furthermore, after the complete decomposition of $(2.iv)$, we end up with an assumption of the form

$$eval\ (single\ (\overline{t}_i\langle M^-\rangle))\ (\overline{x}\langle M^-\rangle)$$

from which we can deduce $\overline{t}_i\langle M^-\rangle = \overline{x}\langle M^-\rangle$ by an application of $single_E$. Thus, using the equations gained from $(2.i)$ and $(2.ii)$, the conclusion of the implication we set out to prove can be rephrased as

$$eval\ (P^M\ \overline{t}_i\langle M\rangle)\ (\overline{t}_i\langle M^-\rangle)$$

Thanks to the introduction rule for P^M, it suffices to prove $P\ \overline{t}_i$, which can easily be done using the introduction rule

$$Q_{i,1}\ \overline{u}_{i,1} \Longrightarrow \cdots \Longrightarrow Q_{i,n_i}\ \overline{u}_{i,n_i} \Longrightarrow P\ \overline{t}_i$$

together with the previous results.

4.5 Animating equations

We have shown in detail how to derive executable equations from the specification of a predicate P for a consistent mode M. The results are always enumerations of type α *pred*. We discuss briefly how to get access to the enumerated values of type α proper.

Membership tests. The type constructor *pred* can be stripped using explicit membership tests. For example, we could define a suffix predicate using *append*:

$$\textit{is-suffix}\ zs\ ys \longleftrightarrow (\exists\, xs.\ \textit{append}\ xs\ ys\ zs)$$

Using the definition of $\textit{append}^{\{2,3\}}$ this can be reformulated as

$$\textit{is-suffix}\ zs\ ys \longleftrightarrow (\exists\, xs.\ \textit{eval}\ (\textit{append}^{\{2,3\}}\ ys\ zs)\ xs)$$

from which follows

$$\textit{is-suffix}\ zs\ ys \longleftrightarrow \textit{eval}\ (\textit{append}^{\{2,3\}}\ ys\ zs \gg\!= (\lambda\text{-}.\ \textit{single}\ ()))\ ()$$

using introduction and elimination rules for *op* $\gg\!=$ and *single*. This equation then is directly executable.

Enumeration queries. When developing inductive specifications it is often desirable to check early whether the specification behaves as expected by enumerating one or more solutions which satisfy the specification. In our framework this cannot be expressed inside the logic: values of type α *pred* are set-like, whereas each concrete enumeration imposes a certain order on elements which is not reflected in the logic. However it can be done directly on the generated code, e.g. in ML using

```
fun nexts [] = NONE
  | nexts (xq :: xqs) = case next xq
     of NONE => nexts xqs
      | SOME (x, xq) => SOME (x, Seq (fn () => Union (xq :: xqs)))
and next (Seq f) = case f ()
 of Empty => NONE
  | Insert (x, xq) => SOME (x, xq)
  | Union xqs => nexts xqs;
```

Wrapped up in a suitable user interface this allows to interactively enumerate solutions fitting to inductive predicates.

5 Extensions to the base framework

5.1 Higher-order modes

A useful extension of the framework presented in §4.3 is to allow inductive predicates that take other predicates as arguments. A standard example for such a predicate is the *reflexive transitive closure* taking a predicate of type $\alpha \Rightarrow \alpha \Rightarrow \textit{bool}$ as an argument, and returning a predicate of the same type:

inductive $\textit{rtc} :: (\alpha \Rightarrow \alpha \Rightarrow \textit{bool}) \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \textit{bool}$
  for $r :: \alpha \Rightarrow \alpha \Rightarrow \textit{bool}$ **where**
    $\textit{rtc}\ r\ x\ x$
  | $r\ x\ y \Longrightarrow \textit{rtc}\ r\ y\ z \Longrightarrow \textit{rtc}\ r\ x\ z$

In addition to its two *arguments* of type α, *rtc* also has a *parameter* r that stays fixed throughout the definition. The general form of a mode for a higher-order predicate P with k arguments and parameters $r_1, \ldots, r_\rho$ with arities $k_1, \ldots, k_\rho$ is $(M_1, \ldots, M_\rho, M)$, where $M_i \subseteq \{1, \ldots, k_i\}$ (for $1 \leq i \leq \rho$) and $M \subseteq \{1, \ldots, k\}$. Intuitively, this mode means that $P\ r_1\ \cdots\ r_\rho$ has mode M, provided that r_i has mode M_i. The possible modes for *rtc* are $(\{\}, \{1\})$, $(\{\}, \{2\})$, $(\{\}, \{1, 2\})$, $(\{1\}, \{1\})$, $(\{2\}, \{2\})$, $(\{1\}, \{1, 2\})$, and $(\{2\}, \{1, 2\})$. The general definition of the function corresponding to the mode $(M_1, \ldots, M_\rho, M)$ of a predicate P is

$$
\begin{aligned}
&P^{(M_1,\ldots,M_\rho,M)} :: \\
&\quad (\overline{\tau}_1\langle M_1\rangle \Rightarrow (\prod \overline{\tau}_1\langle M_1^-\rangle)\ \mathit{pred}) \Rightarrow \cdots \Rightarrow \\
&\quad (\overline{\tau}_\rho\langle M_\rho\rangle \Rightarrow (\prod \overline{\tau}_\rho\langle M_\rho^-\rangle)\ \mathit{pred}) \Rightarrow (\prod \overline{\tau}\langle M^-\rangle)\ \mathit{pred} \\
&P^{(M_1,\ldots,M_\rho,M)}\ s_1\ \ldots\ s_\rho\ \overline{x}\langle M\rangle \equiv \mathit{pred}\ (\lambda(\overline{x}\langle M^-\rangle).\ P \\
&\quad (\lambda\overline{x}_1.\ \mathit{eval}\ (s_1\ \overline{x}_1\langle M_1\rangle)\ (\overline{x}_1\langle M_1^-\rangle))\ \ldots \\
&\quad (\lambda\overline{x}_\rho.\ \mathit{eval}\ (s_1\ \overline{x}_\rho\langle M_\rho\rangle)\ (\overline{x}_\rho\langle M_\rho^-\rangle))\ \overline{x})
\end{aligned}
$$

Since P expects predicates as parameters, but s_i are functions returning sets, these have to be converted back to predicates using *eval* before passing them to P. For *rtc*, the definitions of the functions corresponding to the modes $(\{1\}, \{1\})$ and $(\{2\}, \{2\})$ are

$$
\begin{aligned}
&\mathit{rtc}^{(\{1\},\{1\})} :: (\alpha \Rightarrow \alpha\ \mathit{pred}) \Rightarrow \alpha \Rightarrow \alpha\ \mathit{pred} \\
&\mathit{rtc}^{(\{1\},\{1\})}\ s\ x \equiv \mathit{pred}\ (\lambda y.\ \mathit{rtc}\ (\lambda x'\ y'.\ \mathit{eval}\ (s\ x')\ y')\ x\ y) \\
&\mathit{rtc}^{(\{2\},\{2\})} :: (\alpha \Rightarrow \alpha\ \mathit{pred}) \Rightarrow \alpha \Rightarrow \alpha\ \mathit{pred} \\
&\mathit{rtc}^{(\{2\},\{2\})}\ s\ y \equiv \mathit{pred}\ (\lambda x.\ \mathit{rtc}\ (\lambda x'\ y'.\ \mathit{eval}\ (s\ y')\ x')\ x\ y)
\end{aligned}
$$

The corresponding recursion equations have the form

$$
\begin{aligned}
&\mathit{rtc}^{(\{1\},\{1\})}\ r\ x = \\
&\quad \mathit{single}\ x \gg\!= (\lambda x.\ \mathit{single}\ x) \sqcup \\
&\quad \mathit{single}\ x \gg\!= (\lambda x.\ r\ x \gg\!= (\lambda y.\ \mathit{rtc}^{(\{1\},\{1\})}\ r\ y \gg\!= (\lambda z.\ \mathit{single}\ z)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{rtc}^{(\{2\},\{2\})}\ r\ y = \\
&\quad \mathit{single}\ y \gg\!= (\lambda x.\ \mathit{single}\ x) \sqcup \\
&\quad \mathit{single}\ y \gg\!= (\lambda z.\ \mathit{rtc}^{(\{2\},\{2\})}\ r\ z \gg\!= (\lambda y.\ r\ y \gg\!= (\lambda x.\ \mathit{single}\ x)))
\end{aligned}
$$

5.2 Mixing predicates and functions

When mixing predicates and functions, mode analysis treats functions as predicates where all arguments are *input*. This can restrict the number of consistent mode assignments considerably.

The following mutually inductive predicates model a grammar generating all words containing equally many as and bs. This example, which is originally due to Hopcroft and Ullman, can be found in the Isabelle tutorial by Nipkow [8].

inductive
  $S :: \mathit{alfa\ list} \Rightarrow \mathit{bool}$ **and**
  $A :: \mathit{alfa\ list} \Rightarrow \mathit{bool}$ **and** $B :: \mathit{alfa\ list} \Rightarrow \mathit{bool}$
where
  $S\ []$
$|\ A\ w \Longrightarrow S\ (b \cdot w)$
$|\ B\ w \Longrightarrow S\ (a \cdot w)$
$|\ S\ w \Longrightarrow A\ (a \cdot w)$
$|\ A\ v \Longrightarrow A\ w \Longrightarrow A\ (b \cdot v\ @\ w)$
$|\ S\ w \Longrightarrow B\ (b \cdot w)$
$|\ B\ v \Longrightarrow B\ w \Longrightarrow B\ (a \cdot v\ @\ w)$

By choosing mode {} for the above predicates (i.e. their arguments are all output), we can enumerate all elements of the set S containing equally many as and bs. However, the above predicates cannot easily be used with mode {1}, i.e. for *checking* whether a given word is generated by the grammar. This is because of the rules with the conclusions $A\ (b \cdot v\ @\ w)$ and $B\ (a \cdot v\ @\ w)$. Since the append *function* (denoted by @) is not a constructor, we cannot do pattern matching on the argument. However, the problematic rules can be rephrased as

$$\mathit{append}\ v\ w\ vw \Longrightarrow A\ v \Longrightarrow A\ w \Longrightarrow A\ (b \cdot vw)$$
$$\mathit{append}\ v\ w\ vw \Longrightarrow B\ v \Longrightarrow B\ w \Longrightarrow B\ (a \cdot vw)$$

The problematic expression $v\ @\ w$ in the conclusion has been replaced by a new variable vw. The fact that vw is the result of appending the two lists v and w is now expressed using the *append* predicate from §3. In order to check whether a given word can be generated using these rules, *append* first enumerates all ways of decomposing the given list vw into two sublists v and w, and then recursively checks whether these words can be generated by the grammar.

6 Conclusion and future work

We have presented a *definitional* translation for inductive predicates to equations which can be turned into executable code using existing code generation infrastructure in *Isabelle/HOL*. This is a fundamental contribution to extend the scope of code generation from functional to functional-logic programs embedded into *Isabelle/HOL* without compromising the trusted implementation of the code generator itself. We have applied our translation to two larger case studies, the μJava semantics by Nipkow, von Oheimb and Pusch [7] and the ς-calculus by Henrio and Kammüller [5], resulting in simple interpreters for these two programming languages. Further experiments suggest the following extensions:

– Successful mode inference does not guarantee termination. Like in Prolog, the order of premises in introduction rules can influence termination. Using termination analysis built-in in *Isabelle/HOL*, we can guess which modes lead to terminating functions. The mode analysis can use this and prefer terminating modes over possibly non-terminating ones.

– Rephrasing recursive functions to inductive predicates, as we apply it in §5.2, possibly results in more modes for the mode analysis. But applying the transformation blindly could lead to unnecessarily complicated equations. The mode analysis should be extended to infer modes using the transformation only when required.
– The executable model for enumerations we have presented is sometimes inappropriate: it performs depth-first search which can lead to a non-terminating search in an irrelevant but infinite branch of the search tree. It has to be figured out how alternative search strategies (e.g. iterative depth-first search) can provide a solution for this.

We plan to integrate our procedure into the next *Isabelle* release.

References

1. Berghofer, S., Nipkow, T.: Executing higher order logic. In: P. Callaghan, Z. Luo, J. McKinna, R. Pollack (eds.) Types for Proofs and Programs (TYPES 2000), *Lecture Notes in Computer Science*, vol. 2277. Springer-Verlag (2002)
2. Delahaye, D., Dubois, C., Étienne, J.F.: Extracting purely functional contents from logical inductive types. In: TPHOLs, pp. 70–85 (2007)
3. Haftmann, F., Nipkow, T.: A code generator framework for Isabelle/HOL. Tech. Rep. 364/07, Department of Computer Science, University of Kaiserslautern (2007)
4. Hanus, M.: A unified computation model for functional and logic programming. In: Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97), pp. 80–93 (1997)
5. Henrio, L., Kammüller, F.: A mechanized model of the theory of objects. In: 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), LNCS. Springer (2007)
6. Mellish, C.S.: The automatic generation of mode declarations for prolog programs. Tech. Rep. 163, Department of Artificial Intelligence (1981)
7. Nipkow, T., von Oheimb, D., Pusch, C.: μJava: Embedding a programming language in a theorem prover. In: F. Bauer, R. Steinbrüggen (eds.) Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999, pp. 117–144. IOS Press (2000)
8. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer-Verlag (2002)
9. Slind, K.: Reasoning about terminating functional programs. Ph.D. thesis, Institut für Informatik, TU München (1999)
10. Somogyi, Z., Henderson, F.J., Conway, T.C.: Mercury: an efficient purely declarative logic programming language. In: In Proceedings of the Australian Computer Science Conference, pp. 499–512 (1995)
11. Wasserrab, D., Nipkow, T., Snelting, G., Tip, F.: An operational semantics and type safety proof for multiple inheritance in C++. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications, pp. 345–362. ACM Press (2006)