Unnesting of Copatterns

Anton Setzer[1], Andreas Abel[2], Brigitte Pientka[3], and David Thibodeau[3]

[1] Dept. of Computer Science, Swansea University, Swansea SA2 8PP, UK
a.g.setzer@swan.ac.uk
[2] Computer Science and Engineering, Chalmers and Gothenburg University,
Rännvägen 6, 41296 Göteborg, Sweden, andreas.abel@gu.se
[3] School of Computer Science, McGill University, Montreal, Canada
{bpientka,dthibo1}@cs.mcgill.ca

Abstract. Inductive data such as finite lists and trees can elegantly be defined by constructors which allow programmers to analyze and manipulate finite data via pattern matching. Dually, coinductive data such as streams can be defined by observations such as head and tail and programmers can synthesize infinite data via copattern matching. This leads to a symmetric language where finite and infinite data can be nested. In this paper, we compile nested pattern and copattern matching into a core language which only supports simple non-nested (co)pattern matching. This core language may serve as an intermediate language of a compiler. We show that this translation is conservative, i.e., the multi-step reduction relation in both languages coincides for terms of the original language. Furthermore, we show that the translation preserves strong normalisation: a term of the original language is strongly normalising in one language if and only if it is so in the other.

Keywords: Pattern matching, copattern matching, algebraic data types, codata, coalgebras, conservative extension, strong normalisation

1 Introduction

Finite inductive data such as lists and trees can be elegantly defined via constructors, and programmers are able to case-analyze and manipulate finite data in functional languages using pattern matching. To compile functional languages supporting pattern matching, we typically elaborate complex and nested pattern matches into a series of simple patterns which can be easily compiled into efficient code (see for example [3]). This is typically the first step in translating the source language to a low-level target language which can be efficiently executed. It is also an important step towards developing a core calculus supporting well-founded recursive functions.

Dually to finite data, coinductive data such as streams can be defined by observations such as head and tail. This view was pioneered by Hagino [7] who modelled finite objects via initial algebras and infinite objects via final coalgebras in category theory. This led to the design of symML, a dialect of ML where we can for example define the codata-type of streams via the destructors head and tail which describe the observations we can make about streams [8]. Cockett and Fukushima [6] continued this line of work and designed a language *Charity* where one programs directly with the morphisms of category theory. Our recent work [2] extends these ideas and introduces *copattern* matching for analyzing infinite data. This novel perspective on defining infinite structures via their observations leads to a new symmetric foundation for functional languages where inductive and coinductive data types can be mixed.

In this paper, we elaborate our high-level functional language which supports nested patterns and copatterns into a language of simple patterns and copatterns. Similar to pattern compilation in Idris or Agda, our translation into simple patterns is guided by the coverage algorithm. We show that the translation into our core language of simple patterns is conservative, i.e., the multi-step reduction relations of both languages coincide for terms of the original language. Furthermore, we show that the translation preserves strong normalisation: a term of the original language is strongly normalising in one language if and only if it is normalising in the other.

The paper is organized as follows: We describe the core language including pattern and copattern matching in Section 2. In Section 3, we explain the elaboration into simple patterns. In Section 4 we show its correctness with regard to reduction behavior and normalization.

2 A Core Language for Copattern Matching

In this section, we summarize the basic core language with (co)recursive data types and support for (co)pattern described in previous work [2].

2.1 Types and Terms

A language $\mathcal{L} = (\mathcal{F}, \mathcal{C}, \mathcal{D})$ consists of a finite set $\mathcal{F}$ of *constants* (function symbols), a finite set $\mathcal{C}$ of *constructors*, and a finite set $\mathcal{D}$ of *destructors*. We will in the following assume one fixed language $\mathcal{L}$, with pairwise disjoint $\mathcal{F}$, $\mathcal{C}$, and $\mathcal{D}$. We write f, c, d for elements of $\mathcal{F}, \mathcal{C}, \mathcal{D}$, respectively.

Our type language includes $\mathbf{1}$ (unit), $A \times B$ (products), $A \to B$ (functions), disjoint unions D (labelled sums, "data"), records R (labelled products), least fixed points $\mu X.D$, and greatest fixed points $\nu X.R$.

$$\begin{array}{lll} \text{Types} & A, B, C & ::= X \mid \mathbf{1} \mid A \times B \mid A \to B \mid \mu X.D \mid \nu X.R \\ \text{Variants} & D & ::= \langle c_1 \ A_1 \mid \ldots \mid c_n \ A_n \rangle \\ \text{Records} & R & ::= \{d_1 : A_1, \ldots, d_n : A_n\} \end{array}$$

Variant types $\langle c_1 \ A_1 \mid \ldots \mid c_n \ A_n \rangle$, finite maps from constructors to types, appear only in possibly recursive data types $\mu X.D$. Records $\{d_1 : A_1, \ldots, d_n : A_n\}$, finite maps from destructors to types, list the fields d_i of a possibly recursive record type $\nu X.R$. To illustrate, we define natural numbers Nat, lists and Nat-streams:

$$\begin{array}{ll} \mathsf{Nat} & := \mu X.\langle \mathsf{zero}\ \mathbf{1} \mid \mathsf{suc}\ X \rangle \\ \mathsf{List}\ A & := \mu X.\langle \mathsf{nil}\ \mathbf{1} \mid \mathsf{cons}\ (A \times X) \rangle \\ \mathsf{StrN} & := \nu X.\{\mathsf{head} : \mathsf{Nat}, \mathsf{tail} : X\} \end{array}$$

In our non-polymorphic calculus, type variables X only serve to construct recursive data types and recursive record types. As usual, $\mu X.D$ ($\nu X.R$, resp.) binds type variable X in D (R, resp.). Capture-avoiding substitution of type C for variable X in type A is denoted by $A[X := C]$. A type is *well-formed* if it has no free type variables; in the following, we assume that all types are well-formed.

We write $c \in D$ for $c\,A$ being part of variant D for some A and define the type of constructor c as $(\mu X.D)_c := A[X := \mu X.D]$. Analogously, we write $d \in R$ for $d : A$ being part of the record R for some A and define the type of the destructor d as $(\nu X.R)_d := A[X := \nu X.R]$.

A signature for $\mathcal{L}$ is a map Σ from $\mathcal{F}$ into the set of types. Unless stated differently, we assume one fixed signature Σ. A typed language is a pair $(\mathcal{L}, \Sigma)$ where $\mathcal{L}$ is a language and Σ is a signature for $\mathcal{L}$. We sometimes write Σ instead of $(\mathcal{L}, \Sigma)$. We write $f \in \Sigma$ if $\Sigma(f)$ is defined, i.e., $f \in \mathcal{F}$. Next, we define the grammar of *terms* of a language $\mathcal{L} = (\mathcal{F}, \mathcal{C}, \mathcal{D})$. Herein, $f \in \mathcal{F}$, $c \in \mathcal{C}$, and $d \in \mathcal{D}$.

$$
\begin{array}{rll|ll}
e, r, s, t, u ::= & f & \text{Defined constant (function)} & x & \text{Variable} \\
 | & () & \text{Unit (empty tuple)} & (t_1, t_2) & \text{Pair} \\
 | & c\ t & \text{Constructor application} & t_1\ t_2 & \text{Application} \\
 | & t\ .d & \text{Destructor application} & &
\end{array}
$$

Terms include identifiers (variables x and defined constants f) and introduction forms: pairs (t_1, t_2), unit $()$, and constructed terms $c\ t$, for the *positive* types $A \times B$, $\mathbf{1}$, and $\mu X.D$. There are however no elimination forms for positive types, since we define programs via rewrite rules and employ *pattern matching*. On the other hand we have *eliminations*, application $t_1\, t_2$ and projection $t\ .d$, of *negative* types $A \to B$ and $\nu X.R$ respectively, but omit introductions for these types, since this will be handled by *copattern matching*.

We write term substitutions as $s[x_1 := t_1, \ldots, x_n := t_n]$ or short $s[\vec{x} := \vec{t}]$. Contexts Δ are finite maps from variable to types, written as lists of pairs $x_1 : A_1, \ldots, x_n : A_n$, or short $\vec{x} : \vec{A}$, with $\cdot$ denoting the empty context. We write $\Delta \to A$ or $\vec{A} \to A$ for n-ary curried function types $A_1 \to \cdots \to A_n \to A$ (but A may still be a function type), and $s\ \vec{t}$ for n-ary curried application $s\ t_1\ \cdots\ t_n$.

$$
\frac{\Delta(x) = A}{\Delta \vdash x : A} \qquad \frac{}{\Delta \vdash () : \mathbf{1}} \qquad \frac{\Delta \vdash t : (\mu X.D)_c}{\Delta \vdash c\ t : \mu X.D} \qquad \frac{\Delta \vdash t_1 : A_1 \quad \Delta \vdash t_2 : A_2}{\Delta \vdash (t_1, t_2) : A_1 \times A_2}
$$

$$
\frac{}{\Delta \vdash f : \Sigma(f)} \qquad \frac{\Delta \vdash t : A \to B \quad \Delta \vdash t' : A}{\Delta \vdash t\ t' : B} \qquad \frac{\Delta \vdash t : \nu X.R}{\Delta \vdash t\ .d : (\nu X.R)_d}
$$

Fig. 1. Typing rules

The typing rules for terms (relative to a signature Σ) are defined in Figure 1 as a type assignment system. If we want to explicitly refer to a given typed language $(\mathcal{L}, \Sigma)$ or Σ we write $\Delta \vdash_{\mathcal{L},\Sigma} A$ or $\Delta \vdash_{\Sigma} A$, similarly for later notions of $\vdash$.

2.2 Patterns and copatterns

For each $f \in \mathcal{F}$, we will determine the rewrite rules for f as a set of pairs $(q \longrightarrow r)$ where q is a copattern sometimes referred to as *left hand side*, and r a term, sometimes referred to as *right hand side*. Patterns p and copatterns q are special terms given by the grammar below, where $c \in \mathcal{C}$ and $d \in \mathcal{D}$.

$p ::= x$	Variable pattern	$q ::= f$	Head (constant)
$\mid ()$	Unit pattern	$\mid q\ p$	Application copattern
$\mid (p_1, p_2)$	Pair pattern	$\mid q\ .d$	Destructor copattern
$\mid c\ p$	Constructor pattern		

In addition we require p and q to be *linear*, i.e. each variable occurs at most once in p or q. When later defining typed patterns $\Delta \vdash q : A$ as part of a coverage complete pattern set for a constant f, we will have that this judgement is provable as a typing judgement for terms, the variables in q are exactly the variables in Δ, and f is the head of q.

The distinction between patterns and copatterns is in this article only relevant in this grammar, therefore we usually write simply "pattern" for both.

Example 1 (Cycling numbers). Function cyc of type $\mathsf{Nat} \to \mathsf{StrN}$, when passed an integer n, produces a stream $n, n-1, \ldots, 1, 0, N, N-1, \ldots, 1, 0, N, N-1, \ldots$ for some fixed N. To define this function we match on the input n and also observe the resulting stream, highlighting the mix of pattern and copattern matching. The rules for cyc are the following:

$$\begin{array}{lll} \mathsf{cyc}\ x & .\mathsf{head} & \longrightarrow x \\ \mathsf{cyc}\ (\mathsf{zero}\ ()) & .\mathsf{tail} & \longrightarrow \mathsf{cyc}\ N \\ \mathsf{cyc}\ (\mathsf{suc}\ x) & .\mathsf{tail} & \longrightarrow \mathsf{cyc}\ x \end{array}$$

Example 2 (Fibonacci Stream). Nested destructor copatterns appear in the following definition of the stream of Fibonacci numbers. It uses $\mathsf{zipWith}\ \_{+}\_$ which is the pointwise addition of two streams.

$$\begin{array}{lll} \mathsf{fib}\ .\mathsf{head} & & \longrightarrow 0 \\ \mathsf{fib}\ .\mathsf{tail} & .\mathsf{head} & \longrightarrow 1 \\ \mathsf{fib}\ .\mathsf{tail} & .\mathsf{tail} & \longrightarrow \mathsf{zipWith}\ \_{+}\_\ \mathsf{fib}\ (\mathsf{fib}\ .\mathsf{tail}) \end{array}$$

2.3 Coverage

For our purposes, the rules for a function f are *complete*, if every closed, well-typed term t of positive type can be reduced with exactly one of the rules of f. Alternatively, we could say that all cases for f are uniquely *covered* by the reduction rules. Coverage implies that the execution of a program progresses, i.e., does not get stuck, and is deterministic. Note that by restricting to positive types, which play the role of ground types, we ensure that t is not stuck because f is underapplied. Progress has been proven in previous work [2]; in this work, we extend coverage checking to an algorithm for pattern compilation.

We introduce the judgement $f : A \;\triangleleft|\; Q$, called a *coverage complete pattern set for f (cc-pattern-set for f)*. Here Q is a *set* $Q = (\Delta_i \vdash q_i : C_i)_{i=1,\ldots,n}$ (note our slight abuse of vector notation). If $f : A \;\triangleleft|\; Q$ then constant f of type A can be defined by the coverage complete patterns q_i (depending on variables in Δ_i) together with rewrite rules $q_i \longrightarrow t_i$ for some $\Delta_i \vdash t_i : C_i$.

The rules for deriving cc-pattern-sets are presented in Figure 2. In the variable splitting rules, the split variable is written as the last element of the context. Because contexts are finite maps they have no order—any variable can be split. Note as well that patterns and copatterns are by definition required to be linear.

Result splitting:

$$\frac{}{f : A \;\triangleleft|\; (\cdot \vdash f : A)}\ \mathrm{C_{Head}} \qquad \frac{f : A \;\triangleleft|\; Q\ (\Delta \vdash q : B \to C)}{f : A \;\triangleleft|\; Q\ (\Delta, x : B \vdash q\ x : C)}\ \mathrm{C_{App}}$$

$$\frac{f : A \;\triangleleft|\; Q\ (\Delta \vdash q : \nu X.R)}{f : A \;\triangleleft|\; Q\ (\Delta \vdash q\ .d : (\nu X.R)_d)_{d \in R}}\ \mathrm{C_{Dest}}$$

Variable splitting:

$$\frac{f : A \;\triangleleft|\; Q\ (\Delta, x : \mathbf{1} \vdash q : C)}{f : A \;\triangleleft|\; Q\ (\Delta \vdash q[x := ()] : C)}\ \mathrm{C_{Unit}}$$

$$\frac{f : A \;\triangleleft|\; Q\ (\Delta, x : A_1 \times A_2 \vdash q : C)}{f : A \;\triangleleft|\; Q\ (\Delta, x_1 : A_1, x_2 : A_2 \vdash q[x := (x_1, x_2)] : C)}\ \mathrm{C_{Pair}}$$

$$\frac{f : A \;\triangleleft|\; Q\ (\Delta, x : \mu X.D \vdash q : C)}{f : A \;\triangleleft|\; Q\ (\Delta, x' : (\mu X.D)_c \vdash q[x := c\ x'] : C)_{c \in D}}\ \mathrm{C_{Const}}$$

Fig. 2. Coverage rules

A *coverage complete set of rules (cc-rule-set)* for a constant f

$$f : \Sigma(f) \;\triangleleft|\; (\Delta_i \vdash q_i \longrightarrow t_i : C_i)_{i=1,\ldots,n}$$

consists of a cc-pattern-set $f : \Sigma(f) \;\triangleleft|\; (\Delta_i \vdash q_i : C_i)_{i=1,\ldots,n}$ (called the *underlying cc-pattern-set*) together with terms t_i for $i = 1, \ldots, n$ such that $\Delta_i \vdash t_i : C_i$. The corresponding *term rewriting rules for f* are $q_i \longrightarrow t_i$.

A program $\mathcal{P}$ over signature Σ is a function mapping each constant f to a cc-rule-set $\mathcal{P}_f$. We write $t \longrightarrow_{\mathcal{P}} t'$ for one-step reduction of term t to t' using the compatible closure[4] of the term rewriting rules in $\mathcal{P}$, and drop index $\mathcal{P}$ if clear from the context of discourse. We further write $\longrightarrow^{*}_{\mathcal{P}}$ for its tranistive and reflexive closure and $\longrightarrow^{\geq 1}_{\mathcal{P}}$ for its transitive closure.

Example (Deriving a cc-pattern-set for cyc*)* We start with $\mathrm{C_{Head}}$

$$\mathsf{cyc} : \mathsf{Nat} \to \mathsf{StrN} \;\triangleleft|\; (\cdot \vdash \mathsf{cyc} : \mathsf{Nat} \to \mathsf{StrN})$$

[4] See e.g. Def. 2.2.4 of [11].

We apply x to the head by $\mathrm{C}_{\mathrm{App}}$.

$$\mathsf{cyc} : \mathsf{Nat} \to \mathsf{StrN} \;\triangleleft\mid\; (x : \mathsf{Nat} \vdash \mathsf{cyc}\ x : \mathsf{StrN})$$

Then we split the result by $\mathrm{C}_{\mathrm{Dest}}$.

$$\mathsf{cyc} : \mathsf{Nat} \to \mathsf{StrN} \;\triangleleft\mid\; \begin{array}{l} (x : \mathsf{Nat} \vdash \mathsf{cyc}\ x\ .\mathsf{head} : \mathsf{Nat}) \\ (x : \mathsf{Nat} \vdash \mathsf{cyc}\ x\ .\mathsf{tail} \;\; : \mathsf{StrN}) \end{array}$$

In the second copattern, we split x using $\mathrm{C}_{\mathrm{Const}}$.

$$\mathsf{cyc} : \mathsf{Nat} \to \mathsf{StrN} \;\triangleleft\mid\; \begin{array}{llll} (x : \mathsf{Nat} & \vdash \mathsf{cyc}\ x & .\mathsf{head} & : \mathsf{Nat}) \\ (x : \mathbf{1} & \vdash \mathsf{cyc}\ (\mathsf{zero}\ x) & .\mathsf{tail} & : \mathsf{StrN}) \\ (x : \mathsf{Nat} & \vdash \mathsf{cyc}\ (\mathsf{suc}\ x) & .\mathsf{tail} & : \mathsf{StrN}) \end{array}$$

We finish by applying $\mathrm{C}_{\mathrm{Unit}}$ which replaces x by $()$ in the second clause.

$$\mathsf{cyc} : \mathsf{Nat} \to \mathsf{StrN} \;\triangleleft\mid\; \begin{array}{llll} (x : \mathsf{Nat} & \vdash \mathsf{cyc}\ x & .\mathsf{head} & : \mathsf{Nat}) \\ (\cdot & \vdash \mathsf{cyc}\ (\mathsf{zero}\ ()) & .\mathsf{tail} & : \mathsf{StrN}) \\ (x : \mathsf{Nat} & \vdash \mathsf{cyc}\ (\mathsf{suc}\ x) & .\mathsf{tail} & : \mathsf{StrN}) \end{array}$$

This concludes the derivation of the cc-pattern-set for the cyc function.

In Sect. 3 we prove a more general theorem referring to Abstract Reduction Systems (ARS). An *ARS* is a pair $(\mathcal{A}, \longrightarrow)$, often just written $\mathcal{A}$, such that $\mathcal{A}$ is a set and $\longrightarrow$ is a binary relation on $\mathcal{A}$ written infix. The notions $\longrightarrow^*$ and $\longrightarrow^{\geq 1}$ for ARS are defined as for programs above. The *ARS for* $\mathcal{P}$ *in context* Δ *and type* A is $(\mathsf{Term}_{\mathcal{P}}^{\Delta \vdash A}, \longrightarrow_{\mathcal{P}})$ where $\mathsf{Term}_{\mathcal{P}}^{\Delta \vdash A} = \{t \mid \Delta \vdash_{\Sigma} t : A\}$.

3 Reduction of Nested to Simple Pattern Matching

In the following, we describe a translation of deep (aka nested) (co)pattern matching (i.e. pattern matching as defined before) into shallow (aka non-nested) pattern matching, which we call *simple* pattern matching, as defined below. We are certainly not the first to describe such a translation, except maybe for copatterns, but we have special requirements for our translation. The obvious thing to ask for is *simulation*, *i.e.*, each reduction step in the original program should correspond to one or more reduction steps in the translated program. However, we want the translation also to *preserve and reflect normalization*: A term in the original program terminate, if and only if it terminates in the translated program. Preservation of normalization is important for instance in dependently typed languages such as Agda, where the translated programs are run during type checking and need to behave exactly like the original, user-written programs.

The strong normalization property is lost by some of the popular translations. For instance, translating rewrite rules to fixed-point and case combinators loses normalization, simply because fixed-point combinators reduce by themselves, allowing infinite reduction sequences immediately. But also special fixed-point combinators that only unfold if their principal argument is a constructor term, or dually cofixed-point combinators that only unfold if their result is observed[5] have such problems. Consider the following translation of a function f with deep matching into a function using such a fixed-point combinator.

$$\begin{array}{ll} f\ (\mathsf{z}()) & \longrightarrow \mathsf{z}() \\ f\ (\mathsf{s}\ (\mathsf{z}())) & \longrightarrow \mathsf{z}() \\ f\ (\mathsf{s}\ (\mathsf{s}\ x)) & \longrightarrow f\ (\mathsf{s}\ x)) \end{array} \rightsquigarrow f = \mathsf{fix} f(x).\ \mathsf{case}\ x\ \mathsf{of} \begin{cases} \mathsf{z}() & \longrightarrow \mathsf{z}() \\ \mathsf{s}\ (\mathsf{z}()) & \longrightarrow \mathsf{z}() \\ \mathsf{s}\ (\mathsf{s}\ x) & \longrightarrow f(\mathsf{s}\ x)) \end{cases}$$

While the term $f\,(\mathsf{suc}\,x)$ terminates for the original program simply because no pattern matches (i.e. no rewrite rule applies), it diverges for the translated program since the fixed-point applied to a constructor unfolds to a term containing the original term as a subterm. A closer look reveals that this special fixed-point combinator preserves normalization for simple pattern matching only.

A particular characteristic of term rewrite systems is that unfolding of recursion is tied to reduction by pattern matching to normal forms. In the following we develop a translation of deep patterns that maintains normalisation.

3.1 Simple patterns

A simple copattern q_{s} is of one of the forms $f\ \vec{x}$ (no matching), $f\ \vec{x}\ .d$ (shallow result matching) or $f\ \vec{x}\ p_{\mathsf{s}}$ (shallow argument matching) where $p_{\mathsf{s}} ::= () \mid (x_1, x_2) \mid c\,x$ is a simple pattern.

Definition 1 (Simple coverage-complete pattern sets).

(a) Simple cc-pattern-sets $f : A\ \lhd|_{\mathsf{s}}\ Q$ *are defined as follows ($\Delta = \vec{x} : \vec{A}$):*

$$\begin{array}{rl} f : \Delta \to A & \lhd|_{\mathsf{s}}\ (\Delta \vdash f\ \vec{x} : A) \\ f : \Delta \to \nu X.R & \lhd|_{\mathsf{s}}\ (\Delta \vdash f\ \vec{x}\ .d : (\nu X.R)_d)_{d \in R} \\ f : \Delta \to \mathbf{1} \to A & \lhd|_{\mathsf{s}}\ (\Delta \vdash f\ \vec{x}\ () : A) \\ f : \Delta \to (B_1 \times B_2) \to A & \lhd|_{\mathsf{s}}\ (\Delta, y_1 : B_1, y_2 : B_2 \vdash f\ \vec{x}\ (y_1, y_2) : A) \\ f : \Delta \to (\mu X.D) \to A & \lhd|_{\mathsf{s}}\ (\Delta, x' : (\mu X.D)_c \vdash f\ \vec{x}\ (c\ x') : A)_{c \in D} \end{array}$$

(b) A cc-rule-set is simple *if the underlying cc-pattern-set is simple. A constant in a program is* simple, *if its cc-rule-set is simple. A program is* simple *if all its constants are simple.*

Remark 2. If $f : A\ \lhd|_{\mathsf{s}}\ Q$ then $f : A\ \lhd|\ Q$.

[5] Such fixed-point combinators are used in the *Calculus of Inductive Constructions*, the core language of Coq [9], but have also been studied for sized types [4,1].

3.2 The translation algorithm by example

Neither the cyc function, nor the Fibonacci stream are simple. The translation into simple patterns requires the introduction of auxiliary constants, which are obtained as follows. We have a function for each occurrence of a splitting rule that is not $\mathrm{C}_{\mathrm{Head}}$, nor $\mathrm{C}_{\mathrm{App}}$. We can then start from the bottom of the derivation tree and, if the patterns are not simple, we eliminate this last derivation and create a new function. This function takes as arguments the variables we have not split on from the original function and this function (co)pattern matches just like the last step of the derivation did. Let us walk through the algorithm of transforming patterns into simple patterns for the cyc function.

$$\mathsf{cyc} : \mathsf{Nat} \to \mathsf{StrN} \ \lhd | \begin{array}{llll} (x : \mathsf{Nat} \vdash \mathsf{cyc}\ x & .\mathsf{head} & \longrightarrow x & : \mathsf{Nat}) \\ (\qquad\quad \vdash \mathsf{cyc}\ (\mathsf{zero}\ ()) & .\mathsf{tail} & \longrightarrow \mathsf{cyc}\ N & : \mathsf{StrN}) \\ (x : \mathsf{Nat} \vdash \mathsf{cyc}\ (\mathsf{suc}\ x) & .\mathsf{tail} & \longrightarrow \mathsf{cyc}\ x & : \mathsf{StrN}) \end{array}$$

In the derivation of coverage, the last rule was $\mathrm{C}_{\mathrm{Unit}}$ replacing pattern variable $x : \mathbf{1}$ by pattern $()$. We introduce a new constant g_2 with simple pattern and rule

$$g_2 : \mathbf{1} \to \mathsf{StrN} \ \lhd|_{\mathsf{s}} \ (\cdot \vdash g_2\ () \longrightarrow \mathsf{cyc}\ N : \mathsf{StrN}).$$

This symbol can be interpreted back in terms of the original program by a function int (defined more formally in Sect. 4) where $\mathsf{int}(g_2\ s) = \mathsf{cyc}\ (\mathsf{zero}\ \mathsf{int}(s))\ .\mathsf{tail}$ and int is otherwise the identity. The translated program arises by replacing the right hand side of the split clause with a call to g_2.

$$\mathsf{cyc} : \mathsf{Nat} \to \mathsf{StrN} \ \lhd | \begin{array}{llll} (x : \mathsf{Nat} \vdash \mathsf{cyc}\ x & .\mathsf{head} & \longrightarrow x & : \mathsf{Nat}) \\ (x : \mathbf{1} \quad\ \vdash \mathsf{cyc}\ (\mathsf{zero}\ x) & .\mathsf{tail} & \longrightarrow g_2\ x & : \mathsf{StrN}) \\ (x : \mathsf{Nat} \vdash \mathsf{cyc}\ (\mathsf{suc}\ x) & .\mathsf{tail} & \longrightarrow \mathsf{cyc}\ x & : \mathsf{StrN}) \end{array}$$

The second last step in the derivation of coverage was a split of pattern variable $x : \mathsf{Nat}$ into $\mathsf{zero}\ x$ and $\mathsf{suc}\ x$ using rule $\mathrm{C}_{\mathrm{Const}}$. Again, we introduce a simple auxiliary function, g_1, which performs just this split:

$$g_1 : \mathsf{Nat} \to \mathsf{StrN} \ \lhd|_{\mathsf{s}} \begin{array}{lll} (x : \mathbf{1} \quad\ \vdash g_1\ (\mathsf{zero}\ x) & \longrightarrow g_2\ x & : \mathsf{StrN}) \\ (x : \mathsf{Nat} \vdash g_1\ (\mathsf{suc}\ x) & \longrightarrow \mathsf{cyc}\ x & : \mathsf{StrN}) \end{array}$$

The interpretation of g_1 in terms of the original program would be $\mathsf{int}(g_1\ s) = \mathsf{cyc}\ \mathsf{int}(s)\ .\mathsf{tail}$ and the translated program is

$$\mathsf{cyc} : \mathsf{Nat} \to \mathsf{StrN} \ \lhd|_{\mathsf{s}} \begin{array}{lll} (x : \mathsf{Nat} \vdash \mathsf{cyc}\ x\ .\mathsf{head} & \longrightarrow x & : \mathsf{Nat}) \\ (x : \mathsf{Nat} \vdash \mathsf{cyc}\ x\ .\mathsf{tail} & \longrightarrow g_1\ x & : \mathsf{StrN}) \end{array}$$

This program is simple, thus the translation is finished, resulting in the mutually recursive functions cyc, g_1, and g_2. We note the following:

(a) The translation can be performed by induction on the derivation of coverage; or, one can do the translation while checking coverage.[6]

[6] This is actually happening in the language Idris [5]; Agda [10] has separate phases, but uses the split tree generated by the coverage checker to translate pattern matching into case trees.

(b) The generated functions are simple upon creation and need not be processed recursively. The right hand sides of these functions are either right hand sides of the original program or calls to earlier generated functions applied to exactly the pattern variables in context.
(c) When generating a function, it is invoked on the pattern variables in context. We can interpret this generated function in terms of the original program int, as a "backward-reduction".
(d) Since we gave earlier created functions (here: g_2) a higher index than later created functions (here: g_1), calls between generated functions increase the index. There can only be finitely many calls between generated functions before executing an original right hand side again. This fact ensures preservation of normalization (see later).
(e) Calls between generated functions are undone by the back translation int, thus the corresponding reduction steps vanish under int.

In the case of the Fibonacci stream, the translated simple program is as follows:

$$\begin{array}{llll} \mathsf{fib}\ .\mathsf{head} \longrightarrow 0 & \qquad & g\ .\mathsf{head} \longrightarrow 1 \\ \mathsf{fib}\ .\mathsf{tail}\ \ \longrightarrow g & \qquad & g\ .\mathsf{tail}\ \ \longrightarrow \mathsf{zipWith}\ \ \_+\_\ \ \mathsf{fib}\ (\mathsf{fib}\ .\mathsf{tail}) \end{array}$$

3.3 The translation algorithm

Let $\mathcal{P}$ be the input program. Let $\mathcal{P}_f$ be a non-simple rule of $\mathcal{P}$. Consider the last rule in the derivation of the underlying coverage. Since the rule is non-simple, it cannot be $\mathrm{C}_{\mathrm{Head}}$. Then, $\mathcal{P}_f$ is of the form

$$f : \Sigma(f) \ \lhd|\ Q\ (\Delta_i \vdash q_i \longrightarrow t_i : C_i)_{i \in I}.$$

where in some cases $I = \{0\}$. Let the last rule in the derivation of the underlying coverage be

$$\frac{f : \Sigma(f) \ \lhd|\ Q\ (\Delta' \vdash q : A)}{f : \Sigma(f) \ \lhd|\ Q\ (\Delta_i \vdash q_i : C_i)_{i \in I}}\ \mathsf{C}$$

We split on rule C and obtain a program $\mathcal{P}'$, in which the height of the derivation for f is reduced by 1, and a new constant with simple pattern matching is added. We then recursively apply the algorithm on $\mathcal{P}'$. In all cases, when we speak of a context "Δ", we refer to its variables by $\vec{x}$. Note that we always reorder Δ' such that the variable we split on appears last. Further, g always denotes a fresh constant.
In all cases $\mathcal{P}'$ is obtained from $\mathcal{P}$ by adding a rule for g and replacing one rule case for f by a simpler one that invokes g. Let $\Delta' = \vec{y} : \vec{A}$. For some q'_i to be determined in each case, we will define

$$\mathcal{P}'_h = \begin{cases} f : \Sigma(f) & \lhd| \quad Q\ (\Delta' \vdash q \longrightarrow g\ \vec{y} : A) & \text{if } h = f \\ g : \Delta' \to A & \lhd|_{\mathsf{s}} \ \ (\Delta_i \vdash q'_i \longrightarrow t_i : C_i)_{i \in I} & \text{if } h = g \\ \mathcal{P}_h & & \text{otherwise} \end{cases}$$

Case $q\ x \longrightarrow t$ and C is

$$\frac{f : \Sigma(f) \ \lhd|\ Q\ (\Delta \vdash q : B \to C)}{f : \Sigma(f) \ \lhd|\ Q\ (\Delta, x : B \vdash q\ x : C)}\ \mathrm{C_{App}}$$

Define $q'_0 = g\ \vec{x}\ x$. Therefore,

$$\begin{array}{ll} \mathcal{P}'_f = f : \Sigma(f) & \lhd|\ \ Q\ (\Delta \vdash q \longrightarrow g\ \vec{x} : B \to C) \\ \mathcal{P}'_g = g : \Delta \to B \to C & \lhd|_{\mathsf{s}}\ (\Delta, x : B \vdash g\ \vec{x}\ x \longrightarrow t : C) \end{array}$$

Case $q\ .d \longrightarrow t_d$ for all $d \in R$ and C is

$$\frac{f : \Sigma(f) \ \lhd|\ Q\ (\Delta \vdash q : \nu X.R)}{f : \Sigma(f) \ \lhd|\ Q\ (\Delta \vdash q\ .d : (\nu X.R)_d)_{d \in R}}\ \mathrm{C_{Dest}}$$

Define $q'_d = g\ \vec{x}\ .d$. Therefore,

$$\begin{array}{ll} \mathcal{P}'_f = f : \Sigma(f) & \lhd|\ \ Q\ (\Delta \vdash q \longrightarrow g\ \vec{x} : \nu X.R) \\ \mathcal{P}'_g = g : \Delta \to \nu X.R & \lhd|_{\mathsf{s}}\ (\Delta \vdash g\ \vec{x}\ .d \longrightarrow t_d : (\nu X.R)_d)_{d \in R} \end{array}$$

Case $q[x' := ()] \longrightarrow t$ and C is

$$\frac{f : \Sigma(f) \ \lhd|\ Q\ (\Delta, x' : \mathbf{1} \vdash q : C)}{f : \Sigma(f) \ \lhd|\ Q\ (\Delta \vdash q[x' := ()] : C)}\ \mathrm{C_{Unit}}$$

Define $q'_0 := g\ \vec{x}\ ()$. Therefore,

$$\begin{array}{ll} \mathcal{P}'_f \ = \ f : \Sigma(f) & \lhd|\ \ Q\ (\Delta, x' : \mathbf{1} \vdash q \longrightarrow g\ \vec{x}\ x' : C) \\ \mathcal{P}'_g := g : \Delta \to \mathbf{1} \to C & \lhd|_{\mathsf{s}}\ (\Delta \vdash g\ \vec{x}\ () \longrightarrow t : C) \end{array}$$

Case $q[x' := (x_1, x_2)] \longrightarrow t$ and C is

$$\frac{f : \Sigma(f) \ \lhd|\ Q\ (\Delta, x' : A_1 \times A_2 \vdash q : C)}{f : \Sigma(f) \ \lhd|\ Q\ (\Delta, x_1 : A_1, x_2 : A_2 \vdash q[x' := (x_1, x_2)] : C)}\ \mathrm{C_{Pair}}$$

Define $q'_0 = g\ \vec{x}\ (x_1, x_2)$. Therefore,

$$\begin{array}{ll} \mathcal{P}'_f \ = f : \Sigma(f) & \lhd|\ \ Q\ (\Delta, x' : A_1 \times A_2 \vdash q \longrightarrow g\ \vec{x}\ x' : C) \\ \mathcal{P}'_g := g : \Delta \to (A_1 \times A_2) \to C & \lhd|_{\mathsf{s}}\ (\Delta, x_1 : A_1, x_2 : A_2 \vdash g\ \vec{x}\ (x_1, x_2) \longrightarrow t : C) \end{array}$$

Case $q[x' := c\ x'] \longrightarrow t_c$ for all $c \in D$ and C is

$$\frac{A \ \lhd|\ Q\ (\Delta, x' : \mu X.D \vdash q : C)}{A \ \lhd|\ Q\ (\Delta, x' : (\mu X.D)_c \vdash q[x' := c\ x'] : C)_{c \in D}}\ \mathrm{C_{Const}}$$

Define $q'_c := g\ \vec{x}\ (c\ x')$. Therefore,

$$\begin{array}{ll} \mathcal{P}'_f = f : \Sigma(f) & \lhd|\ \ Q\ (\Delta, x' : \mu X.D \vdash q \longrightarrow g\ \vec{x}\ x' : C) \\ \mathcal{P}'_g = g : \Delta \to \mu X.D \to C & \lhd|_{\mathsf{s}}\ (\Delta, x' : (\mu X.D)_c \vdash g\ \vec{x}\ (c\ x') \longrightarrow t_c : C)_{c \in D} \end{array}$$

Since each step of the algorithm makes the coverage derivation of one non-simple function shorter, it terminates, returning only simple functions.

4 Proof of Correctness of the Translation

In our translation we extend our language by new auxiliary constants while keeping the old ones, including their types. More formally, we define $\Sigma \subseteq_{\mathsf{F}} \Sigma'$, pronounced *$\Sigma'$ extends Σ by constants*, if (1) Σ' and Σ have the same constructor and destructor symbols $\mathcal{C}, \mathcal{D}$, (2) the constants $\mathcal{F}$ of $\mathcal{L}$ form a subset of the constants of $\mathcal{L}'$, and (3) Σ and Σ' assign the same types to F.

It is easy to see that a reduction in the original program $\mathcal{P}$ (with signature Σ) corresponds to possibly multiple reductions in the translated language $\mathcal{P}'$ (with signature Σ'). What is more difficult to prove is that we do not get *additional* reductions, i.e., if $t \not\longrightarrow^*_{\mathcal{P}} t'$ then it is impossible to reduce t to t' using reductions and intermediate terms in $\mathcal{P}'$. We call this notion conservative extension. Even this will not be sufficient as pointed out in Sect. 3, we need in addition preservation of normalisation. We will define the corresponding notions more generally for ARSs.

Let $(\mathcal{A}, \longrightarrow)$ be an ARS, $a \in \mathcal{A}$. a is in *normal form (NF)* if there is no $a' \in \mathcal{A}$ such that $a \longrightarrow a'$. a is *weakly normalising (WN)* if there exists an $a' \in \mathcal{A}$ in NF such that $a \longrightarrow^* a'$. a is *strongly normalising (SN)* if there exist no infinite reduction sequence $a = a_0 \longrightarrow a_1 \longrightarrow a_2 \longrightarrow \cdots$. Let SN, WN, NF be the set of elements in $\mathcal{A}$ which are SN, WN, NF respectively. For a reduction system $(\mathcal{A}', \longrightarrow')$, let SN', WN', NF' be the elements of $(\mathcal{A}', \longrightarrow')$ which are SN, WN, NF.

Let $(\mathcal{A}, \longrightarrow)$, $(\mathcal{A}', \longrightarrow')$ be ARS such that $\mathcal{A} \subseteq \mathcal{A}'$. Then, we say that

$\mathcal{A}'$ is a conservative extension of $\mathcal{A}$ iff $\forall a, a' \in \mathcal{A}.\ a \longrightarrow^* a' \ \Leftrightarrow\ a {\longrightarrow'}^* a'$
$\mathcal{A}'$ is an SN-preserving extension of $\mathcal{A}$ iff $\forall a \in \mathcal{A}.\ a \in \mathsf{SN} \Leftrightarrow a \in \mathsf{SN}'$
$\mathcal{A}'$ is a WN-preserving extension of $\mathcal{A}$ iff $\forall a \in \mathcal{A}.\ a \in \mathsf{WN} \Leftrightarrow a \in \mathsf{WN}'$

Let $\mathcal{P}, \mathcal{P}'$ be programs for signatures Σ, Σ', respectively. $\mathcal{P}'$ is an extension of $\mathcal{P}$ iff $\Sigma \subseteq_{\mathsf{F}} \Sigma'$. If $\mathcal{P}'$ is an extension of $\mathcal{P}$, then $\mathcal{P}'$ is a *conservative*, *SN-preserving*, or *WN-preserving* extension of $\mathcal{P}$ if the corresponding condition holds for the ARSs $(\mathsf{Term}_{\Sigma}^{\Delta \vdash A}, \longrightarrow_{\mathcal{P}})$ and $(\mathsf{Term}_{\Sigma'}^{\Delta \vdash A}, \longrightarrow_{\mathcal{P}'})$.

Lemma 3 (Transitivity of conservative and SN/WN-preserving extensions). Assume $\mathcal{P}'$ is an extension of $\mathcal{P}$ and $\mathcal{P}''$ an extension of $\mathcal{P}'$, both of which are conservative, SN-preserving, and WN-preserving extensions. Then $\mathcal{P}''$ is a conservative, SN-preserving, and WN-preserving extension of $\mathcal{P}$. The same holds for ARSs instead of programs.

In order to show the above properties for our translation of programs, we use the notion of a back-translation from the translated language into the original language. We define this notion more generally for ARS:

Let $(\mathcal{A}, \longrightarrow)$, $(\mathcal{A}', \longrightarrow')$ be ARSs such that $\mathcal{A} \subseteq \mathcal{A}'$. Then a *back-interpretation of $\mathcal{A}'$ into $\mathcal{A}$* is given by

- a set Good such that $\mathcal{A} \subseteq \mathsf{Good} \subseteq \mathcal{A}'$; we say *$a$ is good* if $a \in \mathsf{Good}$;
- a function $\mathsf{int} : \mathsf{Good} \to \mathcal{A}$ such that $\forall a \in \mathcal{A}.\mathsf{int}(a) = a$.

We define 3 conditions for a back-interpretation $(\mathsf{Good}, \mathsf{int})$ where condition (SN 2) refers to a measure $\mathsf{m} : \mathsf{Good} \to \mathbb{N}$:

(SN 1) $\forall a, a' \in \mathcal{A}. a \longrightarrow a' \Rightarrow a \longrightarrow'^{\geq 1} a'$.

(SN 2) If $a \in \mathsf{Good}$, $a' \in \mathcal{A}'$ and $a \longrightarrow' a'$ then $a' \in \mathsf{Good}$ and we have $\mathsf{int}(a) \longrightarrow^{\geq 1} \mathsf{int}(a')$ or $\mathsf{int}(a) = \mathsf{int}(a') \wedge \mathsf{m}(a) > \mathsf{m}(a')$.

(WN) If $a \in \mathsf{Good} \cap \mathsf{NF}'$ then $\mathsf{int}(a) \in \mathsf{NF}$.

Lemma 4. Assume $(\mathcal{A}, \longrightarrow)$, $(\mathcal{A}', \longrightarrow')$ be ARS such that $\mathcal{A} \subseteq \mathcal{A}'$. Let $(\mathsf{Good}, \mathsf{int})$ be a back-interpretation from $\mathcal{A}'$ into $\mathcal{A}$, $\mathsf{m} : \mathsf{Good} \to \mathbb{N}$. Then the following holds:

(a) (SN 1), (SN 2) imply that $\mathcal{A}'$ is a conservative extension of $\mathcal{A}$ preserving SN.
(b) (SN 1), (SN 2), (WN) imply that $\mathcal{A}'$ is an extension of $\mathcal{A}$ preserving WN.

Proof: (a): Proof of Conservativity: $a \longrightarrow^* a'$ implies by (SN 1) $a \longrightarrow'^* a'$. If $a, a' \in \mathcal{A}$, $a \longrightarrow'^* a'$ then by (SN 2) $a = \mathsf{int}(a) \longrightarrow^* \mathsf{int}(a') = a'$.
Proof of preservation of SN: We show the classically equivalent statement
$\forall a \in \mathcal{A}. \neg(a \text{ is } \longrightarrow\text{-SN}) \Leftrightarrow \neg(a \text{ is } \longrightarrow'\text{-SN})$.
For "$\Rightarrow$" assume $a = a_0 \longrightarrow a_1 \longrightarrow a_2 \longrightarrow \cdots$ is an infinite $\longrightarrow$-reduction sequence starting with a. Then by (SN 1) $a = a_0 \longrightarrow'^{\geq 1} a_1 \longrightarrow'^{\geq 1} a_2 \longrightarrow'^{\geq 1} \cdots$ is an infinite $\longrightarrow'$-reduction sequence.
For "$\Leftarrow$" assume $a = a'_0 \longrightarrow' a'_1 \longrightarrow_{\mathcal{P}'} a'_2 \cdots$.
Then by (SN 2) $a = \mathsf{int}(a_0) = \mathsf{int}(a'_0) \longrightarrow^* \mathsf{int}(a'_1) \longrightarrow^* \mathsf{int}(a'_2) \longrightarrow'^* \cdots$. If $\mathsf{int}(a'_i) = \mathsf{int}(a'_{i+1})$ then $\mathsf{m}(a'_i) > \mathsf{m}(a'_{i+1})$, so by (SN 2) after finitely many steps where $\mathsf{int}(a'_i) = \mathsf{int}(a'_{i+1})$ we must have one step $\mathsf{int}(a'_j) \longrightarrow^{\geq 1} \mathsf{int}(a'_{j+1})$. Thus, we obtain an infinite reduction sequence starting with a in $\mathcal{A}$.
(b) Assume $a \in \mathcal{A}$, $a \in \mathsf{WN}$. Then $a \longrightarrow^* a' \in \mathsf{NF}$ for some a', therefore $a' \in \mathsf{SN}$, by (a) $a' \in \mathsf{SN}'$, $a' \longrightarrow'^* a''$ for some $a'' \in \mathsf{NF}'$, therefore $a \longrightarrow'^* a' \longrightarrow'^* a'' \in \mathsf{NF}'$, $a \in \mathsf{WN}'$. For the other direction, assume $a \in \mathcal{A}$, $a \in \mathsf{WN}'$. Then $a \longrightarrow'^* a' \in \mathsf{NF}'$ for some a', by (SN 2), (WN) $a = \mathsf{int}(a) \longrightarrow^* \mathsf{int}(a') \in \mathsf{NF}$, $a \in \mathsf{WN}$.

Our concrete back-interpretations are obtained by replacing in terms $g\, t_1 \ldots t_n$ the new constants g by a term of the original language. Due to lack of λ-abstraction, we only get a term of the original language if g is applied to n arguments. So, for our back translation, we need an $\mathsf{arity}(g) = n$ of new constants, and an interpretation $\mathsf{Int}(g)$ of those terms:

Assume $\Sigma \subseteq_{\mathsf{F}} \Sigma'$. A *concrete back-interpretation* $(\mathsf{arity}, \mathsf{Int})$ *of* Σ' *into* Σ is given by the following:

- An arity $\mathsf{arity}(g) = n$ assigned to each new constant g of Σ' such that $\Sigma'(g) = A_1 \to \cdots \to A_n \to A$ for some types $A_1, \ldots, A_n, A$. Here, A (as well as any A_i) might be a function type.
- For every new constant g of Σ' with $\Sigma'(g) = A_1 \to \cdots \to A_n \to A$, $\mathsf{arity}(g) = n$ a term $\mathsf{Int}(g) = t$ of Σ such that $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$. In this case, we write $\mathsf{Int}(g)[\vec{t}]$ for $t[\vec{x} := \vec{t}]$.

Assume that $(\mathsf{arity}, \mathsf{Int})$ is a concrete back-interpretation of Σ' into Σ, Δ, A a context and type for Σ.

- The set $\mathsf{Good}^{\Delta,A}_{\mathsf{arity},\mathsf{Int}}$ of good terms (written briefly Good) is given by the set of $t \in \mathsf{Term}^{\Delta\vdash A}_{\Sigma}$ such thateach occurrence of a new constant g of arity n in t is applied to at least n arguments.
- If $t \in \mathsf{Good}^{\Delta,A}_{\mathsf{arity},\mathsf{Int}}$, then $\mathsf{int}^{\Delta,A}_{\mathsf{arity},\mathsf{Int}}(t)$, in short $\mathsf{int}(t)$, is obtained by inductively replacing all occurrences of $g\,\vec{t}$ for new constants g by $\mathsf{Int}(g)[\mathsf{int}(\vec{t})]$.

Trivially, concrete back-interpretations are back-interpretations. We now have the definitions in place to prove SN+WN-conservativity of our translation.

Lemma 5 (Some simple facts).

(a) If $f : A \;\triangleleft\,|\; Q\ (\Delta \vdash q : A)$ then each variable in Δ occurs exactly once in q.
(b) If x is a variable occurring in pattern q, then t is a subterm of $q[x := t]$.
(c) Assume s is a maximal subterm of t, i.e. s is a subterm such that there is no term s' such that $s\ s'$ is a subterm starting at the same occurrence as s in t. If t is good, then s is good as well.

Theorem 6. Let $\mathcal{P}$ be a program for the signature Σ. Then there exists a typed language Σ' which extends Σ and a simple program $\mathcal{P}'$, such that $\mathcal{P}'$ is a conservative extension of $\mathcal{P}$ preserving SN and WN.

Proof: Define for a program $\mathcal{P}$ the height of its derivation $\mathsf{height}(\mathcal{P})$ as the sum of the heights of the derivations of those covering patterns in $\mathcal{P}$, which are not simple covering patterns. The proof is by induction on $\mathsf{height}(\mathcal{P})$.

The case $\mathsf{height}(\mathcal{P}) = 0$ is trivial, since $\mathcal{P}$ is simple. Assume $\mathsf{height}(\mathcal{P}) > 0$. We obtain a $\Sigma' \supseteq_{\mathsf{F}} \Sigma$ and corresponding program $\mathcal{P}'$ for Σ' by applying one step of Algorithm 3.3 to $\mathcal{P}$. We show below that $\mathcal{P}'$ is a conservative extension of $\mathcal{P}$ preserving SN and WN. Since the derivations for the covereage complete pattern sets in $\mathcal{P}'$ are the same as for $\mathcal{P}$, except for the one for $\mathcal{P}'_f$, which is reduced in height by one as the algorithm takes out the last derivation of the coverage derivation of $\mathcal{P}_f$, and that for $\mathcal{P}'_g$, which is simple, we have $\mathsf{height}(\mathcal{P}') = \mathsf{height}(\mathcal{P}) - 1$. By IH there exists a conservative extension $\mathcal{P}''$ of $\mathcal{P}'$ preserving SN and WN, which is simple, which is as well a conservative extension of $\mathcal{P}$ preserving SN and WN. This extension is obtained by the recursive call made by the algorithm.

We are going to show that $\mathcal{P}'$ is a conservative extension of $\mathcal{P}$ preserving SN and WN. Let $f, g, \Delta', \vec{y}, q, A, I, \Delta_i, q_i, t_i, C_i, q'_i$ be as stated in Algorithm 3.3, $\Delta_i = \vec{y}_i : \vec{A}_i$, and n be the length of Δ'.

We introduce a concrete back-interpretation of $\mathcal{P}'$ into $\mathcal{P}$ defined by $\mathsf{arity}(g) := n$ and $\mathsf{Int}(g)[\vec{y}] := q$. Let $\mathsf{m}(t)$ be the number of occurrences of f in t. Assume $\Delta \vdash A$ a context and type of Σ, and let $(\mathsf{Good}^{\Delta,A}, \mathsf{int}^{\Delta,A})$ be the corresponding back interpretation, for which we briefly write (Good, A).

Assume $\mathcal{P}'$ fulfills with the given q'_i the following conditions:

(1) $\mathsf{int}(q'_i) = q_i \longrightarrow_{\mathcal{P}'} q'_i$

(2) If $q[\vec{x} := \vec{s}]\ \vec{t} = q_i$, then $g\ \vec{s}\ \vec{t} = q'_i$.

Then $(\mathsf{Good}, \mathsf{int})$ fulfils (SN 1), (SN 2), and (WN), and therefore $\mathcal{P}'$ is a conservative extension of $\mathcal{P}$ preserving SN and WN:

(SN 1) holds since the only changed derivation is based on the original redex $q_i[\vec{y}_i := \vec{t}] \longrightarrow_{\mathcal{P}} t_i[\vec{y}_i := \vec{t}]$ and $q_i[\vec{y}_i := \vec{t}] \longrightarrow_{\mathcal{P}'} q'_i[\vec{y}_i := \vec{t}] \longrightarrow_{\mathcal{P}'} t_i[\vec{y}_i := \vec{t}]$.
(SN 2) holds since the new redexes are the following:

(a) $q[\vec{y} := \vec{t}] \longrightarrow_{\mathcal{P}'} g\ \vec{t}$, where $q[\vec{y} := \vec{t}]$ is good. Since it is good and variables in a pattern are not applied to other terms, by Lem. 5 $\vec{t}$ is good as well, and therefore as well $g\ \vec{t}$. We have $\mathsf{int}(g\ \vec{t}) = q[\vec{y} := \mathsf{int}(\vec{t})] = \mathsf{int}(q[\vec{y} := \vec{t}])$. Furthermore, $\mathsf{m}(q[\vec{y} := \vec{t}]) = \mathsf{m}(g\ \vec{t}) + 1 > \mathsf{m}(g\ \vec{t})$, since pattern q starts with f, and each variable in $\vec{y}$ occurs by Lem. 5 exactly once in q.

(b) $q'_i[\vec{y}_i := \vec{t}] \longrightarrow_{\mathcal{P}'} t_i[\vec{y}_i := \vec{t}]$. Since $q'_i[\vec{y}_i := \vec{t}]$ is good, as in (a) $\vec{t}$ is good and therefore $t_i[\vec{y}_i := \vec{t}]$ is good. Furthermore, $\mathsf{int}(q'_i[\vec{y}_i := \vec{t}]) = \mathsf{int}(q'_i)[\vec{y}_i := \mathsf{int}(\vec{t})]) = q_i[\vec{y}_i := \mathsf{int}(\vec{t})] \longrightarrow_{\mathcal{P}} t_i[\vec{y}_i := \mathsf{int}(\vec{t})] = \mathsf{int}(t_i[\vec{y}_i := \vec{t}])$.

Proof of (WN): We first show that (2) implies
(3) If $s \in \mathsf{Good}$, $\mathsf{int}(s) = q_i$ then $s = q_i \vee s = q'_i$
Since q_i starts with f, s must start with f or g. The only occurrence of a function symbol in q_i is at the beginning, therefore $s = f\ \vec{r}$ or $s = g\ \vec{r}$ where $\mathsf{int}(\vec{r}) = \vec{r}$. If $s = f\ \vec{r}$ then $\mathsf{int}(s) = s = q_i$. If $s = g\ \vec{r} = g\ \vec{s}\ \vec{t}$, $\mathsf{int}(s) = q[\vec{x} := \vec{s}]\ \vec{t} = q_i$, therefore by (2) $s = g\ \vec{s}\ \vec{t} = q'_i$.

Using (3), assume $s \in \mathsf{Good}$, $s \in \mathsf{NF}'$, and assume $\mathsf{int}(s)$ had redex $\widetilde{q}[\vec{x} := \vec{r}]$ for a pattern $\widetilde{q}$ of $\mathcal{P}$. If $\widetilde{q} \neq q_i$, $\widetilde{q}$ starts with some $h \neq f, g$, and has no occurrences of f, g. Then s contains $\widetilde{q}[\vec{x} := \vec{r}']$ where $\mathsf{int}(\vec{r}') = \vec{r}$, and has therefore a redex, contradicting $s \in \mathsf{NF}'$. Therefore $\widetilde{q} = q_i$ for some i. Therefore s contains a subterm $s'[\vec{x} := \vec{r}']$ such that $\mathsf{int}(s') = q_i$, $\mathsf{int}(\vec{r}') = \vec{r}$. But then by (3) $s'[\vec{x} := \vec{r}']$ is a redex of s, again a contradiction.

So the proof is complete provided conditions (1), (2) are fulfilled. These follow straightforwardly from the definition of the translation algorithm.

5 Conclusion

We have described a reduction of deep copattern matching to shallow copattern matching. The translation preserves normalization and thus establishes a weak bisimulation between original and translated program. The translated programs can be used for more efficient evaluation in a checker for dependent types or can serve as intermediate code for translation into a more low-level language that has no concept of pattern at all.

Two more translations might be worthwhile investigating in future work: First, a translation into a variable-free language of combinators. The challenge here lies in the preservation of normalization as in the present translation. Secondly, a translation to a call-by-need lambda-calculus with lazy record constructors. This would allow us to map definitions of infinite structures by copatterns back to Haskell style definitions by lazy evaluation. While there seems to be no (weak) bisimulation in this case, one still hope for preservation of normalization, maybe established by logical relations.

Acknowledgments Anton Setzer acknowledges support by EPSRC (Engineering and Physical Science Research Council, UK) grant EP/C0608917/1. Andreas Abel acknowledges support by a Vetenskapsrådet framework grant 254820104 (Thierry Coquand) to the Department of Computer Science and Engineering at Gothenburg University. Brigitte Pientka achknowledges support by NSERC (National Science and Engineering Research Council Canada). David Thibodeau acknowledges support by a graduate scholarship of Les Fonds Québécois de Recherche Nature et Technologies (FQRNT).

References

1. Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.
2. Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *Proc. of the 40th ACM Symp. on Principles of Programming Languages, POPL 2013*, pages 27–38. ACM Press, 2013.
3. Lennart Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture (FPCA'85)*, volume 201 of *Lect. Notes in Comput. Sci.*, pages 368–381. Springer, 1985.
4. Gilles Barthe, Maria J. Frade, Eduardo Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Math. Struct. in Comput. Sci.*, 14(1):97–141, 2004.
5. Edwin Brady. Implementation of a general purpose programming language with dependent types. Available on the author's homepage, 2014.
6. Robin Cockett and Tom Fukushima. About Charity. Technical report, Department of Computer Science, The University of Calgary, 1992. Yellow Series Report No. 92/480/18.
7. Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science*, volume 283 of *Lect. Notes in Comput. Sci.*, pages 140–157. Springer, 1987.
8. Tatsuya Hagino. Codatatypes in ML. *J. Symb. Logic*, 8(6):629–650, 1989.
9. INRIA. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.4 edition, 2012.
10. Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Dept of Comput. Sci. and Engrg., Chalmers, Göteborg, Sweden, 2007.
11. Terese. *Term Rewriting Systems*. Cambridge University Press Cambridge, 2003.