Rippling on Relational Structures

Vincent Lombart[1] and **Yves Deville**
Département d'Ingénierie Informatique
Université Catholique de Louvain
Place Ste Barbe, 2, B-1348 Louvain-la-Neuve
vl@info.ucl.ac.be   yde@info.ucl.ac.be

Abstract

The rippling heuristic has been rather successfully used to guide inductive proofs in a functional framework. We present here how that heuristic can be formalized in a relational framework, more adapted to logic programming.

The rippling heuristic is first analysed to identify its key components. A formalization of these to a relational framework is then presented. The applicability and the power of our methods is illustrated on two examples: the simulation of the functional rippling's behaviour, and program transformation.

Keywords: Program synthesis and transformation; theorem proving; proof control

1 Introduction

Rippling is a heuristic for guiding inductive proofs [Bundy *et al* 93] and has been rather successfully used in the automatic planning [Bundy 88] of inductive proofs in a functional framework.

Conventional rippling[2] has also been applied in logic program synthesis and transformation by means of the proofs-as-programs approach [Bates & Constable 85, Bundy *et al* 90, Wiggins 92b]. However, the relational nature of logic programming pinpoints the weakness of conventional rippling for dealing with relational expressions. Conventional rippling makes an extensive use of information contained in the syntactic structure of a functional expression. Conventional rippling is basically a tool for functional structures. Used in the logic programming framework, it cannot take into account links via shared variables or operator commutativity, for instance.

In this paper, we present rippling on relational structures. The objective is to provide a rippling heuristic that can be successfully applied on relations. When used in the logic programming context, this will enlarge the class of logic programs automatically synthesisable by means of rippling. The approach followed in the development of the proposed relational rippling was to capture the basic concepts of rippling, then extend and enrich those key concepts into the relational framework.

Several comparisons can be made between conventional rippling and relational rippling[3]:

- The proposed relational rippling is as powerful as functional rippling (in terms of provable theorems): relational rippling can reproduce conventional rippling (see section 4). Moreover, the space searched is exactly the same.

[1] supported by the Belgian National Fund for Scientific Research

[2] In this paper, *conventional rippling* will refer to the heuristic described in [Bundy *et al* 93]

[3] By *relational rippling*, we mean rippling on relational structures, as proposed in this paper.

- The proposed relational rippling allows reasoning in terms of relations rather than functions.
- Conventional rippling relies on the syntactic structure of functional expressions, while the proposed relational rippling takes into account links between expressions in different formulae by means of shared variables. The application of rewriting rules is therefore more complex in relational rippling, but better reflects the logic programming point of view and extends the class of problems where this heuristic can be applied.
- The syntax used for relational structures is close to the body of a Horn clause. Hence, if relational rippling is coupled with a proofs-as-programs logic program synthesis system such as *Whelk*[Wiggins 92b], the proposed proof planning technique can be used for program verification, synthesis and transformation rather easily.
- Conventional rippling uses the functionality information contained in the syntax to control the movement of wave fronts. The proposed relational rippling generalizes this idea by explicitly separating the expressions and properties of those expressions (such as functional dependencies). Those properties, put in explicit form, should allow the use of various types of information to control and guide the rippling process.

Several solutions have been proposed to use the rippling heuristic on relational expressions, mainly by T. Åhs and G. Wiggins [Åhs & Wiggins 94, Åhs 95]. While they have tackled the problem in a general way, we have constrained ourselves to a system in which the functional rippling behaviour is just a particular case, ending up with a different rippling control. The performances of both systems have not yet been extensively compared. More recently, A. Bundy has also investigated some problems linked to the rippling on relational expressions[4].

Another remedy to rippling's poor behaviour on relational structures is the development of *unblocking* techniques [Lombart *et al* 93], which let the rippling process guide most of a proof, but which come into action when it is blocked, to unblock it, using some techniques outside of rippling's scope. Those techniques work fairly well on mixed structures (partially nested, partially flat), but are insufficient for completely flat (relational) structures, basically because rippling is then unable to guide any part of the proof. Another approach using conventional rippling for logic program synthesis can be found in [Kraan *et al* 93, Kraan 94].

In the following sections, we first bring forward the underlying principles of the rippling heuristic and identify its key components. The new rippling on relational structure is then presented and formalized. Two applications are then developed. First, we show how conventional rippling can be reproduced with the proposed relational rippling. Next, our relational rippling heuristic is used in the transformation of a logic program.

The logic used in this paper is that of the *Whelk* system [Wiggins 92b], a first order, typed, constructive logic with equality, based on a Gentzen sequent calculus. Type information will be omitted when not necessary for understanding.

[4] Personal communications.

2 Rippling

The rippling process is basically a clever and structured application of rewriting rules. In this section, we present two key aspects of rippling that make it a successful heuristic, namely focussed rewriting and movement control. Some differences between functional rippling and rippling on relational structures are already underlined. A more complete presentation on functional rippling can be found in [Bundy *et al* 93].

2.1 Focussed Rewriting

A first aspect of rippling is that it focusses rewriting on well-identified parts of an expression. Basically, it tries to match a conjecture (that is, a theorem to be proved) with a given hypothesis by successive rewritings. Rippling focusses the rewriting on *wave fronts*, *i.e.* differences between the conjecture and the hypothesis, so as to remove them, and hence to prove the conjecture.

This is best illustrated on the step case of inductive proofs, where something such as

$$p(x) \vdash p(\boxed{s(\underline{x})}) \tag{1}$$

or

$$p(x), \boxed{succ(x, x')} \vdash p(x') \tag{2}$$

has to be proved. The syntactic difference (called *wave front*) between the hypotheses (left hand side of $\vdash$) and the conclusion (right hand side of $\vdash$) is what is boxed and not underlined. The first form (1) is expressed in the conventional functional framework, and the second one (2) is in a relational form.

To keep the focus on the wave fronts, these are annotated by the above boxes. Rewriting rules are then considered as "moving" rules: their role is to move the wave fronts. Rewriting rules are annotated to identify the wave fronts they move, and are called *wave rules*:

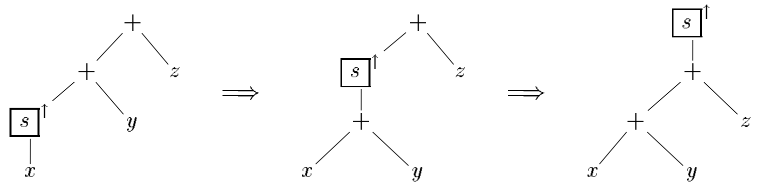
$$\boxed{s(\underline{x})} = \boxed{s(\underline{y})} \implies x = y \tag{3}$$

$$f(\boxed{h(\underline{x})}) \implies \boxed{g(\underline{f(x)})} \tag{4}$$

$$\exists v \,.\, \boxed{p(x, v)} \wedge q(v, y) \implies \exists v' \,.\, q(x, v') \wedge \boxed{r(v', y)} \tag{5}$$

The two first rewriting rules are usual wave rules in the functional framework. The last one is expressed in a relational framework.

It is important to notice that the application of wave rules can transform the wave front while moving it; it can even disappear. But in all cases, the rest of the expression remains the same. Wave rules should move the wave front in a controlled way. With the condition to match the expression as well as the annotations to use a wave rule, the effect is to only move the wave front (difference) in a fixed structure, called the skeleton, which is the image of the hypothesis. This prevents from using rewriting rules where unnecessary or unhelpful.



$$(\boxed{s(\underline{x})}^{\uparrow} + y) + z \quad\Longrightarrow\quad \boxed{s(\underline{x+y})}^{\uparrow} + z \quad\Longrightarrow\quad \boxed{s(\underline{(x+y)+z})}^{\uparrow}$$

Figure 1: Rippling-out: functional version

2.2 Movement Control

If the rewriting process can be viewed as moving a wave front in a skeleton, it is important to give it a well defined direction and to avoid loops. In the conventional rippling, the solution used is to connect the direction to the syntactic tree of the functional skeleton: the wave fronts can move up, down, or sideways in that tree. The initial direction is up, but one sideways move is allowed to change it from up to down, basically to move the wave front down to a *sink*, the place of a universally quantified variable in the hypothesis, which can by definition absorb anything. With a finite skeleton, this means the wave front is moved to the top, to one of the sinks, is removed or is blocked in a finite number of steps. Such a process can be graphically represented as in Fig. 1, with a little arrow to indicate the direction. Such arrows will also be annotations on rewriting rules as control information to guide the rippling process. While this works very well when the syntactic tree has a functional, deeply nested structure, it doesn't work with relational expressions without functional structures, such as

$$p(x, y, z) \wedge q(z, t) \wedge r(t, x, v) \tag{6}$$

where the link between the conjuncts is based on shared variables, and not on the syntactic structure.

This shows one of the problems to be solved when extending rippling to relational structures. In Section 3.2, we will explain how links between shared variables can be used to guide the rippling on relational structures.

2.3 Example

We illustrate conventional rippling on the proof of

$$(x + y) + z = x + (y + z) \tag{7}$$

We have the following rewriting rules:

$$0 + x \Longrightarrow x \tag{8}$$

$$\boxed{s(\underline{x})}^{\uparrow} + y \Longrightarrow \boxed{s(\underline{x+y})}^{\uparrow} \tag{9}$$

$$\boxed{s(\underline{x})}^{\uparrow} = \boxed{s(\underline{y})}^{\uparrow} \Longrightarrow x = y \tag{10}$$

The conjecture to prove is:

$$\vdash (x+y)+z = x+(y+z) \tag{11}$$

In an inductive proof with x as induction variable, the base case is proved by simplification using (8), and the step case is:

$$(x+y)+z = x+(y+z) \vdash (\boxed{s(\underline{x})}^{\uparrow} + y) + z = \boxed{s(\underline{x})}^{\uparrow} + (y+z) \tag{12}$$

The wave fronts are successively rippled using (9) twice then (10).

$$\vdash \boxed{s(\underline{(x+y)})}^{\uparrow} + z = \boxed{s(\underline{x+(y+z)})}^{\uparrow} \tag{13}$$

$$\vdash \boxed{s(\underline{(x+y)+z})}^{\uparrow} = \boxed{s(\underline{x+(y+z)})}^{\uparrow} \tag{14}$$

$$\vdash (x+y)+z = x+(y+z) \tag{15}$$

and the induction hypothesis can be used. The successive transformations of the left hand side are illustrated in Fig. 1.

3 Relational Structures

This section presents and formalizes the new rippling on relational structures.

3.1 ε-expressions

As relational expressions will be treated, let us first formalize the class of expressions that will be handled.

Definition 1 *An ε-expression is a (sub)expression with the following form:*

$$\exists \overline{v} \,.\, \bigwedge_i p_i(\overline{v_i}) \tag{16}$$

with $\overline{v_i} \subseteq \overline{v} \cup \overline{x} \cup \mathcal{C}$,
where $\overline{v}$ are the bound variables, $\overline{x}$ the free variables, and $\mathcal{C}$ the set of constants of the underlying language.

Those expressions are thus flat: there are only variables and constants as predicate arguments.

The form of ε-expressions is not as general as first-order formulae, but present several advantages which have guided our choice. ε-expressions already have a high expressiveness; their form is close to the body of a Horn clause (conjunction of atoms). This form will also keep rewriting simple (and thus automatable). ε-expression will also provide a possible simulation of the conventional rippling.

The form of ε-expressions does not exclude the use of recursive data structures (such as lists). Lists will be treated trough a constant ($[\,]$) and a predicate ($cons(Head, Tail, List)$). A straightforward rewrite can transform such predicates into usual notations.

3.2 Rippling Control

External representation The conventional rippling uses the syntactic tree of a functional structure as the reference skeleton for wave front movements. This is not possible for ε-expressions. We show here how to construct a graph, external representation for ε-expressions, which can play the same role.

Definition 2 *The external representation, or reference skeleton of an ε-expression $\exists \overline{v} \ . \ \bigwedge_i p_i(\overline{v_i})$ is a graph, whose nodes are the predicates p_i, the free and bound variables and the constants. The vertices bind each p_i to its arguments v_{i_j}.*

Example 1 *The external representation of $\exists v \ . \ p_g(x, v) \wedge p_f(v, y)$ is (the existentially quantified variable is encircled):*

$$\begin{tikzcd} & p_g \arrow[dl, dash] \arrow[dr, dash] & & p_f \arrow[dl, dash] \arrow[dr, dash] & \\ x & & v & & y \end{tikzcd} \tag{17}$$

Control annotations In this paper, we will only consider that a wave front is a single conjunct of an ε-expression (rather than groups of such conjuncts). The next problem is the guiding of wave fronts in the reference skeleton. A complete discussion of that subject is far beyond the scope of this paper, but it is important to keep some advantages of the successful conventional rippling, mainly the termination property, the large applicability and the reduced search space.

The experimental solution we propose attributes an in/out pattern to each relation: each parameter of a predicate is labelled with an *in*($\downarrow$) or *out*($\uparrow$) direction, for instance

$$p(\uparrow, \downarrow, \uparrow) \tag{18}$$

and the graphical representation of an ε-expression is directed by binding each p_i to its *in* arguments by a vertex from the argument to the predicate, and to its *out* arguments by a vertex from the predicate to the argument.

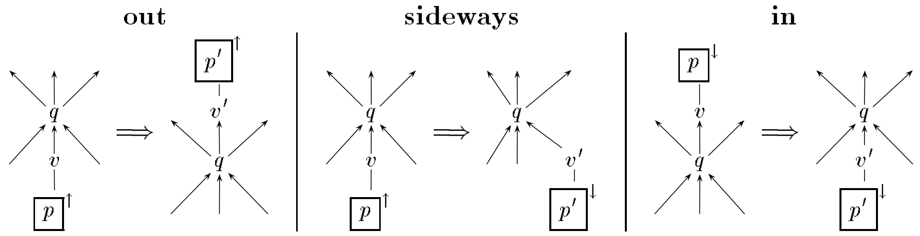
Example 2 *The external representation of $\exists v \ . \ p_g(x, v) \wedge p_f(v, y)$ with $p_g(\downarrow, \uparrow)$ and $p_f(\downarrow, \uparrow)$ is:*

$$\begin{tikzcd} & p_g \arrow[dr] & & p_f \arrow[dr] & \\ x \arrow[ur] & & v \arrow[ur] & & y \end{tikzcd} \tag{19}$$

Definition 3 *Let $\boxed{p(\ldots, v, \ldots)}$ a wave front, $q(\ldots, v, \ldots)$ a conjunct of the skeleton, such that p and q are linked by a shared variable v, the* position of p w.r.t. q *is* ***in*** *if the vertex between q and v is directed from v to q, and* ***out*** *if it is directed from q to v.*

Notice that a wave front can occupy several different positions w.r.t. the same conjunct (if more than one shared variable), or w.r.t. different conjuncts.

This definition allows us to define the *in* and *out* directions for a wave front at each point of the directed graph. A move in the *out* direction is a move from an *in* position to an *out* position w.r.t. the same conjunct of the skeleton. The *sideways* (from *in* to *in*) and *in* (from *out* to *in*) directions are correspondingly defined. This can be illustrated:



The control will follow the conventional rippling behaviour: a wave front can be moved out, then sideways, then in, in the skeleton, and the current direction is indicated by a little arrow. The direction is only based on the skeleton's directed vertices. To ensure that a wave front can be totally rippled in a finite number of steps, it is then sufficient to ensure that the skeleton is cycle-free.

Those definitions can be extended to moves involving multiple wave fronts and multiple conjuncts of the skeleton, as long as such a move follows the directed vertices of the skeleton in a coherent way (*out*, *sideways* or *in* direction). In the other cases, the move is simply forbidden. Also, moves which can be considered in different manners because of multiple shared variables are forbidden to avoid cycling.

All these moves can be expressed without actually building the graph, by only identifying the position of a wave front before and after the move.

Choice of control annotations The last problem is the choice of a good *in/out* pattern for each predicate. As "rules-of-thumb":

- in the case where a relation is actually a function, that is, one of its arguments is functionally determined from the other, a good choice is to label that argument with *out* and all the other arguments with *in*;
- if a relation is recursively defined, the recursive arguments can generally be labelled with *in*;
- this *in/out* labelling bears some similarities with the $+/-/?$ labelling of a Prolog procedure; this could probably be used [Wiggins 92a], and will be explored;
- a labelling which does not allow some rewriting rules because of incoherent directions should be avoided.

3.3 Rewriting Rules

The rewriting rules we use for rippling on relational structures will be based on axioms of the form:

$$P_1 \leftrightarrow P_2 \tag{20}$$

where P_1 and P_2 are both ε-expressions. The rewriting rules corresponding to such axioms will be

$$P_1 \Longrightarrow P_2 \tag{21}$$

$$P_2 \Longrightarrow P_1 \tag{22}$$

From each rewriting rule, several wave rules can be further derived, depending on the added annotations. These added annotations specify the wave front, and the chosen control.

To derive a wave rule, a skeleton is first chosen as a non-empty subset of the predicates which appear both in P_1 and P_2. The chosen *in/out* labelling (see section 3.2) is then used to try to determine the direction of the candidate wave rule. If no coherent direction is determined, the rule is rejected, otherwise it is added with its annotations (wave front and direction) to the wave rule base.

Example 3 *The rewriting rule*

$$\exists v \;.\; succ(x,v) \wedge plus(v,y,z) \Longrightarrow \exists v' \;.\; plus(x,y,v') \wedge succ(v',z) \tag{23}$$

can be annotated, by choosing $plus(\ldots)$ *as skeleton:*

$$\exists v \;.\; \boxed{succ(x,v)} \wedge plus(v,y,z) \Longrightarrow \exists v' \;.\; plus(x,y,v') \wedge \boxed{succ(v',z)} \tag{24}$$

The direction is determined from the in/out *pattern chosen for plus, and more precisely for its first* (v) *and third* (v') *arguments:*

First argument	*Third argument*	*Direction*
in	in	sideways
in	out	out
out	in	in
out	out	*undefined*

This gives us a well defined wave rule. The same could have been done by choosing succ as the skeleton, giving us a second wave rule. Both rules can coexist in the wave rule base.

The main differences with the conventional rippling is that the direction depends here on the chosen *in/out* patterns, and allows several *out* arguments in a relation.

In order to apply rewriting rules on ε-expressions, the concept of "matching subexpression" in the functional sense should be extended as the order and the relative position of disjuncts in a ε-expression are irrelevant. Special care should also be provided to quantified variables to narrow the scope of the necessary existential quantifiers down to the matching subexpression.

The application of a wave rule could be decomposed into three steps:

1. the implied conjuncts of the conjecture are grouped, and the scope of the implied (existential) quantifiers is narrowed to that group;

2. the actual rewriting is performed, with a correct instantiation of the rule;

3. the scope of the quantifiers is broadened again to include the whole conjecture.

Example 4 *The application of*

$$\exists v\ .\ \boxed{succ(x,v)}^{\uparrow} \wedge plus(v,y,z) \Longrightarrow \exists v'\ .\ plus(x,y,v') \wedge \boxed{succ(v',z)}^{\uparrow} \tag{25}$$

to

$$\exists v_1, v_2\ .\ \boxed{succ(x,v_1)}^{\uparrow} \wedge plus(v_2,z,w) \wedge plus(v_1,y,v_2) \tag{26}$$

could be decomposed into

1. *grouping and narrowing*
$$\exists v_2\ .\ plus(v_2,z,w) \wedge [\exists v_1\ .\ \boxed{succ(x,v_1)}^{\uparrow} \wedge plus(v_1,y,v_2)] \tag{27}$$
2. *application of (25)*
$$\exists v_2\ .\ plus(v_2,z,w) \wedge [\exists v_1'\ .\ plus(x,y,v_1') \wedge \boxed{succ(v_1',v_2)}^{\uparrow}] \tag{28}$$
3. *broadening*
$$\exists v_2, v_1'\ .\ plus(v_2,z,w) \wedge plus(x,y,v_1') \wedge \boxed{succ(v_1',v_2)}^{\uparrow} \tag{29}$$

Although those steps are underlying the application of a wave rule, at the planning level the whole is considered an atomic step, and the applicability is checked without building the intermediate steps.

It has been experimentally noticed that in the relational framework we use, very few rules actually remove a wave front, but rather transform it into simple equalities between variables or constants, which has nearly the same effect. Indeed, those equalities are then eliminated by the following simplification wave rule:

$$\exists v\ .\ \boxed{v = x} \wedge P(v) \Longrightarrow P(x) \tag{30}$$

3.4 Induction Schemes

Rippling is particularly well adapted to inductive proofs. The objective is here the possible use of the induction hypothesis in the proof of the inductive step. The introduction of a wave front (difference between conjecture and hypothesis) occurs at a well determined stage, the inductive step, and that wave front is immediately identified as the *step* relation (see below).

In the relational framework, the induction step of a proof will have the form:

$$p(x), \boxed{step(x,x')} \vdash p(x') \tag{31}$$

with the particularly important cases of the naturals (based on the successor relation) and the lists (based on list construction from head and tail):

$$p(x), \boxed{succ(x,x')} \ \vdash\ p(x') \tag{32}$$

$$p(t), \boxed{cons(h,t,l)} \ \vdash\ p(l) \tag{33}$$

As usual, wave fronts (i.e. the content of the boxes) annotate the difference between the hypothesis and the conclusion to be proved. In a relational framework, the position of the wave front raises a problem: the wave front is a hypothesis, and does not appear in the conjecture to prove, which hampers the rewriting. Two solutions are available. The first [Åhs & Wiggins 94], general, is to use conditional wave rules, that is, rewriting rules that can only be used under some hypotheses. This can lead to having two forms of the same rule to maintain, one when the wave front is in the hypothesis, and one when it is in the conjecture. We develop here a second solution which avoids this problem. This solution is applicable when the *step* predicate verifies (which is often the case):

$$step(x, x') \wedge step(x, x'') \rightarrow x' = x'' \tag{34}$$

then (31) can be transformed into

$$p(x), step(x, x') \vdash \exists x'' \,.\, \boxed{step(x, x'')} \wedge p(x'') \tag{35}$$

where the wave front now appears in the conclusion. This is a particular case of the following lemma, formally treated in the appendix:

Lemma 1 *If there is at most one t such that $q(t)$, that is:*

$$\forall x, y \,.\, q(x) \wedge q(y) \Rightarrow x = y \tag{36}$$

then

$$q(t) \quad \vdash \quad (p(t) \Leftrightarrow \exists x \,.\, q(x) \wedge p(x)) \tag{37}$$

If x' appears in several places, various applications of Lemma 1 can lead to different transformations (e.g. a unique application; an application for each occurrence of x'). For instance,

$$p(x) \wedge q(x), \boxed{step(x, x')} \vdash p(x') \wedge q(x') \tag{38}$$

can be transformed into one of those forms:

$$p(x) \wedge q(x), step(x, x') \quad \vdash \quad \exists x'' \,.\, \boxed{step(x, x'')} \wedge p(x'') \wedge q(x'') \tag{39}$$

$$\begin{aligned} p(x) \wedge q(x), step(x, x') \quad \vdash \quad & \exists x'', x''' \,.\, \boxed{step(x, x'')} \wedge \boxed{step(x, x''')} \\ & \wedge p(x'') \wedge q(x''') \end{aligned} \tag{40}$$

A condition for the application of a rewriting rule is that the scope of existentially quantified variables must be local to the conjuncts under rewriting. This means that a rule such as

$$\exists v \,.\, \boxed{step(x, v)} \wedge p(v) \Longrightarrow \ldots \tag{41}$$

can be used to rewrite the right side of (40), but is not applicable for (39). On the other hand, a rule such as

$$\exists v \,.\, \boxed{step(x, v)} \wedge p(v) \wedge q(v) \Longrightarrow \ldots \tag{42}$$

can be used to rewrite the right side of (39), but is not applicable for (40). This is undesirable as it would ruin our objective (a unique rewriting rule). Fortunately, in the case where lemma 1 can be applied, (42) can also be expressed with two independent existentially quantified variables:

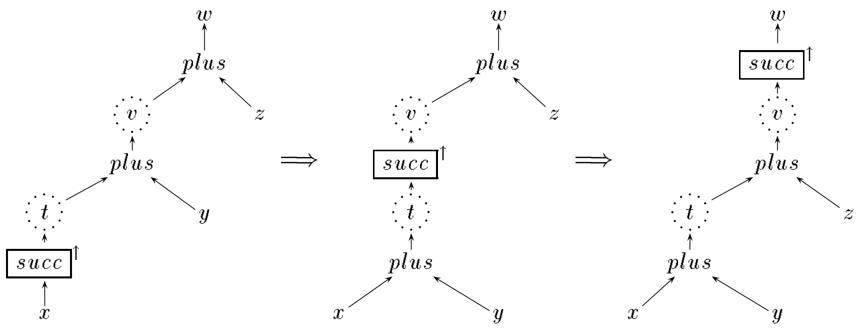


Figure 2: Rippling-out: relational version

$$\exists v, v' . \boxed{step(x, v)} \wedge \boxed{step(x, v')} \wedge p(v) \wedge q(v') \Longrightarrow \ldots \quad (43)$$

We will thus keep independent variables in the induction step conclusion as well as in the rewriting rules.

4 Application: Conventional Rippling

In this section, we show how the conventional rippling behaviour can be reproduced, step by step, by the proposed rippling on relational structures. This is shown for equational theorems and for well formed logic formulae. Conventional rippling is simulated in the sense that, for each conventional proof state, we give a corresponding state in our proof (with relational structures). Fig. 2 is a graphical illustration of such a correspondence. It shows the corresponding proof states (from a proof on relational structures) to the selected proof states in Fig.1 (conventional rippling).

4.1 Equations

We first present how to reproduce the conventional rippling behaviour when the conjecture is an equation and the rewriting rules are transformations from one functional expression to another.

Equation Transformation The equation containing a certain level of function nesting, such as

$$f(x, g(y)) = h(z) \quad (44)$$

should first be transformed into an equivalent ε-expression, such as

$$\exists v_1, v_2 . p_g(y, v_1) \wedge p_f(x, v_1, v_2) \wedge p_h(z, v_2) \quad (45)$$

This is easily done in three steps:

- each n-ary function f is replaced by a $(n+1)$-ary relation p_f, which takes the same arguments as f plus an extra $(n+1)$th argument to express the result, under the hypothesis:

$$p_f(\overline{x}, z) \leftrightarrow f(\overline{x}) = z \tag{46}$$

- an existentially quantified variable is introduced to express the equal result;
- existentially quantified variables are introduced to express function nesting.

An alternative to (45) is to have an equivalence between ε-expressions, such as

$$\forall v \, . \, [\exists v_1 \, . \, p_g(y, v_1) \wedge p_f(x, v_1, v) \leftrightarrow p_h(z, v)] \tag{47}$$

This formulation would more easily allow weak fertilization (replacing in the conjecture one side of the induction hypothesis with the other), but can harden some parts of the rippling. In either case, both alternatives (47) and (45) are actually logically equivalent (see proof in appendix, Lemma 2), and the transformation from one to the other is always possible.

Wave Rule Transformation The technique is nearly identical to the one explained in the previous section, transforming

$$f(\boxed{h(\underline{x})}^{\uparrow}) \Longrightarrow \boxed{g(\underline{f(x)})}^{\uparrow} \tag{48}$$

into

$$\exists v \, . \, \boxed{p_h(x, v)}^{\uparrow} \wedge p_f(v, z) \Longrightarrow \exists v' \, . \, p_f(x, v') \wedge \boxed{p_g(v', z)}^{\uparrow} \tag{49}$$

The only addition is the treatment of annotations: what was enclosed in the expression is now along it. The direction (arrow) is kept.

Rippling Control The rippling control is considerably easier than in the general case: as each relation is actually a function, we label the last argument with *out* and the other arguments with *in*. This gives us a direction corresponding to the *up* direction in the syntactic tree, and the graph is then obviously cycle-free. This is illustrated in Fig. 2.

A further advantage here is that the existentially quantified variables are only used twice, once in an *in* position, and once in an *out* position. The direction can then be deduced from the wave front itself: the *out* direction is its own *out* direction.

Example The simulation of conventional rippling is illustrated on the proof of $(x + y) + z = x + (y + z)$ developed in Section 2.3.

The corresponding rewriting rules in a relational framework are:

$$plus(0, x, y) \Longrightarrow x = y \tag{50}$$

$$\exists v \, . \, \boxed{succ(x, v)}^{\uparrow} \wedge plus(v, y, z) \Longrightarrow \exists v' \, . \, plus(x, y, v') \wedge \boxed{succ(v', z)}^{\uparrow} \tag{51}$$

$$\exists v \, . \, \boxed{succ(x, v)}^{\uparrow} \wedge \boxed{succ(y, v)}^{\uparrow} \Longrightarrow \boxed{x = y}^{\uparrow} \tag{52}$$

The conjecture to prove is:

$$\vdash \exists v_1, v_2, v_3 \; . \; plus(x, y, v_1) \wedge plus(v_1, z, v_2) \wedge plus(y, z, v_3) \wedge plus(x, v_3, v_2) \quad (53)$$

In an inductive proof with x as induction variable, the base case is proved by simplification using (50) (and the simplification rewriting rule (30)), and the step case is:

$$\exists v_1, v_2, v_3 \; . \; plus(x, y, v_1) \wedge plus(v_1, z, v_2) \wedge plus(y, z, v_3) \\ \wedge \; plus(x, v_3, v_2) \quad (54)$$

$$\boxed{succ(x, x')} \quad (55)$$

$$\vdash \exists v_1, v_2, v_3 \; . \; plus(x', y, v_1) \wedge plus(v_1, z, v_2) \wedge plus(y, z, v_3) \\ \wedge \; plus(x', v_3, v_2) \quad (56)$$

The wave front is introduced twice in the conjecture, as explained in Section 3.4:

$$\vdash \exists v_1, v_2, v_3, x'', x''' \; . \; plus(x'', y, v_1) \wedge plus(v_1, z, v_2) \wedge plus(y, z, v_3) \\ \wedge \; plus(x''', v_3, v_2) \wedge \boxed{succ(x, x'')}^{\uparrow} \wedge \boxed{succ(x, x''')}^{\uparrow} \quad (57)$$

The wave fronts are successively rippled using (51) twice then (52) and finally a simplification step. Notice how a wave front moves in the *out* direction, towards the predicate sharing its last argument.

$$\vdash \exists v_1, v_2, v_3, x'', x''' \; . \; plus(x, y, x'') \wedge \boxed{succ(x'', v_1)}^{\uparrow} \wedge plus(v_1, z, v_2) \\ \wedge \; plus(y, z, v_3) \wedge plus(x, v_3, x''') \wedge \boxed{succ(x''', v_2)}^{\uparrow} \quad (58)$$

$$\vdash \exists v_1, v_2, v_3, x'', x''' \; . \; plus(x, y, x'') \wedge plus(x'', z, v_1) \wedge \boxed{succ(v_1, v_2)}^{\uparrow} \\ \wedge \; plus(y, z, v_3) \wedge plus(x, v_3, x''') \wedge \boxed{succ(x''', v_2)}^{\uparrow} \quad (59)$$

$$\vdash \exists v_1, v_3, x'', x''' \; . \; plus(x, y, x'') \wedge plus(x'', z, v_1) \wedge plus(y, z, v_3) \\ \wedge \; plus(x, v_3, x''') \wedge \boxed{v_1 = x'''}^{\uparrow} \quad (60)$$

$$\vdash \exists x'', v_1, v_3 \; . \; plus(x, y, x'') \wedge plus(x'', z, v_1) \wedge plus(y, z, v_3) \\ \wedge \; plus(x, v_3, v_1) \quad (61)$$

and the induction hypothesis can be used.

Let us now analyze and compare the above proof with the corresponding proof using conventional rippling (Section 2.3). The conventional rippling proof is simpler, but there is an equivalence between the steps of each proof. The control is the same, but expressed and used in slightly different ways. The complexity of the relational rippling proof is intrinsic to the relational structure of the expressions.

From a human point of view, the proof with conventional rippling is thus easier to perform, but from a computer-automation perspective, the relational form can be seen as syntactic sugar. The major advantage of the relational approach is that beside its capacity to simulate conventional rippling, it can also be used in the proof of purely relational expressions, hence enlarging the class of formulae where rippling could be applied.

4.2 Well Formed Logic Formulae

Conventional rippling basically considers a formula such as

$$\exists z \ . \ p(x, y, z) \land q(z, t) \land r(t, x, v) \tag{62}$$

just as it considers

$$exists(z, and(p(x, y, z), and(q(z, t), r(t, x, v)))) \tag{63}$$

which is syntactically the same, but in a prefix notation and with named symbols. As has been said before, this limits the conventional rippling: it does not take into account shared variables or the commutativity of operators.

The relational rippling does not have those limitations, and can deal with (62) in a natural way, as explained earlier. However, to be complete, it can even reproduce that crippled behaviour, by considering predicates as boolean-valued functions, logical operators as boolean-valued functions on two boolean arguments, and quantifiers as boolean-valued functions on a typed variable and another boolean argument. This means

$$\forall x \ . \ p(x) \rightarrow q(x) \lor \neg r(x) \tag{64}$$

can be transformed into the ε-expression

$$\begin{aligned} &\exists v_1, v_2, v_3, v_4, v_5, v_6 : \mathit{boole} \ . \\ &\quad p_\forall(x, v_1, v) \land p_\rightarrow(v_2, v_3, v_1) \land p_p(x, v_2) \land p_\lor(v_4, v_5, v_3) \\ &\quad \land p_q(x, v_4) \land p_\neg(v_6, v_5) \land p_r(x, v_6) \end{aligned} \tag{65}$$

Some remarks:

- this is really awkward;
- for type consistency, there must be a functional quantifier ($p_\forall$, $p_\exists$) for each quantifiable type;
- although this seems to mix logics of different orders, it is safe and is used in the conventional rippling; it would of course be better to do it properly, by using relational rippling directly;
- At this stage, this is a possible way of handling wff's that do not quite fit into ε-expressions. Of course, for ε-expressions, it is more appropriate to use the framework we have developed.

5 Application: Program Transformation

In this section, we will illustrate our technique on the transformation of an inefficient `delete(X,Old,New)` procedure definition (stating that the list `New` is the list `Old` where one occurrence of `X` has been removed) to an efficient one, using the *Whelk* program synthesis system [Wiggins 92b]. We will later compare this approach with a similar proof presented in [Lombart *et al* 93]. We start with the definition:

```
delete(X, Old, New) ← app(L1, L2, Old), L2=[X|L2t],
                      app(L1, L2t, New).
```

where `app` is the usual list append.

We can easily transform that into a *Whelk* synthesis conjecture:

$$\vdash \forall x, old, new \,.\, \partial(\exists l_1, l_2, l_{2t} \,.\, app(l_1, l_2, old) \wedge cons(x, l_{2t}, l_2) \wedge app(l_1, l_{2t}, new)) \tag{66}$$

using a list construction relation *cons* and the decidability operator ∂ [Wiggins 92b]. This operator ensures that a program can be extracted from the proof. A (sub)proof can be completed by reducing what is in its scope to a tautology, a contradiction or an equality between (variables or constants).

We will need the following rewriting rules:

$$append([], x, y) \implies x = y \tag{67}$$

$$\exists v \,.\, \boxed{cons(h, t, v)}^{\uparrow} \wedge app(v, x, y) \implies \exists v' \,.\, \boxed{cons(h, v, y)}^{\uparrow} \wedge app(t, x, v) \tag{68}$$

$$app(x, y, []) \implies x = [] \wedge y = [] \tag{69}$$

$$\exists v \,.\, cons(x_1, x_2, v) \wedge cons(y_1, y_2, v) \implies x_1 = y_1 \wedge x_2 = y_2 \tag{70}$$

Definition (68) implies for *app* the *in* direction for the first argument and the *out* direction for the third. The direction of the second argument can be ignored here, but will usually be *in*.

We will also need a case-splitting *proof critic* [Ireland 92] that can be easily adapted to this framework. Basically, it is a method suggesting a promising case split when no more rewriting is possible [Lombart *et al* 93].

We begin with an induction on *old*, suggested by an appropriate recursion analysis. The base case

$$\vdash \forall x, new \,.\, \partial(\exists l_1, l_2, l_{2t} \,.\, app(l_1, l_2, []) \wedge cons(x, l_{2t}, l_2) \wedge app(l_1, l_{2t}, new)) \tag{71}$$

is transformed into

$$\vdash \forall x, new \,.\, \partial(\exists l_1, l_2, l_{2t} \,.\, l_1 = nil \wedge l_2 = [] \wedge cons(x, l_{2t}, l_2) \wedge app(l_1, l_{2t}, new)) \tag{72}$$

which leads to the impossibility $cons(x, l_{2t}, [])$

The step case is

$$\forall x, new \,.\, \partial(\exists l_1, l_2, l_{2t} \,.\, app(l_1, l_2, old_t) \wedge cons(x, l_{2t}, l_2) \wedge app(l_1, l_{2t}, new)) \tag{73}$$

$$\boxed{cons(old_h, old_t, old)}^{\uparrow} \tag{74}$$

$$\vdash \partial(\exists l_1, l_2, l_{2t} \,.\, app(l_1, l_2, old) \wedge cons(x, l_{2t}, l_2) \wedge app(l_1, l_{2t}, new)) \tag{75}$$

We introduce the wave front in the conjecture, as explained in section 3.4:

$$\vdash \partial(\exists l_1, l_2, l_{2t}, old' \,.\, \boxed{cons(old_h, old_t, old')}^{\uparrow} \wedge app(l_1, l_2, old') \wedge cons(x, l_{2t}, l_2) \wedge app(l_1, l_{2t}, new)) \tag{76}$$

The rippling is blocked, as the only argument of the wave front which is shared by another conjunct, old', is in an *out* position (see section 3.2). The proof critic suggests a case split on l_1:

$$l_1 = [\,] \vee \exists l_{1h}, l_{1t} \, . \, cons(l_{1h}, l_{1t}, l_1) \tag{77}$$

In the first case ($l_1 = [\,]$), we get:

$$\vdash \partial(\exists l_2, l_{2t}, old' \, . \, \boxed{cons(old_h, old_t, old')}^{\uparrow} \wedge app([\,], l_2, old') \tag{78}$$

$$\wedge\, cons(x, l_{2t}, l_2) \wedge app([\,], l_{2t}, new)) \tag{79}$$

which is immediately transformed into

$$\vdash \partial(\exists l_2, l_{2t}, old' \, . \, \boxed{cons(old_h, old_t, old')}^{\uparrow} \wedge l_2 = old' \tag{80}$$

$$\wedge\, cons(x, l_{2t}, l_2) \wedge l_{2t} = new) \tag{81}$$

and simplified into

$$\vdash \partial(\exists l_2 \, . \, \boxed{cons(old_h, old_t, l_2)}^{\uparrow} \wedge cons(x, new, l_2)) \tag{82}$$

$$\vdash \partial(\exists l_2 \, . \, old_h = x \wedge old_t = new) \tag{83}$$

In the second case ($\exists l_{1h}, l_{1t} \, . \, cons(l_{1h}, l_{1t}, l_1)$), we get (this needs to be introduced twice, as for an induction):

$$\begin{aligned} \vdash \partial(\exists l_{1a}, l_{1b}, l_{1h}, l_{1t}, l_2, l_{2t}, old' \, . \, & \boxed{cons(l_{1h}, l_{1t}, l_{1a})}^{\uparrow} \\ \wedge \boxed{cons(l_{1h}, l_{1t}, l_{1b})}^{\uparrow} \wedge & \boxed{cons(old_h, old_t, old')}^{\uparrow} \wedge app(l_{1a}, l_2, old') \\ \wedge\, cons(x, l_{2t}, l_2) \wedge & app(l_{1b}, l_{2t}, new)) \end{aligned} \tag{84}$$

which allows a double rippling, as l_{1a} and l_{1b} are both connected to an *in* argument of *app*:

$$\begin{aligned} \vdash \partial(\exists l'_{1a}, l'_{1b}, l_{1h}, l_{1t}, l_2, l_{2t}, old' \, . \, & \boxed{cons(l_{1h}, l'_{1a}, old')}^{\uparrow} \\ \wedge \boxed{cons(l_{1h}, l'_{1b}, new)}^{\uparrow} \wedge & \boxed{cons(old_h, old_t, old')}^{\uparrow} \wedge app(l_{1t}, l_2, l'_{1a}) \\ \wedge\, cons(x, l_{2t}, l_2) \wedge & app(l_{1t}, l_{2t}, l'_{1b})) \end{aligned} \tag{85}$$

which can be simplified into:

$$\begin{aligned} \vdash \partial(\exists l'_{1a}, l'_{1b}, l_{1h}, l_{1t}, l_2, l_{2t} \, . \, & l_{1h} = old_h \wedge l'_{1a} = old_t \\ \wedge \boxed{cons(l_{1h}, l'_{1b}, new)}^{\uparrow} \wedge & app(l_{1t}, l_2, l'_{1a}) \wedge cons(x, l_{2t}, l_2) \\ \wedge\, app(l_{1t}, l_{2t}, l'_{1b})) & \end{aligned} \tag{86}$$

$$\begin{aligned} \vdash \partial(\exists l'_{1b}, l_{1t}, l_2, l_{2t} \, . \, & \boxed{cons(old_h, l'_{1b}, new)}^{\uparrow} \wedge app(l_{1t}, l_2, old_t) \\ \wedge\, cons(x, l_{2t}, l_2) \wedge & app(l_{1t}, l_{2t}, l'_{1b})) \end{aligned} \tag{87}$$

As the rewriting is again blocked, the case-splitting method suggests:

$$new = [\,] \vee \exists new_h, new_t \, . \, cons(new_h, new_t, new) \tag{88}$$

which gives an impossibility in the case $new = [\,]$, because of $cons(old_h, l'_{1b}, [\,])$ and in the case $\exists new_h, new_t \;.\; cons(new_h, new_t, new)$, we must again introduce the new wave front in the conjecture's conclusion, to give:

$$\exists new_h, new_t \;.\; cons(new_h, new_t, new) \tag{89}$$

$$\begin{aligned}\vdash \partial(\exists l'_{1b}, l_{1t}, l_2, l_{2t}, new' \;.\; & \boxed{cons(new_h, new_t, new')}^{\uparrow} \\ & \wedge \boxed{cons(old_h, l'_{1b}, new')}^{\uparrow} \wedge app(l_{1t}, l_2, old_t) \wedge cons(x, l_{2t}, l_2) \\ & \wedge app(l_{1t}, l_{2t}, l'_{1b}))\end{aligned} \tag{90}$$

which is simplified into

$$\begin{aligned}\vdash \partial(\exists l'_{1b}, l_{1t}, l_2, l_{2t} \;.\; & \boxed{new_h = old_h}^{\uparrow} \wedge new_t = l'_{1b} \wedge app(l_{1t}, l_2, old_t) \\ & \wedge cons(x, l_{2t}, l_2) \wedge app(l_{1t}, l_{2t}, l'_{1b}))\end{aligned} \tag{91}$$

$$\begin{aligned}\vdash \partial(\exists l_{1t}, l_2, l_{2t} \;.\; & \boxed{new_h = old_h}^{\uparrow} \wedge app(l_{1t}, l_2, old_t) \wedge cons(x, l_{2t}, l_2) \\ & \wedge app(l_{1t}, l_{2t}, new_t))\end{aligned} \tag{92}$$

and the wave front can be removed from the ε-expression:

$$\begin{aligned}\vdash \boxed{\partial(new_h = old_h)}^{\uparrow} \wedge \partial(\exists l_{1t}, l_2, l_{2t} \;.\; & app(l_{1t}, l_2, old_t) \wedge cons(x, l_{2t}, l_2) \\ & \wedge app(l_{1t}, l_{2t}, new_t))\end{aligned} \tag{93}$$

where the induction hypothesis can be used. From the above proof, a logic program can be extracted. The extraction process, not described in this paper (see [Wiggins 92b, Lombart *et al* 93]), yields the following logic program:

```
delete(X, Old, New) ←
   cons(Old_h,Old_t,Old)
   ∧ (   Old_h=X ∧ Old_t=New
       ∨ cons(New_h,New_t,New) ∧ New_h=Old_h
                                ∧ delete(X, Old_t, New_t) )
```

which can syntactically be rewritten in a more conventional Prolog style as:

```
delete(X, [X|New], New).
delete(X, [Old_h|Old_t], [Old_h|New_t]) ←
               delete(X, Old_t, New_t).
```

We now have obtained an obviously less wasteful[5] version of the original program, which illustrates the potential use of our system for logic program transformation. We can compare this with the synthesis proof of [Lombart *et al* 93], where the same program had been synthesised with the conventional rippling. We have been able to use relational expressions, which is needed to take the initial program as input, whereas the conventional rippling needs a manual construction of a specification, in a functional notation. The two different rippling controls (conventional and relational) have mainly led to the same proof scheme, using different elements to guide them; that had to be expected because of the simplicity of the proof. Once again, the syntax is more human-readable in the functional case, but identically computer-readable in both cases.

[5] in terms of needed computational resources

6 Conclusion and Further Work

In this paper, we have shown how the basic ideas of rippling can be transferred to a relational context. A new rippling method working on a particular type of relation expressions, called ε-expressions, has been proposed, and the necessary framework we have developed has been explained. The desirable properties of rippling have been kept, and a rippling control has been proposed in that sense. How the original rippling behaviour can be simulated through an adequate choice of transformations has been described, and our methods have been illustrated on a program transformation example.

The new form of rippling we have developed has several advantages over the conventional one: it is able to deal with relations; it is more applicable to program transformation; it does not rely on the syntactic structure of expressions; it extends the class of automatically synthesisable programs; and it is still able to plan every proof the conventional rippling can plan. On the same proof (*i.e.* a proof the conventional rippling can prove), the only drawback our method has is a more complex control to cope with its extended power.

Several improvements are still to be done. Our rippling control method, based on an *in/out* pattern for each predicate, should be extended to allow multiple patterns, and the choice of a pattern should be automated. Other control methods are also currently investigated. The syntax of ε-expressions should be extended as much as possible, to tend towards general well formed formulae. Several small improvements, such as the possibility of functional terms, rather than terms limited to variables or constants, have already been considered.

Acknowledgements

Many thanks to Geraint Wiggins for proof-reading several versions of this document, and to Alan Bundy for some useful discussions.

Appendix

Lemma 1 *If there is at most one t such that $q(t)$, that is:*

$$\forall x, y \,.\, q(x) \wedge q(y) \Rightarrow x = y \tag{94}$$

then

$$q(t) \quad \vdash \quad (p(t) \Leftrightarrow \exists x \,.\, q(x) \wedge p(x)) \tag{95}$$

Proof. First, the ($\Rightarrow$) direction:

$$q(t), p(t) \vdash \exists x \,.\, q(x) \wedge p(x) \tag{96}$$

This is of course proved by taking t as existential witness.

Second, the ($\Leftarrow$) direction:

$$q(t) \tag{97}$$

$$\exists x \,.\, q(x) \wedge p(x) \tag{98}$$

$$\vdash p(t) \tag{99}$$

We can name the existential witness of (98) x_0, to get:

$$q(t), q(x_0), p(x_0) \vdash p(t) \tag{100}$$

From $q(t)$ and $q(x_0)$, and from (94), we know that $t = x_0$, and thus $p(t)$. ∎

Lemma 2 *If p_f and p_g are defined from the mappings $f : \tau_1 \rightarrow \tau_2$ and $g : \tau_1 \rightarrow \tau_2$ by $p_f(x, y) \Leftrightarrow f(x) = y$ and $p_g(x, y) \Leftrightarrow g(x) = y$, then*

$$(\forall x : \tau_1 \,.\, \exists y : \tau_2 \,.\, p_f(x, y) \wedge p_g(x, y)) \tag{101}$$
$$\Leftrightarrow (\forall x : \tau_1 \,.\, \forall y : \tau_2 \,.\, p_f(x, y) \Leftrightarrow p_g(x, y)) \tag{102}$$

If we omit the type information for clarity, that is:

$$(\forall x \,.\, \exists y \,.\, p_f(x, y) \wedge p_g(x, y)) \Leftrightarrow (\forall x \,.\, \forall y \,.\, p_f(x, y) \Leftrightarrow p_g(x, y)) \tag{103}$$

Proof. First, the $(\Rightarrow)$ direction:

$$\forall x \,.\, \exists y \,.\, p_f(x, y) \wedge p_g(x, y) \tag{104}$$
$$\vdash \forall x \,.\, \forall y \,.\, p_f(x, y) \Leftrightarrow p_g(x, y) \tag{105}$$

To prove (105), it is sufficient to prove it for the arbitrary values $x_0 : \tau_1$ and $y_0 : \tau_2$, that is, it is sufficient to prove:

$$\vdash p_f(x_0, y_0) \Leftrightarrow p_g(x_0, y_0) \tag{106}$$

We can instantiate (104) with the value x_0 and name the existential witness y_1, to get the new hypothesis:

$$p_f(x_0, y_1) \wedge p_g(x_0, y_1) \tag{107}$$

To prove the $(\Rightarrow)$ part of (106) (the other part is similar), we thus have to prove:

$$p_f(x_0, y_1) \tag{108}$$
$$p_g(x_0, y_1) \tag{109}$$
$$p_f(x_0, y_0) \tag{110}$$
$$\vdash p_g(x_0, y_0) \tag{111}$$

But, given that p_f is in fact a mapping, the first and third hypothesis imply $y_0 = y_1$, and the second hypothesis is thus equivalent to the conclusion.

Now, for the second direction of our lemma $(\Leftarrow)$:

$$\forall x \,.\, \forall y \,.\, p_f(x, y) \Leftrightarrow p_g(x, y) \tag{112}$$
$$\vdash \forall x \,.\, \exists y \,.\, p_f(x, y) \wedge p_g(x, y) \tag{113}$$

We will prove (113) for the arbitrary value $x_0 : \tau_1$:

$$\vdash \exists y \,.\, p_f(x_0, y) \wedge p_g(x_0, y) \tag{114}$$

We will instantiate the hypothesis (112) to the same value:

$$\forall y \,.\, p_f(x_0, y) \Leftrightarrow p_g(x_0, y) \tag{115}$$

Now, as p_f is a mapping, we now that:

$$\exists y : \tau_2 \, . \, p_f(x_0, y) \tag{116}$$

We can thus take an existential witness, y_0, and instantiate our hypothesis (115) with it, to get:

$$p_f(x_0, y_0) \tag{117}$$

$$p_f(x_0, y_0) \Leftrightarrow p_g(x_0, y_0) \tag{118}$$

$$\vdash \exists y \, . \, p_f(x_0, y) \wedge p_g(x_0, y) \tag{119}$$

It is now obvious we can use y_0 as existential witness for the conclusion. ■

References

[Åhs & Wiggins 94] T. Åhs and G. A. Wiggins. Relational rippling for logic program synthesis and transformation, 1994. Presented at the Fourth International Workshop on Logic Program Synthesis and Transformation; available as DAI research paper, forthcoming.

[Åhs 95] T. Åhs. Relational rippling (working title), 1995. Ph.L. Thesis, Computing Science Department, Uppsala University, Sweden.

[Bates & Constable 85] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[Bundy 88] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Bundy *et al* 90] A. Bundy, A. Smaill, and G. A. Wiggins. The synthesis of logic programs from inductive proofs. In J. Lloyd, editor, *Computational Logic*, pages 135–149. Springer-Verlag, 1990. Esprit Basic Research Series. Also available from Edinburgh as DAI Research Paper 501.

[Bundy *et al* 93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Ireland 92] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.

[Kraan 94] I. Kraan. *Proof Planning for Logic Program Synthesis.* Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994. Submitted February 1994.

[Kraan *et al* 93] I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K.K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.

[Lombart *et al* 93] V. Lombart, G. A. Wiggins, and Y. Deville. Guiding synthesis proofs. In Y. Deville, editor, *Proceedings of LoPSTr'93*, Workshops in Computing. Springer Verlag, 1993.

[Wiggins 92a] G. A. Wiggins. Negation and control in automatically generated logic programs. In A. Pettorossi, editor, *Proceedings of META-92*. Springer Verlag, Heidelberg, 1992. LNCS Vol. 649.

[Wiggins 92b] G. A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. R. Apt, editor, *Proceedings of JICSLP-92*, pages 351–368. M.I.T. Press, Cambridge, MA, 1992.