BRICS

Basic Research in Computer Science

BRICS NS-94-2 D. Basin: Induction Based on Rippling and Proof Planning

Induction Based on Rippling and Proof Planning

Mini-Course

David Basin

BRICS Notes Series NS-94-2

ISSN 0909-3206 August 1994

See back inner page for a list of recent publications in the BRICS Notes Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

BRICS

Mini-Course on

Induction Based on Rippling and Proof Planning

David Basin
MPI, Saarbrucken

Aarhus University
August 11, 1994

Induction Based on Rippling and Proof Planning
Part I: Rippling and Proof Planning

David Basin
Max-Planck-Institut für Informatik
Saarbrücken

Example: An Easy Induction Proof

- Definitions and Axioms
  - Plus: $s(X) + Y \Rightarrow s(X + Y)$
  - Multiplication: $s(X) \times Y \Rightarrow Y + X \times Y$
  - Associativity, Simplification
- Prove: $(y + z) \times x = y \times x + z \times x$ (induct y)
  - Step case $\{s(y)/y\}$:

$$\begin{aligned}
(s(y) + z) \times x &= s(y) \times x + z \times x \\
s(y + z) \times x &= (x + y \times x) + z \times x \\
x + (y + z) \times x &= x + (y \times x + z \times x) \\
(y + z) \times x &= y \times x + z \times x
\end{aligned}$$

- Easy but...
  - How did we pick the induction?
  - How did we control rewriting?
  - How did we organize proof?
  - Where do lemmas come from?

A (Slightly) Harder Example

$$\forall t : list(obj).\ rev(t) = qrev(t, nil)$$

- Definitions (+ axioms like Assoc)

$$\begin{aligned}
rev(nil) &= nil \\
rev(X :: Y) &= rev(Y) <> X :: nil \\
qrev(nil, Z) &= Z \\
qrev(X :: Y, Z) &= qrev(Y, X :: Z)
\end{aligned}$$

- Proof by Induction on t

$$\forall t, l : list(obj).\ rev(t) <> l = qrev(t, l)$$

  - Ind Hyp : $\forall l. rev(t) <> l = qrev(t, l)$

$$\begin{aligned}
\forall l. rev(h :: t) <> l &= qrev(h :: t, l) \\
\forall l. (rev(t) <> h :: nil) <> l &= qrev(h :: t, l) \\
\forall l. (rev(t) <> h :: nil) <> l &= qrev(t, h :: l) \\
\forall l. rev(t) <> ((h :: nil) <> l) &= qrev(t, h :: l) \\
\forall l. rev(t) <> (h :: l) &= qrev(t, h :: l)
\end{aligned}$$

Overview: Induction Based on Rippling & Proof Planning

- Part I: Rippling (informal)
  - goal directed rewriting
  - "induction architectures" based on rippling
- Part II: Rippling (formal)
  - rewrite calculus
  - termination orders
  - implementation
- Part III: Proof search and critics
  - Lemma speculation, generalization, case-splitting, ...
- Part IV: Synthesis based on rippling
  - Program and induction synthesis based on h.o. unification
- Part V: Relationship to other approaches
  - Inductive completion, general difference reduction

Overview (cont.) — Focus on Edinburgh MRG Research

- Part I: Rippling
  - Bundy, *et. al.* (CADE9, CADE10, JAR, AIJ)
- Part II: Formalization
  - Basin & Walsh (CADE12, CTRS94)
- Part III: Critics and "patching"
  - Ireland (LPAR92), Hesketh (CADE12)
- Part IV: Synthesis
  - Kraan, Basin & Bundy (LOPSTR92, ICLP93)
  - Basin (LOPSTR94, ICLP94)
- Part V: Relationship
  - Barnett, Basin, Hesketh (Annals AI & Math, 1993)

Goal directed rewriting

- Definitions and axioms as rewrite rules

$$\begin{aligned} s(X)+Y &\Rightarrow s(X+Y)\\ s(X)\times Y &\Rightarrow Y+X\times Y\\ (X+Y)+Z &\Rightarrow X+(Y+Z)\\ X+(Y+Z) &\Rightarrow (X+Y)+Z\\ X+Y=X+Z &\Rightarrow Y=Z \end{aligned}$$

- Prove: $(y+z)\times x = y\times x + z\times x$ (induct y)
  - Step case $\{s(y)/y\}$:

$$\begin{aligned} (s(y)+z)\times x &= s(y)\times x + z\times x\\ s(y+z)\times x &= (x+y\times x)+z\times x\\ x+(y+z)\times x &= x+(y\times x+z\times x)\\ (y+z)\times x &= y\times x+z\times x \end{aligned}$$

- Insight: Rewrite steps are *difference reducing*

Example: Distributivity (with Rippling)

- Definitions and axioms

$$\begin{aligned} \boxed{s(\underline{X})}^{\uparrow}+Y &\Rightarrow \boxed{s(\underline{X+Y})}^{\uparrow}\\ \boxed{s(\underline{X})}^{\uparrow}\times Y &\Rightarrow \boxed{Y+\underline{X\times Y}}^{\uparrow}\\ \boxed{(X+\underline{Y})}^{\uparrow}+Z &\Rightarrow \boxed{X+(\underline{Y+Z})}^{\uparrow}\\ X+\boxed{(\underline{Y}+Z)}^{\uparrow} &\Rightarrow \boxed{(\underline{X+Y})+Z}^{\uparrow}\\ \boxed{X+\underline{Y}}^{\uparrow}=\boxed{X+\underline{Z}}^{\uparrow} &\Rightarrow Y=Z \end{aligned}$$

- Prove: $(y+z)\times x = y\times x+z\times x$ (induct on y)
  - Step case: $\{\boxed{s(\underline{y})}^{\uparrow}/y\}$

$$\begin{aligned} (\boxed{s(\underline{y})}^{\uparrow}+z)\times x &= \boxed{s(\underline{y})}^{\uparrow}\times x+z\times x\\ \boxed{s(\underline{y+z})}^{\uparrow}\times x &= \boxed{(x+\underline{y\times x})}^{\uparrow}+z\times x\\ \boxed{x+\underline{(y+z)\times x}}^{\uparrow} &= \boxed{x+(\underline{y\times x+z\times x})}^{\uparrow}\\ (y+z)\times x &= y\times x+z\times x \end{aligned}$$

Rippling — Idea

- Differences are Contexts:
  - Invariant part (Skeleton): Black
  - Changeable part (Wave-fronts): Red
- Example Induction: $P(x)\vdash P(\boxed{s(\underline{x})}^{\uparrow})$

$$\textit{Skeleton} = P(x) \qquad \textit{Wavefront} = \boxed{s(\underline{\ })}^{\uparrow}$$

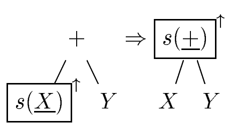
- Rewrite Rules are Context Moving Rules

$$\boxed{s(\underline{X})}^{\uparrow}+Y \Rightarrow \boxed{s(\underline{X+Y})}^{\uparrow}$$

  - Rules structure preserving

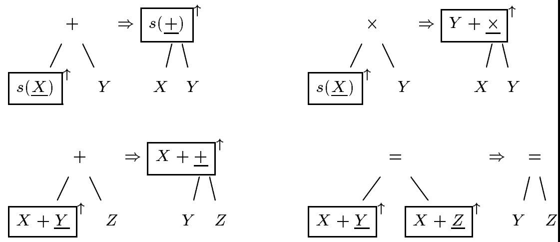
$$\textit{LHS Skeleton} = X+Y \qquad \textit{RHS Skeleton} = X+Y$$

  - Rules reduce differences

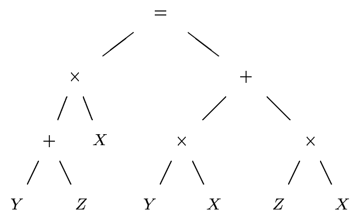


Rippling as Tree Rewriting

- Rules



- Proof



Rippling — Wavefront Directed to Mark Progress

- Definition + Axioms

$$
\begin{array}{rcl}
rev(\boxed{X :: \underline{Y}}^{\uparrow}) & \Rightarrow & \boxed{\underline{rev(Y)} <> X :: nil}^{\uparrow} \\
qrev(\boxed{X :: \underline{Y}}^{\uparrow}, Z) & \Rightarrow & qrev(Y, \boxed{X :: \underline{Z}}^{\downarrow}) \\
\boxed{X :: \underline{Y}}^{\uparrow} <> Z & \Rightarrow & \boxed{X :: (\underline{Y <> Z})}^{\uparrow} \\
\boxed{X :: (\underline{Y <> Z})}^{\downarrow} & \Rightarrow & \boxed{X :: \underline{Y}}^{\downarrow} <> Z \\
(\boxed{\underline{X} <> Y}^{\uparrow}) <> Z & \Rightarrow & X <> (\boxed{Y <> \underline{Z}}^{\downarrow}) \\
\boxed{X :: \underline{Y}}^{\uparrow} = \boxed{X :: \underline{Z}}^{\uparrow} & \Rightarrow & Y = Z \\
X \times (\boxed{\underline{Y} + \underline{Z}}^{\uparrow}) & \Rightarrow & \boxed{\underline{(X \times Y)} + \underline{(X \times Z)}}^{\uparrow}
\end{array}
$$

- Proof $\forall l : list(obj).\ rev(t) <> l = qrev(t, l)$

$$
\begin{array}{rcl}
rev(\boxed{h :: \underline{t}}^{\uparrow}) <> \lfloor l \rfloor & = & qrev(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor) \\
(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}) <> \lfloor l \rfloor & = & qrev(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor) \\
(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}) <> \lfloor l \rfloor & = & qrev(t, \left\lfloor \boxed{h :: \underline{l}}^{\downarrow} \right\rfloor) \\
rev(t) <> \left\lfloor \boxed{(h :: nil) <> \underline{l}}^{\downarrow} \right\rfloor & = & qrev(t, \left\lfloor \boxed{h :: \underline{l}}^{\downarrow} \right\rfloor) \\
rev(t) <> \left\lfloor \boxed{h :: \underline{l}}^{\downarrow} \right\rfloor & = & qrev(t, \left\lfloor \boxed{h :: \underline{l}}^{\downarrow} \right\rfloor)
\end{array}
$$

Rippling — Multiple Invariants

- Generalize Structure Preservation

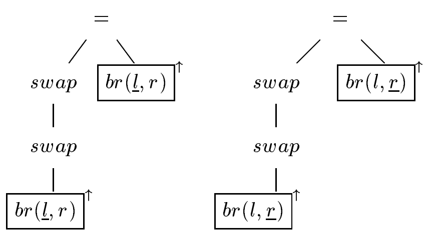
$$\forall t : bin\text{-}tree.swap(swap(t)) = t$$

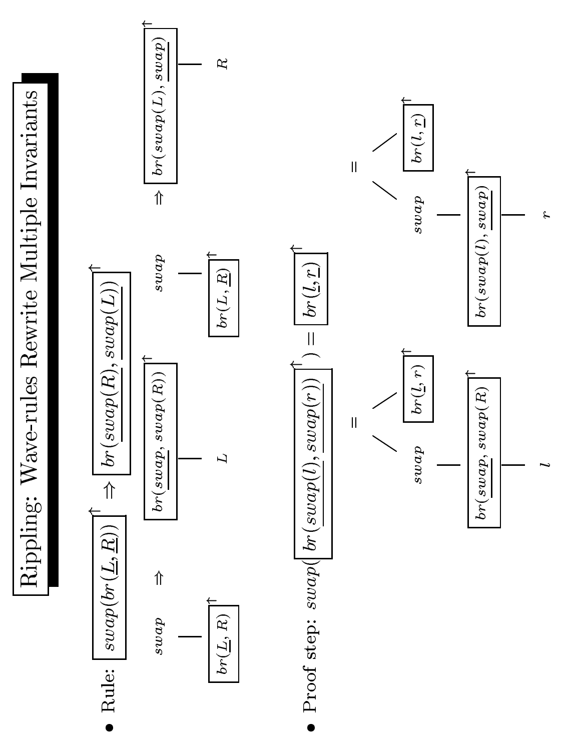
- Two inductions hypotheses

$$swap(swap(l)) = l,$$
$$swap(swap(r)) = r$$

- Conclusion contains two skeletons

$$swap(swap(\boxed{br(\underline{l}, \underline{r})}^{\uparrow})) = \boxed{br(\underline{l}, \underline{r})}^{\uparrow}$$





Types of Rippling

Rippling-Out: Move wave-fronts outwards preserving copy of I.H.

$$\boxed{s(\underline{X})}^{\uparrow} + Y \Rightarrow \boxed{s(\underline{X+Y})}^{\uparrow}$$

Rippling-Sideways/In: Move wave-fronts towards 'sinks'

$$qrev(\boxed{X :: \underline{Y}}^{\uparrow}, Z) \Rightarrow qrev(Y, \boxed{X :: \underline{Z}}^{\downarrow})$$

Conditional Rippling: Use conditional wave rules to suggest case splits.

$$X \neq Y \rightarrow member(X, \boxed{Y :: \underline{Z}}^{\uparrow}) \Rightarrow member(X, Z)$$

Generalised Rippling: Combinations of above

$$palin(\boxed{H :: \underline{T}}^{\uparrow}, Acc) \Rightarrow \boxed{H :: \underline{palin(T, \boxed{H :: \underline{Acc}}^{\downarrow})}}^{\uparrow}$$

Proof Planning

Formula ϕ

$\Downarrow$

Clam $\Leftarrow$ Methods (Rippling, Generalization, ...)

$\Downarrow$

Tactic

$\Downarrow$

Prover $\Leftarrow$ Type Theory, FOL, ...

$\Downarrow$

Proof of ϕ

Proof Plans/Clam

- Capture common structure of family of proofs, e.g., induction
- Planners construct proof-plans for conjectures
  - *Tactics* are programs which construct proofs
  - *Methods* are specifications of tactics
- Plans and Methods explicit objects
  - Success/Failure can be analyzed (by humans or computers)
- Leads to understanding of structure of inductive theorem proving

Methods

- Specification of tactics
- Example: *Ind_Strat*

  $ind\_strat(IndTerm(X), X) \equiv$
  $\quad induction(IndTerm(X), X)$ *then*
  $\quad\quad [\ sym\_eval,\ ripple\ then\ fertilize\ ]$

- Preconditions:

  Conj is conjecture, X is induction variable,

  IndTerm is wave front in induction rule.

  X must occur in *Conj*;

  Let $Conj' = Conj\{IndTerm(X')/X\}$.

  For each occurrence of $IndTerm(X')$ in $Conj'$ a wave rule must apply so that its wave front matches *IndTerm*.

Plan Examples

```
ind_strat(s(x),x) then          ind_strat(s(x),x) then
   [ sym_eval,                     [ ind_strat(s(y),y) then
     sym_eval                            [ sym_eval,
   ]                                       sym_eval
                                         ],
                                     ind_strat(s(y),y) then
                                         [ sym_eval,
                                           sym_eval
                                         ]
                                   ]
```

Associativity of $+$: $x + (y + z) = (x + y) + z$

Commutativity of $+$: $x + y = y + x$

Planning (cont.)

- 4 Forward Planners: depth first, breadth first, iterative deepener, best first.
- Match current conjecture to input formula and check preconditions.

  E.g. $Conj \equiv x + (y + z) = (x + y) + z$

  Fit *ind_strat*

  Fits with $IndTerm(X) \equiv \boxed{s(\underline{x})}^{\uparrow} \qquad X \equiv x$
- Calculate new result from output formula and effects

  Base case: $\ldots \vdash x + (y + z) = (x + y) + z,$

  Step case: $\ldots \vdash \boxed{s(\underline{x})}^{\uparrow} + (y + z) = \boxed{s(\underline{x})}^{\uparrow} + (y + z)$
- Execute tactic when whole plan found

Planning (cont.)

- Example Theorem: Invariance of Count After Sorting

$$\forall a : nat,\ l : list(nat).\ count(a, sort(l)) \quad = \quad count(a, l)$$

- Proof plan: 58 nodes, 3 inductions, 7 case splits, 41.5 cpu secs, no search
- Proof: 4,204 steps, 1,535 cpu secs
- Case splits suggested by rippling

Planning (cont.)

- Some Theorems attempted

Name	Theorem
ass+	$x + (y + z) = (x + y) + z$
com+	$x + y = y + z$
com+$_2$	$x + (y + z) = y + (x + z)$
dist	$x \times (y + z) = (x \times y) + (x \times z)$
ass×	$x \times (y \times z) = (x \times y) \times z$
com×	$x \times y = y \times x$
even+	$(even(x) \wedge even(y)) \rightarrow even(x + y)$
primes	$x \neq 0 \rightarrow \exists xl : list(primes).prod(xl) = x$
tailrev$_2$	$app(rev(a), n :: nil) = rev(n :: a)$
assapp	$app(l, app(m, n)) = app(app(l, m), n)$
lensum	$len(app(x, y)) = len(x) + len(y)$
tailrev	$rev(app(a, n :: nil)) = n :: rev(a)$
lenrev	$len(x) = len(rev(x))$
revrev	$x = rev(rev(x))$
comapp	$len(app(x, y)) = len(app(y, x))$
apprev	$app(rev(l), rev(m)) = rev(app(m, l))$
applast	$n = last(app(x, n :: nil))$
tailrev$_3$	$rev(app(rev(a), n :: nil)) = n :: a$

- Search space constrained by methods/rippling

Suggested Reading on Rippling/Proof Planning

The following reports are available by anonymous ftp from dream.dai.ed.ac.uk:

- Bundy, " The Use of Explicit Plans to Guide Inductive Proofs", CADE9.
- Bundy *et. al.*, "Extensions to the Rippling-Out Tactic for Guiding Inductive Proofs", CADE10.
- Bundy *et. al.*, "Experiments with Proof Plans for Induction", JAR, Vol 7, 1991.
- Bundy *et. al.*, "Rippling: A Heuristic for Guiding Inductive Proofs", AI Journal, Vol 62, 1993.

Induction Based on Rippling and Proof Planning
Part II: Formalization of Rippling

David Basin
Max-Planck-Institut für Informatik
Saarbrücken

Overview — Formalization of Rippling

- Intuitive idea of rippling
  - "Ordinary rewriting" with annotated rules
- Formalize rewrite calculus for rippling
  - "Ordinary rewriting" is insufficient
- Formalize structure preservation
  - Successful rippling ⇒ can use induction hypotheses
- Formalize termination: rippling always terminates
  - Termination important: other inductions, critics, ...

Annotation (WATs)

- Wave-fronts are *Contexts*:

$$P(n) \vdash P(\boxed{s(\underline{n})}^{\uparrow}) \quad \equiv \quad P(n) \vdash P(\mathrm{wf}(s(\mathrm{wh}(n))))$$

- WATs: Context is meaningfully marked (relative to WFFs)

$$even(\boxed{s(\underline{\boxed{s(\underline{x})}^{\uparrow}})}^{\uparrow}) \quad \equiv \quad even(\mathrm{wf}(s(\mathrm{wh}(\mathrm{wf}(s(\mathrm{wh}(x)))))))$$

$$\boxed{\underline{x \times \boxed{s(\underline{y})}^{\uparrow}} + s(y)}^{\uparrow} \quad \equiv \quad \mathrm{wf}(\mathrm{wh}(x \times \mathrm{wf}(s(\mathrm{wh}(y)))) + s(y))$$

$$swap(\boxed{br(\underline{t_1}, \underline{t_2})}^{\uparrow}) \quad \equiv \quad swap(\mathrm{wf}(br(\mathrm{wh}(t_1), \mathrm{wh}(t_2))))$$

- Examples of non-WATs

$$even(\boxed{s(s(x))}^{\uparrow}) \quad \equiv \quad even(\mathrm{wf}(s(s(x))))$$

$$\boxed{\underline{x \times \boxed{s(\underline{y})}^{\uparrow}} + \boxed{s(\underline{y})}^{\uparrow}}^{\uparrow} \quad \equiv \quad \mathrm{wf}(\mathrm{wh}(x \times \mathrm{wf}(s(\mathrm{wh}(y)))) + \mathrm{wf}(s(\mathrm{wh}(y))))$$

Annotation (cont.)

- Skeleton: WAT → $\mathcal{P}$(WFF)
  - $skel(a) = \{a\}$ (a atomic)
  - $skel(f(..., t_i, ...)) = \{f(..., skel(t_i), ...)\}$
  - $skel(\boxed{f(..., \underline{t_i}, ...)}^{\uparrow}) = \{s \mid s \in skel(t_i) \wedge t_i \text{ in wave-hole}\}$
- Erase: WAT → WFF by dropping annotation

t	$erase(t)$	$skel(t)$
$even(\boxed{s(\underline{\boxed{s(\underline{x})}^{\uparrow}})}^{\uparrow})$	$even(s(s(x)))$	$\{even(x)\}$
$\boxed{\underline{x \times \boxed{s(\underline{y})}^{\uparrow}} + s(y)}^{\uparrow}$	$(x \times s(y)) + s(y)$	$\{x \times y\}$
$swap(\boxed{br(\underline{t_1}, \underline{t_2})}^{\uparrow})$	$swap(br(t_1, t_2))$	$\{swap(t_1), swap(t_2)\}$

Desired Properties of Rippling

Well-formedness: if s is a WAT, and s ripples to t, then t is also a WAT.

Skeleton preservation: if s ripples to t then $skel(t) \subseteq skel(s)$;

Correctness: if s ripples to t then $erase(s)$ rewrites to $erase(t)$ in the original (unannotated) theory;

$$\begin{array}{ccccccc} t_1 & \Rightarrow & t_2 & \Rightarrow & \ldots & & t_n \\ & & & \Downarrow & & & \\ erase(t_1) & \rightarrow & erase(t_2) & \rightarrow & \ldots & & erase(t_n) \end{array}$$

Termination: rippling terminates.

First Order Rewriting $\neq$ Rippling

$$\begin{array}{ccc} l & \longrightarrow & r \\ {\scriptstyle Match \atop \sigma(l)=s} \downarrow & & \downarrow {\scriptstyle Apply \atop \sigma(r)} \\ t[s] & \twoheadrightarrow & t[\sigma(r)] \end{array}$$

- Rippling as First-Order Rewriting?

$$\boxed{s(\underline{a})}^{\uparrow} \times \boxed{s(\underline{b})}^{\uparrow} \mapsto \mathrm{wf}(s(\mathrm{wh}(a))) \times \mathrm{wf}(s(\mathrm{wh}(b)))$$

- Fails: $t = s = \boxed{s(\underline{a})}^{\uparrow} \times \boxed{s(\underline{b})}^{\uparrow}$
  - Rule: $\boxed{s(\underline{X})}^{\uparrow} \times Y \rightarrow \boxed{Y + \underline{X \times Y}}^{\uparrow}$
  - Match: $\sigma = \{a/X, \boxed{s(\underline{b})}^{\uparrow}/Y\}$

$$\boxed{\boxed{s(\underline{b})}^{\uparrow} + \underline{a \times \boxed{s(\underline{b})}^{\uparrow}}}^{\uparrow}$$

First Order Rewriting $\neq$ Rippling (cont.)

- Termination also fails.
  - Can annotate $h(f(U, s(V))) = s(h(f(s(U), V)))$ as wave-rule

$$h(\boxed{f(U, s(\underline{V}))}^{\uparrow}) \Rightarrow \boxed{s(\underline{h(\boxed{f(s(U), \underline{V})}^{\uparrow})})}^{\uparrow}.$$

$$\boxed{s(\underline{h(\boxed{f(s(\underline{U}), V)}^{\uparrow})})}^{\uparrow} \Rightarrow h(\boxed{f(\underline{U}, s(V))}^{\uparrow}).$$

  - leads to cycling (Note: all terms WATs)

$$h(\boxed{f(\underline{a}, s(\underline{a}))}^{\uparrow}) \Rightarrow \boxed{s(\underline{h(\boxed{f(s(\underline{a}), \underline{a})}^{\uparrow})})}^{\uparrow} \Rightarrow h(\boxed{f(\underline{a}, s(\underline{a}))}^{\uparrow}) \Rightarrow \ldots.$$

- Conclude: first-order rewriting cannot directly implement rippling.

A Rippling Calculus

- Term Replacement Must Respect Annotation

$$\boxed{f(a, \underline{b})}^{\uparrow}$$

  - Replacement in wavefront requires erasure

$$a \mapsto \boxed{s(\underline{a})}^{\uparrow} \text{ then } \boxed{f(s(a), \underline{b})}^{\uparrow}$$

  - Replacement in skeleton as usual

$$b \mapsto \boxed{s(\underline{b})}^{\uparrow} \text{ then } \boxed{f(a, \underline{\boxed{s(\underline{b})}^{\uparrow}})}^{\uparrow}$$

- Replacement effects substitution $\sigma = \{a/X, \boxed{s(\underline{b})}^{\uparrow}/Y\}$

$$\sigma(\boxed{Y + \underline{X \times Y}}^{\uparrow}) = \boxed{s(b) + \underline{a \times \boxed{s(\underline{b})}^{\uparrow}}}^{\uparrow}$$

- New substitution requires new matching
  - Match $\boxed{f(X, \underline{0})}^{\uparrow}$ with $\boxed{f(s(0), \underline{0})}^{\uparrow}$.
  - Answers: $\{s(0)/X\}$ and $\{\boxed{s(\underline{0})}^{\uparrow}/X\}$

Matching: Standard Matching

DELETE rule:
$S \cup \{t = t\} \quad \Rightarrow \quad S$

DECOMPOSE rule
$S \cup \{f(s_1, .., s_n) = f(t_1, ..., t_n)\} \quad \Rightarrow \quad S \cup \{s_i = t_i \mid 1 \leq i \leq n\}$

- Normalize starting equation

$$\begin{array}{cc} \{f(a, X, Y) = f(a, s(a), b)\} & \text{Start} \\ \{a = a, X = s(a), Y = b\} & \text{Decompose} \\ \{X = s(a), Y = b\} & \text{Delete} \end{array}$$

- Succeeds if result compatible (each var occurs once)

$$\{X = s(a), X = b, Y = b\}$$

- Result is substitution

$$\sigma = \{s(a)/X, b/Y\}$$

Annotated Matching

DELETE rule:
$S \cup \{t = t : Pos\} \quad \Rightarrow \quad S$

DECOMPOSE rules:
$S \cup \{f(s_1, .., s_n) = f(t_1, ..., t_n) : Pos\} \quad \Rightarrow \quad S \cup \{s_i = t_i : Pos \mid 1 \leq i \leq n\}$
$S \cup \{\boxed{f(\underline{s_1}, ..., s_n)}^{\uparrow} = \boxed{f(\underline{t_1}, ..., t_n)}^{\uparrow} : sk\} \quad \Rightarrow \quad S \cup \{s_1 = t_1 : sk, ..., s_n = t_n : wf\}$

- Normalize start equation

$$\{\boxed{f(\underline{X}, X)}^{\uparrow} = \boxed{f(\underline{\boxed{s(\underline{a})}^{\uparrow}}, s(a))}^{\uparrow} : sk\}$$

- Assignments for variables must be compatible

$$\{X = \boxed{s(\underline{a})}^{\uparrow} : sk, X = s(a) : wf\}.$$

- Answer substitution can be extracted

$$\sigma = \{\boxed{s(\underline{a})}^{\uparrow}/X\}$$

Rewrite Calculus

- Rippling = Annotated Matching + Term Replacement

$$\begin{tikzcd} l \arrow[r] \arrow[d, "\substack{Match \\ \sigma(l)=s}"'] & r \arrow[d, "\substack{Apply \\ \sigma(r)}"] \\ t[s] \arrow[r, two heads] & t[\sigma(r)] \end{tikzcd}$$

- $l \Rightarrow r$ is *Proper Rewrite Rule* iff
  - l and r are WATs & $erase(l) \rightarrow erase(r)$
  - $skel(r) \subseteq skel(l)$,
- Correctness: for s a WAT, $s \Rightarrow^* t$ with proper equations
  - t is a *wat*,
  - $skel(t) \subseteq skel(s)$, and
  - $erase(s) \rightarrow^* erase(t)$.

Termination

- Termination Important!
- Use orders analogous to *Reduction Orders*
  $>$ is well-founded: I.e., no $t_1 > t_2 > ...$
  $>$ is monotonic: $l > r$ then $s[l] > s[r]$
  $>$ is stable: $l > r$ then $\sigma(l) > \sigma(r)$
- Rewriting with rules R terminating

$$\exists \text{ reduction order } >. \ \forall l \rightarrow r \in R. \, l > r$$

- Analog: Rippling reduction order
  $>$ well-founded
  $>$ monotonic with respect to WATs
  stability: if $s > t$, σ well-annotated, then $\sigma(s) > \sigma(t)$
- Rippling orders weaker than reduction orders
  - Only consider monotonicity/stability relative to *well-annotated* replacement

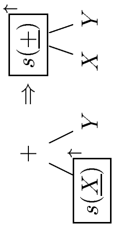
Wave-rules & Correctness

- Correctness of Rippling depends on two independent properties
  1. Wave-rules are structure preserving

     $$\boxed{s(\underline{X})}^{\uparrow} + Y \to \boxed{s(\underline{X+Y})}^{\uparrow}$$

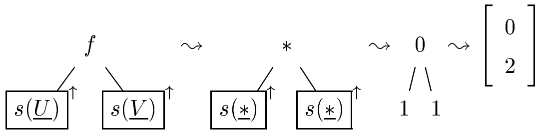
  2. Rippling makes well-founded progress

     Second Property achieved by insisting LHS > RHS where > is a reduction order for rippling
- What counts as such orderings?
  - Orientation determines direction of wave-front movement!

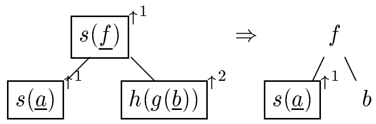


Termination Orders

- Measure Depth/Weight of annotation — consider single hole/skeleton



- Example



- Weight function: μ: Term $\to$ Weight

  $$[1,3] \Rightarrow [0,1]$$

- Progress: wave-fronts move upwards
  - Reverse lexicographic order ">"
  - Above example is a wave-rule

Termination Orders — Generalization

- Multi-hole wave-fronts

  $$swap(\boxed{node(\underline{L},\underline{R})}^{\uparrow})$$

  - Reduce to simple annotation

    $$\{swap(\boxed{node(\underline{L},R)}^{\uparrow}), swap(\boxed{node(L,\underline{R})}^{\uparrow})\}$$

  - Measure are Multisets: $\{[0,1],[0,1]\}$
  - Take multiset extension of $>_{revlex}$
- Inward wave-fronts: lex-order

  $$\boxed{s(\underline{f(x)})}^{\downarrow} \to f(\boxed{s(\underline{x})}^{\downarrow}) \quad as \quad \{[1,0]\} >_{lex} \{[0,1]\}$$

- Inward & Outward

  $$palin(\boxed{H :: \underline{T}}^{\uparrow}, Acc) \Rightarrow \boxed{H :: \underline{palin(T, \boxed{H :: \underline{Acc}}^{\downarrow})}}^{\uparrow}$$

  - Find measures relative to inward and outward

    $$palin(\boxed{H :: \underline{T}}^{\uparrow}, Acc) \quad \Rightarrow \quad \boxed{H :: \underline{palin(T, Acc)}}^{\uparrow}$$

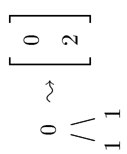
    $$palin(T, Acc) \quad \Rightarrow \quad palin(T, \boxed{H :: \underline{Acc}}^{\downarrow})$$

  ... lexicographically combine $\langle OUT, IN\rangle$

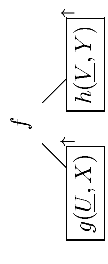
  $$\langle\{[0,1]\},\{[0,0]\}\rangle > \langle\{[1,0]\},\{[0,1]\}\rangle$$

Termination Orders (cont.)

- > is well-founded: lexicographic combinations and multi-set extensions
- > is monotonic: order defined via level-by-level mapping



- > is stable: Vars in wave-fronts have no effect



  - Vars in skeleton in $l \to r$ occur at same positions.
- Conclude: rippling with > terminates

Orderings Domain Dependent

- Induction: "Up" out of the way or "Down" to sinks
- Other domains require different orderings

$$\begin{aligned}
\boxed{log_e(\underline{x}+1) + log_e(\underline{x}-1)}^{\uparrow} &= c \\
\boxed{log_e((\underline{x}+1) \times (\underline{x}-1))}^{\uparrow} &= c \\
\boxed{log_e(\underline{x}^2-1)}^{\uparrow} &= c \\
\boxed{\underline{x}^2-1}^{\uparrow} &= \boxed{e^{\underline{c}}}^{\downarrow} \\
\boxed{\underline{x}^2}^{\uparrow} &= \boxed{e^{\underline{c}}+1}^{\downarrow} \\
x &= \boxed{\pm\sqrt{e^{\underline{c}}+1}}^{\downarrow}
\end{aligned}$$

- Rippling can be used for domain independent difference reduction

Rippling for Algebraic Problem Solving

- Use "Press" strategies (Bundy 1981)

$$\begin{aligned}
&COLLECTION: & \boxed{\underline{U} \times \underline{U}}^{\uparrow} &\Rightarrow \boxed{\underline{U}^2}^{\uparrow} \\
&ATTRACTION: & \boxed{\log(\underline{U}) + \log(\underline{V})}^{\uparrow} &\Rightarrow \boxed{\log(\underline{U} \times \underline{V})}^{\uparrow} \\
&ISOLATION: & \boxed{\underline{U}^2}^{\uparrow} = V &\Rightarrow U = \boxed{\pm\sqrt{\underline{V}}}^{\downarrow}
\end{aligned}$$

- Termination based on preconditions
- Simplify and generalize via Rippling
  - Unknown variables are invariants
  - Ordering lexicographically combines:

Collection	Attraction	Isolation
# "Wave-Holes"	Distance between holes (Path in tree)	Annotation Weight

Implementing Rippling

- Simple orders yield simple parsers

$$\begin{array}{ccc}
(X+Y)+Z & \to & X+(Y+Z) \\
 & \Downarrow & \\
\boxed{(X+\underline{Y})}^{\uparrow}+Z & \to & \boxed{X+(\underline{Y+Z})}^{\uparrow} \\
\boxed{(\underline{X}+Y)}^{\uparrow}+Z & \to & X+\boxed{(Y+\underline{Z})}^{\downarrow} \\
 & \vdots & \\
\boxed{(\underline{X}+Y)+Z}^{\uparrow} & \to & \boxed{\underline{X}+(Y+Z)}^{\downarrow}
\end{array}$$

- Can generate in advance: *Parsing = Orientation + Annotation*
- Preferable to generate rules as needed for rewriting

Implementing Rippling — Wave-rule Parsing

```
rewrite(T,NT) :-                   % pick term position
  pick_an_pos(T,X,ST,NT),          % subterm ST in NT marked by X
  pick_rule(L,R),                  % pick a rule L -> R
  match_erasure(ST,L),             % check L against erasure of T
  replace(ST,L,R,NST),             % perform replacement yielding NST
  X = NST.                         % do subterm replacement

replace(T, L, R, ST) :-
  copy_an(T,L,AL,Subs),            % copy annotations and generate subs
  parse(AL,R,AR),                  % find compatible R from annotated L
  apply_subs(Subs,AS,ST).          % Apply substitutions to parsed R

parse(AL,R,AR) :-
  pick_an(R,A),                     % annotate R  (generate annotations)
  skel_preserving(AL,A),            % skeletons equal? (test equality)
  orient(AL,A,AR).                  % Orient R (and test measure)
```

Wave-rule Parsing Example

- Example: $t = \boxed{s(\underline{x})}^{\uparrow} \times \boxed{s(\underline{y})}^{\uparrow}$
  - Wave rule: $s(U) \times V \Rightarrow (U \times V) + V$
- Erasures match: $erase(t) = s(x) \times s(y)$
- Copy annotation onto LHS: $\boxed{s(\underline{U})}^{\uparrow} \times V$

  ... generate substitutions: $\{x/U\}$, $\{\boxed{s(\underline{y})}^{\uparrow}/V\}$
- Generate RHS testing skeleton and measure

$$\boxed{(\underline{U \times V}) + V}^{\uparrow}$$

- Apply substitution (using annotated replacement)

$$\boxed{\underline{(x \times \boxed{s(\underline{y})}^{\uparrow})} + s(y)}^{\uparrow}$$

History/Comparison — Edinburgh

- Calculus based on first-order rewriting + restrictions
  - Restrictions disallow annotated substitutions

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

  ... cannot be rewritten with $\boxed{s(\underline{X})}^{\uparrow} + Y \Rightarrow \boxed{s(\underline{X + Y})}^{\uparrow}$
  - restrictions partially lifted in implementation
- Wave-rules given by complex schemata
  - combine structure preservation + termination
  - Less flexible, i.e., disallow

$$f(\boxed{s(s(\underline{x}))}^{\uparrow}) \to f(\boxed{g(\underline{x})}^{\uparrow})$$

History/Comparison — INKA

- Hutter developed rigorous rewrite calculus for rippling
- More complex, but also more general
- Not concerned with termination
- Have also explored "difference reduction" based on rippling (See part V)

Suggested Reading

The following reports are available by anonymous ftp from mpi-sb.mpg.de and dream.dai.ed.ac.uk:

- Basin & Walsh, "A Calculus for Rippling", CTRS94.
- Basin & Walsh, "Termination Orderings for Rippling", CADE12.
- Yoshida *et. al.*, "Coloured Rippling: An Extension of a Theorem Proving Heuristic".

Induction Based on Rippling and Proof Planning
Part III: Proof Search and Critics

David Basin
Max-Planck-Institut für Informatik
Saarbrücken

(With thanks to Andrew Ireland
for slide material on critics)

Overview — Rippling and Proof Search

- Rippling & Lemma Discovery
  - Discovery based on “weak fertilization” + rippling-in
- Critics & Productive Use of Failure
  - create new wave-rules
  - nested inductions
  - generalize theorem to introduce accumulators
  - case analyses

Rippling-in & Lemma Discovery

- Schematic Induction Hypothesis

$$+ \ + \ + \ + \ + \ + \ + = * \ * \ * \ * \ * \ * \ *$$

- Schematic Induction Conclusion

$$+ \ + \ + \boxed{?\,\underline{+}\,?}^{\uparrow} + \ + \ + = * \ * \ * \boxed{?\,\underline{*}\,?}^{\uparrow} * \ * \ *$$

- Imagine we can ripple out on one side

$$+ \ + \boxed{?\,\underline{+ \ + \ +}\,?}^{\uparrow} + \ + = \boxed{?\,\underline{* \ * \ * \ * \ * \ * \ *}\,?}^{\uparrow}$$

- We can fertilize RHS

$$+ \ + \boxed{?\,\underline{+ \ + \ +}\,?}^{\uparrow} + \ + = \boxed{?\,\underline{+ \ + \ + \ + \ + \ + \ +}\,?}^{\downarrow}$$

- Progress: two sides only differ in “?” and position!
- Further Progress: Ripple in

$$+ \ + \boxed{?\,\underline{+ \ + \ +}\,?}^{\uparrow} + \ + = + \boxed{?\,\underline{+ \ + \ + \ + \ +}\,?}^{\downarrow} +$$

- And cancel

$$+ \boxed{?\,\underline{+ \ + \ +}\,?}^{\uparrow} + = \boxed{?\,\underline{+ \ + \ + \ + \ +}\,?}^{\downarrow}$$

- Result is generalized to missing wave-rule

Rippling-in & Lemma Discovery

- Example: $\mathit{half}(x + x) = x$
- Definition for + and half

$$\mathit{half}(0) = \mathit{half}(s(0)) = 0$$
$$\mathit{half}(\boxed{s(s(\underline{X}))}^{\uparrow}) = \boxed{s(\underline{\mathit{half}(X)})}^{\uparrow}$$

- Proof

$$\begin{aligned}
\mathit{half}(\boxed{s(\underline{x})}^{\uparrow} + \boxed{s(\underline{x})}^{\uparrow}) &= \boxed{s(\underline{x})}^{\uparrow} \\
\mathit{half}(\boxed{s(\underline{x + \boxed{s(\underline{x})}^{\uparrow}})}^{\uparrow}) &= \boxed{s(\underline{x})}^{\uparrow} \\
\mathit{half}(\boxed{s(\underline{x + \boxed{s(\underline{x})}^{\uparrow}})}^{\uparrow}) &= \boxed{s(\underline{\mathit{half}(x + x)})}^{\downarrow} \\
\mathit{half}(\boxed{s(\underline{x + \boxed{s(\underline{x})}^{\uparrow}})}^{\uparrow}) &= \mathit{half}(\boxed{s(s(\underline{x + x}))}^{\downarrow})
\end{aligned}$$

- Further rippling in doesn't help cancellation

$$x + s(x) = s(x + x)$$

- Generalize to missing wave-rule

$$x + s(Y) = s(x + Y)$$

Critic Approach to Failure Analysis

- Developed by Andrew Ireland, U. of Edinburgh
- Based on declarative specification of kinds of rippling
- Failure of conditions suggests patches
  - Example: generalize to introduce sinks
- Natural extension of planning hierarchy
  - *Tactic:* controls application of inference rules.
  - *Method:* declaratively specifies a tactic.
  - *Critic:* analyses proof failures and suggests patches.
- We begin by specifying rippling methods

Specification: (longitudinal) wave-rules

$$rev(\boxed{h :: \underline{t}}^{\uparrow}) <> \lfloor l \rfloor \quad = \quad \ldots$$

$$(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}) <> \lfloor l \rfloor \quad = \quad \ldots$$

- Method preconditions 1: wave (longitudinal)

  1.1 There exists a wave-occurrence, *e.g.*

  $$rev(\boxed{h :: \underline{t}}^{\uparrow}) \ldots$$

  1.2 and a matching wave-rule, *e.g.*

  $$rev(\boxed{X :: \underline{Y}}^{\uparrow}) \Rightarrow \boxed{\underline{rev(Y)} <> X :: nil}^{\uparrow}$$

  1.3 and any condition attached to the rewrite is provable.

Specification: (transverse) wave-rules

$$\ldots \quad = \quad qrev(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor)$$

$$\ldots \quad = \quad qrev(t, \left\lfloor \boxed{h :: \underline{l}}^{\downarrow} \right\rfloor)$$

- Method preconditions 2: wave (transverse)

  2.1 There exists a wave-occurrence, *e.g.*

  $$\ldots qrev(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor)$$

  2.2 and a matching wave-rule, *e.g.*

  $$qrev(\boxed{X :: \underline{Y}}^{\uparrow}, Z) \Rightarrow qrev(Y, \boxed{X :: \underline{Z}}^{\downarrow})$$

  2.3 and any condition attached to the rewrite is provable,

  2.4 and the selected wave-occurrence is sinkable, *e.g.*

  $$\ldots = qrev(t, \left\lfloor \boxed{h :: \underline{l}}^{\downarrow} \right\rfloor)$$

Wave-rules: General Pattern

$$g(x, Y)$$
$$\vdash g(\boxed{c(\underline{x})}^{\uparrow}, \lfloor y \rfloor)$$

Longitudinal $\swarrow$ $\qquad\qquad$ $\searrow$ Transverse

$$g(x, Y) \qquad\qquad\qquad g(x, Y)$$

$$\vdash \boxed{c'(\underline{g(x, \lfloor y \rfloor)})}^{\uparrow} \qquad\qquad \vdash g(x, \left\lfloor \boxed{c''(\underline{y})}^{\downarrow} \right\rfloor)$$

Exception: missing wave-rule

$$\forall t, l : list(obj).\ \ rev(rev(t <> l)) = rev(rev(t)) <> rev(rev(l))$$

- Induction hypothesis (induction on t, l is sink)

$$rev(rev(t <> L)) = rev(rev(t)) <> rev(rev(L))$$

- Induction conclusion

$$\begin{aligned}
rev(rev(\boxed{h :: \underline{t}}^{\uparrow} <> \lfloor l \rfloor)) &= rev(rev(\boxed{h :: \underline{t}}^{\uparrow})) <> rev(rev(\lfloor l \rfloor)) \\
rev(rev(\boxed{h :: \underline{t <> \lfloor l \rfloor}}^{\uparrow})) &= rev(rev(\boxed{h :: \underline{t}}^{\uparrow})) <> rev(rev(\lfloor l \rfloor)) \\
rev(\boxed{\underline{rev(t <> \lfloor l \rfloor)} <> h :: nil}^{\uparrow}) &= rev(rev(\boxed{h :: \underline{t}}^{\uparrow})) <> rev(rev(\lfloor l \rfloor)) \\
\underbrace{rev(\boxed{\underline{rev(t <> \lfloor l \rfloor)} <> h :: nil}^{\uparrow})}_{\text{blocked}} &= \underbrace{rev(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}) <> rev(rev(\lfloor l \rfloor))}_{\text{blocked}}
\end{aligned}$$

Critic preconditions (lemma speculation)

- Precondition *(1.1)* of the wave method holds, *e.g*

  1.1 There exists a wave-occurrence, *e.g.*

$$f(\boxed{c(\underline{x})}^{\uparrow}, y)$$

  Preconditions *(1.2)* and *(1.3)* are false:

  1.2 but no matching wave-rule

  1.3 so no condition to check

- and for each most nested wave-occurrence there does not exist a potential unblocking wave-rule, *e.g.*

$$f(\boxed{c(C'(\underline{X}))}^{\uparrow}, \boxed{C''(\underline{Y})}^{\uparrow}) \Rightarrow \ldots$$

Critic preconditions (lemma speculation)

- Precondition *(1.1)* of the wave method holds, *e.g*

  1.1 There exists a wave-occurrence, *e.g.*

$$\ldots = rev(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}) <> \ldots$$

- Preconditions *(1.2)* and *(1.3)* are false:

  1.2 but no matching wave-rule

  1.3 so no condition to check

- and for each most nested wave-occurrence there does not exist a potential unblocking wave-rule, *e.g.*

$$rev(\boxed{C(\underline{X}) <> Y :: nil}^{\uparrow}) \Rightarrow \ldots$$

Patch: wave-rule speculation (cont.)

- Speculative wave-rule: $\boxed{C(\underline{s})}^{\uparrow}, \boxed{C(\underline{s})}^{\downarrow}$,

$$rev(\boxed{\underline{X} <> Y :: nil}^{\uparrow}) \Rightarrow \boxed{C(\underline{rev(X)}, X, Y)}^{\uparrow}$$

  – Possible Instances

$$\begin{aligned}
rev(\boxed{\underline{X} <> Y :: nil}^{\uparrow}) &\Rightarrow rev(X) \\
rev(\boxed{\underline{X} <> Y :: nil}^{\uparrow}) &\Rightarrow \boxed{Y :: \underline{rev(X)}}^{\uparrow}
\end{aligned}$$

- Actual wave-rules

$$\begin{aligned}
\boxed{X :: \underline{Y}}^{\uparrow} <> Z &\Rightarrow \boxed{X :: \underline{(Y <> Z)}}^{\uparrow} \\
\boxed{X :: \underline{Y}}^{\uparrow} = \boxed{X :: \underline{Z}}^{\uparrow} &\Rightarrow Y = Z
\end{aligned}$$

Patch: wave-rule speculation

- Continue with speculative rippling

$$
\begin{array}{rcl}
rev(\boxed{\underline{rev(\ldots)} <> h :: nil}^{\uparrow}) & = & rev(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}) <> \ldots \\
\boxed{C(\underline{rev(rev(\ldots))}, \ldots, h)}^{\uparrow} & = & \boxed{C(\underline{rev(rev(t))}, \ldots, h)}^{\uparrow} <> \ldots \\
\boxed{C'(\ldots, h) :: \underline{\boxed{C''(\underline{rev(rev(\ldots))}, \ldots)}^{\uparrow}}}^{\uparrow} & = & \boxed{C'(\ldots, h) :: \underline{\boxed{C''(\underline{\ldots <> \ldots}, \ldots)}^{\uparrow}}}^{\uparrow} \\
\boxed{C''(\underline{rev(rev(\ldots))}, \ldots)}^{\uparrow} & = & \boxed{C''(\underline{\ldots <> \ldots}, \ldots)}^{\uparrow} \\
rev(rev(t <> l)) & = & rev(rev(t)) <> rev(rev(l))
\end{array}
$$

Exception to General Pattern — Lemma Speculation

$$
\begin{array}{c}
f(g(x, h(Z))) \\
\vdash f(g(\boxed{c(\underline{x}, y)}^{\uparrow}, h(\lfloor z \rfloor)))
\end{array}
$$

Longitudinal $\swarrow$ $\searrow$ Transverse

$$
\begin{array}{c}
f(g(x, h(Z))) \\
\vdash f(\boxed{C(\underline{g(x, h(\lfloor z \rfloor))}, x, y)}^{\uparrow})
\end{array}
\qquad
\begin{array}{c}
f(g(x, h(Z))) \\
\vdash f(g(x, \boxed{C'(\underline{h(\lfloor z \rfloor)}, x, y)}^{\downarrow}))
\end{array}
$$

Exception: nested induction

$$\forall t : list(obj).\ evenl(t <> nil) \leftrightarrow even(length(t))$$

- Wave-rules:

$$
\begin{array}{rcl}
length(\boxed{X :: \underline{Y}}^{\uparrow}) & \Rightarrow & \boxed{s(\underline{length(Y)})}^{\uparrow} \\
even(\boxed{s(s(\underline{X}))}^{\uparrow}) & \Rightarrow & even(X) \\
evenl(\boxed{X :: Y :: \underline{Z}}^{\uparrow}) & \Rightarrow & evenl(Z) \\
\boxed{X :: \underline{Y}}^{\uparrow} <> Z & \Rightarrow & \boxed{X :: \underline{(Y <> Z)}}^{\uparrow}
\end{array}
$$

Exception: nested induction (cont.)

- First induction suggestion $P(\boxed{h :: \underline{t}}^{\uparrow})$

$$\underbrace{evenl(\boxed{h :: \underline{(t <> nil)}}^{\uparrow})}_{\text{blocked}} \leftrightarrow \underbrace{even(\boxed{s(\underline{length(t)})}^{\uparrow})}_{\text{blocked}}$$

- Second induction suggestion $P(\boxed{h_1 :: h_2 :: \underline{t}}^{\uparrow})$

$$
\begin{array}{rcl}
evenl(\boxed{h_1 :: \underline{(\boxed{h_2 :: \underline{t}}^{\uparrow} <> nil)}}^{\uparrow}) & \leftrightarrow & even(\boxed{s(\underline{length(\boxed{h_2 :: \underline{t}}^{\uparrow})})}^{\uparrow}) \\
evenl(\boxed{h_1 :: h_2 :: \underline{(t <> nil)}}^{\uparrow}) & \leftrightarrow & even(\boxed{s(s(\underline{length(t)}))}^{\uparrow}) \\
evenl(t <> nil) & \leftrightarrow & even(length(t))
\end{array}
$$

Critic preconditions (nested induction)

- Precondition *(1.1)* of the wave method holds, *e.g*
  1.1 There exists a wave-occurrence, *e.g.*

$$\ldots \leftrightarrow even(\boxed{s(\underline{length(t)})}^{\uparrow})$$

- Preconditions *(1.2)* and *(1.3)* are false:
  1.2 but no matching wave-rule
  1.3 so no condition to check
- and there exists a most nested wave-occurrence for which there exists a potential unblocking wave-rule, *e.g.*

$$even(\boxed{s(C(\underline{X}))}^{\uparrow}) \;\Rightarrow\; \ldots$$
$$even(\boxed{s(s(\underline{X}))}^{\uparrow}) \;\Rightarrow\; even(X)$$

Exception to General Pattern — nested induction

$$f(g(x,h(Y)))$$
$$\vdash f(g(\boxed{c(\underline{x})}^{\uparrow}, h(\lfloor y \rfloor)))$$

Longitudinal ↙ ↘ Transverse

$$\vdash f(\boxed{c_1(\underline{g(x,h(\lfloor y \rfloor))})}^{\uparrow}) \qquad \vdash f(g(x, \boxed{c_3(\underline{h(\lfloor y \rfloor)})}^{\downarrow}))$$
$$\vdash f(\boxed{c_1(\underline{g(\boxed{c(\underline{x})}^{\uparrow}, h(\lfloor y \rfloor))})}^{\uparrow}) \qquad \vdash f(g(\boxed{c(\underline{x})}^{\uparrow}, \boxed{c_3(\underline{h(\lfloor y \rfloor)})}^{\downarrow}))$$
$$\vdash f(\boxed{c_1(c_1(\underline{g(x,h(\lfloor y \rfloor))}))}^{\uparrow}) \qquad \vdash f(g(x, \boxed{c_3(c_3(\underline{h(\lfloor y \rfloor)}))}^{\downarrow}))$$
$$\vdash \boxed{c_2(\underline{f(g(x,h(\lfloor y \rfloor)))})}^{\uparrow} \qquad \vdash f(g(x, h(\left\lfloor \boxed{c_4(\underline{y})}^{\downarrow} \right\rfloor)))$$

Exception: missing sink

$$\forall t : list(obj).\;\; rev(t) = qrev(t, nil)$$

- Induction hypothesis

$$rev(t) = qrev(t, nil)$$

- Induction conclusion

$$rev(\boxed{h :: \underline{t}}^{\uparrow}) = qrev(\boxed{h :: \underline{t}}^{\uparrow}, nil)$$
$$\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow} = \underbrace{qrev(\boxed{h :: \underline{t}}^{\uparrow}, nil)}_{\text{missing sink}}$$

Critic preconditions (sink speculation)

- Preconditions *(2.1)*, *(2.2)* and *(2.3)* of the wave method hold, e.g.
  2.1 There exists a wave-occurrence, *e.g.*

$$\ldots = qrev(\boxed{h :: \underline{t}}^{\uparrow}, nil)$$

  2.2 and a matching wave-rule, *e.g.*

$$qrev(\boxed{X :: \underline{Y}}^{\uparrow}, Z) \Rightarrow qrev(Y, \boxed{X :: \underline{Z}}^{\downarrow})$$

  2.3 and any condition attached to the rewrite is provable.
- Precondition *(2.4)* is false, *i.e.*
  2.4 no sink occurrence.

Patch: sink speculation

$$\forall t, l : list(obj).\ \ F(rev(t), l) = qrev(t, G(l))$$

- Induction hypothesis

$$F(rev(t), L) = qrev(t, G(L))$$

- Induction conclusion

$$
\begin{array}{rcl}
F(rev(\boxed{h :: \underline{t}}^{\uparrow}), \lfloor l \rfloor) & = & qrev(\boxed{h :: \underline{t}}^{\uparrow}, G(\lfloor l \rfloor)) \\
F(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}, \lfloor l \rfloor) & = & qrev(\boxed{h :: \underline{t}}^{\uparrow}, G(\lfloor l \rfloor)) \\
F(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}, \lfloor l \rfloor) & = & qrev(t, \boxed{h :: \underline{G(\lfloor l \rfloor)}}^{\downarrow}) \\
rev(t) <> (\left\lfloor \boxed{h :: nil <> \underline{l}}^{\downarrow} \right\rfloor) & = & qrev(t, \boxed{h :: \underline{G(\lfloor l \rfloor)}}^{\downarrow}) \\
rev(t) <> (\left\lfloor \boxed{h :: \underline{l}}^{\downarrow} \right\rfloor) & = & qrev(t, \boxed{h :: \underline{G(\lfloor l \rfloor)}}^{\downarrow}) \\
rev(t) <> (\left\lfloor \boxed{h :: \underline{l}}^{\downarrow} \right\rfloor) & = & qrev(t, \left\lfloor \boxed{h :: \underline{l}}^{\downarrow} \right\rfloor)
\end{array}
$$

- Wave-rule

$$(\boxed{\underline{X} <> Y}^{\uparrow}) <> Z \Rightarrow X <> (\boxed{Y <> \underline{Z}}^{\downarrow})$$

- Generalized conjecture

$$\forall t, l : list(obj).\ \ rev(t) <> l = qrev(t, l)$$

Exception to General Pattern — sink speculation

$$
\begin{array}{lcl}
f(x, \ldots) & & f(x, \ldots, F(Y), \ldots) \\
\vdash f(\boxed{c(\underline{x})}^{\uparrow}, \ldots) & \Rightarrow & \vdash f(\boxed{c(\underline{x})}^{\uparrow}, \ldots, F(\lfloor y \rfloor), \ldots) \\
 & & \vdash f(x, \ldots, \boxed{c'(\underline{F(\lfloor y \rfloor)})}^{\downarrow}, \ldots)
\end{array}
$$

Exception: missing casesplit

$$\forall a : obj. \forall t, s : list(obj).\ member(a, t) \to member(a, t <> s)$$

- Wave-rules

$$
\begin{array}{rcl}
\boxed{X :: \underline{Y}}^{\uparrow} <> Z & \Rightarrow & \boxed{X :: \underline{(Y <> Z)}}^{\uparrow} \\
X \neq Y \to member(X, \boxed{Y :: \underline{Z}}^{\uparrow}) & \Rightarrow & member(X, Z) \\
X = Y \to member(X, \boxed{Y :: \underline{Z}}^{\uparrow}) & \Rightarrow & \text{true}
\end{array}
$$

- Induction hypothesis

$$member(a, t) \to member(a, t <> S)$$

- Induction conclusion

$$
\begin{array}{ccc}
member(a, \boxed{h :: \underline{t}}^{\uparrow}) & \to & member(a, \boxed{h :: \underline{t}}^{\uparrow} <> \lfloor s \rfloor) \\
\underbrace{member(a, \boxed{h :: \underline{t}}^{\uparrow})}_{\text{blocked}} & \to & \underbrace{member(a, \boxed{h :: \underline{t <> \lfloor s \rfloor}}^{\uparrow})}_{\text{blocked}}
\end{array}
$$

Critic preconditions (casesplit):

- Preconditions *(1.1)* and *(1.2)* of the wave method hold, *e.g*

  1.1 There exists a wave-occurrence, *e.g.*

$$member(a, \boxed{h :: \underline{t}}^{\uparrow}) \to \ldots$$

  1.2 and a matching wave-rule, *e.g.*

$$X \neq Y \to member(X, \boxed{Y :: \underline{Z}}^{\uparrow}) \Rightarrow \ldots$$

- Precondition *(1.3)* is false, *i.e.*

  1.3 $a \neq h$ is not provable given the hypotheses.

Patch: casesplit

- Induction hypothesis

$$member(a,t) \rightarrow member(a, t <> S)$$

- Induction conclusion

$$member(a, \boxed{h :: \underline{t}}^{\uparrow}) \rightarrow member(a, \boxed{h :: \underline{(t <> \lfloor s \rfloor)}}^{\uparrow})$$

- Case: $a \neq h$

$$\begin{array}{lcl} member(a,t) & \rightarrow & member(a, \boxed{h :: \underline{(t <> \lfloor s \rfloor)}}^{\uparrow}) \\ member(a,t) & \rightarrow & member(a, t <> \lfloor s \rfloor) \end{array}$$

- Case: $a = h$

$$\begin{array}{lcl} \text{true} & \rightarrow & member(a, \boxed{h :: \underline{(t <> \lfloor s \rfloor)}}^{\uparrow}) \\ \text{true} & \rightarrow & \text{true} \end{array}$$

Critics — Examples

No	Theorem
1	$\forall x{:}nat.\ even(x + x)$
2	$\forall x, y{:}int.\ even(x + y) \leftrightarrow even(x - y)$
3	$\forall x{:}list(obj).\ evenl(x <> nil) \leftrightarrow even(length(x))$
4	$\forall x, y{:}list(obj).\ rev(x) <> (y <> x) = qrev(x, y) <> x$
5	$\forall x{:}list(obj).\ rev(x) = qrev(x, nil)$
6	$\forall x, y{:}list(obj).\ rev(rev(x <> y)) = rev(rev(x)) <> rev(rev(y))$
7	$\forall x{:}list(obj).\ rev(x) <> x = qrev(x, x)$
8	$\forall x{:}list(obj).\ \forall y{:}obj.\ rev(rev(x) <> y :: nil) = y :: x$
9	$\forall x, y{:}list(obj).\ rotate(length(x), x <> y) = y <> x$
10	$\forall x{:}list(obj).\ rotate(length(x), x) = x$

Critics — Examples (cont.)

No	Lemmas	Generalize	Nested Ind
1	$X + s(Y) = s(X + Y)$		
2			x
3			x
4	$rev(X <> Y :: nil) = Y :: rev(X)$ $qrev(X, Y) <> Z = qrev(X, Y <> Z)$		
5	$(X <> Y) <> Z = X <> (Y <> Z)$	x	
6	$rev(X <> Y :: nil) = Y :: rev(X)$		
7	$X <> Y :: Z = (X <> Y :: nil) <> Z$	x	
8	$X <> Y :: Z = (X <> Y :: nil) <> Z$	x	
9	$(X <> Y) <> Z = X <> (Y <> Z)$ $X <> Y :: Z = (X <> Y :: nil) <> Z$		
10	$(X <> Y) <> Z = X <> (Y <> Z)$ $X <> Y :: Z = (X <> Y :: nil) <> Z$	x	

Suggested Reading

The following reports are available by anonymous ftp from dream.dai.ed.ac.uk:

- Bundy *et. al.*, "Rippling: A Heuristic for Guiding Inductive Proofs", AI Journal, Vol 62, 1993.
- Ireland, "The Use of Planning Critics in Mechanizing Inductive Proofs", LPAR92.
- Ireland and Bundy, "Productive Use of Failure in Inductive Proof", DAI Report (forthcoming in July).

Induction Based on Rippling and Proof Planning
Part IV: Synthesis

David Basin
Max-Planck-Institut für Informatik
Saarbrücken

Induction & Synthesis

- Synthesis of programs by inductive theorem proving
  - Non-type theoretic approaches
  - Induction corresponds to well-founded recursion

Problem	Assertions
Functional Programs	$f(x) = t$ s.t. $S(x, f(x))$
Logic Programs	$\forall x.prog(x) \leftrightarrow spec(x)$
Hardware	$\forall x.prog(x) \rightarrow spec(x)$

- Synthesis of induction schemata
  - Construct induction schema during proof
- Combination: schema/program synthesis

Idea: Generalize Unification Based Synthesis

- Consider Prolog — synthesis/verification in Horn clause theory

$$H(\overline{x}) \leftarrow C_1(\overline{x}_1), ...C_n(\overline{x}_n)$$

- Example

$$prof(alan).$$
$$liked(alan).$$
$$niceprof(X) \leftarrow prof(X), liked(X).$$

- Query constructs proof

$$\dfrac{\dfrac{prof(alan)}{prof(alan)} \quad \dfrac{liked(alan)}{liked(alan)}}{niceprof(alan)} \qquad \dfrac{\dfrac{prof(alan)}{prof(X)}\{alan/X\} \quad \dfrac{liked(alan)}{liked(X)}\{X = alan\}}{niceprof(X)}\{X = alan\}$$

Generalize Unification Based Synthesis (cont.)

- Verification/synthesis identical (modulo unification base-case)
- Goes back (at least) to Cordell Green (1969)
- Generalize idea
  - Theory: Axiomatized clauses $\Longrightarrow$ general derived rules
  - Proof: SLD resolution $\Longrightarrow$ non-uniform proof search
  - Syntax: First order syntax $\Longrightarrow$ higher-order syntax
  - Objects Built: relations, recursive programs, ...

Generalization requires HO-syntax/unification

- Generalization based on idea of "schematic proofs"

$$\forall l\, m.\, (\forall z.\, z \in l \to z \in m) \leftrightarrow E(l,m)$$

  – Proof instantiates E with program — what is E?
- Proof rules generally schematic (e.g., over formulas)

$$\frac{A \quad B}{A \wedge B}$$

  – Can be applied with 1st-order matching/unification
- Verification = Synthesis when instantiation commutes with proof rules
  – E.g., Pure Prolog

Generalization — HO-syntax/unification

- Binding Requires H.O. syntax

$$\frac{\forall x.\, A[x]}{A[t/x]}\ \forall\text{-}E\ (t)$$

  – $\forall x.\, A$ fails, A of type o.
  ... Application of σ to $\forall x.\, A$: can't refer to x.
  ... Substitution $[t/x]$ to A: only defined for A ground
- HO-syntax & Unification suffice — $\forall : (i \to o) \to o$, $A : (i \to o)$
  – Instantiation commutes with proof rules ($\sigma = \{\lambda y.y + 3 = 5/A\}$)

$$\forall x.\, A(x)) \xrightarrow{\sigma} \forall x.\, x+3=5 \xrightarrow{\forall E\ (t)} t+3=5 \xleftarrow{\sigma} A(t) \xleftarrow{\forall E\ (t)} \forall x.\, A(x))$$

- Often only pattern unification needed: $P \leftarrow \lambda r\, s.a(s)$

$$\lambda x\, y\, z.P(x,z) =^{?} \lambda x\, y\, z.a(z)$$

Extended Example — Logic Program Synthesis

- First-order logic sufficient for formalizing/manipulating LPs
- Formalization — represent as "Pure Logic Programs"

$$\forall x\, y.mem(x,y) \leftrightarrow (x = [] \wedge False) \vee$$
$$(\exists v_0\, v_1.x = v_0.v_1 \wedge (y = v_0 \vee mem(v_1,y)))$$

  – represents

$$\begin{array}{lll} mem([],y). \leftarrow fail. & & \\ mem([V_0|V_1],Y). \leftarrow Y = V_0. & i.e. & mem([V_0|V_1],V_0). \\ mem([V_0|V_1],Y) \leftarrow mem(V_1,Y). & & mem([V_0|V_1],Y) \leftarrow mem(V_1,Y). \end{array}$$

- Correctness: Reason about equivalence in appropriate logical theory.

$$\forall l\, m.\, (\forall z.\, z \in l \to z \in m) \leftrightarrow E(l,m)$$

Logic Program Synthesis (cont.)

- Relation between objects/spec is $\leftrightarrow$
- Derive calculus for reasoning about $\leftrightarrow$
  – Trivial equivalences

$$\frac{P \leftrightarrow P' \quad Q \leftrightarrow Q'}{P \wedge Q \leftrightarrow P' \wedge Q'}\ \wedge \leftrightarrow\text{-}I$$

  – Induction

$$\frac{A_1 \quad A_2 \quad A_3}{A(x,y) \leftrightarrow P(x,y)}\ \leftrightarrow\text{-}Ind$$

where ...

$$\begin{array}{lll} A_1 & \equiv & \forall x\, y.\, P(x,y) \leftrightarrow (x = [] \wedge B(y)) \vee (\exists h\, t.\, x = h.t \wedge S(h,t,y)) \\ A_2 & \equiv & \forall y.\, A([],y) \leftrightarrow B(y) \\ A_3 & \equiv & \forall t.\, (\forall y.\, A(t,y) \leftrightarrow P(t,y)) \to \forall h\, y.\, A(h.t,y) \leftrightarrow S(h,t,y) \end{array}$$

Logic Program Synthesis (cont.)

- Subset Example: $\forall l\, m.\, (\forall z.\, z \in l \to z \in m) \leftrightarrow E(l,m)$
  - Unify with $\leftrightarrow$-*Ind* rule — induction on l when $\{l/x\}$

$$\frac{A_1 \quad A_2 \quad A_3}{\forall l\, m.\, (\forall z.\, z \in l \to z \in m) \leftrightarrow P(l,m)} \leftrightarrow\text{-}Ind$$

$$\begin{array}{lcl} A_1 & \equiv & \forall x\, y.\, P(x,y) \leftrightarrow x = [\,] \wedge B(y) \vee (\exists h\, t.x = h.t \wedge S(h,t,y)) \\ A_2 & \equiv & \forall y.\, ((\forall z.\, z \in [\,] \to z \in y) \leftrightarrow B(y)) \\ A_3 & \equiv & \forall y.\, (IH \to (\forall z.z \in h.t \to z \in y) \leftrightarrow S(h,t,y)) \\ IH & \equiv & (\forall z.\, z \in t \to z \in y) \leftrightarrow P(t,y)) \end{array}$$

Logic Program Synthesis (cont.)

- Base case: normalize

$$\cfrac{\cfrac{}{True \leftrightarrow B(y)}\ resolve\ A \leftrightarrow A}{\forall y.\, ((\forall z.\, z \in [\,] \to z \in y) \leftrightarrow B(y))}\ Normalize$$

- Produces substitution

$$\{True/B(y)\}$$

Logic Program Synthesis (cont.)

- Step Case: Use rewrite rules

$$\begin{array}{rcl} (A \vee B \to C) & \leftrightarrow & (A \to C) \wedge (B \to C) \\ \forall v.A(v) \wedge B(v)) & \leftrightarrow & (\forall v.A(v)) \wedge (\forall v.B(v)) \\ (\forall v.v = w \to A(v)) & \leftrightarrow & A(w) \end{array}$$

- Simplify A_3

$$\cfrac{\cfrac{A_4 \qquad \cfrac{}{A_5}\ Resolve\ IH}{\forall y.\, IH \to h \in y \wedge (\forall z.\, z \in t \to z \in y) \leftrightarrow S(h,t,y)}\ \wedge\leftrightarrow\text{-}I}{\forall y.\, (IH \to (\forall z.z \in h.t \to z \in y) \leftrightarrow S(h,t,y))}\ Rewriting$$

$$\begin{array}{lcl} A_4 & \equiv & \forall y.\, IH \to (h \in y \leftrightarrow S_1(h,t,y)) \\ A_5 & \equiv & \forall y.\, IH \to ((\forall z.\, z \in t \to z \in y) \leftrightarrow S_2(h,t,y)) \end{array}$$

- Substitutions $\{S_1(h,t,y) \wedge S_2(h,t,y)/S(h,t,y)\}$ $\{P(t,y)/S_2(h,t,y)\}$

Subset Example (cont.)

- Current proof state (including instantiations)

$$\frac{A_1 \quad A_4}{\forall l\, m.\, (\forall z.\, z \in l \to z \in m) \leftrightarrow P(l,m)} \leftrightarrow\text{-}Ind$$

$$\begin{array}{lcl} A_1 & \equiv & \forall x\, y.\, P(x,y) \leftrightarrow x = [\,] \wedge True \vee (\exists h\, t.x = h.t \wedge (S_1(h,t,y) \wedge P(h,y))) \\ A_4 & \equiv & \forall y.\, IH \to (h \in y \leftrightarrow S_1(t,h,y)) \end{array}$$

- A_4 also proved by induction $S_1(h,t,y) = Q(y,h)$
  - As before 3 goals
  ... 1st is schematic definition for $Q(y,h)$
  ... 2nd is base case: normalize
  ... 3rd is step-case: Rewrite/use new IH

Subset Example (cont.)

- Final proof state

$$\frac{A_1 \quad A_6}{\forall l\, m.\, (\forall z.\, z \in l \rightarrow z \in m) \leftrightarrow P(l,m)} \leftrightarrow\text{-}Ind$$

$$\begin{array}{lcl} A_1 & \equiv & \forall x\, y.\, P(x,y) \leftrightarrow x = [\,] \wedge True \vee (\exists h\, t.x = h.t \wedge (Q(y,h) \wedge P(t,y))) \\ A_6 & \equiv & \forall x\, y.\, Q(x,y) \leftrightarrow x = [\,] \wedge False \vee (\exists h\, t.x = h.t \wedge (x = h \vee Q(t,y))) \end{array}$$

- Translate to multi-mode Prolog program

$$\begin{array}{ll} p([\,],Y). & q([H|T],H). \\ p([H|T],Y) \leftarrow q(Y,H), p(T,Y). & q([H|T],Y) \leftarrow q(T,Y). \end{array}$$

Automating Synthesis

- Proof follows "schematic verification" plan

$$\forall x.\, spec(x) \leftrightarrow P(x)$$

- Induction: set up schematic definition

$$\forall x.\, P(x) \leftrightarrow ...B...S(h,t)...$$

  – Base case: normalize specification, produce assignment for B

$$spec(0) \leftrightarrow B$$

  – Step case: normalize specification, fertilize with IH.

$$(spec(t) \leftrightarrow P(t)) \rightarrow (spec(\boxed{h.\underline{t}}^{\uparrow}) \leftrightarrow S(h,t))$$

  – Fertilization produces assignment for S based on P

Automating Synthesis (cont.)

- Logic Programming Proof Plans have been implemented (Kraan's PhD)
- Simplification based on rippling
- Only pattern unification is required
  – Starting program is a pattern, i.e., $P(x_1, x_2)$
  – Rules manipulate patterns on RHS (program side)

$$\begin{array}{lcl} A_2 & \equiv & \forall y.\, A([\,],y) \leftrightarrow B(y) \\ A_3 & \equiv & \forall t.\, (\forall y.\, A(t,y) \leftrightarrow P(t,y)) \rightarrow \forall h\, y.\, A(h.t,y) \leftrightarrow S(h,t,y) \end{array}$$

  – Unification under patterns is "closed".

Synthesizing Induction Schemata

- Problem: induction variables & induction schemata

$$even(x) \wedge even(y) \rightarrow even(x+y)$$

- Axioms

$$\boxed{s(\underline{x})}^{\uparrow} + y \Rightarrow \boxed{s(\underline{x+y})}^{\uparrow}$$
$$even(\boxed{s(s(\underline{x}))}^{\uparrow}) \Rightarrow even(x)$$

- Induction schemata

$$\frac{P(0) \quad P(x) \rightarrow P(\boxed{s(\underline{x})}^{\uparrow})}{\forall x.P(x)}\ \text{1-step}$$

$$\frac{P(0) \quad P(s(0)) \quad P(x) \rightarrow P(\boxed{s(s(\underline{x}))}^{\uparrow})}{\forall x.P(x)}\ \text{2-step}$$

- Questions:
  – How do we pick induction variables?
  – How do we pick induction schemata?

Synthesizing Induction Schemata (cont.)

- Proof

$$even(\boxed{s(s(\underline{x}))}^{\uparrow}) \wedge even(y) \rightarrow even(\boxed{s(s(\underline{x}))}^{\uparrow} + y)$$
$$even(x) \wedge even(y) \rightarrow even(\boxed{s(\underline{\boxed{s(\underline{x})}^{\uparrow} + y})}^{\uparrow})$$
$$even(x) \wedge even(y) \rightarrow even(\boxed{s(s(\underline{x+y}))}^{\uparrow})$$
$$even(x) \wedge even(y) \rightarrow even(x+y)$$

- Induction on y would be flawed
- One step induction on x would fail
- Recursion analysis works here — but only 1 step look-ahead!

Synthesizing Induction Schemata

- Idea: Pick induction variable/schemata "middle-out"

$$even(C(x)) \wedge even(D(y)) \rightarrow even(C(x) + D(y))$$

- Begin with rippling (e.g., on RHS)

$$even(\boxed{s(\underline{C_1(x)})}^{\uparrow}) \wedge even(D(y)) \rightarrow even(\boxed{s(\underline{C_1(x) + D(y)})}^{\uparrow}) \qquad \{\lambda x.\boxed{s(\underline{C_1(x)})}^{\uparrow}/C\}$$
$$even(\boxed{s(s(\underline{C_2(x)}))}^{\uparrow}) \wedge even(D(y)) \rightarrow even(\boxed{s(s(\underline{C_2(x) + D(y)}))}^{\uparrow})) \qquad \{\lambda x.\boxed{s(\underline{C_2(x)})}^{\uparrow}/C_1\}$$
$$even(C_2(x)) \wedge even(D(y)) \rightarrow even(C_2(x) + D(y))$$
$$even(x) \wedge even(y) \rightarrow even(x+y) \qquad \{\lambda x.x/C_2,\ \lambda x.x/D\}$$

- Synthesized $\{\lambda x.\boxed{s(s(\underline{x}))}^{\uparrow}/C,\ \lambda x.x/D\}$

$$even(s(s((x)))) \wedge even(y) \rightarrow even(s(s(x)) + y)$$

  - Suggests 2 step schemata and induction on x
  - Delayed commitment allows arbitrary large look-ahead into rippling
  - Rippling constrains rewriting & unification

Synthesizing Noetherian Induction

$$\frac{\forall x.(\forall y.R(y,x) \rightarrow P(y)) \rightarrow P(x) \quad wf(R)}{P(x)}$$

- Synthesize logic programs

$$\frac{\forall x.P(x) \leftrightarrow B(x) \quad \forall x.(\forall y.R(y,x) \rightarrow (Spec(y) \leftrightarrow P(y)) \rightarrow Spec(y) \leftrightarrow B(x) \quad wf(R)}{Spec(x) \leftrightarrow P(x)}$$

- Synthesize functional programs

$$\frac{\forall x.F(x) = B(x) \quad \forall x.(\forall y.R(y,x) \rightarrow Spec(y, F(y)) \leftrightarrow Spec(x, B(x)) \quad wf(R)}{Spec(x, F(x))}$$

- Realization in HOL — derive induction from wf definition

$$wf(R) \equiv (\forall P.(\forall x.(\forall y.R(y,x) \rightarrow P(y)) \rightarrow P(x)) \rightarrow (\forall x.P(x)))$$

- Idea: Can synthesize both program and R

Synthesizing Noetherian Induction (cont.)

- Sketch: quick sort

$$\forall x.perm(x, F(x)) \wedge ordered(F(x))$$

- After Induction
  - Build program $F(x) = B(x)$ by proving

1) $\forall y.R(y,x) \rightarrow (perm(y, F(y)) \wedge ordered(F(y))) \vdash (perm(x, B(x)) \wedge ordered(B(x))))$
2) $\vdash wf(R)$

- After case split ($x = []$), use IH with $\{less(hd(x), tl(x))/y\}$

$$R(less(hd(x), tl(x)), x) \rightarrow$$
$$(perm(less(hd(x), tl(x))), F(less(hd(x), tl(x)))) \wedge ordered(F(less(hd(x), tl(x))))$$

- To use IH ($\rightarrow$-E), get new goal

$$R(less(hd(x), tl(x)), x)$$

  - Along with subgoal (2) forms constraint for MOR induction

Comparison — Type Theory

- Type theory is alternative foundation for deriving programs
- Well-founded induction corresponds to well-founded recursion
  - Analogous to Nordstrom's Acc type
- Type theory has advantage that all terms compute
  - Disadvantage: terms are *functional programs*
  - Must show faithfulness of other (e.g., logic programming) encodings
- Synthesis also understandable using unification
  - Proofs-as-programs = proofs-as-objects + objects-as-programs
  - PAO explained via unification
  - OAP explained via constructivity

$$t \in A \to B$$

  - Two parts are independent

Summary of Results on Synthesis

- Induction + meta-variables $\Rightarrow$ "recursive structures"
  - Multi-mode logic programs from non-executable specification
- Can *derive* development calculi
  - Logic & functional calculi derived in HOL in Isabelle
- Can *partially automate* rewriting
  - Structure of proofs amenable to rippling
- Can *synthesize/partially automate* induction
  - Constructor style
  - Well-founded

Suggested Reading

The following reports are available by anonymous ftp from mpi-sb.mpg.de and dream.dai.ed.ac.uk:

- Basin, "Logic Frameworks for Logic Programs", LOPSTR94.
- Kraan, Basin, Bundy, "Logic Program Synthesis via Proof Planning", *Logic Program Synthesis and Transformation*, 1993.
- Kraan, Basin, Bundy, "Middle-Out Reasoning for Logic Program Synthesis", ICLP93.
- Ayari, Basin, "Deductive Tableaux as Higher-Order Resolution" (in preparation).

Induction Based on Rippling and Proof Planning
Part V: Rippling in Other Settings (& Comparisons)

David Basin
Max-Planck-Institut für Informatik
Saarbrücken

Generality/Comparison of Rippling Based Approach

- Rippling applicable in other induction settings
  - Comparison with induction completion
- Rippling applicable in non-induction settings
  - Serves as "goal directed problem solving"
  - Difference unification = identification of differences
  - Rippling = removal of differences

Implicit ("Inductionless") Induction

- Truth with respect to standard model: $\mathcal{I}(\mathcal{E}) \models E$

$$\{0 + X = X,\ s(X) + Y = s(X+Y)\} \models X + Y = Y + X$$

- Incomplete methods based on completion and consistency under extension
- Do not explicitly construct induction proofs
- Tight theoretical/practical relationship between Explicit Induction and IC
  - Recursion analysis/rippling in explicit induction
  - Critical pair generation/simplification in IC

Inductive Completion (abstractly)

- Input: (ground) confluent set $\mathcal{R}$, equations $\mathcal{L}$, equations $\mathcal{C}$.
- Proof
  - Deduction: the generation of critical pairs between *Rules* and $\mathcal{C}$.
  - Simplification: rewriting with $\mathcal{R}$, $\mathcal{L}$ and $\mathcal{C}$.
  - Ends when all equations in $\mathcal{C}$ proven ("deleted")
  - Or an inconsistency is detected

  ... $true = false$, $C(x) = D(x)$, ...

Deduction Phase — Similar to Explicit Induction

- Consider definition by primitive recursion

$$\begin{aligned} f(0,\vec{Y}) &\rightarrow g(\vec{Y}) \\ f(s(X),\vec{Y}) &\rightarrow h(X,\vec{Y},f(X,\vec{Y})) \end{aligned}$$

- Critical pairs with goal

$$P[f(X,\vec{Y})] = Q(X,\vec{Y})$$

Critical Expression	Critical Pair	Explicit Induction
$P[f(0,\vec{Y})]$	$\langle P[g(\vec{Y})], Q(0,\vec{Y})\rangle$	$P[f(0,\vec{Y})] = Q(0,\vec{Y})$
$P[f(s(X),\vec{Y})]$	$\langle P[h(X,\vec{Y},f(X,\vec{Y}))], Q(X,\vec{Y})\rangle$	$P[f(s(X),\vec{Y})] = Q(s(X),\vec{Y})$

- Correspondence:
  - critical pairs express desired equalities
  - pairs one rewrite step removed from explicit structural induction
  - One step is the application rewrite to LHS of pair
  - Rewrite step insures IH applied to "smaller" terms

Inductive Completion — Even Example

- Axioms

$$\begin{aligned} 0+X &\rightarrow X \\ s(X)+Y &\rightarrow s(X+Y) \\ \mathit{even}(0) &\rightarrow \mathit{true} \\ \mathit{even}(s(0)) &\rightarrow \mathit{false} \\ \mathit{even}(s(s(X))) &\rightarrow \mathit{even}(X) \end{aligned}$$

- Goal

$$\mathit{even}(X) \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(X+Y)) \rightarrow \mathit{true}$$

Even Example — critical pairs/inductions

Critical Expression/Critical Pair	Scheme	Subgoal
$\mathit{even}(0) \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(0+Y)$	sX	base: 0
$\langle \mathit{true}, \mathit{even}(0) \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(Y)\rangle$		$\underline{X}+Y$
$\mathit{even}(sX) \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(sX+Y)$	sX	inductive
$\langle \mathit{true}, \mathit{even}(sX) \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(s(X+Y))\rangle$		$\underline{X}+Y$
$\mathit{even}(0) \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(0+Y)$	ssX	base: 0
$\langle \mathit{true}, \mathit{true} \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(0+Y)\rangle$		$\mathit{even}(\underline{X})$
$\mathit{even}(s0) \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(s0+Y)$	ssX	base: $s0$
$\langle \mathit{true}, \mathit{false} \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(s0+Y)\rangle$		$\mathit{even}(\underline{X})$
$\mathit{even}(ssX) \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(ssX+Y)$	ssX	inductive
$\langle \mathit{true}, \mathit{even}(X) \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(ssX+Y)\rangle$		$\mathit{even}(\underline{X})$
$\mathit{even}(X) \wedge \mathit{even}(0) \Rightarrow \mathit{even}(X+0)$	ssY	base: 0
$\langle \mathit{true}, \mathit{even}(X) \wedge \mathit{true} \Rightarrow \mathit{even}(X+0)\rangle$		$\mathit{even}(\underline{X})$
$\mathit{even}(X) \wedge \mathit{even}(s0) \Rightarrow \mathit{even}(X+s0)$	ssY	base: $s0$
$\langle \mathit{true}, \mathit{even}(X) \wedge \mathit{false} \Rightarrow \mathit{even}(X+s0)\rangle$		$\mathit{even}(\underline{Y})$
$\mathit{even}(X) \wedge \mathit{even}(ssY) \Rightarrow \mathit{even}(X+ssY)$	ssY	inductive
$\langle \mathit{true}, \mathit{even}(X) \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(X+ssY)\rangle$		$\mathit{even}(\underline{Y})$

Even Example — Recursion Analysis

- Superposition performs recursion analysis
  - In practice chose from "complete positions" (induction variables)

$$\mathit{even}(X) \wedge \mathit{even}(Y) \Rightarrow \mathit{even}(X+Y) \rightarrow \mathit{true}$$

Variable	Function	Schema	Recursion Term	Status
X	*even*	2-step	ssX	unflawed
Y	*even*	2-step	ssY	flawed
X	$+$	1-step	sX	subsumed

- Only 1-step lookahead at one position
- No schema merging
- Can only use schemata suggested by recursive definitions

Simplification/Rippling in Inductive Completion

- Simplification corresponds to rippling

$$R = \{0 + X \to X,\ s(X) + Y \to s(X+Y)$$
$$0 * X \to X,\ s(X) * Y \to Y + X * Y\}$$

- Conjecture $(y+z) * x = y * x + z * x$
- Induction Step: $\langle s(y+z) * x, s(y) * x + z * x \rangle$
- Simplifies to: $\langle x + (y+z) * x, (x + y * x) + z * x \rangle$
- Can use conjecture to further simplify

$$\langle x + (y * x + z * x), (x + y * x) + z * x \rangle.$$

- Blocked — terms are irreducible

Rippling in IC

- Rippling directly applicable to simplification
- Recursive definitions in $\mathcal{R}$ parsed as wave-rules.

$$f(s(X), \vec{Y}) \to h(X, \vec{Y}, f(X, \vec{Y}))$$

  – Parses as

$$f(\boxed{s(\underline{X})}^{\uparrow}, \vec{Y}) \to \boxed{h(X, \vec{Y}, \underline{f(X, \vec{Y})})}^{\uparrow}$$

- Critical pairs can be annotated (see following slides)
- Lemmas in $\mathcal{L}$ often wave-rules.
- Previous example

$$\langle \boxed{s(\underline{y+z})}^{\uparrow} * x, \boxed{s(\underline{y})}^{\uparrow} * x + z * x \rangle$$

  – Ripples to

$$\langle \boxed{x + \underline{(y+z) * x}}^{\uparrow}, \boxed{(x + \underline{y * x})}^{\uparrow} + z * x \rangle.$$

  – Associative of plus

$$\boxed{(X + \underline{Y})}^{\uparrow} + Z \to \boxed{X + \underline{(Y + Z)}}^{\uparrow}$$

  – Complete proof with fertilization

IC versus Clam/Proof Planning

- Many IC algorithms/techniques
  – Basic, i.e., Huet/Hullot
  – Cover-set procedures (Bachmair, Reddy)
  – Complete positions — induction variables (Fribourg)
  – Lemmata (Gobel, Kuchlin), Generalization (Gramlich), ...
- As restrictions drop, control becomes important
  – Which cover set does one pick?
  – How does simplify with lemmata
- Suggests scope for cross-fertilization of ideas

Rippling Outwidth Induction

- Rippling well suited for induction
  – Reason: Can direct proof towards desired result (Induction Hyp)
- Can be used in other domains for goal-directed rewriting
  – Algebraic problem solving: isolate unknowns
- Example: arithmetic series — use "standard results"
- Problem: must identify differences first!

Difference Identification = Difference Unification

- Homeomorphic Embedding — $s \trianglerighteq t$ if $s \xrightarrow{*} t$

$$f(x_1, ..., x_n) \rightarrow x_i$$

  – Example $f(s(x)) \trianglerighteq f(x)$, rewriting first subterm

$$skel(f(\boxed{s(\underline{x})}^{\uparrow})) = f(x)$$

- Idea generalizes — "difference unification"

$$unif(f(X,a), f(s(x),a)) = \{a/X\}$$

  – Return annotation suitable for rippling

$$s = h(f(X,a),a) \qquad t = f(s(a),X)$$

$$du(s,t) = \langle \boxed{h(\underline{f(X,a)},a)}^{\uparrow}, f(\boxed{s(\underline{a})}^{\uparrow}, X), \{a/X\}\rangle$$

- Wanted: $du(s,t) = \langle s', t', \sigma \rangle$
  – where $\sigma(s')$ & $\sigma(t')$ share the same skeleton.

Difference Unification

$$..., x < y \times y, ... \vdash x < s(y) \times s(y)$$

- Rule-based difference identification

Decompose	$f(s_1, ..., s_k) =^? f(t_1, ..., t_k)$	$\rightarrow$	$\{s_1 =^? t_1, ..., s_k =^? t_k\}$
Delete	$t =^? t$	$\rightarrow$	$\{\ \}$
Hide	$t =^? f(s_1, ..., s_i, ...s_k)$	$\rightarrow$	$\{t =^? s_i\} + \boxed{f(s_1, ..., \underline{s_i}, ...s_k)}^{\uparrow}$

- Example

Rule	Remaining Equations
Decompose	$\{x < y \times y =^? x < s(y) \times s(y)\}$
Decompose	$\{x =^? x,\ y \times y =^? s(y) \times s(y)\}$
Hide (2)	$\{x =^? x,\ y =^? \boxed{s(\underline{y})}^{\uparrow},\ y =^? \boxed{s(\underline{y})}^{\uparrow}\}$
Delete (3)	$\{x =^? x,\ y =^? y,\ y =^? y\}$
	$\{\ \}$

- Result

$$..., x < y \times y, ... \vdash x < \boxed{s(\underline{y})}^{\uparrow} \times \boxed{s(\underline{y})}^{\uparrow}$$

Series

- Problem find "solved form" (without summation) of series
- Proofs often guided by standard results (C constant)

$$\sum_{i=0}^{N} i = \frac{N \times s(N)}{2} \qquad \sum_{i=0}^{N} C = s(N) \times C$$

- Proof technique: difference unify problem with solved forms

$$\sum_{j=0}^{m} \boxed{\sum_{k=0}^{n} k \times \underline{j} + c}^{\uparrow}$$

- and ripple with wave-rules

$$\sum_{j=A}^{B} \boxed{\sum_{k=C}^{D} \underline{U}}^{\uparrow} \rightarrow \boxed{\sum_{k=C}^{D} \underline{\sum_{j=A}^{B} U}}^{\uparrow} \qquad (1)$$

$$\sum_{j=A}^{B} \boxed{C \times \underline{U}}^{\uparrow} \rightarrow \boxed{C \times \underline{\sum_{j=A}^{B} U}}^{\uparrow} \qquad (2)$$

$$\sum_{j=A}^{B} \boxed{\underline{U} \times C}^{\uparrow} \rightarrow \boxed{\underline{\sum_{j=A}^{B} U} \times C}^{\uparrow} \qquad (3)$$

$$\sum_{j=A}^{B} \boxed{\underline{U} + V}^{\uparrow} \rightarrow \boxed{\underline{\sum_{j=A}^{B} U} + \sum_{j=A}^{B} V}^{\uparrow} \qquad (4)$$

Series (cont.)

$$\sum_{j=0}^{m} \boxed{\sum_{k=0}^{n} k \times \underline{j} + c}^{\uparrow}$$

- Rewrite with (??), (??), (??):

$$\boxed{\sum_{k=0}^{n} \underline{\sum_{j=0}^{m} \boxed{k \times \underline{j} + c}^{\uparrow}}}^{\uparrow}$$

$$\boxed{\sum_{k=0}^{n} \underline{(\sum_{j=0}^{m} \boxed{k \times \underline{j}}^{\uparrow} + \sum_{j=0}^{m} c)}}^{\uparrow}$$

$$\boxed{\sum_{k=0}^{n} (k \times \underline{\sum_{j=0}^{m} j} + \sum_{j=0}^{m} c)}^{\uparrow}$$

- Fertilize with standard result

$$\sum_{k=0}^{n} (k \times \frac{m \times s(m)}{2} + s(m) \times c)$$

Series (cont.)

- Re-difference unify against standard forms

$$\sum_{k=0}^{n} \boxed{(\underline{k} \times \frac{m \times s(m)}{2} + s(m) \times c)}^{\uparrow}$$

- Ripple with (??), (??)

$$\boxed{\underline{\sum_{k=0}^{n} \boxed{(\underline{k} \times \frac{m \times s(m)}{2})}^{\uparrow}} + \sum_{k=0}^{n} s(m) \times c}^{\uparrow}$$

$$\boxed{(\underline{\sum_{k=0}^{n} k}) \times \frac{m \times s(m)}{2} + \sum_{k=0}^{n} s(m) \times c}^{\uparrow}$$

- Fertilize and iterate again; conclude with

$$\frac{n \times s(n)}{2} \times \frac{m \times s(m)}{2} + s(n) \times s(m) \times c$$

Series (cont.)

- DU + Rippling is a flexible approach to solving series

Series	Closed Form
$\sum i^2$	$\frac{2.n^3+3.n^2+n}{6}$
$\sum a^i$	$\frac{a^{s(n)}-1}{a-1}$
$\sum \frac{1}{i.(i+1)}$	$\frac{n}{s(n)}$
$\sum F_i$	$F_{n+2} - 1$
$\sum \begin{pmatrix} s(i) \\ s(m) \end{pmatrix}$	$\begin{pmatrix} s(n) \\ s(s(m)) \end{pmatrix} + \begin{pmatrix} s(n) \\ s(m) \end{pmatrix}$

All sums are from 0 to n, $a \neq 1$, F_i is the ith Fibonacci number.

Table 1: Some series summed by CLAM using difference unification and rippling.

Historical/Comparison — General Difference Reduction

- History of similarity & difference reduction heuristics
- Pedagogic based work: simulate humans on same task
  - Logic Machine of Newell, Shaw, and Simon (1950s)
- Search space reduction: Reorder infinite space
  - Resolution Theorem Proving: E and RUE Resolution

$$P(h(a), b) \leftrightarrow P(g(a), b) \quad (\{P(h(a), b)\}, \{\neg P(g(a), b)\})$$

... Failed unifier represents differences

$$h(a) =^? g(a) \quad (\{\neg Eq(h(a), g(a))\})$$

... Differences are manipulated by equality reasoning

... Yields demand driven paramodulation

Historical/Comparison — General Difference Reduction

	RUE	Rippling Based
Difference Identification	Failed Unification	Difference Unification
Type	Term	Context
Difference Manipulation	Paramodulation	Context Rewriting
Difference Reduction	(Term Orderings)	Context Orderings
	(May diverge)	(terminating)
Calculus	Unification Based	“Modified” Rewriting

- Difference Reduction Strategies Important

  Although paramodulation was a substantial improvement compared to the axiomatical formalization of the equality relation, it still leads to enormous search spaces, as this inference rule can be applied almost everywhere in the clause space. .... Whereas most research is based with remarkable success on simplification mechanisms, especially on demodulation and term rewriting, difference reduction methods have found less attention, although they are at least as important for an automated deduction system.

  Bläsius & Siekmann, CADE 9

- Especially successful in induction given “strong hint” of I.H.

Suggested Reading

The following reports are available by anonymous ftp from mpi-sb.mpg.de and dream.dai.ed.ac.uk:

- Barnett, Basin, & Hesketh, “A Recursion Planning Analysis of Inductive Completion”, Annals of Mathematics and AI, 1993.
- Basin & Walsh “Difference Unification”, IJCAI93.
- Basin & Walsh “Symbolic Reasoning by Difference Reduction”, unpublished report.
- Walsh, Nunes, & Bundy, “The Use of Proof Plans to Sum Series”, CADE 11.

MAX-PLANCK-INSTITUT FÜR INFORMATIK

Logic Frameworks for Logic Programs

David A. Basin

MPI–I–94–218 June 1994

INFORMATIK

Im Stadtwald
D 66123 Saarbrücken
Germany

Authors' Addresses

David Basin Max-Planck-Institut für Informatik Im Satdwald, D-66123 Saarbrücken, Germany
basin@mpi-sb.mpg.de

Publication Notes

A version of this paper will appear at the Fourth International Workshop on Logic Program Synthesis and Transformation (LOPSTR'94), 19 – 21 June, 1994, Pisa, Italy.

Acknowledgements

I thanks Tobias Nipkow and Larry Paulson for their advice on Isabelle. I also thank Alan Bundy, Ina Kraan, and Sean Matthews for discussions on this work and collaboration on related work. This research was funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9102. The responsibility for the contents lies with the author.

Abstract

We show how logical frameworks can provide a basis for logic program synthesis. With them, we may use first-order logic as a foundation to formalize and derive rules that constitute program development calculi. Derived rules may be in turn applied to synthesize logic programs using higher-order resolution during proof that programs meet their specifications. We illustrate this using Paulson's Isabelle system to derive and use a simple synthesis calculus based on equivalence preserving transformations.

1 Introduction

Background

In 1969 Dana Scott developed his Logic for Computable Functions and with it a model of functional program computation. Motivated by this model, Robin Milner developed the theorem prover LCF whose logic PPλ used Scott's theory to reason about program correctness. The LCF project [13] established a paradigm of formalizing a programming logic on a machine and using it to formalize different theories of functional programs (e.g., strict and lazy evaluation) and their correctness; although the programming logic was simple, within it complex theories could be developed and applied to functional program verification.

This paradigm can be characterized as *formal development from foundations*. Type theory and higher-order logic have been also used in this role. A recent example is the work of Paulson with ZF set theory. Although this theory appears primitive, Paulson used it to develop a theory of functions using progressively more powerful derived rules [24].

Most work in formalized program development starts at a higher level; foundations are part of an informal and usually unstated meta-theory. Consider, for example, transformation rules like Burstall and Darlington's well known fold-unfold rules [7]. Their rules are applied to manipulate formulas and derive new ones; afterwards some collection of the derived formulas defines the new program. The relationship of the new formulas to the old ones, and indeed which constitute the new program is part of their informal (not machine formalized) metatheory. So is the correctness of their rules (see [18, 8]). In logic programming the situation is similar; for example, [30, 29] and others have analyzed conditions required for fold-unfold style transformations to preserve equivalence of logic programs and indeed what "equivalence" means.

Development from Foundations in Logic Programming

We propose that, analogous to LCF, we may begin with a programming logic and derive within it a program development calculus. Derived rules can be applied to statements about program correctness formalized in the logic and thereby verify or synthesize logic programs. Logic programming is a particularly appropriate domain to formalize such development because under the declarative interpretation of logic programs as formulas, programs are formalizable within a fragment of first-order logic and are therefore amenable to manipulation in proof systems that contain this fragment. Indeed, there have been many investigations of using first-order logic to specify and derive correct logic programs [9, 10, 11, 17, 19]. But this work, like that of Burstall and Darlington, starts with the calculus rather than the foundations. For example in [17] formulas are manipulated using various simplification rules and at the end a collection of the resulting formulas constitutes the program. The validity of the rules and the relationship of the final set of formulas (which comprise the program) to the specification is again part of the informal meta-theory.

Our main contribution is to demonstrate that without significant extra work much of the informal metatheory can be formalized; we can build calculi from foundations and carry out proofs where our notion of correctness is more explicit. However, to do this, a problem must be overcome: first-order logic is too weak to directly formalize and derive proof rules. Consider for example, trying to state that in first-order logic we may replace any formula $\forall x.A$ by $\neg\exists x.\neg A$. We might wish to formulate this as $\forall x.A \to \neg\exists x.\neg A$. While this is provable for any instance A, such a generalized statement cannot be made in first-order logic itself; some kind of second-order quantification is required.[1] In particular, to formalize proof rules of a logic, one must express rules that (in the terminology of [15]) are *schematic* and *hypothetical*. The former means that rules may contain variables ranging over formula. The latter means that one may represent

[1] First-order *logic* is too weak, but it is possible to formalize powerful enough first-order *theories* to express such rules by axiomatizing syntax, e.g., [32, 3, 23]. However, such approaches require some kind of reflection facility to establish a link between the formalized meta-theory and the desired theory where such rules are used. See [4] for a further discussion of this. Moreover, under such an approach unification cannot be used to identify program verification and synthesis.

logical consequence; in the above example consequence has been essentially internalized by implication in the object language.

Rather than moving to a more powerful logic like higher-order logic, we show that one can formalize program development using weak logics embedded in logical frameworks such as Paulson's Isabelle system [25] or Pfenning's ELF [28]. In our work, a programming logic (also called the *object logic*) is encoded in the logic of the logical framework (the *meta-logic*). For example, the meta-logic of Isabelle, which we use, is fragment of higher-order logic containing implication (to formalize hypothetical rules) and universal quantification (to formalize schematic rules). Within this meta-logic we formalize a theory of relevant data-types like lists and use this to specify our notion of program correctness and derive rules for building correct programs. Moreover, Isabelle manipulates rules using higher-order unification and we use this to build programs during proof where meta-variables are incrementally instantiated with logic programs.

We illustrate this development paradigm by working through a particular example in detail. Within an Isabelle theory of first-order logic we formulate and derive a calculus for reasoning about equivalences between specifications and representations of logic programs in first-order logic. The derived calculus can be seen as a formal development of a logic for program development proposed in Wiggins (see Section 3.4). After derivation, we apply these rules using higher-order unification to verify that logic programs meet their specifications; the logic programs are given by meta-variables and each unification step during proof incrementally instantiates them.

Our experience indicates that this development is quite manageable. Isabelle comes with well-developed tactic support for rewriting and simplification. As a result, our derivation of rules was mostly trivial and involved no more than typing them in and invoking the appropriate first-order simplification tactics. Moreover, proof construction with these rules was partially automated by the use of Isabelle's standard normalization and simplification procedures. We illustrate this by developing a program for list subset.

2 Background to Isabelle

What follows is a brief overview of Isabelle [25, 26, 27] as is necessary for what follows. Isabelle is an interactive theorem prover developed by Larry Paulson. It is a logical framework: its logic serves as a meta-logic in which object logics (e.g., first-order logic, set theory, etc.) are encoded. Proofs are interactively constructed by applying proof rules using higher-order resolution. Proof construction may be automated using *tactics* which are ML programs in the tradition of LCF that construct proofs.

Isabelle provides a fine basis for our work. Since it is a logical framework, we may encode in it the appropriate object logic, first-order logic (although we indicate in Section 5 other possible choices). Isabelle's metalogic is based on the implicational fragment of higher-order logic where implication is represented by "`==>`" and universal quantification by "`!!`"; hence we can formalize and derive proof rules which are both hypothetical and schematic. Rules, primitive and derived, may be applied with higher-order unification during higher-order resolution; unification permits meta-variables to occur both in rules and proofs. We use this to build logic programs by theorem proving where the program is originally left unspecified as a higher-order meta-variable and is filled in incrementally during the resolution steps; the use of resolution is similar to the use of "middle out reasoning" to build logic programs as demonstrated in [20, 21].

Isabelle manipulates objects of the form[2] `[|F1; ...; Fn|] ==> F`. A proof proceeds by applying rules to such formulas which result in zero or more subgoals, possibly with different assumptions. When there are no subgoals, the proof is complete. Although Isabelle proof rules are formalized natural deduction style, the above implication can be read as an intuitionistic sequent where the `Fi` are the hypotheses. Isabelle has resolution tactics which apply rules in a way the maintains this illusion of working with sequents.

[2] We shall use `typewriter font` to display concrete Isabelle syntax.

3 Encoding A Simple Equivalence Calculus

We give a simple calculus for reasoning about equivalence between logic programs and their specifications. Although simple, it illustrates the flavor of calculus and program development we propose.

3.1 Base Logic

We base our work on standard theories that come with the Isabelle distribution. We begin by selecting a theory of constructive first-order predicate calculus and augment this with a theory of lists to allow program development over this data-type (See *IFOL* and *List* in [27]). The list theory, for example, extends Isabelle's first-order logic with constants for the empty list "[]", cons ".", and standard axioms like structural induction over lists. In addition, we have extended this theory with two constants called `Wfp` (well-formed program) and `Def` with the property that `Wfp(P) = Def(P) = P` for all formulas `P`; their role will be clarified later.

The choice of lists was arbitrary; to develop programs over numbers, trees, etc. we would employ axiomatizations of these other data-types. Moreover, the choice of a constructive logic was also arbitrary. Classical logic suffices too as the proof rules we derive are clearly valid after addition of the law of excluded middle. This point is worth emphasizing: *higher-order unification, not any feature of constructivity, is responsible for building programs from proofs in our setting.*

In this theory, we reason about the equivalence between programs and specifications. "Equivalence" needs clarification since even for logic programs without impure features there are rival notions of equivalence. The differences though (see [22, 5]) are not so relevant in illustrating our suggested methodology (they manifest themselves through different formalized theories). The notion of equivalence we use is equivalence between the specification and a logic program represented as a *pure logic program* in the above theory. *Pure logic programs* themselves are equivalences between a universally quantified atom and a formula in a restricted subset of first-order logic (see [6] for details); they are similar to the *logic descriptions* of [12].

For example, the following is a pure logic program for list membership (where *cons* is ".").[3]

$$\forall x\, y.p(x,y) \leftrightarrow (x = [] \wedge False) \vee (\exists v_0\, v_1.x = v_0.v_1 \wedge (y = v_0 \vee p(v_1, y))) \tag{1}$$

Such programs can be translated to Horn clauses or run directly in a language like Gödel [16].

3.2 Problem Specification

As our notion of correctness is equivalence between programs and specifications, our proofs begin with formulas of the form $\forall \overline{x}.(spec(\overline{x}) \leftrightarrow E(\overline{x}))$. The variables in $\overline{x}$ represent parameters to both the specification *spec* and the logic program E; we do not distinguish input from output. *spec* is a first-order specification and E is either a concrete (particular) pure logic program or a schematic (meta) variable standing in for such a program. If E is a concrete formula then a proof of this equivalence constitutes a *verification* proof as we are showing that E is equivalent to its specification. If E is a second-order meta-variable then a proof of this equivalence that instantiates E serves as a *synthesis* proof as it builds a program that meets the *spec*. If *spec* is already executable we might consider such a proof to be a *transformation* proof.

An example we develop in this report is synthesizing a program that given two lists l and m is true when l is a subset of m. This example has been proposed by others, e.g., [17, 33]. Slipping into Isabelle syntax we specify it as

```
ALL l m. (ALL z. In(z,l) --> In(z,m)) <-> ?E(l,m).
```

[3] Unfortunately, "." is overloaded and also is used in the syntax of quantifiers; e.g., $\forall x\, y.\phi$ which abbreviates $\forall x.\forall y.\phi$.

Note that `ALL`, `-->` and `<->` represent first-order universal quantification, implication, and equivalence, and are declared in the definition of first-order logic. The "`?`" symbol indicates metavariables in Isabelle. Note that `?E` is a function of the input lists `l` and `m` but `z` is only part of the specification. Higher-order unification, which we use to build an instance for `?E` will ensure that it is only a function of `l` and `m`.

3.3 Rules

We give natural deduction rules where the conclusion explains how to construct `?E` from proofs of the subgoals. These rules form a simple calculus for reasoning about equivalences and can be seen as a reconstruction of those of the Whelk system (see Section 3.4). Of course, since `A <-> A` is valid, the synthesis specification for subset can be immediately proven by instantiating `?E` with the specification on the left hand side of the equivalence. While this would lead to a valid proof, it is uninteresting as the specification does not suggest an algorithm for computing the subset relation. To make our calculus interesting, we propose rules that manipulate equivalences with restricted right-hand sides where the right hand side can be directly executed.

Specifically, we propose rules that admit as right hand sides formulas like the body of the membership predicate given above, but exclude formula like `ALL z. In(z,l) --> In(z,m)`. To do this we define inductively the set of such admissible formulas. They are built from a collection of (computable) base-relations and operators for combining these that lead to computable algorithms provided their arguments are computable. In particular, our base relations are the relations `True`, `False`, equality and inequality. Our operators will be the propositional connectives and existential quantification restricted to a use like that in the membership example, i.e., of the form $\exists v_0 v_1 . x = v_0 . v_1 \wedge P$ where P is admissible. This limited use of existential quantification is necessary for constructing recursive programs in our setting; it can be trivially compiled out in the translation to Horn clauses.

Note that to be strictly true to our "foundations" paradigm, we would specify the syntax of such well-formed logic programs in our theory (which we could do by recursively defining a unary well-formedness predicate that captures the above restrictions). However, to simplify matters we capture it by only deriving rules for manipulating these equivalences where the right-hand sides meet these restrictions. To ensure that only these rules are used to prove equivalences we will resort to a simple trick. Namely, we wrap all the right hand sides of equivalences in our rule, and in the starting specification with the constructor `Wfp`. E.g., our starting goal for the subset proof would really be

```
ALL l m. (ALL z. In(z,l) --> In(z,m)) <-> Wfp(?E(l,m))).
```

As `Wfp` was defined to be the identity (i.e., `Wfp(P)` equals `P`) it does not effect the validity of any of the rules. It does, however, affect their applicability. That is, after rule derivation we remove the definition of `Wfp` from our theory so the only way we can prove the above is by using rules that manipulate equivalences whose right hand side is also labeled by `Wfp`. In particular, we won't be able to prove

```
ALL l m. (ALL z. In(z,l) --> In(z,m)) <-> Wfp(ALL z. In(z,l) --> In(z,m)).
```

Basic Rules

Figure 1 contains a collection of typical derived rules about equivalences. Many of the rules serve simply to copy structure from the specification to the program. These are trivially derivable, for example

$$\frac{A \leftrightarrow Wfp(ExtA) \quad B \leftrightarrow Wfp(ExtB)}{A \wedge B \leftrightarrow Wfp(ExtA \wedge ExtB)}.$$

Translating this into Isabelle we have

```
[| A <-> Wfp(ExtA); B <-> Wfp(ExtB) |] ==> A & B <-> Wfp(ExtA & ExtB).
```

```
RAllI: [| !!x. A(x) <-> Wfp(Ext) |] ==> (ALL x.A(x)) <-> Wfp(Ext)

RAndI: [| A <-> Wfp(ExtA); B <-> Wfp(ExtB) |] ==> A & B <-> Wfp(ExtA & ExtB)

ROrI: [| A <-> Wfp(ExtA); B <-> Wfp(ExtB) |] ==> A | B <-> Wfp(ExtA | ExtB)

RImpI: [| A <-> Wfp(ExtA); B <-> Wfp(ExtB) |] ==> (A --> B) <-> (Wfp(ExtA --> ExtB))

RTrue: [| A |] ==> A <-> Wfp(True)

RFalse: [| ~A |] ==> A <-> Wfp(False)

RAllE: [| ALL x.A(x) <-> Wfp(Ext(x)) |] ==> A(x) <-> Wfp(Ext(x))

ROrE: [| A ==> (C <-> Wfp(ExtA)); B ==> (C <-> Wfp(ExtB)); A | B |] ==> C <-> Wfp(ExtA | ExtB)

EqInstance:  A = B <-> Wfp(A = B)
```

Figure 1: Examples of Basic Rules

This rule is derivable (recall that `Wfp(P) = P`) in one step with Isabelle's simplification tactic for intuitionistic logic, so it is a valid rule. The rule allows us essentially to decompose synthesizing logic programs for a conjunction into synthesizing programs for the individual parts. Note that this rule is initially postulated with free variables like `A` and `ExtA` which are treated as constants during proof of the rule; this prevents their premature instantiation, which would lead to a proof of something more specialized. When the proof is completed, these variables are replaced by metavariables, so the rule may be later applied using unification.

There are two subtleties in the calculus we propose: parameter variables and induction rules. These are explained below.

Predicate Parameters

Recall that problems are equivalences between specifications and higher-order meta-variables applied to parameters, e.g., `l` and `m` in the subset specification. We would like our derived rules to be applicable independent of the number of parameters involved. Fortunately, these do not need to be mentioned in the rules themselves (with one exception noted shortly) as Isabelle's higher-order unification properly propagates these parameters to subgoals. This is explained below.

Isabelle automatically *lifts* rules during higher-order resolution (see [25, 26]); this is a sound way of dynamically matching types of meta-variables during unification by applying them to new universally quantified parameters when necessary. This idea is best explained by an example. Consider applying the above conjunction rule to the following (made-up) goal.

```
ALL l m. ((ALL z. In(z,l)) & (Exists z. ~In(z,m))) <-> Wfp(?E(l,m)))
```

In our theory, we begin proving goals by "setting up a context" where initial universally quantified variables become eigenvariables.[4] Applying ∀-I (∀-intro of first-order logic) twice yields the following.

```
!! l m. ((ALL z. In(z,l)) & (Exists z. ~In(z,m))) <-> Wfp(?E(l,m)))
```

Now if we try to apply the above derived rule for conjunction, Isabelle will automatically lift this rule to

[4]By eigenvariables, we mean variables universally quantified outermost in the context. Recall universal quantification is the operator "`!!`" in Isabelle's meta-logic. See [26] for more details.

```
!! l m. [| ?A(l,m) <-> Extract(?ExtA(l,m)); ?B(l,m) <-> Extract(?ExtB(l,m)) |] ==>
    ?A(l,m) & ?B(l,m) <-> Extract(?ExtA(l,m) & ?ExtB(l,m)),
```

which now resolves (by unifying the conclusion) with `?A(l,m) = ALL z. In(z,l)`, `?B(l,m) = Exists z. ~In(z,m)`, and the program is instantiated with `?E(l,m) = ?ExtA(l,m) & ?ExtB(l,m)`. As the proof proceeds `?ExtA` and `?ExtB` are further instantiated.

Recursive Definitions

Our calculus so far is trivial; it copies structure from specifications into programs. One nontrivial way of transforming specifications is to admit proofs about equivalence by induction over the recursively defined data-types. But this introduces a problem of how predicates recursively call themselves.

We solve this by proving theorems in a context and proof by induction can extend this context with new predicate definitions.[5] In particular, the context will contain not only axioms for defined function symbols (e.g., like `In` in the subset example) but it also contains a meta-variable ("wrapped" by `Def`) that is instantiated during induction with new predicate definitions.

Back to the subset example; our starting goal actually includes a context which defines the axioms for `In` and includes a variable `?H` which expands to a definition or series of definitions. These will be called from the program that instantiates `?E`.

```
[| ALL x. ~In(x,[]);  ALL x h t. In(x,h.t) <-> x = h | In(x,t) |]
   ==> Def(?H) --> (ALL l m. (ALL z. In(z,l) --> In(z,m)) <-> Wfp(?E(l,m)))
```

The wrapper `Def` (recall this, like `Wfp` is the identity) also serves to restrict unification; in particular, only the induction rule which creates a schematic pure logic program can instantiate `Def(?H)`.

Definitions are set up during induction. Consider the following rule corresponding to structural induction over lists. This rule builds a schematic program `P(x,y)` contained in the first assumption. The second and third assumption correspond to the base case and step case of a proof by induction showing that this definition is equivalent to the specification formula `A(x,y)`. This rule is derived in our theory by list induction.

```
[| Def(ALL x y. P(x,y) <-> (x = [] & EA(y)) | (EX v0 v1. x = v0.v1 & EB(v0,v1,y)));
   ALL y. A([],y) <-> Wfp(EA(y));
   !!m. ALL y. A(m,y) <-> Wfp(P(m,y)) ==> ALL h y. A(h.m,y)  <-> Wfp(EB(h,m,y)) |]
 ==> A(x,y) <-> Wfp(P(x,y))
```

As in [2] we have written a tactic that applies induction rules. Resolution with this rule yields three subgoals (corresponding to the three assumptions above) but the first is discharged by unifying against a `Def(?H)` in the context which sets up a recursive definition. This is precisely the role that `Def(?H)` serves. Actually, to allow for multiple recursive definitions, the induction tactic first duplicates the `Def(?H)`[6] before resolving with the induction rule. Also, it thins out (weakens) the instantiated definition in the two remaining subgoals.

There is an additional subtlety in the induction rule which concerns parameter arguments. Other rules did not take additional parameters but this is the exception; `P` takes two arguments even though the induction is on only one of them. This is necessary as the rule must establish (in the first assumption) a definition for a predicate with a fixed number of universally quantified parameters and the number of these

[5] The ability to postulate new predicate definitions can, of course, lead to inconsistency. We lack space here for details, but it is not hard to prove under our approach that defined predicates are defined by well-founded recursion and may be consistently added as assumptions.

[6] This follows as `Def(?H)` equals `?H` and if we have an hypothesis `?H` then we can instantiate it with `?H1 & ?H2`. Instantiation is performed by unification with `&`-elimination and results in the two new assumptions `?H1` and `?H2` which are rewrapped with `Def`. This "engineering with logic" is accomplished by resolving with a derived rule that performs these manipulations.

cannot be predicted at the time of the induction. Our solution to this problem is ad hoc; we derive in Isabelle a separate induction rule for each number of arguments needed in practice (two are needed for the predicates synthesized in the subset example). Less ad hoc, but more complex, solutions are also possible.

3.4 Relationship to Other Calculi

The derived calculus, although very simple, is motivated by and is similar to the Whelk Calculus developed by Wiggins in [33]. There Whelk is presented as a new kind of logic where specifications are manipulated in a special kind of "tagged" formal system. A tagged formula A is of the form $[\![A]\!]_{P(\overline{x})\leftrightarrow\phi}$. Both formulas and sequents are tagged and the tag subscript represents part of a pure logic program. The Whelk logic manipulates these tags so that the tagged (subscripted) program should satisfy two properties. First, the tagged program should be logically *equivalent* to formula it tags in the appropriate first-order theory. To achieve this the proof rules state how to build programs for a given goal from programs corresponding to the subgoals. Second, the tagged program should be *decidable*, which means as a program it terminates in all-ground mode. One other feature of Whelk is that a proof may begin with a subformula of the starting goal labeled by a special operator ∂ . At the end of the proof the Whelk system *extracts* the tagged program labeling this goal; hence Whelk may be used to synthesize logic programs.

The rules I give can be seen as a reinterpretation of the rules of Whelk where tagged formulas are formulated directly as equivalences between specifications and program schemas (for full details see [1]); hence, seen in this light, the Whelk rules constitute a simple calculus for manipulating equivalences. For example, the Whelk rule for reasoning about conjunctions is

$$\partial\wedge\text{-}I\ \frac{[\![,\ \vdash \partial A]\!]_{P(\mathcal{E})\leftrightarrow\phi} \quad [\![,\ \vdash \partial B]\!]_{P(\mathcal{E})\leftrightarrow\psi}}{[\![,\ \vdash \partial (A\wedge B)]\!]_{P(\mathcal{E})\leftrightarrow\phi\wedge\psi}}$$

and can be straightforwardly translated into the rule RAndI given in Section 3.3 (ϕ and ψ play the role of $ExtA$ and $ExtB$ and P and its parameters $\mathcal{E}$ are omitted.) Our derivation of many of these rules provides a formal verification that they are correctness preserving with respect to the above mentioned equivalence criteria. Interestingly, not all of the Whelk rules given could be derived; the reinterpretation led to rules which were not derivable (counter models could be given) and hence helped uncover mistakes in the original Whelk calculus (see [1]). This confirms that just as it is useful to have machine checked proofs of program correctness, it is also important to certify calculi formally.

4 Program Development

We now illustrate how the derived rules can be applied to program synthesis. Our example is synthesizing the subset predicate (over lists). We choose this as it is a standard example from the literature. In particular, our proof is almost identical to one given in [33].

Our proof requires 15 steps and is given in Figure 2 with comments. Here we replay Isabelle's response to these proof steps, i.e., the instantiated top-level goal and subgoals generated after each step. The output is taken directly from an Isabelle session except, to save space, we have combined a couple of steps, "pretty printed" formulas, and abbreviated variable names.

The proof begins by giving Isabelle the subset specification. Isabelle prints back the goal to be proved (at the top) and the subgoals necessary to establish it. As the proof proceeds, the theorem we are proving becomes specialized as `?H` is incrementally instantiated with a program. We have also given the names `inbase` and `instep` to the context assumptions that define the membership predicate `In`.

```
Def(?H) --> (ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Wfp(?E(l, m)))
 1. Def(?H) --> (ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Wfp(?E(l, m)))
val inbase = "ALL x. ~ In(x, [])"
val instep = "ALL x h t. In(x, h . t) <-> x = h | In(x, t)"
```

```
val [inbase,instep] = goal thy
"  [| ALL x. ~In(x,[]); \
\     ALL x h t. In(x,h.t) <-> x = h | In(x,t) |] \
\ ==> Def(?H) --> (ALL l m. (ALL z. In(z,l) --> In(z,m)) <-> Wfp(?E(l,m)))";

(* After performing forall introductions, perform induction *)
by SetUpContext;
by (IndTac WListInduct2  [("x","l"),("y","m")] 1);

(* Base Case *)
br RAllI 1;
br RTrue 1;
by (cut_fast_tac [inbase] 1);

(* Step Case *)
by(SIMP_TAC (list_ss addrews [instep,AndImp,AllAnd,AllEqImp]) 1);
br RAndI 1;

(* Prove 2nd case with induction hypothesis! *)
by (etac allE 2 THEN assume_tac 2);

(* First Case --- Do an induction on y to synthesize member(h,y) *)
by (IndTac WListInduct2  [("x","y"),("y","h")] 1);
br RFalse 1;    (* Induction Base Case *)
by(SIMP_TAC (list_ss addrews [inbase]) 1);
by(SIMP_TAC (list_ss addrews [instep]) 1);  (* Induction Step Case *)
br ROrI 1;
br EqInstance 1;
by (etac allE 1 THEN assume_tac 1);  (* Apply induction hypothesis *)
```

Figure 2: Isabelle Proof Script for Subset Proof

The first proof step, invoked by the tactic `SetUpContext`, moves the definition variable `?H` into the assumption context and, as discussed in the previous section, promotes universally quantified variables to eigenvariables so our rules may be used via lifting.

```
Def(?H) --> (ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Wfp(?E(l, m)))
 1. !!l m. Def(?H) ==> (ALL z. In(z, l) --> In(z, m)) <-> Wfp(?E(l, m))
```

Next, we invoke our induction tactic that applies the derived list induction rule, specifying induction on `l`. The execution of the tactic instantiates our schematic definition `?H` with the first schematic definition `?P` and a placeholder `?Q` for further instantiation. Note too that `?E` has been instantiated to this schematic program `?P`.

```
Def((ALL x y. ?P(x, y) <-> x = [] & ?EA10(y) | (EX v0 v1. x = v0 . v1 & ?EB11(v0, v1, y))) &
    ?Q) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Wfp(?P(l, m)))
 1. !!l m y. Def(?Q) ==> (ALL z. In(z, []) --> In(z, y)) <-> Wfp(?EA10(y))
 2. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y)) |] ==>
       (ALL z. In(z, h . ma) --> In(z, y)) <-> Wfp(?EB11(h, ma, y))
```

We now prove the first case, which is the base-case (and omit printing the step case in the next two steps — Isabelle maintains a goal stack). First we apply `RAllI` which promotes the ∀-quantified variable z to an eigenvariable. The new subgoal becomes (as this step does not instantiate the theorem we are proving, we omit redisplaying it) the following.

```
 1. !!l m y z. Def(?Q) ==> (In(z, []) --> In(z, y)) <-> Wfp(?EA10(y))
```

Next we apply `RTrue` which states if `?EA10(y)` is `True`, the above is provable provided `In(z, []) --> In(z, y)` is provable. This reduces the goal to one of ordinary logic (without `Wfp`) as it instantiates the base case with the proposition `True`.

```
Def((ALL x y. ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?EB11(v0, v1, y))) &
    ?Q) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Wfp(?P(l, m)))
 1. !!l m y z. Def(?Q) ==> In(z, []) --> In(z, y)
```

Finally we complete this step by applying Isabelle's predicate-calculus simplification routines augmented with base case of the definition for `In`. Isabelle leaves us with the following step case (which is now the top goal on the stack and hence numbered 1).

```
 1. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y)) |] ==>
       (ALL z. In(z, h . ma) --> In(z, y)) <-> Wfp(?EB11(h, ma, y))
```

We now normalize this goal by applying the tactic

```
SIMP_TAC (list_ss addrews [instep,AndImp,AllAnd,AllEqImp]) 1
```

This calls Isabelle's simplification tactic which applies basic simplifications for the theory of lists, `list_ss`, augmented with the recursive case of the definition for `In` and the following lemmas `AndImp`, `AllAnd` and `AllEqImp`.

```
(A | B --> C) <-> (A --> C) & (B --> C)
ALL v. A(v) & B(v)) <-> (ALL v.A(v)) & (ALL v.B(v))
(ALL v. v = w --> A(v)) <-> A(w)
```

Each of these had been previously (automatically!) proven with Isabelle's predicate calculus simplifier. This normalization step simplifies our subgoal to the following.

```
 1. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y)) |] ==>
       In(h, y) & (ALL v. In(v, ma) --> In(v, y)) <-> Wfp(?EB11(h, ma, y))
```

We decompose the conjunction with `RAndI`, which yields two subgoals.

```
Def((ALL x y. ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?EA21(v1, v0, y) & ?EB22(v1, v0, y))) &
    ?Q) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Wfp(?P(l, m)))
 1. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y)) |] ==>
       In(h, y) <-> Wfp(?EA21(ma, h, y))
 2. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y)) |] ==>
       (ALL v. In(v, ma) --> In(v, y)) <-> Wfp(?EB22(ma, h, y))
```

We immediately solve the second subgoal by resolving with the induction hypothesis. I.e., after ∀-E we unify the conclusion with the induction hypothesis using Isabelle's assumption tactic. This instantiates the program we are building by replacing `?EB22` with a recursive call to `?P` as follows.

```
Def((ALL x y. ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?EA21(v1, v0, y) & ?P(v1, y))) & ?Q) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Wfp(?P(l, m)))
```

Returning to the first goal (to build a program for `?EA21`), we perform another induction; the base case is proved as in the first induction except rather than introducing `True` with `RTrue` we introduce `False` with `RFalse` and solve the remaining goal by simplification. This leaves us with the step case.

```
Def((ALL x y. ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?Pa(v0, y) & ?P(v1, y))) &
    (ALL x y. ?Pa(y, x) <-> x = [] & False | (EX v0 v1. x = v0 . v1 & ?EB32(v0, v1, y))) &
    ?Q27) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Wfp(?P(l, m)))
 1. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Wfp(?Pa(y, mb)) |] ==>
       In(ya, ha . mb) <-> Wfp(?EB32(ha, mb, ya))
```

As before, we normalize this subgoal with Isabelle's standard simplifier.

```
 1. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Wfp(?Pa(y, mb)) |] ==>
       ya = ha | In(ya, mb) <-> Wfp(?EB32(ha, mb, ya))
```

Applying `ROrI` unifies `?EB32(v0, v1, y)` with `?EA40(v1, v0, y) | ?EB41(v1,v0, y)` and yields a subgoal for each disjunct.

```
 1. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Wfp(?Pa(y, mb)) |] ==>
       ya = ha <-> Wfp(?EA40(mb, ha, ya))
 2. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Wfp(?Pa(y, mb)) |] ==>
       In(ya, mb) <-> Wfp(?EB41(mb, ha, ya))
```

In the first we apply `EqInstance` which instantiates `?EA40(v1, v0, y)` with `y = v0`. This completes the first goal leaving only the following.

```
Def((ALL x y.
        ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?Pa(v0, y) & ?P(v1, y))) &
    (ALL x y. ?Pa(y, x) <-> x = [] & False | (EX v0 v1. x = v0 . v1 & (y = v0 | ?EB41(v1, v0, y)))) &
    ?Q27) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Wfp(?P(l, m)))
 1. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Wfp(?Pa(y, mb)) |] ==>
       In(ya, mb) <-> Wfp(?EB41(mb, ha, ya))
```

We complete the proof by resolving with the induction hypothesis. Isabelle prints back the following proven formula with no remaining subgoals.

```
   [| ALL x. ~ ?In(x, []);
      ALL x h t. ?In(x, h . t) <-> x = h | ?In(x, t) |] ==>
   Def((ALL x y. ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?Pa(y, v0) & ?P(v1, y))) &
       (ALL x y. ?Pa(x, y) <-> x = [] & False | (EX v0 v1. x = v0 . v1 & (y = v0 | ?Pa(v1, y)))) &
       ?Q) -->
   (ALL l m. (ALL z. ?In(z, l) --> ?In(z, m)) <-> Wfp(?P(l, m)))
```

Note that the context remains open (`?Q`) as we might have needed to derive additional predicates. Also observe that Isabelle never forced us to give the predicates built (`?P` and `?Pa`) concrete names; these were picked automatically during resolution when variables were renamed apart by the system.

The constructed program can be simplified and translated into a Gödel program similar to the one in [33]. Alternatively it can be directly translated into the following Prolog program.

```
p([],Y).
p([V0|V1],Y)  :- pa(Y,V0), p(V1,Y).
```

```
pa([],Y)      :- false.
pa([V0|V1],V0).
pa([V0|V1],Y) :- pa(V1,Y).
```

5 Conclusion, Comparisons, and Future Work

The ideas presented here have applicability, of course, outside logic programming and Isabelle can be used to derive other calculi for verification and synthesis. [2, 4] describes other applications of this methodology. But logic programming seems an especially apt domain for such development due to the close relationship between the specification and programming language.

Other authors have argued that first-order logic is the proper foundation for reasoning about and transforming logic programs (e.g., [11, 9]). But there are benefits to using even richer logics to manipulate first-order, and possibly higher-order, specifications. For example, in this paper we used a recursion schema corresponding to structural induction over lists. But synthesizing logic programs with more complicated kinds of recursion (e.g., quick sort) requires general well-founded induction. But providing a theory where the user can provide his own well-founded relations necessitates formalizing well-foundedness which in turn requires quantifying over sets or predicates and, outside of set-theory, this is generally second-order. We are currently exploring synthesis based on well-founded induction in higher-order logic.

Another research direction is exploring other notions of equivalence. Our calculus has employed a very simple notion based on provability in a theory with induction principles over recursive data-types. There are other notions of equivalence and ways of proving equivalence that could be formalized of course. Of particular interest is exploring schematic calculi like that proposed by Waldau [31]. Waldau presents a calculus for proving the correctness of transformation schemata using intuitionistic first-order logic. In particular he showed how one can prove the correctness of fold-unfold transformations and schemata like those which replace recursion by tail recursion. The spirit of this work is similar to our own: transformation schemata should be proven correct using formal proofs. It would be interesting to carry out the kinds of derivations he suggests in Isabelle and use Isabelle's unification to apply his transformation schemata.

We conclude with a brief comparison of related approaches to program synthesis based on unification. This idea can be traced back to [14] who proposed the use of resolution not only for checking answers to queries, but also for synthesizing programs and the use of second-order matching by Huet and Lang to apply schematic transformations. Work in unification based program synthesis that is closest in spirit to what we described here is the work of [20, 21], which used higher-order (pattern) unification to synthesize logic programs in a "middle-out" fashion. Indeed, synthesis with higher-order resolution in Isabelle is very similar as in our work, the meta-variable standing in for a program is a second-order pattern and it is only unified against second-order patterns during proof. [20, 21] emphasizes, however, the automation of such proofs via rippling while we concentrate more on the use of logical frameworks to give formal guarantees to the programming logic itself. Of course, these approaches are compatible and can be combined.

References

[1] David Basin. Isawhelk: Whelk interpreted in Isabelle. Abstract accepted at the 11th International Conference on Logic Programming (ICLP94). Full version available via anonymous ftp to mpi-sb.mpg.de in pub/papers/conferences/Basin-ICLP94.dvi.Z.

[2] David Basin, Alan Bundy, Ina Kraan, and Sean Matthews. A framework for program development based on schematic proof. In *7th International Workshop on Software Specification and Design*, Los Angeles, December 1993. IEEE Computer Society Press.

[3] David Basin and Robert Constable. Metalogical frameworks. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993.

[4] David Basin and Sean Matthews. A conservative extension of first order logic and its applications to theorem proving. In *13th Conference of the Foundations of Software Technology and Theoretical Computer Science*, pages 151–160, December 1993.

[5] A. Bundy. Tutorial notes; reasoning about logic programs. In G. Comyn, N.E. Fuchs, and M.J. Ratcliffe, editors, *Logic programming in action*, pages 252–277. Springer Verlag, 1992.

[6] 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 Re search Paper 501.

[7] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

[8] Wei Ngan Chin. *Automatic Methods for Program Transformation.* Ph. D. thesis, Imperial College Department of Computer Science, March 1990.

[9] K. L. Clark and S-Å. Tärnlund. A first order theory of data and programs. In B. Gilchrist, editor, *Information Processing*, pages 939–944. IFIP, 1977.

[10] K.L. Clark. Predicate logic as a computational formalism. Technical Report TOC 79/59, Imperial College, 1979.

[11] K.L. Clark and S. Sickel. Predicate Logic: a calculus for deriving programs. In R. Reddy, editor, *Proceedings of IJCAI-77*, pages 419–420. IJCAI, 1977.

[12] Pierre Flener and Yves Deville. Towards stepwise, schema-guided synthesis of logic programs. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation*, pages 46–64. Springer-Verlag, 1991.

[13] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[14] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the IJCAI-69*, pages 219–239, 1969.

[15] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[16] P. Hill and J. Lloyd. The Gödel Report. Technical Report TR-91-02, Department of Computer Science, University of Bristol, March 1991. Revised in September 1991.

[17] C.J. Hogger. Derivation of logic programs. *JACM*, 28(2):372–392, April 1981.

[18] L. Kott. About a transformation system: A theoretical study. In *Proceedings of the 3rd International Symposium on Programming*, pages 232–247, Paris, 1978.

[19] Robert A. Kowalski. Predicate logic as a programming language. In *IFIP-74*. North-Holland, 1974.

[20] Ina Kraan, David Basin, and Alan Bundy. Logic program synthesis via proof planning. In *Proceedings of LoPSTr-92*. Springer Verlag, 1992.

[21] Ina Kraan, David Basin, and Alan Bundy. Middle-out reasoning for logic program synthesis. In *10th International Conference on Logic Programming (ICLP93)*, pages 441–455, Budapest Hungary, 1993.

[22] M.J. Maher. Equivalences of logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1987.

[23] Sean Matthews, Alan Smaill, and David Basin. Experience with FS_0 as a framework theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 61–82. Cambridge University Press, 1993.

[24] Lawrence C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*. In press; draft available as Report 271, University of Cambridge Computer Laboratory.

[25] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[26] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, Cambridge University Computer Laboratory, Cambridge, January 1993.

[27] Lawrence C. Paulson. Isabelle's object-logics. Technical Report 286, Cambridge University Computer Laboratory, Cambridge, February 1993.

[28] Frank Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149 – 181. Cambridge University Press, 1991.

[29] Taisuke Sato. Equivalence-preserving first-order unfold/fold transformation systems. *Theoretical Computer Science*, 105:57–84, 1992.

[30] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proceedings of 2nd ICLP*, 1984.

[31] Mattias Waldau. Formal validation of transformation schemata. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation*, pages 97–110. Springer-Verlag, 1991.

[32] Richard W. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

[33] Geraint A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. R. Apt, editor, *Proceedings of JICSLP-92*, 1992.

MAX-PLANCK-INSTITUT FÜR INFORMATIK

Termination Orderings for Rippling

David A. Basin
Toby Walsh

MPI–I–94–209 June 1994

INFORMATIK

Im Stadtwald
D 66123 Saarbrücken
Germany

Authors' Addresses

David Basin Max-Planck-Institut für Informatik Im Satdwald, D-66123 Saarbrücken, Germany
basin@mpi-sb.mpg.de
Toby Walsh INRIA-Lorraine, 615, rue du Jardin Botanique, 54602 Villers-les-Nancy, France
walsh@loria.fr

Publication Notes

A version of this paper will appear at CADE-12 in Nancy France, June 1994.

Acknowledgements

Many of the ideas described here stem from conversations with members of the Edinburgh MRG group, in particular with Alan Bundy, Ian Green, and Andrew Ireland. We also wish to thank Sean Matthews, Michael Rusinowitch, and Andrew Stevens for comments on earlier drafts. The first author was funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9102. The second author was supported by a SERC and a HCM Postdoctoral Fellowship.

Abstract

Rippling is a special type of rewriting developed for inductive theorem proving. Bundy *et. al.* have shown that rippling terminates by providing a well-founded order for the annotated rewrite rules used by rippling. Here, we simplify and generalize this order, thereby enlarging the class of rewrite rules that can be used. In addition, we extend the power of rippling by proposing new domain dependent orders. These extensions elegantly combine rippling with more conventional term rewriting. Such combinations offer the flexibility and uniformity of conventional rewriting with the highly goal directed nature of rippling. Finally, we show how our orders simplify implementation of provers based on rippling.

1 Introduction

Rippling is a form of goal directed rewriting developed at Edinburgh [5, 3] and in parallel in Karlsruhe [11, 12] for inductive theorem proving. In inductive proof, the induction conclusion typically differs from the induction hypothesis by the addition of some constructors or destructors. Rippling uses special annotations, called *wavefronts*, to mark these differences. They are then removed by annotated rewrite rules, called *wave-rules*. Rippling has several attractive properties. First, it is highly goal directed, attempting to remove just the differences between the conclusion and hypothesis, leaving the common structure preserved. And second, it terminates yet allows rules like associativity to be used both ways.

The contributions of this paper are to simplify, improve, and generalize the specification of wave-rules and their associated termination orderings. Wave-rules have previously been presented via complex schematic definitions that intertwine the properties of structure preservation and the reduction of a well-founded measure (see [3] and §7). As these properties may be established independently, our definition of wave-rules separates these two concerns. Our main focus is on new measures. We present a family of measures that, despite their simplicity, admit strictly more wave-rules than the considerably more complex specification given in [3].

This work has several practical applications. By allowing rippling to be combined with new termination orderings, the power of rippling can be greatly extended. Although rippling has been designed primarily to prove inductive theorems it has recently been applied to other problem domains. We show that in rippling, as in conventional rewriting, the ordering used should be domain dependent. We provide several new orderings for applying rippling to new domains within induction (e.g. domains involving mutually recursive functions) and outside of induction (e.g. equational problem solving). In doing so, we show for the first time how rippling can be combined with conventional rewriting.

Another practical contribution is that our work greatly simplifies the implementation of systems based on rippling. Systems like Clam [4] require a procedure, called a *wave-rule parser*, to annotate rewrite rules. Clam's parser is based upon the complex definition of wave-rules in [3] and as a result is itself extremely complex and faulty. We show how, given a simple modular order, we can build simple modular wave-rule parsers. We have implemented such parsers and they have pleasant properties that current implementations lack (e.g. notions of correctness and completeness); our work hence leads to a simpler and more flexible mechanization of rippling.

The paper is organized as follows. In §2 we give a brief overview of rippling. In §3 we define an order on a simple kind of annotated term and use this in §4 to build orderings on general annotated terms. Based on this we show in §5 how rewrite rules may be automatically annotated. In §6 we describe how new orders increase the power and applicability of rippling. In §7 we compare this work to previous work in this area and discuss some practical experience. Finally we draw conclusions.

2 Background

We provide a brief overview of rippling. For a complete account please see [3].

Rippling arose out of an analysis of inductive proofs. For example, if we wish to prove $P(x)$ for all natural numbers, we assume $P(n)$ and attempt to show $P(s(n))$. The hypothesis and the conclusion are identical except for the successor function $s(.)$ applied to the induction variable n. Rippling marks this difference by the annotation, $P(\boxed{s(\underline{n})})$. Deleting everything in the box that is not underlined gives the skeleton, which is preserved during rewriting. The boxed but not underlined term parts are wavefronts, which are removed by rippling.

Formally, a *wavefront* is a term with at least one proper subterm deleted. We represent this by marking a term with *annotation* where wavefronts are enclosed in boxes and the deleted subterms, called *waveholes*, are underlined. Schematically, a wavefront looks like $\boxed{\xi(\underline{\mu_1}, ..., \underline{\mu_n})}$, where $n > 0$ and μ_i may be similarly annotated. The part of the term not in the wavefront is called the *skeleton*. Formally, the skeleton is a non-empty set of terms defined as follows.

Definition 1 (Skeleton)

1. $skel(t) = \{t\}$ *for* t *a constant or variable*
2. $skel(f(t_1, ..., t_n)) = \{f(s_1, ..., s_n) | \forall i.\, s_i \in skel(t_i)\}$
3. $skel(\boxed{f(\underline{t_1}, ..., \underline{t_n})}) = skel(t_1) \cup ... \cup skel(t_n)$ *for the* t_i *in waveholes.*

We call a term *simply annotated* when all its wavefronts contain only a single wavehole and *generally annotated* otherwise. In the simply annotated case, the skeleton function returns a singleton set whose member we call the skeleton. E.g. the skeleton of $f(\boxed{s(\underline{a})}, \boxed{s(\underline{b})})$ is $f(a, b)$.

We define *wave-rules* to be rewrite rules between annotated terms that meet two requirements: they are skeleton preserving and measure decreasing. This is a simpler and more general approach to defining wave-rules than that given in [3] where these requirements were intertwined into the syntactic specification of a wave-rule.[1] *Skeleton preservation* in the simply-annotated case means that both the LHS (left-hand side) and RHS (right-hand side) of the wave-rule have an identical skeleton. In the multi-hole case we demand that *some* of the skeletons on the LHS are preserved on the RHS and no new skeletons are introduced, i.e. $skel(LHS) \supseteq skel(RHS)$.

Wavefronts in wave-rules are also *oriented*. This is achieved by marking the wavefront with an arrow indicating if the wavefront should move up through the skeleton term tree or down towards the leaves. Oriented wavefronts dictate a measure on terms that rippling decreases. The focus of this paper is on these measures.

Below are some examples of wave-rules (s is successor and $<>$ is infix append).

$$\boxed{s(\underline{U})}^{\uparrow} \times V \Rightarrow \boxed{(\underline{U \times V}) + V}^{\uparrow} \tag{1}$$

$$\boxed{s(\underline{U})}^{\uparrow} \geq \boxed{s(\underline{V})}^{\uparrow} \Rightarrow U \geq V \tag{2}$$

$$\boxed{\underline{U} + V}^{\uparrow} \times W \Rightarrow \boxed{\underline{U \times W} + V \times W}^{\uparrow} \tag{3}$$

$$(\boxed{\underline{U} <> V}^{\uparrow}) <> W \Rightarrow U <> (\boxed{V <> \underline{W}}^{\downarrow}) \tag{4}$$

$$U <> (\boxed{\underline{V} <> W}^{\uparrow}) \Rightarrow \boxed{(\underline{U <> V}) <> W}^{\uparrow} \tag{5}$$

$$\boxed{\underline{U} + \underline{V}}^{\uparrow} = \boxed{\underline{W} + \underline{Z}}^{\uparrow} \Rightarrow \boxed{\underline{U = W} \wedge \underline{V = Z}}^{\uparrow} \tag{6}$$

(1) and (2) are typical of wave-rules based on a recursive definitions. The remainder come from lemmas. Methods for turning definitions and lemmas into wave-rules is the subject of §5. Note that annotation in the wave-rules must match annotation in the term being rewritten. This allows use of rules like associativity of append, (4) and (5), in both directions; these would loop in conventional rewriting. Note also that in (6) the skeletons of the RHS are a strict subset of those of the LHS.

[1] This generalization is, however, briefly discussed in their further work section.

As a simple example of rippling, consider proving the associativity of multiplication using structural induction. In the step-case, the induction hypothesis is

$$(x \times y) \times z = x \times (y \times z)$$

and the induction conclusion is

$$(\boxed{s(\underline{x})}^{\uparrow} \times y) \times z = \boxed{s(\underline{x})}^{\uparrow} \times (y \times z).$$

The wavefronts in the induction conclusion mark the differences with the induction hypothesis. Rippling on both sides of the induction conclusion using (1) yields (7) and then with (3) on the LHS gives (8).

$$(\boxed{\underline{x \times y} + y}^{\uparrow}) \times z \quad = \quad \boxed{\underline{(x \times (y \times z))} + y \times z}^{\uparrow} \tag{7}$$

$$\boxed{\underline{((x \times y) \times z)} + y \times z}^{\uparrow} \quad = \quad \boxed{\underline{(x \times (y \times z))} + y \times z}^{\uparrow} \tag{8}$$

As the wavefronts are now at the top of each term, we have successfully rippled-out both sides of the equality. We can complete the proof by simplifying with the induction hypothesis.

The example illustrates how rippling preserves skeletons during rewriting. Provided rippling does not get *blocked* (no wave-rule applies yet we are not completely rippled-out), we are guaranteed to be able to simplify with the induction hypothesis (called *fertilization* in [2]). This explains the highly goal directed nature of rippling.

We can also ripple wavefronts towards the position of universally quantified variables in the induction hypothesis. Such positions are called *sinks* because wavefronts can be absorbed there; when we appeal to the induction hypothesis, universally quantified variables will be matched with the content of the sinks. Rippling towards sinks at the leaves of terms is called *rippling-in*. Wavefronts are oriented with arrows pointing out (upwards) or in (downwards) indicating if they are moving towards the root or leaves. *Transverse* wave-rules like (4) are used to turn outward directed wavefronts inwards.

3 Ordering Simple Wave-Rules

In this section we consider only simply annotated terms (whose wavefronts have a single wavehole). In the next section we generalize to orders for generally annotated terms with multiple waveholes. We begin with motivation, explaining generally the kinds of orders we wish to define. Afterwards, we propose several concrete measures that are similar, though simpler, to those given by Bundy *et. al.* in [3]. They are able to order all the wave-rules given in [3] and in addition allow rule orientations not possible using the measure given there (see §7).

We consider annotated terms as decorated trees where the tree is the skeleton and the wavefronts are boxes decorating the nodes. See, for example, the first tree in Fig. 1 which represents the term $\boxed{s(\underline{U})}^{\uparrow} \geq \boxed{s(\underline{V})}^{\uparrow}$. Our orders are based on assigning measures to annotation in these trees. We can define progressively simpler orders by simplifying these annotated trees to capture the notion of progress during rippling that we wish to measure.

To begin with, since rippling is skeleton preserving, we needn't account for the contents of the skeleton in our orderings. That is, we can abstract away function symbols in the skeleton, for example, mapping each function to a variadic function constant "*". This gives, for example, the second tree in Fig. 1. In §6.2, we return

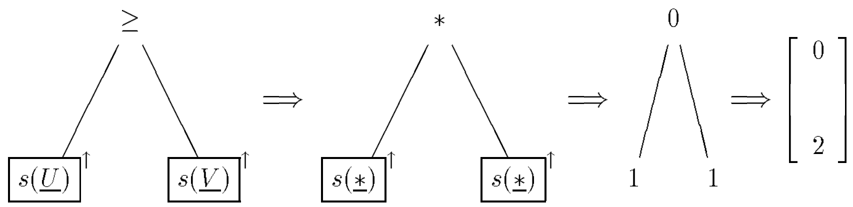


Figure 1: Defining a measure on annotated terms.

to this abstraction and examine termination orderings that do allow the skeleton to be changed during rewriting.

A further abstraction is to ignore the names of function symbols within wave-fronts and assign some kind of numeric weight to wave-fronts. For example, we may tally up the values associated with each function symbol as in a Knuth-Bendix ordering. The simplest kinds of weights that we may assign to wave-fronts measure their *width* and their *size*. Width is the number of nested function symbols between the root of the wavefront and the wavehole. Size is the number of function symbols and constants in a wavefront. For simplicity, we will consider just the width unless otherwise stated. This gives, for example, the third tree in Fig. 1. Of course, there are problem domains where we want our measure to reflect more of the structure of wave-fronts. §6.1 contains an example of this showing how the actual contents may be compared using a conventional term ordering.

Finally, a very simple notion of progress during rippling is simply that wave-fronts move up or down through the skeleton tree. Under this view, the tree structure may be ignored: it is not important which branch a wave-front is on, only its height in the skeleton tree. Under this notion of progress, we can apply an abstraction that maps the tree onto a list, level by level. For instance, we can use the sum of the weights at a given depth. Applying this abstraction gives the final tree in Fig. 1. Again, note that depths are *relative* to the skeleton and not depth in the erased term tree.

To make this more formal and concrete, we introduce some definitions. A *position* is simply a path address (written "Dewey decimal style") in the term tree of the skeleton and the subterm of t at position p is denoted by t/p. If s is a subterm of t at position p, its *depth* is the length of p. The *height* of t, written $|t|$, is the maximal depth of any subterm in t. For an annotated term t, the *out-weight of a position p* is the sum of the weights of the (possibly nested) outwards oriented wavefronts at p. The *in-weight* is defined identically except for inward directed wavefronts. We now define a measure on terms corresponding to the final tree in Fig. 1 based on weights of annotation relative to their depths.

Definition 2 (Out/In Measure) *The* out-measure*, $MO(t)$ (*in-measure*, $MI(t)$) of an annotated term t is a list whose i-th element is the sum of out-weights (in-weights) for all term positions in t at depth i.*

For example, in the following palindrome function over lists ("::" is infix cons)

$$palin(\boxed{H :: \underline{T}}^{\uparrow}, Acc) \Rightarrow \boxed{H :: \underline{palin(T, \boxed{H :: \underline{Acc}}^{\downarrow})}}^{\uparrow} \qquad (9)$$

and the skeleton of both sides is $palin(T, Acc)$ and the out-measure of the LHS is [0,1] and the RHS is [1,0]. The in-measures are [0,0] and [0,1] respectively.

We now define a well-founded ordering on these measures which reflects the progress that we want rippling to make during rewriting. Consider, a simple wave-rule like (1),

$$\boxed{s(\underline{U})}^{\uparrow} \times V \Rightarrow \boxed{(\underline{U \times V}) + V}^{\uparrow}.$$

The LHS out-measure is $[0, 1]$, and the RHS is $[1, 0]$. Rippling has progressed here as the one out-oriented wavefront has moved up the term. In general, rippling progresses if one out-oriented wavefront moves up or disappears, while nothing deeper moves downwards. If the out-measure on a term before rippling is $[l_1, ..., l_k]$ and after $[r_1, ..., r_k]$ then there must be some depth j where $l_j > r_j$ and for all $i > j$ we have $l_i = r_i$. This is simply the lexicographic order on the reverse of the two lists (compared with $>$ on the natural numbers).[2] Progress for in-oriented wavefronts is similar and reflects that these wavefronts should move towards leaves; that is, we use the lexicographic order on the in-measures. Of course, both outward and inward oriented wavefronts may occur in the same rule. For example, consider (9). As in [3], we define a composite ordering on the out and in measures. We order the out measure before the in measure since this enables us to ripple wavefronts out and either to reach the top of the term, or at some point to turn the wavefront down and to ripple it in towards the leaves.

Definition 3 (Composite Ordering) *$t \succ s$ iff $\langle MO(t), MI(t)\rangle >_o \langle MO(s), MI(s)\rangle$ where $>_o$ is the lexicographic order on pairs whose first components are compared with $>_{revlex}$ and the second with $>_{lex}$, the reversed and unreversed lexicographic order on lists of equal length.*

Given the well-foundedness of $>$ on the natural numbers and that lexicographic combinations of well-founded orders are well-founded we can conclude the following.

Lemma 1 *The composite ordering is well-founded.*

We lack space here to discuss implementations of rippling. Two different implementations are considered in [3] and [12]. For both calculi, $\succ$ (and $\succ^*$ of the next section) is monotonic and stable over the substitutions produced during rippling. It follows from standard techniques that if all wave-rules are oriented so that $l \succ r$ then rippling terminates [8].

4 Ordering Multi-Wave-Rules

We now generalize our order for simply annotated terms to those with generalized annotation, that is, multiple waveholes in a single wavefront. Wave-rules involving such terms are called *multi-wave-rules* in [3] and we have already seen an example of this in (6). The binomial equation is another example.

$$binom(\boxed{s(\underline{X})}^{\uparrow}, \boxed{s(\underline{Y})}^{\uparrow}) = \boxed{\underline{binom(X, \boxed{s(\underline{Y})}^{\uparrow})} + \underline{binom(X, Y)}}^{\uparrow} \qquad (10)$$

We define orders for generally annotated terms in a uniform way from the previous ordering by reducing generally annotated terms to sets of simply annotated terms and extending $\succ$ to such sets. This reduction is accomplished by considering ways that general annotation can be *weakened* to simple annotation by "absorbing" waveholes. Weakening a multi-wave term like (10) erases some of the waveholes (underlining) though always leaving at least one wavehole. A wavefront is *maximally weak* when it has exactly one wavehole. A term is *maximally weak* when all

[2]Note that these lists are the same length as the skeletons of both sides are identical; however, when we generalize the measure to multi-holed waves, the skeletons may have different depths and we pad with trailing zeros where necessary.

its wavefronts are maximally weak. Maximally weak terms are simply annotated and this allows us to use the previously defined measure $\succ$ on these terms.

Returning to the binomial example, (10) has only the following two weakenings.

$$
\begin{aligned}
binom(\boxed{s(\underline{X})}^{\uparrow}, \boxed{s(\underline{Y})}^{\uparrow}) &= \boxed{\underline{binom(X, \boxed{s(\underline{Y})}^{\uparrow})} + binom(X,Y)}^{\uparrow} && (11)\\
binom(\boxed{s(\underline{X})}^{\uparrow}, \boxed{s(\underline{Y})}^{\uparrow}) &= \boxed{binom(X, s(Y)) + \underline{binom(X,Y)}}^{\uparrow} && (12)
\end{aligned}
$$

Both of these are maximally weak as each wavefront has a single hole.

Let *weakenings*(s) be the set of maximal weakenings of a term s. We now define an ordering on generally annotated terms l and r.

Definition 4 (General ordering) $l \succ^* r$ *iff* weakenings(s) $\succ\!\succ$ weakenings(t) *where* $\succ\!\succ$ *is the multiset ordering over the order* $\succ$ *on simply annotated terms.*

This order is sensible as all the elements of the weakening sets are simply annotated and can be compared with $\succ$. Also observe that if l and r are simply annotated then their weakenings are $\{l\}$ and $\{r\}$ and $l \succ^* r$ agrees with $l \succ r$. In general, we will drop the superscript on $\succ^*$ and use context (e.g., at least one argument has multiple holes) to disambiguate.

As the multi-set extension of a well-founded ordering is well-founded [10] we immediately have the following lemma.

Lemma 2 $\succ^*$ *is well-founded.*

As an example, consider (10). The LHS weakenings are

$$\{binom(\boxed{s(\underline{X})}^{\uparrow}, \boxed{s(\underline{Y})}^{\uparrow})\}\,.$$

The RHS weakenings are

$$\{\boxed{\underline{binom(X, \boxed{s(\underline{Y})}^{\uparrow})} + binom(X,Y)}^{\uparrow}, \boxed{binom(X + s(Y)) + \underline{binom(X,Y)}}^{\uparrow}\}\,.$$

The sole member of the first set is $\succ$-greater than both members of the second set. This equation is measure decreasing and hence a wave-rule when used left to right.

5 Parsing

These orders are simple and admit simple mechanization. We begin with simply annotated terms and then sketch the generalization to multi-waves. We have implemented the routines we describe and in §7 we report on practical experience.

A wave-rule $l \to r$ must satisfy two properties: the preservation of the skeleton, and a reduction of the measure. We achieve these separately. An *annotation phase* first annotates l and r with unoriented wavefronts so their skeletons are identical; this guarantees that rippling is skeleton preserving. An *orientation phase* then orients the wavefronts so that $l \succ r$. We sum this up by the slogan

$$\textit{WAVE-RULE} \quad = \quad \textit{ANNOTATION} + \textit{ORIENTATION}\,. \qquad (13)$$

5.1 Annotation

To annotate terms we can use the *ground difference unification* algorithm given in [1]. Since parsing is an off-line computation (performed once before theorem proving), it is also reasonable to find skeleton preserving annotation via generate-and-test: generate candidate annotations and test if the resulting terms have the same skeleton. Consider, for example, annotating the recursive definition of the palindrome function. There are four possible skeletons: $palin(T, Acc)$, T, Acc, and H. The first of these corresponds to the annotation

$$palin(\boxed{H :: \underline{T}}, Acc) \Rightarrow \boxed{H :: \underline{palin(T, \boxed{H :: \underline{Acc}})}}. \tag{14}$$

The remaining annotations are *trivial* in that both sides are completely within wavefronts except for some subterm at the leaves. For example,

$$\boxed{palin(H :: T, \underline{Acc})} \Rightarrow \boxed{H :: palin(T, H :: \underline{Acc})}.$$

Such trivial wave-rules can usually be ignored as they they make no progress moving wavefronts (although they can be used for wavefront normalization, see §6.1).

5.2 Orientation

Given annotated, but unoriented rules, we must now orient them by placing arrows on the wavefronts. We do this by picking an orientation for wavefronts on the LHS of the wave-rule and finding an orientation on the RHS such that $l \succ r$. In Clam the wave-rules used are oriented with wavefronts on the LHS exclusively out or in. Other combinations are, of course, possible. In general the number of wavefronts, n in the LHS is very small, typically one or two in [3]; hence, it is not much extra effort to consider all 2^n orientations and for each of these generate an orientation for the RHS.[3] In practice this is manageable; see §7.

For each orientation of l we must orient r. If l contains at least one outward oriented wavefront there will always be a measure decreasing orientation of r, namely with all wavefronts oriented in. However, orienting wavefronts inwards prohibits later rippling out whilst orienting outwards does not. If rippling-out blocks, we can always redirect wave-rules inwards with the rewrite rule. $\boxed{F(\underline{X})}^{\uparrow} \Rightarrow \boxed{F(\underline{X})}^{\downarrow}$. This rule is structure preserving and measure decreasing. Hence, we orient r's annotation so that it is measure decreasing and $\succ$*-maximal*; that is, for all orientations r_o, if $l \succ r_o$ then $r \succeq r_o$ ($\succeq$ is the union of the identity relation with $\succ$).

One can find a maximal orientation using generate and test, but it is possible to do much better. Below we sketch an algorithm, linear in $|r|$. Its input is two annotated terms l and r where l is oriented and r unoriented. The output is r oriented and $\succ$-maximal. In what follows, suppose $|l|$ (and hence $|r|$) equals k. Let $t_i^{\uparrow}$ be the sum of out-weights at depth i, $t_i^{\downarrow}$ be the sum of in-weights at depth i, and $flip(t, d, n)$ be the operation that non-deterministically flips down n arrows in t at depth d (there may be multiple choices corresponding to different branches or multiple wavefronts at the same position). We assume below that l has at least one wavefront oriented up. If this is not the case then all of r's wavefronts must be oriented down and this is a maximal orientation iff $l \succ r$. Otherwise orientation proceeds as follows. We first orient all the wavefronts in r upwards and then execute the first of the following statements that succeeds.

1. choose the maximum i such that $l_i^{\uparrow} > r_i^{\uparrow}$ and $\forall j \in \{i+1..k\}.flip(r, j, r_j^{\uparrow} - l_j^{\uparrow})$

[3] This requires of course an implementation that efficiently indexes wave-rules so that extra wave-rules do not degrade the performance of rippling.

2. $\forall i \in \{0..k\}.flip(r, i, r_i^{\uparrow} - l_i^{\uparrow})$ and succeed if $\text{MI}(l) >_{lex} \text{MI}(r)$
3. choose the minimum i such that $l_i^{\uparrow} \neq 0$, $flip(r, i, r_i^{\uparrow} - l_i^{\uparrow} - 1)$ and $\forall j \in \{i + 1..k\}.flip(r, j, r_j^{\uparrow} - l_j^{\uparrow})$

Each of the three statements can be executed in linear time. Note that the first two may fail (there does not exist a maximum i in the first case, or in the second the test $\text{MI}(l) >_{lex} \text{MI}(r)$ fails) but the third case will always succeed.

Lemma 3 *The orientation algorithm computes all $\succ$-maximal r where $l \succ r$.*

Proof (sketch): If the first statement succeeds then $\forall j \in \{i+1..k\}.l_j^{\uparrow} = r_j^{\uparrow}$ and $l_i^{\uparrow} > r_i^{\uparrow}$ so $\text{MO}(l) >_{revlex} \text{MO}(r)$ and r is maximal. Otherwise, $\forall i.l_i^{\uparrow} \leq r_i^{\uparrow}$ so we flip arrows down to equate out-orders and test $\text{MI}(l) >_{lex} \text{MI}(r)$. If this succeeds, we have a maximal r. Otherwise we still have $\forall i.l_i^{\uparrow} \leq r_i^{\uparrow}$ but flipping arrows in r to equate out-orders is insufficient as r then has a larger in-order. However, by assumption, l has at least one outward wavefront with a least depth i, so we can flip enough arrows at this depth so $r_i = l_i - 1$. Thus $l \succ r$ and r is maximal. □

This parser for simply annotated terms is correct (it only returns wave-rules) and complete (it returns all maximal wave-rules under the orderings we define). As an example, consider (9) with the LHS oriented all out. We begin by orienting both wavefronts in the RHS out. The two sides thus have the measures $\langle[0, 1], [0, 0]\rangle$ and $\langle[1, 1], [0, 0]\rangle$ respectively. Hence step 1 fails. Moreover, if we equate the out-measures by turning down the annotation at depth 0, this gives the RHS a measure of $\langle[0, 1], [1, 0]\rangle$ so step 2 fails. Finally we succeed in step 3 by turning down the arrow at depth 1 giving the RHS a measure of $\langle[1, 0], [0, 1]\rangle$. The resulting oriented annotation is given in (9).

5.3 Multi-waves and sinks

The above ideas generalize easily to multi-wave-rules. For reasons of space we only sketch this. We generate skeleton preserving annotations analogous to the single-hole case but allow multi-holed wavefronts. Usually both sides are simply annotated and we may use the above orientation algorithm. Alternatively, after fixing an orientation for the LHS of the wave-rule we may orient the RHS by cycling through possible orientations. For each orientation we compare the weakenings of the two sides under the multi-set ordering over our measure and we pick the RHS orientation with the greatest measure. There are various ways the efficiency of this can be enhanced. E.g. we need only compute weakenings of each side once; with "orientation variables" we may propagate the different orientations we select for the RHS to orientations on the weakening set before comparison under the multi-set measure.

One kind of annotation we haven't yet discussed in our measures is *sinks* (see §2). This is deliberate as we can safely ignore sinks in both the measure and the parser. Sinks only serve to decrease the applicability of wave-rules by creating additional preconditions; that is, we only ripple inwards if there is a sink underneath the wavefront. But if rippling terminates without such a precondition, it terminates with it as well. Sinks (and also recent additions to rippling such as colours [15]) can be seen as not effecting the termination of rippling but rather the *utility* of rippling. That is, they increase the chance that we will be able to fertilize with the hypothesis successfully.

6 Extensions to Rippling

By introducing new termination orders for rippling, we can combine rippling with conventional term rewriting. Such extensions greatly extend the power and applicability of rippling both within and outwith induction. In addition, by design, our orderings are not dependent upon rippling preserving skeletons. This allows us to use rippling in new domains involving, for example, mutual recursion or definition unfolding where the skeleton needs to be modified; such applications were previously outside the scope of rippling. We feel that these extensions offer the promise of the "best of both worlds": that is, the highly goal directed nature of rippling combined with the flexibility and uniformity of conventional rewriting. To test these ideas, we have implemented an *Annotated Rewrite System*, a simple PROLOG program which manipulates annotated terms, and in which we can mix conventional term rewriting and rippling. All the examples below have been proven by this system.

6.1 Unblocking

Rippling can sometimes become blocked. Usually the blockage occurs due to the lack of a wave-rule to move the differences out of the way; in such a situation the wave-rule may be speculated automatically using techniques presented in [13]. However, sometimes the proof becomes blocked because a wavefront needs to be rewritten so that it matches either a wave-rule (to allow further rippling) or a sink (to allow fertilization). This is best illustrated by an example.

Consider the following theorem, where rev is naive reverse, $qrev$ is tail-recursive reverse using an accumulator, $<>$ is infix append, and $::$ infix cons.

$$\forall L, M.\ qrev(L, M) = rev(L) <> M \tag{15}$$

To prove this theorem, we perform an induction on L. The induction hypothesis is

$$qrev(l, M) = rev(l) <> M$$

and the induction conclusion is

$$qrev(\boxed{h :: \underline{l}}^{\uparrow}, \lfloor m \rfloor) \quad = \quad rev(\boxed{h :: \underline{l}}^{\uparrow}) <> \lfloor m \rfloor\ . \tag{16}$$

where m is a skolem constant which sits in a sink, annotated with "$\lfloor\,\rfloor$".

We will use wave-rules taken from the recursive definition of $qrev$, and rev.

$$rev(\boxed{H :: \underline{T}}^{\uparrow}) \quad \Rightarrow \quad \boxed{\underline{rev(T)} <> (H :: nil)}^{\uparrow} \tag{17}$$

$$qrev(\boxed{H :: \underline{T}}^{\uparrow}, L) \quad \Rightarrow \quad qrev(T, \boxed{H :: \underline{L}}^{\downarrow}) \tag{18}$$

On the LHS, we ripple with (18) to give

$$qrev(l, \left\lfloor \boxed{h :: \underline{m}}^{\downarrow} \right\rfloor) \quad = \quad rev(\boxed{h :: \underline{l}}^{\uparrow}) <> \lfloor m \rfloor\ .$$

On the RHS, we ripple with (17) and then (4), the associativity of $<>$ to get

$$qrev(l, \left\lfloor \boxed{h :: \underline{m}}^{\downarrow} \right\rfloor) \quad = \quad rev(l) <> (\left\lfloor \boxed{(h :: nil) <> \underline{m}}^{\downarrow} \right\rfloor). \tag{19}$$

Unfortunately, the proof is now blocked. We can neither further ripple nor fertilize with the induction hypothesis. The problem is that we need to simplify the wavefront on the righthand side. Clam currently uses an ad-hoc method to try to perform wavefront simplification when rippling becomes blocked. In this case (19) is rewritten to

$$qrev(l, \left\lfloor \boxed{h :: \underline{m}}^{\downarrow} \right\rfloor) = rev(l) <> (\left\lfloor \boxed{h :: \underline{m}}^{\downarrow} \right\rfloor).$$

Fertilization with the induction hypothesis can now occur.

In general, unblocking steps are not sanctioned under the measure proposed earlier, or that given in [3]; their uncontrolled application during rippling can lead to non-termination. But we can easily create new orders where unblocking steps are measure decreasing. These new orders allows us to combine rippling with conventional rewriting of wavefronts in an elegant and powerful way. Namely, unblocking rules will be measure decreasing wave-rules accepted by the parser and applied like other wave-rules.

We define an unblocking ordering by giving (as before) an ordering on simply annotated terms, which can then be lifted to an order on multi-wave terms. To order simply annotated terms, we take the lexicographic order of the simple wave-rule measure proposed above (using size of the wavefront as the notion of weight) paired with $>_{wf}$, an order on the *contents* of wavefronts. As a simply annotated term may still contain multiple wavefronts, this second order is lifted to a measure on sets of wavefronts by taking its multi-set extension. The first part of the lexicographic ordering will ensure that anything which is normally measure decreasing remains measure decreasing and the second part will allow us to orient rules that only manipulate wavefronts. This combination provides a termination ordering that allows us to use rippling to move wavefronts about the skeleton and conventional rewriting to manipulate the contents of these wavefronts.

For the reverse example, the normalization ordering is very simple; we use the following wave-rules.

$$\boxed{nil <> \underline{L}}^{\downarrow} \quad \Rightarrow \quad L \tag{20}$$

$$\boxed{(H :: T) <> \underline{L}}^{\downarrow} \quad \Rightarrow \quad \boxed{H :: (T <> \underline{L})}^{\downarrow} \tag{21}$$

The first is already parsed as a wave-rule using our standard measures, but we need to add the second. This rule doesn't change the size of the wavefront or its position but only its form. Hence we want this to be decreasing under some normalization ordering. There are many such orderings; here we take $>_{wf}$ to be the recursive path ordering [7] on the terms in the wavefront where $<>$ has a higher precedence than $::$ and all other function symbols have an equivalent but lower priority. The measure of the LHS of (21) is now greater than that of the RHS as its wavefront is $(H :: T) <> *$ which is greater than $H :: (T <> *)$ in the recursive path ordering (to convert wavefronts into well formed terms, waveholes are marked with the new symbol $*$).

Unblocking steps which simplify wavefronts are useful in many proofs enabling both immediate fertilization (as in this example) and continued rippling. Wavefronts can even be unblocked using a different set of rules to that used for rippling.

6.2 Mutual Recursion and Skeleton Simplification

Rippling can also become blocked because the skeleton (and not a wavefront) needs to be rewritten. This happens in proofs involving mutually recursive functions, definition unfolding, and other kinds of rewriting of the skeleton. Consider

$$\forall x. even(s(s(0)) \times x)$$

where *even* has the following wave-rules.

$$even(\boxed{s(\underline{U})}^{\uparrow}) \quad \Rightarrow \quad odd(U) \tag{22}$$

$$odd(\boxed{s(\underline{U})}^{\uparrow}) \quad \Rightarrow \quad even(U) \tag{23}$$

Note that (22) and (23) are not wave-rules in the conventional sense since they are not skeleton preserving. However, they do decrease the annotation measure. Rules (22) and (23) can be viewed as a more general type of wave-rule of the form $LHS \Rightarrow RHS$ which satisfy the constraint $skeleton(LHS) \equiv skeleton(RHS)$ where $\equiv$ is some equivalence relation. In this case, the equivalence relation includes the granularity relation in which $even(x)$ and $odd(x)$ are in the same equivalence class. Rippling with this more general class of wave-rules still gives us a guarantee of termination. However weakening the structure preservation requirement can reduce the utility of rippling since now we are only guaranteed to rewrite the conclusion into a member of the equivalence class of the hypothesis.

To prove the theorem, we will also need the following wave-rules.

$$\boxed{s(\underline{U})}^{\uparrow} + V \quad \Rightarrow \quad \boxed{s(\underline{U+V})}^{\uparrow} \tag{24}$$

$$U + \boxed{s(\underline{V})}^{\uparrow} \quad \Rightarrow \quad \boxed{s(\underline{U+V})}^{\uparrow} \tag{25}$$

The theorem can be proved without (25) but this requires a nested induction and generalization, complications which need not concern us here.

The proof begins with induction on x. The induction hypothesis is

$$even(s(s(0)) \times n)$$

and the induction conclusion is

$$even(s(s(0)) \times \boxed{s(\underline{n})}^{\uparrow}). \tag{26}$$

Unfortunately rippling is immediately blocked. To continue the proof, we simplify the skeleton of the induction conclusion by exhaustively rewriting (26) using the unannotated version of (1) and the following rules.

$$0 \times V \quad \Rightarrow \quad 0 \tag{27}$$

$$0 + V \quad \Rightarrow \quad V \tag{28}$$

This gives

$$even(\boxed{s(\underline{n})}^{\uparrow} + \boxed{s(\underline{n})}^{\uparrow}). \tag{29}$$

Note that the skeleton was changed by this rewriting. The induction hypothesis can, however, be rewritten using the same rules so that it matches the skeleton of (29). Of course, arbitrary rewriting of the skeleton may not preserve the termination of rippling. To justify these unblocking steps we therefore introduce a new termination order which combines lexicographically a measure on the skeleton with the measure on annotations.[4] We then admit rewrite rules provided their application decreases this combined measure. This new order allows us to combine rippling with conventional rewriting of the skeleton in an elegant and powerful way. In this case, the

[4] With more complex theorems, the height of the skeleton may increase; the addition of the height of the skeleton to the order ensures termination in such situations.

recursive path order on skeletons (with precedence $\times > + > s > 0$) is again adequate. Note that though termination is guaranteed, again skeleton preservation has been weakened. Since the skeleton can be changed during rippling, we are no longer able to guarantee that we can fertilize at the end of rippling. However, provided the skeleton is rewritten identically in both the hypotheses and the conclusion, we will still be able to fertilize.

To return to the proof, rippling (29) with (24) gives

$$even(\boxed{s(\underline{n + \boxed{s(\underline{n})}^{\uparrow}})}^{\uparrow}).$$

Then with (25) gives

$$even(\boxed{s(s(\underline{n+n}))}^{\uparrow}).$$

We now ripple with the mutually recursive definition of even, (22),

$$odd(\boxed{s(\underline{n+n})}^{\uparrow}).$$

Note that this step also changes the skeleton. However, as the measure decreases and as the skeleton stays in the same equivalence class, such rewriting is permitted. Finally rippling with (23) gives

$$even(n+n).$$

This matches the (rewritten) induction hypothesis and so completes the proof.

The power of rippling is greatly enhanced by its combination with traditional rewriting. For example, proofs involving mutually recursive functions, or other kinds of skeleton simplification (e.g., definition unfolding) were not previously possible with rippling. The use of conventional term rewriting to simplify the skeleton is a natural dual to the use of conventional rewriting to simplify wavefronts; indeed they are orthogonal and can be combined to allow even more sophisticated rewriting.

6.3 Other Applications

Rippling has found several novel uses of outside of induction. For example, it has been used to sum series [14], to prove limit theorems [15], and guide equational reasoning [11]. However, new domains, especially non-inductive ones, require new orderings to guide proof. For example, consider the PRESS system [6].[5] To solve algebraic equations, PRESS uses a set of methods which apply rewrite rules. The three main methods are: *isolation*, *collection*, and *attraction*. Below are examples of rewrite rules used by each of these methods.

$$\begin{array}{lrcl}
ATTRACTION: & \boxed{\log(\underline{U}) + \log(\underline{V})}^{\uparrow} & \Rightarrow & \boxed{\log(\underline{U} \times \underline{V})}^{\uparrow} \\
COLLECTION: & \boxed{\underline{U} \times \underline{U}}^{\uparrow} & \Rightarrow & \boxed{\underline{U}^2}^{\uparrow} \\
ISOLATION: & \boxed{\underline{U}^2}^{\uparrow} = V & \Rightarrow & U = \boxed{\pm\sqrt{\underline{V}}}^{\downarrow}
\end{array}$$

PRESS uses preconditions and not annotation to determine rewrite rule applicability. Attraction must bring occurrences of unknowns closer together. Collection must reduce the number of occurrences of unknowns. Finally, isolation must make progress towards isolating unknowns on the LHS of the equation. These requirements can easily be captured by annotation and PRESS can thus be implemented

[5] Due to space constraints, we only sketch this application. The idea of reconstructing PRESS with rippling was first suggested by Nick Free and Alan Bundy.

by rippling. The above wave-rules suggest how this would work. PRESS wave-rules are structure preserving, where the preserved structure is the unknowns. The ordering defined on these rules reflects the well-founded progress achieved by the PRESS methods. Namely, we lexicographically combine orderings on the number of waveholes for collection, their distance (shortest path between waveholes in term tree) for attraction, and our width measure on annotation weight for isolation.

7 Related Work and Experience

The measures and orders we give are considerably simpler than those in [3]. There, the properties of structure preservation and the reduction of a measure are intertwined. Bundy *et al.* describe wave-rules schematically and show that any instance of these schemata is skeleton preserving and measure decreasing under an appropriately defined measure. Mixing these two properties makes the definition of wave-rules very complex. For example, the simplest kind of wave-rule proposed are so-called *longitudinal* wave-rules (which ripple-out) defined as rules of the form,

$$\eta(\boxed{\xi_1(\underline{\mu_1^1},..,\underline{\mu_1^{p_1}})}^{\uparrow},..,\boxed{\xi_n(\underline{\mu_n^1},..,\underline{\mu_n^{p_n}})}^{\uparrow}) \Rightarrow \boxed{\zeta(\underline{\eta(\varpi_1^1,..,\varpi_n^1)},..,\underline{\eta(\varpi_1^k,..,\varpi_n^k)})}^{\uparrow}$$

that satisfy a number of side conditions. These include: each ϖ_i^j is either an unrippled wavefront, $\boxed{\xi_i(\underline{\mu_i^1},\ldots,\underline{\mu_i^{p_i}})}$, or is one of the waveholes, μ_i^l; for each j, at least one ϖ_i^j must be a wavehole. η, the ξ_is, and ζ are terms with distinguished arguments; ζ may be empty, but the ξ_is and η must not be. There are other schemata for *traverse* wave-rules and *creational* wave-rules[6]. These schemata are combined in a general format, so complex that in [3] it takes four lines to print. It is notationally involved although not conceptually difficult to demonstrate that any instance of these schemata is a wave-rule under our size and width measures.

Consider the longitudinal schema given above. It is clear that evey skeleton on the RHS is a skeleton of the LHS because of the constraint on the ϖ_j^i. What is trickier to see is that it is measure decreasing. Under our order this is the case if LHS $\succ^*$ RHS. We can show something stronger, namely, for every $r \in$ *weakenings*(RHS). $\exists l \in$ *weakenings*(LHS). $l \succ r$. To see this observe that any such r must be a maximal weakening of

$$r' = \boxed{\zeta(\eta(\varpi_1^1,\ldots,\varpi_n^1),\ldots,\underline{\eta(\varpi_1^j,\ldots,\varpi_n^j)},\ldots\eta(\varpi_1^k,\ldots,\varpi_n^k))}^{\uparrow}$$

for some $j \in \{1..k\}$. Corresponding to r' is an l' which is a weakening of the LHS where $l' = \eta(t_1,...,t_n)$ and the t_i correspond to the ith subterm of $\eta(\varpi_1^j,\ldots,\varpi_n^j)$ in r': if ϖ_i^j is an unrippled wavefront then $t_i = \varpi_i^j = \boxed{\xi_i(\underline{\mu_i^1},\ldots,\underline{\mu_i^{p_i}})}$, and alternatively if ϖ_i^j a wavehole μ_i^l then $t_i = \boxed{\xi_i(\mu_i^1,\ldots,\underline{\mu_i^l},\ldots,\mu_i^{p_i})}$. Now r is a maximal weakening of r' so there is a series of weakening steps from r to r'. Each of these weakenings occurs in a ϖ_i^j and we can perform the identical weakening steps in the corresponding t_i in l' leading to a maximal weakening l. As l and r are maximally weak they may be compared under $\succ$. Their only differences are that r has an additional wavefront at its root and is missing a wavefront at each ϖ_i^j corresponding to a wavehole. The depth of ϖ_i^j is greater than the root and at this depth the

[6] Creational wave-rules are used to move wavefronts between terms during induction proofs by destructor induction. They complicate rippling in a rather specialized and uninteresting way. Our measures could be easily generalized to include such creational rules.

out-measure of l is greater than r (under any of the weights defined in §3) and at all greater depths they are identical. Hence $l \succ r$.

Similar arguments hold for the other schemata given in [3] and from this we can conclude that wave-rules acceptable under their definition are acceptable under ours. Moreover it is easy to construct simple examples that are wave-rules under our formalism but not theirs; for example, the following two rules are measure decreasing but are not instances of their schema.

$$rot(\boxed{s(\underline{X})}^{\uparrow}, \boxed{H :: \underline{T}}^{\uparrow}, Acc) \Rightarrow rot(X, T, \boxed{H :: \underline{Acc}}^{\uparrow})$$

$$\boxed{0 + \underline{X}}^{\uparrow} \Rightarrow X$$

Aside from being more powerful, there are additional advantages to the approach taken here. Our notion of wave-rules and measures are significantly simpler and therefore easier to understand. As a result, they are easier to implement. The definition of wave-rules given in [3] is not what is recognized by the Clam wave-rule parser as it returns invalid wave-rules under either our definition or that of [3] and misses many valid ones. For example, Clam's current parser fails to find even wave-rules as simple as the following.

$$divides(\boxed{\underline{X} + Y}^{\uparrow}, Y) \quad \Rightarrow \quad \boxed{s(\underline{divides(X, Y)})}^{\uparrow}$$

We have therefore implemented the parser described in §5. The parser is simple, just a couple of pages of Prolog, yet allows new orderings based on different annotation measures to be easily incorporated. Although parsing is in the worst case exponential in the size of the rewrite rule, the parser typically takes under 5 seconds to return a complete set of maximal wave-rules (which seems reasonable for an off-line procedure). The parser is part of our annotated rewrite system and will be shortly integrated into the Clam theorem prover.

8 Conclusions

An ordering for proving the termination of rippling along with a schematic description of wave-rules was first given in [3]. We have simplified, generalized and improved both this termination ordering, and the description of wave-rules. In addition, we have shown that different termination orderings for rippling can be profitably used within and outwith induction. Such new orderings can combine the highly goal directed features of rippling with the flexibility and uniformity of more conventional term rewriting. We have, for example, given two new orderings which allow unblocking, definition unfolding, and mutual recursion to be added to rippling in a principled (and terminating) fashion; such extensions greatly extend the power of the rippling heuristic. To support these extensions, we have implemented a simple *Annotated Rewrite System* which annotates and orients rewrite rules, and with which we can rewrite annotated terms. We have used this system to perform experiments combining rippling and conventional term rewriting. We confidently expect that this combination of rippling and term rewriting has an important rôle to play in many areas of theorem proving and automated reasoning.

References

[1] D. Basin and T. Walsh. Difference unification. In *Proceedings of the 13th IJCAI*. International Joint Conference on Artificial Intelligence, 1993.

[2] R.S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, 1979. ACM monograph series.

[3] 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.

[4] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*. 1990.

[5] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag, 1990.

[6] A. Bundy and B. Welham. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence*, 16(2):189–212, 1981.

[7] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.

[8] N. Dershowitz. Termination of Rewriting. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*. Academic Press, 1987.

[9] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. North-Holland, 1990.

[10] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Comms. ACM*, 22(8):465–476, 1979.

[11] D. Hutter. Guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*. 1990.

[12] D. Hutter. Colouring terms to control equational reasoning. An Expanded Version of PhD Thesis: Mustergesteuerte Strategien für Beweisen von Gleichheiten (Universität Karlsruhe, 1991), in preparation.

[13] A. Ireland and A. Bundy. Using failure to guide inductive proof. Technical report, Dept. of Artificial Intelligence, University of Edinburgh, 1992.

[14] T. Walsh, A. Nunes, and A. Bundy. The use of proof plans to sum series. In D. Kapur, editor, *11th Conference on Automated Deduction*. 1992.

[15] T. Yoshida, A. Bundy, I. Green, T. Walsh, and D. Basin. Coloured rippling: the extension of a theorem proving heuristic. Technical Report, Dept. of Artificial Intelligence, University of Edinburgh, 1993. Under review for ECAI-94.

Recent Publications in the BRICS Notes Series

NS-94-2 **David Basin.** ***Induction Based on Rippling and Proof Planning. Mini-Course*****. August 1994, 62 pp.**

NS-94-1 **Peter D. Mosses, editor.** ***Proc. 1st International Workshop on Action Semantics*** **(Edinburgh, 14 April, 1994), number NS-94-1 in BRICS Notes Series, Department of Computer Science, University of Aarhus, May 1994. BRICS. 145 pp.**